RAL-92-010

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-92-010

Reasoning about VDM Developments using The VDM Support Tool in Mural

J C Bicarregui and B Ritchie

January 1992

Reasoning about VDM Developments using The VDM Support Tool in Mural

J.C. Bicarregui and B. Ritchie
Systems Engineering Division
Informatics Department
Rutherford Appleton Laboratory

Abstract

Mural is an interactive mathematical reasoning environment designed to assist the kind of theorem proving tasks that arise when following a formal methods approach to software engineering. It is the result of work carried out at Manchester University and the Rutherford Appleton Laboratory under the Alvey IPSE 2.5 project.

Considerable design emphasis has been placed upon the user interface, using the power of workstation technology to present information and to give the user freedom of action backed up by careful dependency tracking. Through this emphasis on the user interface it is hoped to enable users to maintain their intuition of the problem domain and hence guide the proof in the right direction, whilst the mechanical symbolic manipulation of the machine can maintain the integrity of the proof.

The Mural proof assistant is generic in that it can be instantiated for reasoning in a variety of logics. Logical theories are constructed in a hierarchical store where collections of declarations and axioms are structured along with derived rules and their proofs. Some effort has been spent on the instantiation of the proof assistant for the formal method VDM. This instantiation includes theories of the logic LPF upon which VDM is based, and of the basic types and functions of VDM.

The system includes tools for the construction of VDM specifications and reifications between them and for the generation of the proof obligations that provide the basis of the formal verification of the refinement relationship. It also supports the construction of theories in the proof assistant where it is possible to reason about specifications, reifications and proof obligations. Though there are many more features that would be desirable in a complete environment for VDM, this degree of support has shown that the Mural proof assistant could be used as an integral part of a generic support environment including provision for the formal development of software.

This paper concentrates upon the VDM support aspects of Mural: how users can build specifications and reifications between them; and how these are "translated" into Mural theories including the generation of the corresponding proof obligations.

Keywords: formal methods, formal specification, refinement, proof, integrated support environments.

1 Introduction

One frequently cited reason for the slow take-up of formal methods in industry is the lack of tools for supporting the processes involved. Tools are required that can provide integrated support for the full formal development method, including the construction of specifications,

incremental refinement of specifications to implementations, the generation of proof obligations for the verification of the refinements and the formal reasoning required to discharge them. Such tools could also address the management of the formal objects involved. However, fully integrated tools for all the processes of formal software development may still be some way off.

The Mural system, developed by Manchester University and Rutherford Appleton Laboratory under the Alvey funded IPSE 2.5 project, is primarily concerned with providing generic support for what is perhaps the most intricate of these processes, that is the construction of fully formal mathematical proofs. The project has also concerned itself with how this proof assistant could be integrated with support for the other processes of formal development. To this end, support facilities for specification and verification for the formal development method VDM have been built alongside the formal reasoning assistant. With this combined facility it is possible to construct VDM specifications and reifications between them and generate the theories in which proof obligations can be discharged, thus providing a level of integration not previously available.

This paper discusses the support for VDM provided by the Mural system. We assume some knowledge of VDM, including data reification and operation modelling.

2 Background

2.1 The Mural System

The major component of Mural is a generic mathematical reasoning environment designed to assist in the theorem proving tasks that arise when following a formal methods approach to software engineering. Using a carefully specified and constructed kernel, and through the use of "state-of-the-art" user interface design, Mural is intended to harness the mechanical precision of the computer in combination with the intuitive insight of the user. Mural is founded on a generic logic that can be instantiated to a variety of logics [1] and thus is capable of providing support for the reasoning required in many formal development methods. Logical theories are organised in a hierarchical store where collections of declarations and axioms are structured along with derived rules and their proofs. Proofs are constructed interactively in a natural deduction style. The user is given the freedom to construct the proof in any manner – top-down, bottom-up, middle-out,..., with Mural providing assistance in determining which existing rules in the theory store are applicable at each stage, and checking correctness of the reasoning in the proof.

The second component of Mural is a support facility for VDM, the VDM Support Tool (VST). This facility provides support for the construction of VDM specifications and refinements between them. It also generates theories in the hierarchy in which the specifications can be reasoned about together with the proof obligations that must be discharged to verify the design decisions. Considerable effort has also gone into populating the proof assistant with the theories of the basic constructs of VDM. By intention, the Mural VDM support tool does not provide full support for specification and refinement in VDM, as the primary reason for its development was to investigate how a fuller support facility could be integrated with the proof assistant. Only a small part of the VDM specification language is catered for, and only a limited notion of reification is supported. Operation decomposition is not addressed. At present, the only way to determine the status of proofs about a specification or reification is by looking at the status of the rules in the corresponding Mural theories. The VST does however provide a level of integration of support for the processes of specification refinement and proof that was not previously available. Though limited in its original objectives,

the degree of support provided has attracted much interest and the system has been used on several non-trivial specifications as part of evaluation work.

2.2 Related Work

Following almost twenty years of development and practical use in a wide variety of applications, work is currently under way to develop a BSI standard for the VDM specification language [2]. The standardisation effort concentrates upon the abstract and concrete syntaxes but also addresses semantics and context conditions. It is hoped that this work will be adopted by ISO. The standardisation effort, however, does not consider issues related to refinement and proof.

The use of VDM as a development method including refinement and proof is described in [3]. There have been a number of papers addressing semantic issues in VDM. In [4, 5], the formulation of the proof obligations are justified with respect to a semantic model for VDM. Monahan [6] has presented a semantic model for VDM type definitions; this has been used by Arentoft and Larsen as the starting point for a semantic model for BSI-VDM [7]. Operation decomposition has been addressed by Milne [8] and Ah-Kee [9]. A new formulation of some of the proof rules for operation decomposition is presented in [3].

In addition to Mural, there are a number of other tools for the construction of VDM specifications. STC's VDM Toolset [10] provides a structure editor for specifications, which performs some static type-checking and syntax conformance. Adelard's SpecBox provides a parser for BSI-VDM which performs arity checking. IST's GENESIS has been instantiated for VDM, giving a structure editor and static semantics checker; there is also a proof editor for LPF, but proof obligations are not generated from VDM specifications.

A number of tools exist which provide support for specification construction and (patent) type-checking in Z, including Spivey's "Fuzz". Several proof tools have been used to discharge Z "proof opportunities" (including the "B" tool [13] and the Boyer-Moore theorem prover). There have been some recent interesting developments in reasoning about Z specifications by embedding the notation in other systems, in the "zedB" rule base for the B tool, and in the FST project's work on Z in HOL.

The RAISE project [14] developed a "wide-spectrum" specification language in which specifications can be written in both model-oriented and algebraic styles. Part of the RAISE work has involved the design of tools to support the construction of specifications, though little or no support for formal proof is provided.

3 A Small Example

In this section we give an example of a small development in the VDM support tool. The example is a simple address book with a lookup operation.

Figure 1 shows the development browser, which comprises two (duplicate) lists of specifications ("abstract" and "concrete", relative to the reifications) and a list of reifications. Each reification links two specifications, with the intention of showing that one specification can be considered a "more concrete" version of the other. When a specification is selected in the "abstract" list, the list of reifications is filtered to show only those reifications "from" that specification, and similarly for the "concrete" list. When a specification is selected in each list, it is possible to add a new reification between them (in addition to any already present). Selecting a reification causes its abstract and concrete specifications to be selected in the two lists, as in the figure.

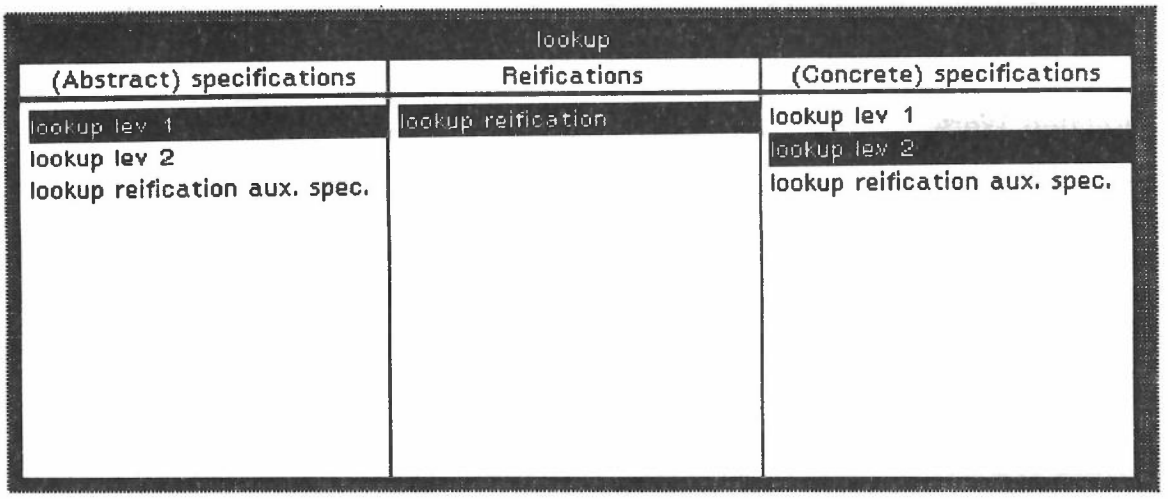

Figure 1: **A development browser**

From the development browser, we can add, browse and edit specifications using a specification browser. Figure 2 shows the specification browser on the level 1 (most abstract) specification of "lookup". This consists of lists of the type, constant, function and operation

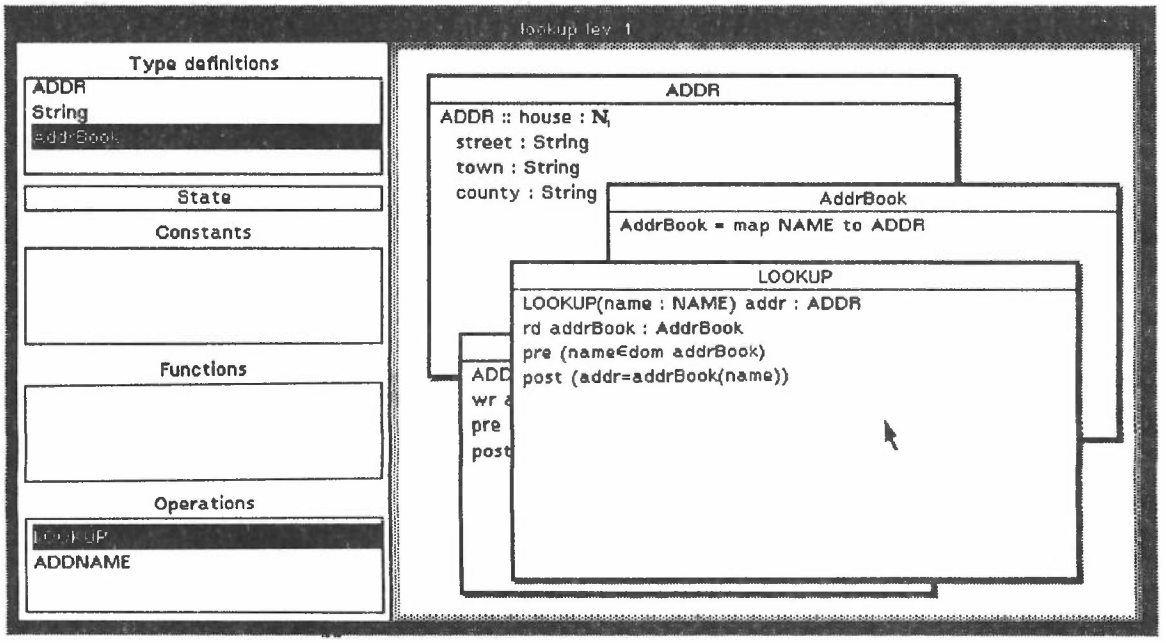

Figure 2: **Specification browser on the "abstract" lookup specification**

definitions in the specification, a button to show the state, and a "noticeboard" in which the definitions can be displayed and edited in individual "notices". (The noticeboard is a miniature window environment in which notices may be moved, resized, iconised and exposed). The visible notices here show definitions of the types $ADDR$ and $AddrBook$, and the specification of the $LOOKUP$ operation. At this level, the address book is simply a map from names to addresses, and $LOOKUP$ is specified in terms of map functions. These definitions were constructed using a structure editor which uses an abstract syntax drawn from the BSI draft standard.

Another instance of the browser on the "level 2 lookup" specification is shown in fig-

ure 3. The address book in this case is a sequence of pairs of names and addresses. To

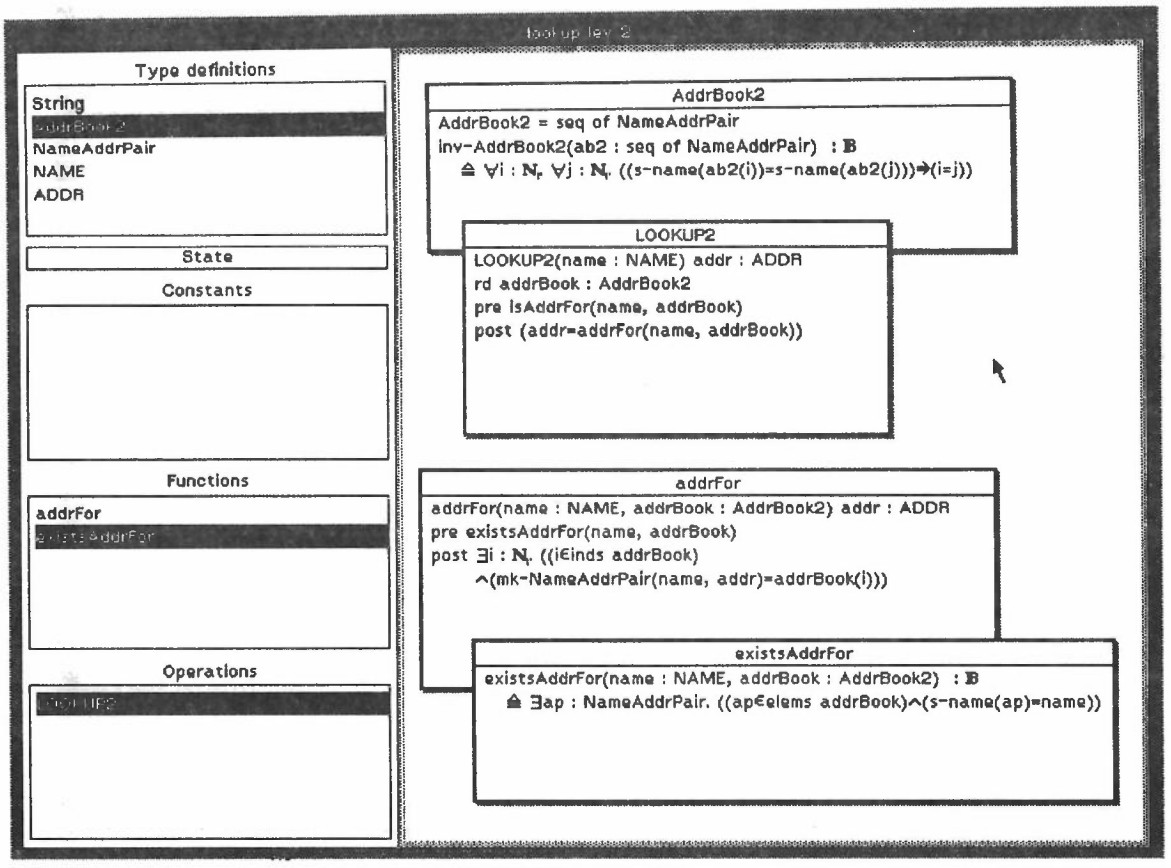

Figure 3: **Specification browser on the "concrete" lookup specification**

ensure that it is still "map-like" we need to restrict this by an invariant that no two distinct elements of the sequence can have the same name. To simplify the specification of the new operation *LOOKUP2*, we choose to define two auxiliary function definitions, *addrFor* (which is implicitly defined) and *existsAddr* (which is explicitly[1] defined).

When we construct a reification between two specifications, or look at an existing reification, it is presented in a reification browser (figure 4). This contains component lists for the two specifications, and lists for definitions specific to the reification (e.g. auxiliary functions used to define retrieve functions). Components of specifications may be viewed via notices in an attached noticeboard. At present, the reification browser is incomplete; the only interesting thing we can do here is to select an operation in each specification and create an operation model between them. Doing this for the operations *LOOKUP* and *LOOKUP2* gives us an operation modelling browser (figure 5). This lists the arguments, read and write externals and result of the two operations (the operation may also be viewed as a notice in the attached noticeboard), and a central list which is a "pool" of available data reifications. The component lists of the two operations are tied, so that when (for example) the (single) read-only state component of the abstract operation is selected, the corresponding component of the concrete operation is also selected. The intention is to link each of these component-pairs by a data retrieval describing how the type of the concrete component can be viewed as a data reification of the type of its abstract counterpart. When a pair is selected,

[1] Note: though explicit, this definition is non-constructive, through its use of existential quantification; however, the function is "clearly implementible" in this case.

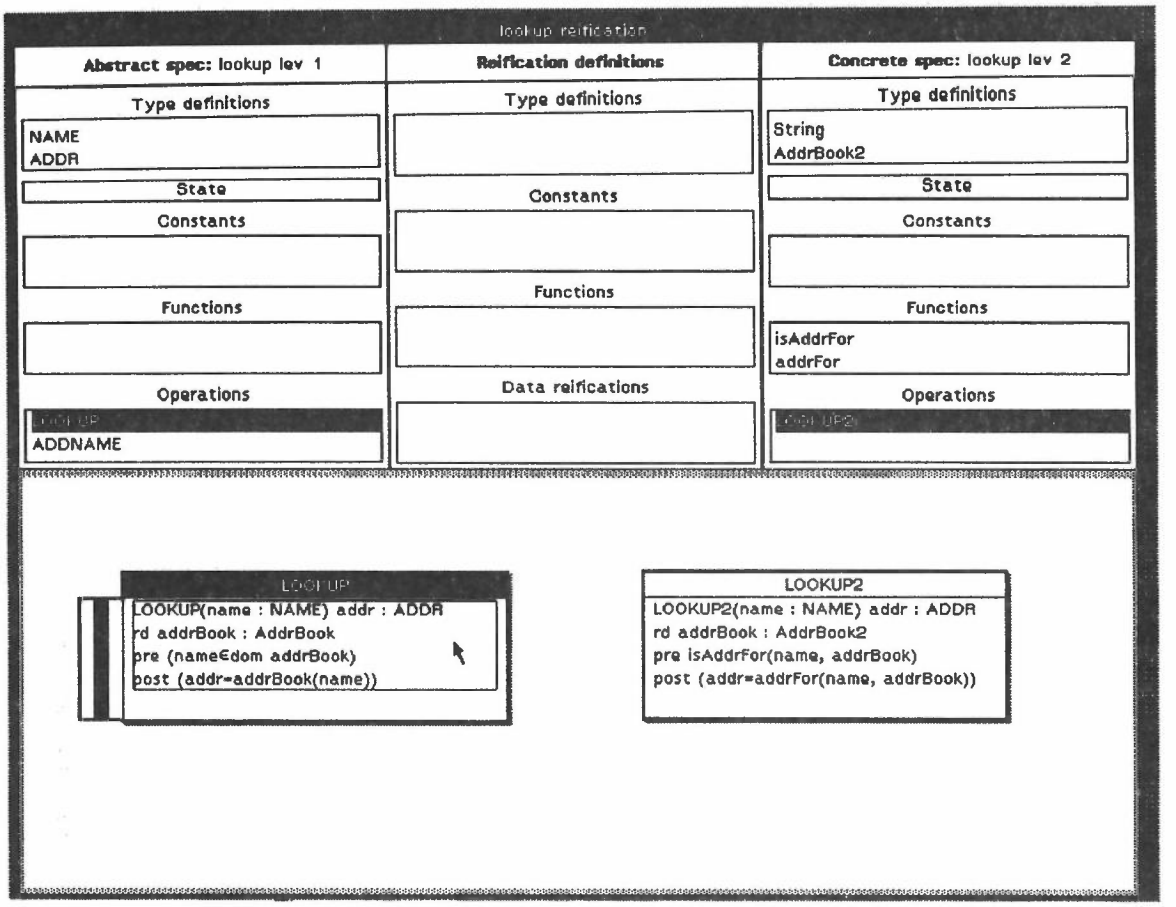

Figure 4: **A reification browser**

this can be done by selecting a data reification in the central list and "tying" it to the pair. These links are used in the generation of the domain and result obligations arising from the operation model. The noticeboard in the example shows a data reification between $AddrBook$ and $AddrBook2$; in addition to the abstract and concrete types, this contains the definition of the retrieve function from $AddrBook2$ to $AddrBook$, here defined in terms of an auxiliary function $mapFromSeq$. Note that we permit interface refinement (i.e. reification of argument and result types); and that our reification model is restricted to those cases where the abstract and concrete operations have the same number of read and write components.

We are free to construct abstract and concrete specifications in any order, and the components of each specification can be added in any order. It is even possible to begin building a reification between two specifications before they are complete (indeed, we have done so here).

4 Theories from Developments

In this section we describe the approach taken in the translation of VDM specifications and reifications into the Mural theory store, to create theories within which we can reason about the proof obligations that arise. As Mural is generic, there is a "once-off" effort to instantiate the theory store for VDM, creating a hierarchy of theories (propositional and predicate LPF,

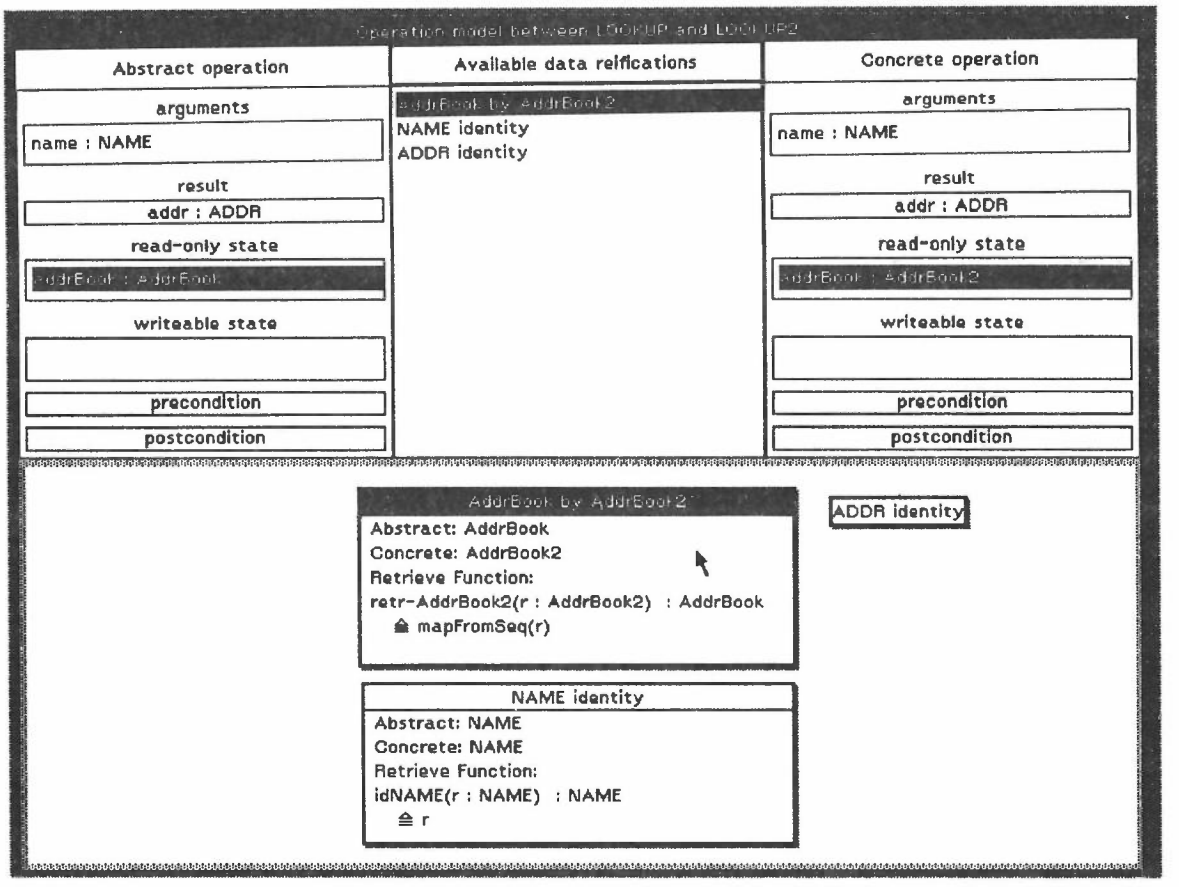

Figure 5: **An operation model browser**

numbers, sets, maps, sequences etc.) culminating in a "VDM Primitives" theory. This theory should be an ancestor of every theory generated from a specification or reification in the VDM support tool. Such a theory store is not static: not only will new theories be created corresponding to new specifications and reifications, but more derived rules can be added to the VDM Primitives hierarchy as required.

4.1 Notation used for Mural theory components

There is insufficient space in this paper to describe Mural's generic logic; here we give only a brief summary of the notation used in this paper to describe components of Mural theories.

The Mural theory store is organised as a hierarchy of individual "theories". Each theory can declare new type and expression symbols, which can be defined either through axioms or by direct definitions. The syntax permits higher-order metavariables and includes dependent type constructs, but beta-reduction is not built-in (to make matching of proof rules decidable). In addition to axioms, a theory can also contain derived rules; each rule may have an associated proof which justifies it with respect to the axioms or other derived rules. (It is possible to use derived rules in proofs before they have been proven; Mural's dependency tracking can determine the set of unproven rules used in a proof (directly or indirectly), and will also prevent circular arguments). A theory can have one or more parent theories, in which case all the declarations, definitions, axioms and rules of these theories (and their ancestors) may be used in the theory.

We will use the following notation to describe components of Mural theories[2].

[2]This notation is not precisely the same notation as used in Mural, and does not capture the full power of

A Mural *declaration* such as

$$mk\text{-}T \mapsto [2, 0]$$

declares a constant symbol $mk\text{-}T$ of expression arity 2 and type arity 0. We will use this notation for both expression and type declarations. If $mk\text{-}T$ is an expression constant declared as above, then we may construct expressions such as $mk\text{-}T[x, y]$ for any expressions x and y. Whether or not such an expression is meaningful depends upon axioms (e.g. formation rules) given in the theory[3].

Expression and type symbols may be bound to *definitions*, e.g.

$$\wedge \mapsto \neg(\neg([\![e1]\!]) \vee \neg([\![e2]\!]))$$

defines $\wedge$ as an expression constant of (expression) arity 2 (using $[\![e1]\!]$ and $[\![e2]\!]$ as "placeholders" in the definition). Instances of definitions in expressions may be folded and unfolded; for example the expressions

$$\wedge[P, Q] \quad \text{and} \quad \neg(\neg P \vee \neg Q)$$

are interchangeable. (In Mural, it is also possible to specify a "pretty-printing" format for each constant (declared or defined). For $\wedge$, this is defined as $([\![e1]\!] \wedge [\![e2]\!])$, so $\wedge[P, Q]$ will actually appear as $(P \wedge Q)$. In much of what follows, we will use pretty-printed notation without introduction.)

We will use the following notation for inference rules:

$$\boxed{\text{mk-T formn}}(a, b)\frac{a: A, b: B}{mk\text{-}T(a, b): T}$$

defines an inference rule (which may be either an axiom or a derived rule) called "mk-T formn". This rule has expression metavariables a and b (which may be "instantiated" to any expressions when applying the rule). It has two hypotheses (which in this case are typing assertions) and a conclusion. This rule states that if we have two expressions a and b of types A and B respectively, then the expression $mk\text{-}T[a, b]$ is of type T.

4.2 Theories from Specifications

How best to divide the objects corresponding to specifications into theories is largely a matter of taste. For example, we could try to place each type definition in a theory of its own; this would allow reuse of the type definition when reasoning about other specifications. However, we would then have to make the theory structure reflect the inter-dependency of type definitions. We choose a simpler approach where each specification becomes a single Mural theory. If a "finer-grained" structure is required, then the theories must be rebuilt by hand.

The "translated theory" of a specification is a single Mural theory, with the VDM Primitives theory as its parent. The theory contains:

- Mural type definitions, type formation and checking axioms corresponding to the type definitions of the specification;

Mural's declaration mechanism and inference rules (for example, an inference rule may also contain sequent hypotheses), but is sufficient for our purposes.

[3] All of the constants we will declare during the translation of specifications and reifications will have type arity 0. An example of a constant with a non-zero type arity is the set type constructor X-set, which is declared as $setof \mapsto [0, 1]$ (i.e. it takes one type as an argument).

- definitions of any constants and explicit functions;
- definitions of the pre- and post-conditions of implicit functions and operations;
- (initially unproven) rules corresponding to proof obligations and type-checking (including invariant checking) of functions and operations

As we shall see later, many proof obligations can be presented as extra hypotheses to axioms, rather than as unproven rules. In such cases, the axiom can then be used in practice only when the proof obligation is discharged.

The translation of specifications is incremental, but demand-driven: for example, if a function definition is modified after translation, then its translation is undone, but will be regenerated if the specification is subsequently re-translated.

4.3 Translation of types and type definitions

Generally, a type definition in a specification will be translated to a similar type definition in Mural. Most of the type constructors of VDM can be described in "general" theories which form part of the VDM Primitives theory. Sets, maps, sequences, (binary) type unions and optional types are in this category. Type definitions using these constructors can be translated as Mural definitions using instances of the generic constructors. N-ary type unions can be translated as a composition of binary unions.

4.3.1 Invariants

For a definition of type T with an invariant where the type shape uses the above constructors, translation is straightforward. First, the definition of the invariant $inv\text{-}T$ (an explicit function definition) is translated (as will be described later). Then the type definition is translated to a $SubType$ definition. For example, the type definition:

$T = Texp$

where

inv-T(v) $\triangleq$ P(v)

is translated to a definition of $inv\text{-}T$ and to the Mural definition of the type symbol T:

$$T \mapsto < v: Texp' \mid inv\text{-}T(v) >$$

where $Texp'$ is the translation of the type shape $Texp$. This defines type T as a subtype of type $Texp'$ such that all values v of type T satisfy the predicate $inv\text{-}T(v)$. Note that we adopt the view that the invariant is an intrinsic part of the type T, and that there is no intermediate type "T without the invariant".

4.3.2 Composite types

Composite type definitions require an alternative treatment. As different composite type definitions may have different numbers of fields, and as Mural does not allow constants of variable arities, it is not possible (or rather, not easy) to define a general "composite type constructor" in Mural. Therefore composite type shapes cannot be translated into Mural type expressions directly. Furthermore, a greater amount of information is associated with

composite types than other type constructions, in that a composite type definition also defines constructor and destructor functions. Instead, we create an axiomatic definition, with formation and typing rules for expressions involving the constructor and destructors.

For example, the composite type definition

$T :: f1 : A$
$\quad\quad f2 : B$

should be "translated" to:

a declaration of the constructor function (an expression constant with expression arity two):

$mk\text{-}T \mapsto [2, 0];$

declarations of the destructor functions (expression constants with expression arity one):

$s\text{-}f1, s\text{-}f2 \mapsto [1, 0];$

appropriate typing axioms:

$$\boxed{\text{s-f1 formn}}(t)\frac{t\colon T}{s\text{-}f1(t)\colon A}$$

$$\boxed{\text{s-f2 formn}}(t)\frac{t\colon T}{s\text{-}f2(t)\colon B}$$

$$\boxed{\text{mk-T formn}}(a, b)\frac{a\colon A, b\colon B}{mk\text{-}T(a, b)\colon T}$$

and axioms defining $s\text{-}f1$ and $s\text{-}f2$:

$$\boxed{\text{s-f1 intro}}(a, b)\frac{mk\text{-}T(a, b)\colon T}{s\text{-}f1(mk\text{-}T(a, b)) = a}$$

$$\boxed{\text{s-f2 intro}}(a, b)\frac{mk\text{-}T(a, b)\colon T}{s\text{-}f2(mk\text{-}T(a, b)) = b}$$

(note the use of typing assertions to ensure that $mk\text{-}T(a, b)$ is well-formed)

$$\boxed{\text{mk-T intro}}(t)\frac{t\colon T}{mk\text{-}T(s\text{-}f1(t), s\text{-}f2(t)) = t}$$

When a composite type definition has an associated invariant, this must also be declared and mentioned in the $mk\text{-}T$ formation axiom, e.g.:

$inv\text{-}T \mapsto [2, 0]$

$$\boxed{\text{mk-T formn}}(a, b)\frac{a\colon A, b\colon B, inv\text{-}T(a, b)}{mk\text{-}T(a, b)\colon T}$$

and we must also provide an axiom for asserting the invariant:

$$\boxed{\text{inv-T intro}}(a, b)\frac{mk\text{-}T(a, b)\colon T}{inv\text{-}T(a, b)}$$

It should be pointed out that this is only a partial translation, as an induction scheme for T is not generated.

4.4 Translation of function definitions

4.4.1 Definition of pre- and post-conditions

A VDM definition of a function f declares a new function name for use in expressions. Therefore the first part of the translation creates a Mural declaration of an expression symbol f of the appropriate arity. If the function has a pre-condition, then a definition of the symbol *pre-f* is also created; similarly for the post-condition of an implicit function. For example, given:

$f\ (x{:}\mathsf{N}, y{:}\mathsf{N})\ r{:}\mathsf{N}$
pre $x \geq y$
post $r = x{-}y$

then the following definitions will be created in translation:

$$\textit{pre-f} \triangleq ([\![1]\!] \geq [\![2]\!])$$
$$\textit{post-f} \triangleq [\![3]\!] = [\![1]\!]{-}[\![2]\!]$$

The symbols *pre-f* and *post-f* can then be used in the definition axiom for the function. We must guard against ill-formed pre- and post-conditions; though some of these checks could be performed by the VST, we prefer to phrase these checks as proof obligations in the translated theory. Given arguments of the correct types, the pre-condition must be a Boolean expression:

$$\boxed{\text{wf-pre-f}}(x, y)\frac{x{:}\mathsf{N}, y{:}\mathsf{N}}{\textit{pre-f}\,(x, y){:}\mathsf{B}}$$

It may be argued that this is overstrict, in that it forces pre-conditions to be total; however, for approaches such as animation, it may be useful to determine when a function or operation *cannot* be applied or invoked, in addition to determining when it can.

Given arguments of the correct types such that the pre-condition holds, the post-condition must also be a Boolean expression:

$$\boxed{\text{wf-post-f}}(x, y, r)\frac{x{:}\mathsf{N}, y{:}\mathsf{N}, r{:}\mathsf{N}, \textit{pre-f}\,(a, b)}{\textit{post-f}\,(a, b, r){:}\mathsf{B}}$$

4.4.2 Translation of explicit function definitions

An explicit function definition such as:

$f : \mathsf{N} \times \mathsf{N} \to \mathsf{N}$
$f(x, y) \quad \triangleq \quad x{-}y$

could be translated into a Mural definition $f \mapsto [\![e1]\!]{-}[\![e2]\!]$ where instances of x and y in the function body have been replaced in the Mural definition by $[\![e1]\!]$ and $[\![e2]\!]$ respectively.

(Note that this function is ill-defined when $x < y$; this will be reflected in the inability to complete the proof of the well-formedness obligation given later.)

However if the body of a function does not refer to some argument, then such a simple translation would not work, as the Mural definition would be ill-formed. (The expansion of any Mural definition must mention all of its arguments; this allows us to treat definition folding and unfolding as proper inverses. It also prevents us from replacing a non-denoting

term by a denoting term.) The only well-formed definition we can make will have an arity which is less than that of the original function; this would greatly complicate the translation process. Therefore we have chosen to translate explicit functions into symbol declarations and axioms rather than direct definitions. This has the further advantage of making the translation of explicit functions similar to that for implicit functions.

For an explicitly-defined function:

$f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

$f(x, y) \triangleq x-y$

pre $x \geq y$

we want to be able to substitute occurrences in expressions of applications of the function by the body of the definition (with suitable substitutions). However, we should only be allowed to do this when the function application is well-formed, that is, when the arguments are of the correct types and satisfy the pre-condition. Thus, we want a rule of the form

$$\boxed{\text{"f defn"}}(x, y)\frac{x:\mathbb{N}, y:\mathbb{N}, pre\text{-}f(x, y)}{f(x, y) = x-y}$$

Note that instances of x and y in the body of the function are translated to meta-variables.

However, this is not strict enough: we must also ensure that the function definition is itself well-formed. When the arguments are of the correct type and satisfy the pre-condition, then the body should have the same type as the result type claimed for the definition. In theory, the VDM support tool could perform some of the well-formedness checks on explicit function definitions, but for some cases — those where the result type of the function has an associated invariant — this cannot be done mechanically, and the well-formedness check must be presented as a rule to the proof assistant. Such a rule might appear as:

$$\boxed{\text{f well-formed}}(x, y)\frac{x:\mathbb{N}, y:\mathbb{N}, pre\text{-}f(x, y)}{x-y:\mathbb{N}}$$

In preference to giving this as a separate (unproven) rule, we add its conclusion as an extra hypothesis to the final definition axiom, to make the dependency clear:

$$\boxed{\text{f defn}}(x, y)\frac{x:A, y:B, pre\text{-}f(x, y), x-y:\mathbb{N}}{f(x, y) = x-y}$$

The "working version" of this axiom can then be derived by proving the general well-formedness obligation rule and discharging the new hypothesis.

4.4.3 Translation of implicit function definitions

Given an implicit function definition:

$f\ (x:\mathbb{N}, y:\mathbb{N})\ r:\mathbb{N}$

pre $x \geq y$

post $r = x-y$

we need rules for reasoning about expressions which contain subexpressions of the form $f(e_1, e_2)$.

If f is implicitly defined as above, then we know that the expression $f(x, y)$ is of type $\mathbb{N}$ when x, y are of type $\mathbb{N}$ and $pre\text{-}f$ holds for them; i.e., we need a rule such as:

$$\boxed{\text{"f-formn"}}(x, y)\frac{x{:}\mathbf{N}, y{:}\mathbf{N}, pre\text{-}f(x, y)}{f(x, y){:}\mathbf{N}}$$

As f is implicitly defined, we cannot, in general, give a precise value z such that $f(x, y) = z$; the most we can say is that $f(x, y)$ satisfies its post-condition (given the same conditions as above):

$$\boxed{\text{"f-prop"}}(x, y)\frac{x{:}\mathbf{N}, y{:}\mathbf{N}, pre\text{-}f(x, y)}{post\text{-}f(x, y, f(x, y))}$$

Before we can use these rules, we should discharge the implementibility proof obligation for f (otherwise, there is no guarantee that the expression $f(x, y)$ denotes a value). The implementibility obligation can be described as the (initially unproven) rule:

$$\boxed{\text{f implementible}}(x, y)\frac{x{:}\mathbf{N}, y{:}\mathbf{N}, pre\text{-}f(x, y)}{\exists r{:}\mathbf{N} \cdot post\text{-}f(x, y, r)}$$

In preference to generating the first two rules in translation, we create two axioms which incorporate "one-point" versions of the implementibility obligation:

$$\boxed{\text{f-formn}}(x, y)\frac{x{:}\mathbf{N}, y{:}\mathbf{N}, pre\text{-}f(x, y), \exists r{:}\mathbf{N} \cdot post\text{-}f(x, y, r)}{f(x, y){:}\mathbf{N}}$$

$$\boxed{\text{f-prop}}(x, y)\frac{x{:}\mathbf{N}, y{:}\mathbf{N}, pre\text{-}f(x, y), \exists r{:}\mathbf{N} \cdot post\text{-}f(x, y, r)}{post\text{-}f(x, y, f(x, y))}$$

Thus, before we can derive any properties of an expression which applies f to a particular x and y, we must show that f is implementible for those particular arguments. Note that if we prove the more general implementibility obligation, we can then use it to discharge the one-point versions in the axioms and thus derive the "working versions" first given.

4.5 Translation of operation definitions

As operation decomposition is not supported by the VST, only implicit operation definitions are translated. The pre- and post-conditions are translated in a similar manner as for function definitions. Their translation is slightly more complicated in that the pre-condition can refer to the "initial state" of an operation, and the post-condition can refer to the initial and final states, in addition to the arguments and result.

The implementibility obligation for an operation

$OP\ (x{:}X)\ r{:}R$
ext rd rd : Rd
wr wr : Wr
pre ... expression in x, rd and wr ...
post ... expression in x, r, rd, $\overleftarrow{wr}$ and wr ...

is translated as an unproven rule[4]:

$$\boxed{\text{OP implementibility}}(x, rd, \overleftarrow{wr})\frac{x{:}X, rd{:}Rd, \overleftarrow{wr}{:}Wr, pre\text{-}OP(x, r, rd, \overleftarrow{wr})}{\exists r{:}R \cdot \exists wr{:}Wr \cdot post\text{-}OP(x, r, rd, \overleftarrow{wr}, wr)}$$

[4]If operation decomposition were supported, then the conclusion could appear as a hypothesis to an axiom concerning calls of the operation within an "explicit" operation, by analogy with the definition axiom for an implicit function definition.

4.6 Theories from Reifications

In order to reason about a reification of one specification by another, we need the information contained in the two specifications. Therefore, the theory within which we reason about the reification should inherit from both their theories. This is achieved by making the theories of the two specifications parents of the reification theory.

Roughly, the theory of a reification will contain definitions of the retrieve functions used in the data reifications, together with all the concomitant proof obligations of data reification and operations modelling. It will also contain definitions of auxiliary types and functions used.

4.6.1 Translation of data reifications

Within a reification, a particular data reification from a concrete type C to an abstract type A via a retrieve function *retr-A* translates to:

- a definition or declaration of the retrieve function (in the same manner as for explicit function definitions), as a function from C to A:

  $$retr\text{-}A \mapsto [1, 0];$$

- and an unproven rule expressing the adequacy obligation for the data reification:

$$\boxed{\text{retr-A adequacy}}(c)\frac{c : C}{\exists a : A \cdot retr\text{-}A(c) = a}$$

Both of these are situated in the theory of the overall reification.

4.6.2 Translation of operation models

Recall that an operation modelling refers to the abstract and concrete operations and a number of *DataReifs*. Translation of the *OpModel* amounts to the translation of each of these along with the creation of two unproven rules in the reification theory: one expressing the domain obligation, and another expressing the result obligation.

Suppose an abstract operation AOP is modelled by a concrete operation COP:

$AOP\ (aa : AA)\ ar : AR$	$COP\ (ca : CA)\ cr : CR$
ext rd ard : ARD	ext rd crd : CRD
wr awr : AWR	wr cwr : CWR
pre ...	pre ...
post ...	post ...

and that *retr-a*, *retr-r*, ..., *retr-wr* are the retrieve functions associated with each component. Then the domain proof obligations generated is of the form:

$$\boxed{COP \text{ domain obln}}(ca, crd, cwr)\frac{ca : CA, crd : CRD, cwr : CWR,\ pre\text{-}AOP(retr\text{-}a(ca), retr\text{-}rd(crd), retr\text{-}wr(cwr))}{pre\text{-}COP(ca, crd, cwr)}$$

that is, the concrete pre-condition should hold whenever the abstract pre-condition holds for the retrieved values. The generated result proof obligation has the form:

$$\boxed{OP_C \text{ result obligation}}(ca, \ldots, cwr)\frac{\begin{array}{c} a_c{:}\,A_C, r_c{:}\,R_C, rd_c{:}\,RD_C, \overleftarrow{wr_c}{:}\,WR_C, wr_c{:}\,WR_C, \\ pre\text{-}OP_A(retr\text{-}a(a_c), retr\text{-}rd(rd_c), retr\text{-}wr(\overleftarrow{wr_c})), \\ post\text{-}OP_C(a_c, r_c, rd_c, \overleftarrow{wr_c}, wr_c) \end{array}}{\begin{array}{c} post\text{-}OP_A(retr\text{-}a(a_c), retr\text{-}r(r_c), retr\text{-}rd(rd_c), \\ retr\text{-}wr(\overleftarrow{wr_c}), retr\text{-}wr(wr_c)) \end{array}}$$

that is: the abstract post-condition should hold on the retrieved values whenever the abstract pre-condition and concrete post-condition hold. The generalisation to multiple arguments and state references is straightforward.

4.7 An example

An important design decision in the development of Mural was to preserve as far as possible the freedom of the user to construct objects (such as theories, proofs, specifications and refinements) in any order. However (partly as a consequence of this freedom of construction order) care must be taken when we come to translate specifications and reifications into the Mural theory store. For example, an attempt to translate (say) an operation which refers to a function or type which has no definition in the specification will fail; a reification cannot be translated before its abstract and concrete specifications; and before an operation model can be translated, all of its component-pairs must be tied to data reifications. Furthermore, the VDM support tool's translation mechanism does not track dependencies, so the user must ensure that components of specifications and reifications are translated in the correct order.

With these provisos in mind (and with the addition of "identity retrievals" on $NAME$ and $ADDR$ in our example operation model), we can translate our mini-development into the Mural store. This creates a Mural theory for each specification, extending the basic VDM theory, containing translations of the type, function and operation definitions (including well-formedness and implementibility proof obligations). Translating the reification creates a theory which inherits from the theories of the two specifications and contains definitions of the retrieve functions, the corresponding adequacy proof obligations, and the domain and result obligations of the operation model. Figure 6 shows a Mural theory browser on the theory generated from our reification. Its parent theories are the theories generated from the two specifications (which in turn have the basic "VDM" theory as a common parent). Thus the reification theory inherits all the declarations and definitions in the two specification theories (e.g. definitions of the pre- and post-conditions of $LOOKUP$ and $LOOKUP2$). The reification theory contains declarations and definitions of the retrieve functions used in our operation model, together with their adequacy obligations. The noticeboard contains notices showing the statements of the domain and result obligations for our operation model, together with the statement of the adequacy obligation for $retr\text{-}AddrBook$.

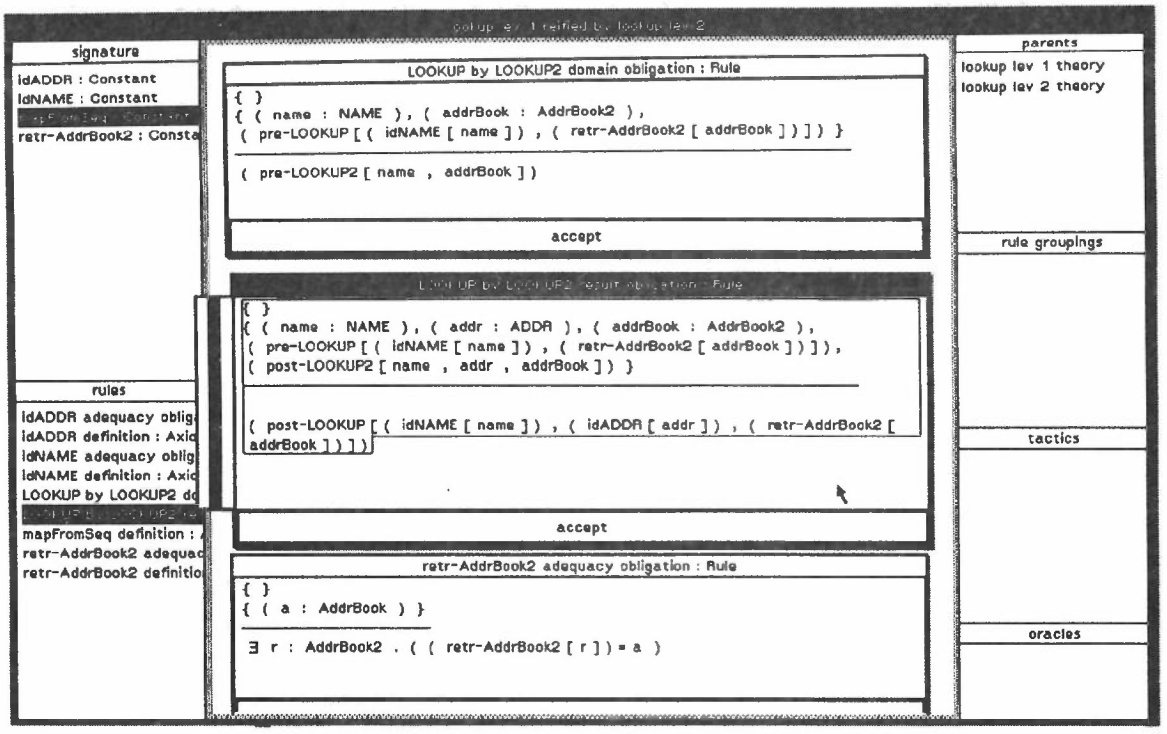

Figure 6: **A theory browser on the *LOOKUP* reification theory**

5 Conclusions

As stated earlier, the VDM support tool component of Mural is not a fully-fledged support environment for VDM. There are many ways in which it could be improved or extended, for example:

- Only a small subset of VDM is catered for.
- Many well-formedness checks could be carried out prior to the translation phase (including, for example, "patent" type-checking, detection of references to undeclared types/functions).
- The theory structure should be finer-grained, allowing re-use of common type definitions or even common data reifications.
- The support tool does not "keep track" of proof obligations etc, so users cannot easily determine which obligations are still outstanding for a particular specification or reification (or component thereof). (This would be easy to remedy.)
- The reification model chosen greatly limits the kinds of operations that can be modelled.
- There is no support for operation decomposition; much work needs to be done to determine a practical interface model for this.

However, the VDM support tool has satisfied our original intentions of providing "interesting" proof obligations to exercise the Mural proof assistant, and of demonstrating that a generic formal reasoning environment can be extended and instantiated to provide support for a particular development method.

References

[1] Lindsay, P. *A formal system with inclusion polymorphism.* IPSE 2.5 working document 060/pal014/2.3, 1987.

[2] BSI IST/5/50. *VDM Specification Language Proto-Standard.* Working paper IST/5/50/170, 1990.

[3] Jones, C. *Systematic Software Development Using VDM.* Prentice-Hall, 1990 (second edition).

[4] Jones, C. *Program Specification and Verification in VDM.* Technical report UMCS-86-10-5, Department of Computer Science, Manchester University, 1986.

[5] Jones, C. VDM Proof Obligations and Their Justification. In: *VDM '87: VDM — A Formal Method at Work.* LNCS 252, Springer-Verlag, 1987.

[6] Monahan, B. A Type Model for VDM In: *VDM '87: VDM — A Formal Method at Work.* LNCS 252, Springer-Verlag, 1987.

[7] Arentoft, M.M. and Larsen, P.G. *The Dynamic Semantics of the BSI/VDM Specification Language.* M.Sc.E.-thesis, Department of Computer Science, Technical University of Denmark, 1988.

[8] Milne, R. Proof Rules for VDM Statements. In *VDM '88: VDM — The Way Ahead.* LNCS 328, Springer–Verlag 1988.

[9] Ah-Kee, J.A. *Operation Decomposition Proof Obligations for Blocks and Procedures.* Ph.D. thesis, Department of Computer Science, Manchester University, 1989.

[10] Crispin, R.J. Experience Using VDM in STC. In: *VDM '87: VDM — A Formal Method at Work.* LNCS 252, Springer-Verlag, 1987.

[11] Spivey, J.M. *The Z notation.* Prentice–Hall, 1989.

[12] Spivey, J.M. *Understanding* Z. Cambridge University Press, 1988.

[13] Abrial, J-R. The B Tool (Abstract). In *VDM '88: VDM — The Way Ahead.* LNCS 328, Springer–Verlag 1988.

[14] Nielson, M, Klaus, H, Wagner, K.R and George, C. The RAISE Language, Method and Tools. *Formal Aspects of Computing* Vol.1 No.1 pp 85-114, 1989.