ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/2510095

# On the Verification of VDM Specification and Refinement with PVS

**Article** · January 1998

Source: CiteSeer

CITATION
1

READS
76

**2 authors:**

Savi Maharaj
University of Stirling
**40** PUBLICATIONS **657** CITATIONS
SEE PROFILE

Juan Carlos Bicarregui
Science and Technology Facilities Council
**137** PUBLICATIONS **1,844** CITATIONS
SEE PROFILE

**Some of the authors of this publication are also working on these related projects:**

Social distancing View project

Towards an Interactive Simulation Framework for effective E-learning in University Classroom environments View project

All content following this page was uploaded by Savi Maharaj on 08 September 2014.

The user has requested enhancement of the downloaded file.

# Chapter 6

# On the Verification of VDM Specification and Refinement with PVS

**Sten Agerholm, Juan Bicarregui and Savi Maharaj**

## Summary

This chapter describes using the PVS system as a tool to support VDM-SL. It is possible to translate from VDM-SL into the PVS specification language in a very easy and direct manner, thus enabling the use of PVS for typechecking and verifying properties of VDM-SL specifications and refinements. The translation is described in detail and illustrated with examples. The drawbacks of the translation are that it must be done manually (though automation may be possible), and that the "shallow embedding" technique which is used does not accurately capture the proof rules of VDM-SL. The benefits come from the facts that the portion of VDM-SL which can be represented is substantial and that it is a great advantage to be able to use the powerful PVS proof-checker. A variety of examples of verifications using PVS are described in the chapter.

## 6.1 Introduction

The PVS system[13], developed at SRI Menlo Park, in California, combines an expressive specification language with a powerful theorem-proving tool in which interactive proof is tightly integrated with many automatic procedures which speed up routine parts of proof development. The combination makes for a user-friendly system which is easy to learn and to use for doing non-trivial theorem-proving. The specification language is based on classical higher-order logic, enriched with a type system based on Church's simple theory of types with extensions for subtyping and dependent types. The result is an expressive language which has much in common with formal notations such as VDM-SL. A system of parameterized theories provides structuring for developments and extensive libraries add definitions of many useful mathematical constructs similar to those used in VDM-SL.

The striking, if superficial, similarity between VDM-SL and the PVS specification language means that a significant portion of VDM-SL may be represented very directly in PVS. This gives rise to an informal "shallow embedding" [10] of a substantial portion of VDM-SL. The embedding makes use of the parameterized theories of PVS to represent VDM-SL specifications and refinement relationships between pairs of specifications. The result is a simple and direct means of harnessing the power of the PVS proof-checker for proving validation conditions and proof obligations arising from specifications and refinements. The embedding also provides automatic generation of some proof obligations which can be made to arise as *type-checking constraints* (TCCs) in PVS. The drawbacks of the method are that translation must (at present) be performed manually, and that the proof rules of VDM-SL are not accurately captured because of differences between the logics of VDM-SL and PVS. The latter fact implies that what we are doing would be more accurately described as VDM-*style* specification.

Using this approach outlined above, we have hand-translated a number of VDM-SL specifications and refinements into the PVS logic, and used the prover to check various properties of these specifications. This process resulted in the detection of several errors in some of these specifications. In this chapter we describe one medium-sized example which illustrates most of the aspects of our approach. The structure of the chapter is as follows. In Section 6.2 we briefly describe the PVS system. This is followed in Section 6.3 by an informal presentation of the translation from VDM-SL to the PVS specification language. Section 6.4 complements Section 6.3 by illustrating through the use of an example how a specification as a whole is represented as a PVS theory. In this section we also discuss using the PVS proof-checker to typecheck the example specification and to prove various validation conditions about it. In Section 6.5 we turn to the subject of refinement, showing how the example of Section 6.4 can be represented in PVS as a refinement of a more abstract specification, and how the PVS proof-checker was used to discharge all the resulting proof obligations. In the final sections we discuss the validity of our informal translation from VDM-SL to PVS, make some observations about what was learned during this exercise, and present our conclusions.

## 6.2 The PVS System

The PVS system is produced by a team of researchers at SRI International, in California. The philosophy behind the system is that it should provide a high degree of support for requirements specification, since errors at this initial stage are considered to be the most costly. However, the system itself is general purpose and lends itself to a variety of applications [13], including verifications of algorithms and hardware systems and embeddings of other logics.

The foundation of the system is the expressive PVS Specification Language. This is based on classical higher-order logic, enhanced with a rich system of types including subtyping, dependent types and recursive types. Specifications may be structured using a system of parameterized theories. Various aspects of the syntax of the PVS specification language will be explained where they arise in the chapter.

The PVS specification language is supported by a collection of tools including a type checker and an interactive proof checker. The tools run under emacs which provides a basic user interface. Typechecking in PVS is undecidable and generally results in the generation of proof obligations called *type-checking constraints* (TCCs). TCCs may be proved by invoking a powerful tactic called `tcp` or, if this tactic fails, by use of the PVS proof-checker.

One of the main attractions of the PVS proof-checker is that basic proof commands are tightly integrated with powerful decision procedures and tactics. The tactics may be combined in various ways to form proof strategies. Some of the most commonly used tactics embody simple proof rules (splitting conjunctions and disjunctions, instantiating quantifiers, skolemizing, etc). Others implement sophisticated decision procedures. Examples are the tactic `PROP` for propositional simplification and the all-powerful tactic `GRIND`, which subsumes `PROP` and does much more. Other tactics such as `HIDE` allow management of the proof state.

The version of PVS referred to in this chapter is the "alpha-plus" release available in June 1996.

## 6.3 From VDM-SL to the Higher Order Logic of PVS

A large subset of VDM-SL can be translated into a higher order logic such as the PVS specification language. This section describes such a translation in sufficient detail to support manual translations, though more formality might be required in order to support an implementation of the translation process. The translation method may be viewed as a (very) shallow embedding of VDM-SL in PVS. The syntax of VDM-SL constructs is not embedded: we work with the "semantics" of the constructs directly. Therefore it can be argued that the translation is not safe (in a logical sense). This is a disadvantage of shallow embeddings in general, though some shallow embeddings are more safe than others. The translation of VDM-SL to PVS is relatively safe in this sense since PVS and VDM-SL share many concepts and constructs.

One of the differences between VDM-SL and PVS is that a VDM-SL expression can be undefined; there is one common undefined element to all types, which can be viewed as sets. For example, an expression is undefined if a divisor is zero, if a finite sequence is applied to an index outside its indices, or if a pattern match in a let expression fails. Most of such situations are handled implicitly by restrictions on the translation of various constructs, in combination with type checking in PVS. For instance, PVS itself generates type checking conditions to capture division by zero. A related factor is that PVS does not support partial functions. All VDM-SL functions are translated to PVS functions and PVS generates an obligation stating that a given function must be total. Partial functions are discussed in more depth in Section 6.6.

Below we discuss how various constructs of VDM-SL specifications can be translated. We show VDM-SL expressions in the ASCII notation supported by the IFAD VDM-SL Toolbox.

### 6.3.1 Basic Types, the Product Type and Type Invariants

Boolean and number types can be viewed as sets and translated directly to the corresponding types in PVS. It is not necessary to edit arithmetic expressions and logical connectives since they have the same symbols and names in PVS. Specially, the connectives `and` and `or` are used as "and-also" and "or-else" in both the Toolbox version of VDM-SL and PVS. Hence, it is allowed to write "`P(x) and x/x = 1`" if one can prove the automatically generated condition "`P(x) IMPLIES x /= 0`".

Restricted quantifications are translated to quantifications over PVS subtypes. Hence, a universal quantification like `forall x in set s & P[x]` can be translated to `forall (x:(s)): P[x]`. Note that the `&` is replaced with `:` and we insert brackets around the binding.

In VDM-SL it is possible to specify a subtype of a type by writing a predicate that the elements of the subtype must always satisfy. This predicate is called an invariant. As an example consider the definition of a type of positive reals:

```
Realp = real
inv r == r >= 0
```

This introduces a new constant `inv_Realp` implicitly, which equals the invariant predicate. The definition is translated to PVS as follows:

```
inv_Realp(r:real) : bool = r >= 0
Realp: TYPE = (inv_Realp)
```

The type `Realp` is defined as a subtype of the built-in type `real`.

The product type of VDM-SL can be translated directly to the product type of PVS. VDM-SL tuples are written using a constant `mk_`, e.g. `mk_(1,2,3)`. This constant could be ignored in the translation, but instead we have introduced and usually use a dummy constant `mk_`, defined as the identity function. The only way to split a tuple in VDM-SL is using pattern matching on `mk_`, e.g. in a let expression like `let mk_(x,y,z) = t in e`. The translation of pattern matching is discussed in more detail in Section 6.3.6.

### 6.3.2 Record Types

VDM-SL records are translated directly to records in PVS. There is only a slight difference in syntax. PVS does not automatically introduce a constructor function `mk_A` as in VDM-SL for a record definition called `A`, but the function is easy to define. Below, we first give an example of how to translate a standard record type definition and then give a more complicated example where an invariant is also specified.

#### Standard Records

We consider a simple example, which defines a record of points in the two-dimensional positive real plane. The VDM-SL definition of the record type is (`Realp` was introduced above):

```
Point:: x: Realp
        y: Realp
```

This is translated to PVS as follows:

```
Point: TYPE = [# x: Realp, y: Realp #]
```

In both cases, field selectors `x` and `y` are introduced implicitly. Unfortunately, PVS does not support the VDM-SL notation for field selection, which is the standard one using a dot and the field name, e.g. `p.x` for some point `p`. Instead fields are viewed as functions that can be applied to records. Thus the VDM-SL expression `p.x` translates to the PVS expression `x(p)`. The PVS notation is less convenient, e.g. a nested field selection as in `r.a.b.c` translates to `c(b(a(r)))`.

In VDM-SL each new record definition generates a new "make" function for building elements of the record. In order to allow a direct translation of such expressions to the same expressions in PVS, the constructor function is defined along with a record definition. The constructor for points is:

```
mk_Point(a:Realp,b:Realp) : Point = (# x:= a, y:= b #)
```

In VDM-SL such make constructors are also used for pattern matching in function definitions and in let expressions but this is not possible in PVS. Instead we use Hilbert's choice operator (`choose`), as described in Section 6.3.6.

VDM-SL also provides an `is_` test function for records, which is sometimes used to test where elements of union types come from. Since we do not support union types properly (see Section 6.3.4) we shall ignore this constant.

#### Records with Invariants

Sometimes a VDM-SL record definition is written with an invariant. For instance, an equivalent way of defining the point record above would be the following definition

```
Point:: x: real
        y: real
inv mk_Point(x,y) == x >= 0 and y >= 0
```

where an invariant is used to specify that we are only interested in the positive part of the two-dimensional real plane. We translate this as follows:

```
inv_Point(x:real,y:real) : bool = x >= 0 and y >= 0

Point: TYPE = {p: [# x:real, y:real #] | inv_Point(x(p),y(p))}

mk_Point(z:(inv_Point)) : Point = (# x:= x(z), y:= y(z) #)
```

Note that we restrict the arguments of `mk_Point`. For instance, the following definition does not work

```
mk_Point(a:real,b:real) : Point = (# x:= a, y:= b #)
```

since we cannot prove that this yields a point for all valid arguments.

There is one small semantic difference between VDM-SL records and the above representation in PVS. In VDM-SL it is allowed to write `mk_Point(1,-1)` though this will not have type `Point`. The only use of such a (kind of) junk term would be in `inv_Point(mk_Point(1,-1))`, which would equal false. In the PVS representation, this invariant expression is written as `inv_Point(1,-1)`.

## 6.3.3 Sequences, Sets and Maps

In this section, we briefly consider the translation of finite sequences, finite sets and finite maps. Invariants on these types are treated much like in the previous section on records.

### Finite Sets

A type of finite sets is provided in the PVS `finite_sets` library. The type is defined as a subtype of the set type, which represents sets as predicates. Most operations on sets exist, or can be defined easily. One annoying factor is that for instance set membership, set union and set intersection are all prefix operations in PVS. E.g. one must write `member(x,s)` for the VDM-SL expression "`x in set s`". Moreover, user-defined constants must be prefix and one cannot define new symbols. PVS supports only a simple and restricted syntax of expressions.

### Finite Sequences

VDM-SL finite sequences can be represented as finite sequences or as finite lists in PVS. The difference is that finite sequences are represented as functions and finite lists as an abstract datatype. There is more support for finite lists, so we have usually chosen this type as the representation. An advantage of sequences is that indexing is just function application. However, with both representations one must be careful since indexing of sequences starts from one in VDM-SL and from zero in PVS. The safest thing to do is therefore to define new indexing operations in PVS and use these for the translation.

#### Finite Maps

PVS does not support finite maps, so an appropriate theory must be derived from scratch. In doing this, one could probably benefit from the paper on finite maps in HOL by Collins and Syme [12], who have implemented their work in a HOL library. However, many operations on finite maps are not supported in this library, so an extended theory of finite maps must be worked out.

As a start one could just axiomatize maps in PVS, e.g. by introducing maps as an uninterpreted subtype of the function type, with a few appropriate definitions (and axioms). In fact, for the examples very little support was needed.

A representation of maps using functions has advantages. Map application will just be function application and map modification can be translated to PVS `with` expressions. For example, the VDM-SL map modification `m ++ { 1 |-> 2, 2 |-> 3 }`, where a map `m` from numbers to numbers is modified to send 1 to 2 and 2 to 3, translates to `m with [ 1 |-> 2, 2 |-> 3 ]`.

### 6.3.4 Union Types

In VDM-SL, the union of two or more types corresponds to the set union of the types. Thus, the union type is a non-disjoint union, if two types have a common element this will be just one element in the union type. Higher order logics do not support non-disjoint unions, but support disjoint sums (unions) or abstract datatypes as in PVS. In general, a VDM-SL union type cannot be translated easily to a PVS datatype. However, if the component types of the union are disjoint then this is partly possible. The translation is only satisfactory when the component types are quote types; these correspond to singleton sets.

#### Union of Disjoint Types

The union of disjoint types can be represented as a new datatype with constructor names for the different types. This representation is not perfect, the component types does not become subtypes of the union type as in VDM-SL. For example this means that the operators defined on the individual types are not inherited as in VDM-SL, where the dynamic type checking ensures that arguments of operators have the right types. In the special case where all components of the union type are new types, it might be possible to define the union type first and then define each of the component type as subtypes of this. Such tricks would not be easy to employ in an automatic translation.

#### Enumerated Types

An enumerated type is a union of quote types, which is written using the following ASCII syntax in VDM-SL: `ABC = <A>|<B>|<C>`. This can be translated almost directly to PVS, with some minor syntax changes: `ABC: TYPE = {A,B,C}`. PVS does not support identifiers enclosed in `<` and `>`.

### 6.3.5 Function Definitions

Total functions are translated directly to PVS functions. As mentioned before, partial functions cannot be translated in this way. As we shall see in Section 6.6, it is possible to encode some partial functions as total functions in PVS by using subtypes to represent their domains of definition. Other formalizations are also possible (see e.g. [2, 1]). Polymorphic functions are not considered at the moment.

Standard explicit function definitions, which are function definitions that do not have postconditions, can be translated directly to PVS, if they are not recursive. A precondition will be translated to a subtype predicate. If functions are recursive we must demonstrate that they are total functions in PVS. It is up to the translator to specify an appropriate measure, which is decreased in each recursive call, for the termination proof. Moreover, VDM-SL supports mutual recursive function definitions which would not be easy to translate. The example specifications used only few recursive definitions and these were very simple.

Implicit function definitions, which are specified using pre- and postconditions only and have no function body, can be represented using the choice operator. Almost equivalently, one can also use function specification, which is a way of defining partially specified functions; it is only specified on a subset of a type how a function behaves, and this is specified by an "underdetermined" relation, not an equation [18, 15].

#### Implicit Definition

Let us first consider the following semi-abstract example of an implicit function definition in VDM-SL:

```
f(x:real,y:real) z:Point
pre p[x,y]
post q[x,y,z]
```

where the variables in square brackets may occur free in the precondition `p` and the postcondition `q`. This translates to the following PVS definitions:

```
pre_f(x:real,y:real) : bool = p[x,y]
```

```
post_f(t:(pre_f))(z:Point) : bool = let (x,y) = t in q[x,y,z]

f: FUNCTION[t:(pre_f) -> (post_f(t))]
```

The precondition is translated to a predicate on the arguments of the function and the postcondition is translated to a binary relation on the arguments and the result. The function itself is defined as an uninterpreted constant using a dependent function type: given arguments `t` satisfying the precondition it returns a result satisfying the postcondition, or more precisely, a result related to `t` by the postcondition. This relation may be underdetermined, i.e. it may specify a range of possible values for a given input, but the function will always return a fixed value in this range. If the precondition is not satisfied the result is an arbitrary value.

As a result of an uninterpreted constant definition, the PVS type checker generates an existence condition, which says that we must prove there exists a value in the specified type. Hence, above we must prove there exists a function from the precondition to the postcondition. In general, proving this condition can be non-trivial, since one must usually provide a witness, i.e. a function of the specified form. (For instance, it would be difficult to prove that there exists a square root function.)

### Explicit Definition

Explicit definitions of recursive functions can be problematic for automatic translation since a translator must insert a well-founded measure for proofs of termination. This is easy enough when the recursion is simple, which it is for primitive recursive functions over numbers and abstract datatypes, but for more general recursive functions this can be hard. PVS has some strategies for proving termination in simple cases.

An explicit function definition has no postcondition but instead a direct definition, and perhaps a precondition. Let us consider a standard example of a primitive recursive function on the natural numbers:

```
fac: nat -> nat
fac(n) == if n = 0 then 1 else n * fac(n-1)
pre 0<=n
```

This translates to the following PVS definitions:

```
pre_fac(n:nat) : bool = n >= 0

fac(n:(pre_fac)) : recursive nat =
  if n = 0 then 1 else n * fac(n-1) endif
  measure (lambda (n:(pre_fac)): n)
```

Note that we have inserted "`recursive`" and "`measure`", which are part of the syntax for recursive function definitions in PVS. The measure is used to generate conditions for termination of recursive calls. (The precondition is redundant above, it is included for illustration.)

### 6.3.6 Pattern Matching

Pattern matching plays an important role in VDM-SL specifications. It is used frequently to get access to the values at fields of a record, and it is the only way to get access to the values of the components of a tuple. We can represent a successful pattern matching but not a failing one, since we do not represent undefined expressions. However, undefined expressions are either avoided due to type checking, or else represented by arbitrary values, i.e. values of a certain type that we do not know anything about.

#### Pattern Matching in Let Expressions

Here are some examples which use a record type `A` with three fields `a`, `b` and `c`. Assuming `x`, `y` and `z` are variables, the following VDM-SL let expression

```
let mk_(x,y,z) = e1 in e2[x,y,z]
```

can be translated to exactly the same term in PVS, except that the tuple constructor `mk_` must be omitted for the expression to parse. The following VDM-SL let expression with a pattern match on the record type `A`

```
let mk_A(x,y,z) = e1 in e2[x,y,z]
```

can be translated to the following PVS term:

```
let x = a(e1), y = b(e1), z = c(e1) in e2[x,y,z]
```

The field selector functions are used to destruct the expression. This corresponds to the way that PVS itself represents pattern matching on tuples (using project functions). If one of the variables in the VDM-SL expression was the don't care pattern, written as an minus sign `-`, then we could just replace this with a new variable. We do not allow constants in patterns in let expressions, since they do not make much sense (they are however allowed in VDM-SL).

The following VDM-SL "let-be-such-that" expression

```
let mk_A(x,y,z) in set s be st b[x,y,z] in e[x,y,z]
```

can be translated to

```
let v = (choose ({w:(s) | let x = a(w), y = b(w), z = c(w)
                          in b[x,y,z]})
in
  let x = a(v), y = b(v), z = c(v) in e[x,y,z]
```

where we use the choice operator, `choose`, to represent the looseness in the VDM-SL specification. Don't care patterns are translated as suggested above, by introducing new variables. We allow constants and other values in let-be-st expressions. For instance, we can translate

```
let mk_A(x,y,0) in set s be st b[x,y] in e[x,y]
```

into the PVS term

```
let
  v = (choose ({w:(s) | let x = a(w), y = b(w), n = c(w)
                        in n = 0 and b[x,y]})
in
  let x = a(v), y = b(v) in e[x,y]
```

where we include a test in the body of the `choose`.

### Pattern Matching in Cases Expressions

The following VDM-SL cases expression

```
cases e:
  mk_A(0,-,z) -> e1,
  mk_A(x,1,z) -> e2,
  others      -> e3
end
```

can be translated to the following conditional expression in PVS:

```
cond
  a(e) = 0 -> let z = c(e) in e1,
  b(e) = 1 -> let x = a(e), z = c(e) in e2,
  else     -> e3
endcond
```

PVS's built-in cases expression only works on abstract datatypes.

### Pattern Matching in Function Definitions

Pattern matching can be used on arguments in a function definition, where the patterns are typically variables (or don't care patterns which are translated to new variables). We can treat this by inventing a new variable using the function definition and then extending the body with a let expression to represent the pattern match. This approach is also used in the formal semantics of VDM-SL.

### 6.3.7 State and Operations

A VDM-SL specification may contain a state definition, which specifies a number of variable names for elements of the state space. The state space is known to operations and nowhere else. The state definition is essentially a record definition and is therefore represented as a record type in PVS. Operations are represented as state transformations, i.e. functions which, in addition to the operation's input values, take the initial state as an argument and return the output state as a result (and possibly an explicit result value). Hence, operation definitions can be translated in a similar way as functions.

The body of operation definitions may contain assignments and sequential compositions. Assignments are translated to PVS `with` expressions and sequential compositions are represented using let expressions. In this chapter we do not consider conditions (which should be easy) and while loops (which probably could be translated to recursive functions). More exotic features such as exception handling are also excluded from consideration.

Assume we have the following state definition in VDM-SL:

```
state ST of
      x: real
      y: real
      z: real
end
```

This can be translated to:

```
 ST: TYPE = [# x: real, y: real, z: real #]
 mk_ST(x:real,y:real,z:real): ST = (# x:=x, y:=y, z:=z #)
```

Now assume we have the sequence:

```
x:=5; y:=3; z:=1
```

This can be translated to

```
lambda (s:ST):
  let s1 = s  with [x:=5],
      s2 = s1 with [y:=3],
      s3 = s2 with [z:=1]
  in s3
```

or simply to

```
lambda (s:st): s with [x:=5, y:=3, z:=1]
```

since the assignments are independent in this example.

## 6.4 A Specification Example: MSMIE

The Multiprocessor Shared-Memory Information Exchange (MSMIE) is a protocol for "inter-processor communications in distributed, microprocessor-based nuclear safety systems" [17], which has been used in the embedded software of Westinghouse nuclear systems designs.

The protocol uses multiple buffering to ensure that no "data-tearing" occurs as separate processors communicate via some shared memory. In other words, data should never be overwritten by one process while it is still being read by another. One important requirement is that neither writing nor reading processes should have to wait for a buffer to become available; another is that recent information should be passed, via the buffers, from writers to readers. The example has previously been analyzed using CCS in [11] and using VDM and B in [9]. In common with these analyses, we shall be working with a simplified system in which it is assumed that information is being passed from a single "slave" processor to several "master" processors. Thus, there are several reading processors, "masters", but only one writing, "slave" process.

The information exchange is realised by a system with three buffers. At any time, one buffer is available for writing, one for reading, and the third is either between a write and a read and hence contains the most recently written information, or between a read and a write and so is idle.

The status of each buffer is recorded by a flag which can take one of four values:

s - "assigned to slave" This buffer is reserved for writing. It may actually be being written at the moment or just marked as available for writing.

n - "newest" This buffer has just been written and contains the latest information. It is not being read at the moment.

m - "assigned to master" This buffer is being read by one or more processors.

i - "idle" This buffer is not being read or written and does not contain the latest data.

The names of the master processors that are currently reading are also stored in the state.

The VDM specification of [8] and [9] is concerned with various "data models" of MSMIE: the state of the device is modelled but not the slave and master processors nor the dynamic evolution of the system as they access the buffers in parallel. This analysis concerns only the operations which modify the buffer status flags. In the system as a whole, these operations are protected by a system of semaphores which allows each operation uninterrupted access to the state, and thus their behaviour is purely sequential.

There are three such operations:

*slave* This operation is executed when a write finishes. The buffer that was being written is given the status "newest" thereby replacing any other buffer with this status.

*acquire* This is executed when a read begins. The new reader name (passed as a parameter) is added to the set of readers and status flags are updated as appropriate.

*release* This is executed when a read ends. The given reader name is removed from the set of readers and status flags are updated as appropriate.

The precise description of these operations is left to the formal specification in the following section.

We note, in passing, that the MSMIE protocol has the the undesirable property that it is possible for information flow from slave to master to be held up indefinitely. This problem is dealt with in the original paper [17] by the use of timing constraints. The paper [11] suggests an improvement to the protocol in which the problem is avoided by the use of a fourth buffer. This improved protocol is quite intricate, and is modelled in [8] by using non-standard extensions to VDM which provide a more concise means of expression than standard VDM-SL. In this report we restrict ourselves to standard VDM-SL specifications and so do not treat the improved MSMIE protocol.

The paper [8] explores several ways of specifying the MSMIE system in VDM with varying degrees of abstraction. These specifications can be translated more or less "as is" into PVS, which can then be used to carry out proof obligations. The specifications can be written in such a way that some of the proof obligations are automatically generated by PVS, though it is still necessary to type in others by hand.

The VDM specifications of MSMIE which we show are in VDM-SL and therefore differ slightly from those in [8]. They were developed with the help of the IFAD VDM-SL Toolbox [14].

## 6.4.1 The VDM Specification

The specification uses an auxiliary function called *count*, which counts the number of occurrences of a given item in a sequence.

**functions**

```
$count[@T] : @T \times @T^* \to \mathbb{N}$

$count\ (s, ss) \triangleq$
  cases $ss$ :
    $[] \to 0$,
    others $\to$ if hd $ss = s$
                then $1 + count[@T]\ (s, \mathsf{tl}\ ss)$
                else $count[@T]\ (s, \mathsf{tl}\ ss)$
  end
```

The possible values of the status flags are given via an enumerated type; the type of the names of master processes is deferred.

types

```
$Status = \text{S} \mid \text{M} \mid \text{N} \mid \text{I};$

$MName =$ token
```

The state records the status of each of the three buffers, and the names of all currently reading master processes. These are represented by, respectively, a sequence of status values and a set of master names. The invariant captures constraints on the possible states that are reachable: there is exactly one buffer assigned to the writing slave process; at most one buffer is currently being read, and at most one holds newest data that is not being read; the set of readers is empty precisely when no buffer is being read. In the initial state, one buffer is assigned to the slave and the other two buffers are marked as idle.

```
state $\Sigma$ of
  $b : Status^*$
  $ms : MName$-set
  inv mk-$\Sigma\ (b, ms) \triangleq$
      len $b = 3 \land$
      $count[Status]\ (\text{S}, b) = 1 \land$
      $count[Status]\ (\text{M}, b) \in \{0, 1\} \land$
      $count[Status]\ (\text{N}, b) \in \{0, 1\} \land$
      $(count[Status]\ (\text{M}, b) = 0 \ \Leftrightarrow\ ms = \{\})$
  init $s \triangleq s =$ mk-$\Sigma\ ([\text{S}, \text{I}, \text{I}], \{\})$
end
```

The *slave* operation is executed when a slave process completes writing. The buffer that has just been written, previously of status S, is given the status N reflecting that it now contains the newest data. This buffer replaces any other buffer with the N status. The operation also selects one of the available buffers and assigns to it status S, making it the new buffer available for writing.

operations

```
slave ()
ext wr $b : (Status^*)$
pre  true
post $\forall\, i \in \{1,2,3\} \cdot$
          $(\overleftarrow{b}\,(i) = \mathrm{S} \;\Rightarrow\; b\,(i) = \mathrm{N}) \wedge$
          $(\overleftarrow{b}\,(i) = \mathrm{M} \;\Rightarrow\; b\,(i) = \mathrm{M})$ ;
```

The *acquire* operation is executed when a master process is about to begin reading. The new reader's name, passed as a parameter, is added to the set of active readers, and status flags are updated as necessary. If there is already a buffer being read, then the new reader also begins to read that buffer and no status changes are needed. Otherwise it reads the buffer with the newest data, status N, and the status of that buffer is changed to M.

```
acq ($l : MName$)
ext wr $b : (Status^*)$
    wr $ms : (MName\text{-set})$
pre  $(\neg\,(l \in ms)) \wedge$
     $(\exists\, i \in \{1,2,3\} \cdot b\,(i) = \mathrm{N} \vee b\,(i) = \mathrm{M})$
post $ms = \overleftarrow{ms} \cup \{l\} \wedge$
       $\forall\, i \in \{1,2,3\} \cdot$
          if $\overleftarrow{b}\,(i) = \mathrm{N} \wedge \overleftarrow{ms} = \{\}$
          then $b\,(i) = \mathrm{M}$
          else $b\,(i) = \overleftarrow{b}\,(i)$ ;
```

The release operation takes place when a master process has finished reading. The master's name is removed from the set of readers and buffer flags reassigned as appropriate. If there are still other masters reading then the status flags do not need to be changed. Otherwise, the buffer that has just been relinquished must have its status flag reassigned. There are two possibilities. If there is some other buffer which was written while the read was taking place, and therefore has status N, then the released buffer no longer contains the newest data and must have its status set to I. Otherwise, it still contains the freshest data and must have its status reset to N.

```
rel ($l : MName$)
ext wr $b : (Status^*)$
    wr $ms : (MName\text{-set})$
pre  $l \in ms$
```

$$
\begin{array}{l}
\textsf{post } ms = \overleftarrow{ms} \setminus \{l\} \wedge \\
\quad \forall\, i \in \{1,2,3\} \cdot \\
\qquad \textsf{if } \overleftarrow{b}\,(i) = \mathrm{M} \wedge ms = \{\} \\
\qquad \textsf{then } b\,(i) \in \{\mathrm{N},\mathrm{I}\} \wedge count[Status]\,(\mathrm{N}, b) = 1 \\
\qquad \textsf{else } b\,(i) = \overleftarrow{b}\,(i)
\end{array}
$$

### 6.4.2 PVS Translation

The VDM specification is represented as a PVS theory called msmie_sigma2. The theory is parameterized over a non-empty type of master names.

```
msmie_sigma2[MName : TYPE+] : THEORY

  BEGIN
```

The theory begins by importing another theory containing definitions of functions on lists. The "count" function is defined within this imported theory.

```
  IMPORTING list_funs
```

As in VDM, the possible values of the status flags are represented by an enumerated type.

```
  Status : TYPE = {Slave,Master,Newest,Idle}
```

The state and invariant of the VDM specification are represented by a single (dependent) type in PVS. The definition is best understood in two parts. First, the state is represented as a record containing two fields: a list, `b` of Status values, and a set of master names, `ms`. The set of valid states is then represented by forming the subtype, `sigma2`, of all such records which satisfy the invariant.

```
  sigma2 : TYPE =
   {x : [# b  : list[Status],
           ms : setof[MName] #] |

   ((length (b(x)) = 3) AND
    (count (Slave,b(x)) = 1) AND
    member(count(Master,b(x)),{x:nat|x=0 OR x=1}) AND
    member(count(Newest,b(x)),{x:nat|x=0 OR x=1}) AND
    ((count (Master,b(x)) = 0) <=> (ms(x) = emptyset)))}
```

The initial state is defined as a record containing appropriate values and its type is explicitly constrained to be `sigma2`. When this definition is typechecked, PVS will automatically generate as a type-checking constraint (TCC) the condition that the initial state satisfies the invariant. We shall look more closely at these TCCs in the next section.

```
  initial_sigma2 : sigma2 =
    (# b := (: Slave, Idle, Idle :), ms := emptyset #)
```

For each operation, the precondition is represented as a predicate over the valid states (`sigma2`). The postcondition is represented as a predicate over pairs of valid states. The operation is then defined as a function which maps each value from the subtype defined by the precondition to some unspecified valid state which, when paired with the input value, satisfies the postcondition. Typechecking such a definition will cause PVS to generate a TCC stating that a suitable value for the post state does indeed exist. In other words, we are asked to show that the specified operation is *feasible*.

Our first example is the `slave` operation. The definitions of the pre- and postconditions resemble very closely the original VDM specification. One difference is that there is no analogue in PVS to the frames in VDM, so that the fact that the variable `ms` has read-only status in the VDM specification must be explicitly stated in the PVS postcondition.

```
  pre_slave  : [sigma2 -> bool] = LAMBDA (st:sigma2) :
    true

  post_slave : [(pre_slave),sigma2 -> bool] =
    LAMBDA (st:(pre_slave),st2:sigma2) :
      ((FORALL (i:{x:nat| x=0 OR x=1 OR x=2}) :
         (nth(b(st),i) = Slave IMPLIES nth(b(st2),i) = Newest)
         AND
         (nth(b(st),i) = Master IMPLIES nth(b(st2),i) = Master))
       AND
       (ms(st2) = ms(st)))
```

The slave operation is then defined using the PVS choice operator `choose`. Our use of `choose` leads to a rather different interpretation of looseness from that adopted in VDM-SL. We defer discussion of this point until Section 6.6.3.

```
  slave : [(pre_slave) -> sigma2] =
          LAMBDA (st:(pre_slave)) :
             choose({st2:sigma2 | post_slave(st,st2)})
```

The specification of acquire is done similar. First the precondition and postcondition are defined as predicates.

```
  pre_acq : [MName ->[sigma2 -> bool]] =
    LAMBDA (l:MName)(st:sigma2) :
      (NOT member(l,ms(st))) AND
      (EXISTS (i:{x:nat| x=0 OR x=1 OR x=2}) :
          (nth(b(st),i) = Newest OR nth(b(st),i) = Master))

  post_acq : [l:MName -> [(pre_acq(l)),sigma2 -> bool]]  =
             LAMBDA (l:MName)(st:(pre_acq(l)),st2:sigma2) :
               (ms(st2) = union(ms(st),singleton(l))) AND
               (FORALL (i:{x:nat| x=0 OR x=1 OR x=2}) :
                   IF nth(b(st),i) = Newest AND ms(st) = emptyset
                   THEN nth(b(st2),i) = Master
                   ELSE nth(b(st2),i) = nth(b(st),i)
                   ENDIF)
```

Next, the acquire operation is defined as a function which, when given a master name `l` and a state belonging to the type `(pre_acq(l))`, returns a nondeterministically chosen state in sigma2 such that the two states together satisfy the postcondition `post_acq(l)`

```
  acq : [l:MName, (pre_acq(l)) -> sigma2] =
         LAMBDA (l:MName, st:(pre_acq(l))) :
              choose({st2:sigma2 | post_acq(l)(st,st2)})
```

The specification of the release operation is similar to that of acquire and is not described here.

### 6.4.3 Typechecking Constraints

When the theory `msmie_sigma2` is typechecked by PVS, a number of typechecking constraints (TCCs) are generated. These must be proved in order to demonstrate that the theory is well-typed. Simple TCCs can be handled by invoking an automatic TCC-prover, `tcp`, but more difficult ones must be proved interactively by the user. In the case of the theory `msmie_sigma2`, there are only 4 TCCs which are too difficult for `tcp` to prove. Interestingly, these TCCs correspond to the satisfiability proof obligations for the specification. We are required to show that the initial state satisfies the invariant, and that each of the three operations is feasible.

#### Showing that the Initial State Satisfies the Invariant

The first TCC generated for `msmie_sigma2` is shown below. It results from the fact that we have explicitly stated that the initial state, `initial_sigma2`, is of type `sigma2`; we are required to prove that `initial_sigma2` satisfies the predicate which defines the subtype `sigma2`. In other words, we must show that the initial state satisfies the invariant. The proof is too difficult for `tcp`, but can be done quickly using the interactive prover. The main goal, which consists of a conjunction of 5 formulae, is split, making each conjunct into a separate subgoal. Each of these is then proved by repeatedly expanding definitions until a statement is obtained which PVS recognises to be trivially true.

```
initial_sigma2_TCC1: OBLIGATION
  ((length[Status]((: Slave, Idle, Idle :)) = 3)
      AND (count[Status](Slave, (: Slave, Idle, Idle :)) = 1)
        AND
      member[nat](count[Status](Master, (: Slave, Idle, Idle :)),
                  {x: nat | x = 0 OR x = 1})
          AND
        member[nat](count[Status](Newest, (: Slave, Idle, Idle :)),
                    {x: nat | x = 0 OR x = 1})
            AND
          ((count[Status](Master, (: Slave, Idle, Idle :)) = 0)
               <=> (emptyset[MName] = emptyset[MName])));
```

### Showing that the Operations Are Feasible

To specify the operations we used the nondeterministic choice operator of PVS. For this to be correctly typed, PVS requires us to demonstrate that there exist possible candidates for the nondeterministically chosen values. In other words, we must show that each operation is feasible.

We show the statement of this proof obligation for the acquire operation. Given any master name `l`, and any state `st` within the type defined by the precondition, `(pre_acq(l))`, we must prove that the set of all states, `st2`, satisfying the postcondition `post_acq(l)(st,st2)` is nonempty.

```
acq_TCC1: OBLIGATION
      (FORALL (l: MName, st: (pre_acq(l))):
         nonempty?[sigma2]({st2: sigma2 | post_acq(l)(st, st2)}));
```

This TCC has been proved interactively using PVS. The proof is unsurprising but not trivial: the user must supply a suitable candidate for the nondeterministic choice and then verify that all the various conditions imposed by the postcondition and the invariant are satisfied.

We describe only the main highlights of the proof. After skolemizing, expanding definitions, and making some hidden hypotheses explicit, we are in a position where we may supply a possible candidate for the value that is nondeterministically chosen. This is done using the tactic `INST`, which is given two arguments: an integer representing the appropriate subgoal and a value for instantiation consisting of a record representing the state after the operation is carried out. In this case, the postcondition of the acquire operation happens to be very explicit so the calculation of the post state simply reflects the postcondition. Note that identifiers such as `st!1` are skolem variables generated by PVS.

```
(INST -8
  "(# ms := union(ms(st!1), singleton(l!1)),
      b := (: IF nth(b(st!1), 0) = Newest AND ms(st!1) = emptyset
                THEN Master ELSE nth(b(st!1), 0) ENDIF,
              IF nth(b(st!1), 1) = Newest AND ms(st!1) = emptyset
                THEN Master ELSE nth(b(st!1), 1) ENDIF ,
              IF nth(b(st!1), 2) = Newest AND ms(st!1) = emptyset
                THEN Master ELSE nth(b(st!1), 2) ENDIF :) #)")
```

This gives us two subgoals: we must show that the witness satisfies both the post-condition and the invariant. We shall not describe these proofs in detail because they are lengthy and not particularly instructive. Briefly, each subgoal consisted of a conjunction which was split to give several new subgoals. These subgoals were then proved by case analysis (splitting the hypotheses), rewriting and simplifying each individual case, and then using a decision procedure or some general tactic such as `GRIND` to either verify the conclusion or discover a contradiction within the hypotheses. Much use was made of the `HIDE` command to hide irrelevant formulae and hence speed up the workings of the tactics.

Using a proof approach similar to that described above, we were able to show that the release operation is also feasible. The slave operation should have been similarly easy to handle, but, unfortunately, we were unable to complete the proof because of a bug in the PVS system concerning equality. In certain situations arising after a lengthy sequence of tactics, it seems that the system fails to recognise goals which are simply instances of the reflexivity of equality. This is a known bug and will, hopefully, be corrected in future versions of PVS.

## 6.4.4 Some Validation Conditions

In both the VDM and PVS specifications of the slave operation, the postconditions explicit specify what happens to those buffers which have status Slave or Master, but do not describe the effect on buffers which have status Newest or Idle. However, in conjunction with the invariant (and the frame in the VDM version) the postcondition ensures that no other Newest buffer remains, exactly one new Slave buffer is chosen, and no new Master buffers are added. This fact was stated as a validation condition in [8], and we have verified it using PVS. We show its statement in both VDM and PVS notations.

$$\forall i \in \{1,2,3\} \cdot (\overleftarrow{b}(i) \in \{\textsc{n},\textsc{i}\} \;\Rightarrow\; b(i) \in \{\textsc{i},\textsc{s}\})$$

```
slave_prop : CONJECTURE
   (FORALL (i:{x:nat| x=0 OR x=1 OR x=2}) :
      member(nth(b(st),i), {x:Status|x=Newest OR x=Idle})
      IMPLIES
      member(nth(b(slave(st)),i), {x:Status|x=Idle OR x=Slave}))
```

Our PVS proof of this statement required about 500 tactics and is difficult to follow intuitively, although neither its structure nor any of the individual steps is particularly sophisticated. After some initial preparation including skolemization, definition expansion, and adding some simple lemmas, the tactic PROP is invoked. This has the effect of splitting the many disjunctions among the hypotheses and thereby breaking the top goal down into about 200 subgoals. The majority of these were proved easily by some definition expansion, simplification, rewriting, and use of the tactic GRIND to detect contradictions among the hypotheses of the subgoal. Of the 54 subgoals which remained, 36 were proved by simply using GRIND. For the remaining subgoals, further case analysis was used to split each one into smaller subgoals which were then proved by GRIND.

The lemmas stated below were required later in order to prove a refinement proof obligation. We present them as validation conditions since they are reasonable properties to require of the MSMIE system. Their proofs were carried out interactively and required about 15 tactics each.

```
slave_prop2 : LEMMA
   (FORALL (st,st2:sigma2) :
      post_slave(st,st2) IMPLIES count(Newest,b(st2)) = 1)

slave_prop3 : LEMMA
   (FORALL (st,st2:sigma2) :
      post_slave(st,st2) IMPLIES
        count(Master,b(st)) = count(Master,b(st2)))
```

## 6.5 Representing Refinement

The states of the MSMIE system may be described more abstractly by ignoring the identity of individual buffers and distinguishing only the possible combinations of buffers which satisfy the invariant. The two binary choices in the invariant concerning the number of buffers assigned to Master and Newest mean that there are four such combinations: (Slave, Idle, Idle), (Slave, Idle, Newest), (Slave, Idle, Master), and (Slave, Newest, Master).

### 6.5.1 The VDM Specification

In VDM, the possible status combinations are represented by giving a new enumerated type comprising four tokens.

**types**

$$Status1 = \textsc{sii} \mid \textsc{sin} \mid \textsc{sim} \mid \textsc{snm}$$

The state simply records which combination is current and the invariant and initial state are the "images under retrieval" of the concrete ones given previously.

```
state $\Sigma 1$ of
    $bs : Status1$
    $ms : MName$-set
    inv mk-$\Sigma 1\,(bs, ms) \triangleq$
        $ms = \{\} \;\Leftrightarrow\; bs \in \{\mathrm{SII}, \mathrm{SIN}\}$
    init $s \triangleq s =$ mk-$\Sigma 1\,(\mathrm{SII}, \{\})$
end
```

The operations are similar to those given in the previous specifications. In particular, the postconditions rely on the same case distinctions.
operations

```
$slave\,()$
ext wr $bs : Status1$
    rd $ms : (MName$-set$)$
pre true
post $(\overleftarrow{bs} \in \{\mathrm{SII}, \mathrm{SIN}\} \;\Rightarrow\; bs = \mathrm{SIN}) \wedge$
    $(\overleftarrow{bs} \in \{\mathrm{SIM}, \mathrm{SNM}\} \;\Rightarrow\; bs = \mathrm{SNM})$ ;

$acq\,(l : MName)$
ext wr $bs : Status1$
    wr $ms : (MName$-set$)$
pre $(\neg\,(l \in ms)) \wedge (\neg\,(bs = \mathrm{SII}))$
post $ms = \overleftarrow{ms} \cup \{l\} \wedge$
    if $\overleftarrow{ms} = \{\}$
    then $bs = \mathrm{SIM}$
    else $bs = \overleftarrow{bs}$ ;

$rel\,(l : MName)$
ext wr $bs : Status1$
    wr $ms : (MName$-set$)$
pre $l \in ms$
post $ms = \overleftarrow{ms} \setminus \{l\} \wedge$
    if $ms = \{\}$
    then $bs = \mathrm{SIN}$
    else $bs = \overleftarrow{bs}$
```

### 6.5.2 The PVS Specification

We have formalised the more abstract specification as a theory in PVS. The techniques used are the same as for the concrete specification. The TCCs generated by typechecking are also similar, but they are easier to prove because this is a less elaborate specification.

```
msmie_sigma1[MName : TYPE+] : THEORY

  BEGIN

  Status1 : TYPE = {SII,SIN,SIM,SNM}

  sigma1 : TYPE =
   {x : [# bs : Status1,
           ms : setof[MName] #] |
   (member(bs(x),{x:Status1|x=SII OR x=SIN}) <=> ms(x) = emptyset)}

  initial_sigma1 : sigma1 =
    (# bs := SII, ms := emptyset #)


  pre_slave  : [sigma1 -> bool] = LAMBDA (st:sigma1) :
    true

  post_slave : [(pre_slave),sigma1 -> bool] =
    LAMBDA (st:(pre_slave),st2:sigma1) :
      ((member(bs(st),{x:Status1|x=SII OR x=SIN}) IMPLIES
           bs(st2) = SIN)
       AND
       (member(bs(st),{x:Status1|x=SIM OR x=SNM}) IMPLIES
           bs(st2) = SNM))
       AND
       (ms(st2) = ms(st))

  slave : [(pre_slave) -> sigma1] =
    LAMBDA (st:(pre_slave)) :
       choose({st2:sigma1 | post_slave(st,st2)})

  pre_acq : [MName -> [sigma1 -> bool]] =
    LAMBDA (l:MName)(st:sigma1) :
       (NOT member(l,ms(st))) AND (bs(st) /= SII)

  post_acq : [l:MName -> [(pre_acq(l)),sigma1 -> bool]] =
              LAMBDA (l:MName)(st:(pre_acq(l)),st2:sigma1) :
                ((ms(st2) = union(ms(st),singleton(l)))) AND
                 (IF
                      ms(st) = emptyset
                  THEN
```

```
                    bs(st2) = SIM
                ELSE
                    bs(st2) = bs(st)
                ENDIF)

  acq : [l:MName, (pre_acq(l)) -> sigma1] =
         LAMBDA (l:MName, st:(pre_acq(l))) :
              choose({st2:sigma1 | post_acq(l)(st,st2)})
```

The specification of the release operation is not shown.

```
END msmie_sigma1
```

### 6.5.3 The Refinement Relationship

We formalise, as a PVS theory, the statement that `sigma2` is a refinement of `sigma1`. The refinement relationship is modelled by a theory which imports the theories representing the concrete and abstract specifications. The proof obligations that pertain to refinement must be typed in by hand; they are not automatically generated by the system. They are declared to be CONJECTURES in the theory, as described below and then PVS is used to prove them.

```
sigma1_sigma2[MName : TYPE+] : THEORY

  BEGIN

  IMPORTING msmie_sigma1[MName], msmie_sigma2[MName]
```

First, we define the "retrieve" operation mapping concrete states to abstract ones. The definition is given by cases, and is much the same as that given in [8]. Typechecking generates a TCC stating that the retrieved state does indeed satisfy the invariant of `sigma1`. This is proved automatically by `tcp`.

```
  retr2_1 : [sigma2 -> sigma1] = LAMBDA (st:sigma2) :
     (# bs :=
             LET cc : [nat,nat] =
                 (count(Newest,b(st)),count(Master,b(st)))
             IN
               IF    cc = (0,0) THEN SII
               ELSIF cc = (1,0) THEN SIN
               ELSIF cc = (0,1) THEN SIM
               ELSE                  SNM
               ENDIF,
        ms := ms(st) #)
```

Next we typed in the various proof obligations required to demonstrate that `sigma2` is a refinement of `sigma1`. All of these have been verified using PVS. We shall indicate how difficult it was to carry out each verification.

Showing adequacy was a relatively non-trivial task, taking a few hours to complete. The proof itself is conceptually simple: we consider each of the possible values of the system status in the abstract specification, and supply a suitable value for the concrete state in each case. For each of these we must then show two things: that it is the value returned by the , and that it satisfies the invariant of sigma2. The first can be proved almost automatically by expanding some definitions and invoking the tactic `GRIND`. To prove the second we must use the invariant of the abstract state. The entire proof script for adequacy is about 130 lines long.

```
  adeq2_1 : CONJECTURE
     (FORALL (st1:sigma1) :
         (EXISTS (st2:sigma2) : retr2_1(st2) = st1))
```

Next we proved that the maps the concrete initial state to the abstract initial state. This was very simple: after expanding one definition, the "grind" tactic completed the proof.

```
  init2_1 : CONJECTURE
     (retr2_1 (initial_sigma2) = initial_sigma1)
```

Next was the domain rule for the slave operation. This is trivial and was proved automatically by "grind".

```
  slave_dom2_1 : CONJECTURE
     (FORALL (st2:sigma2) :
         (pre_slave (retr2_1(st2))) IMPLIES pre_slave(st2))
```

To prove the result rule for slave we first added the lemmas `slave_prop1` and `slave_prop2` which were validation conditions in the theory `msmie_sigma2`. Next, we used the tactic `TYPEPRED` to make the invariant of `sigma2` visible to the prover. Once this preparation was in place, the `GRIND` tactic completed the proof.

```
 slave_result2_1 : CONJECTURE
     (FORALL (st2_:((pre_slave:[sigma2->bool])),st2:sigma2) :
         ((pre_slave (retr2_1(st2_)) AND (post_slave(st2_,st2)))
          IMPLIES
          (post_slave (retr2_1(st2_),retr2_1(st2)))))
```

The domain rule for acq required a proof about 30 tactics long as well as a lemma about the `count` function. The proof structure is as follows: first definitions were expanded, the lemma was added, hidden type information was made visible, and the proof was split into two subgoals; then the `GRIND` tactic was called upon to complete the proof.

```
  acq_dom2_1 : CONJECTURE
     (FORALL (st2:sigma2) : (FORALL (l:MName) :
         (pre_acq (l)(retr2_1(st2))) IMPLIES pre_acq(l)(st2)))
```

The result rule for acq required a complicated, interactive proof comprising hundreds of tactics which took several days to complete.

```
 acq_result2_1 : CONJECTURE
     (FORALL (l:MName) :
     (FORALL (st2_:(pre_acq(l):[sigma2->bool]),st2:sigma2) :
         ((pre_acq (l)(retr2_1(st2_)) AND (post_acq(l)(st2_,st2)))
          IMPLIES
          (post_acq (l)(retr2_1(st2_),retr2_1(st2)))))
```

The domain rule for the release operation was proved automatically by `GRIND`.

```
  rel_dom2_1 : CONJECTURE
     (FORALL (st2:sigma2) : (FORALL (l:MName) :
         (pre_rel (l)(retr2_1(st2))) IMPLIES pre_rel(l)(st2)))
```

The result rule for this operation required a complex, interactive proof very similar to that of the result rule for acq.

```
 rel_result2_1 : CONJECTURE
    (FORALL (l:MName) :
    (FORALL (st2_:(pre_rel(l):[sigma2->bool]),st2:sigma2) :
       ((pre_rel (l)(retr2_1(st2_)) AND (post_rel(l)(st2_,st2)))
        IMPLIES
        (post_rel (l)(retr2_1(st2_),retr2_1(st2)))))

 END sigma1_sigma2
```

## 6.6 Discussion

In this section we note various observations made during our experiments. These include some points about the PVS system, as well as a discussion of some differences between the logics of VDM-SL and the PVS specification language.

### 6.6.1 Using the PVS System

PVS facilitates proofs at a fairly non-tedious level, due to the integrated decision procedures and rewriting techniques. Low level proof hacking using for instance associativity and commutation properties of arithmetic operations is usually not necessary. Of course, the real difficult side of theorem proving is still difficult, for instance, understanding the application (and formalizing it correctly), inventing proofs, and generating suitable lemmas. However, we were impressed by the fact that we were actually able to prove all (but one) of the proof obligations, including refinement proof obligations, for the MSMIE example. This augers well for the usability of the system for further applications.

In all of our examples we made use of the TCC mechanism of PVS to obtain some automatic proof obligation generation. This results in TCCs which are, in general, too complicated to be solved by the PVS command "typecheck-prove", which is good at automatically finding proofs of simple TCCs. Unfortunately, in the present implementation of PVS it is impossible to prevent this command from embarking on time-consuming attempts to prove all existing TCCS for the current theory, even though the user may know that certain ones are too difficult to be solved. A more flexible version of "typecheck-prove", or perhaps simply a time limit to its operation, would be welcome.

One may like or dislike the PVS Emacs interface. Though all of the authors were used to Emacs, we disliked some of its features relating to PVS. For instance, we found that the way in which buffers popped up and destroyed existing Emacs windows was confusing and irritating. We also felt that the quite frequent switching between buffers that we had to do became somewhat of a bottleneck. Moreover, the interface was unreliable and it was often necessary to restart PVS when Emacs ended up in a state where you could not execute important PVS commands.

## 6.6.2 Partiality in VDM and PVS

The most notable difference between the specification languages of PVS and VDM is that PVS deals only in total functions. In practice, much of the language flexibility of partial functions in LPF can be captured in PVS by the use of subtypes and dependent types to express the domain of definition of a function. A good example is the *nth* function on lists which is defined recursively in PVS as follows:

```
nth(l, (n:nat | n < length(l))): RECURSIVE nat =
   IF n = 0 THEN car(l) ELSE nth(cdr(l), n-1) ENDIF
  MEASURE length(l)
```

As the following examples show, this function may be used freely when writing specifications: the fact that it is partial imposes no special syntactic constraints. For example we can write:

```
ex1:nat = nth((: 1, 3 , 2 :), 1)

ex2:nat = nth((: 1, 3 , 2 :), 4)
```

Correctness is maintained by typechecking. The first example causes no problems. However, the second example results in the false TCC

```
4 < length[nat]((: 1, 3, 2 :))
```

An example of a use of partial functions which is possible in LPF but not in PVS involves the `subp` function, due to Cliff Jones. In PVS it may be defined as follows[1].

```
subp(i:nat,j:nat | i >= j): RECURSIVE nat =
  IF i=j THEN 0 ELSE 1 + subp(i,j+1) ENDIF
  MEASURE abs(i-j)
```

When applied to natural numbers `i` and `j`, where `i > j`, this function returns the difference `i-j`. This property can be formalised and proved in both PVS and LPF. However, the following property which is also true of `subp` in LPF, cannot be proved in PVS — in fact, it cannot even be typechecked.

```
subp_lemma : CONJECTURE
 FORALL (i,j:nat) : (subp(i,j) = i - j) OR (subp(j,i) = j - i)
```

Attempting to typecheck this results in the false TCC:

```
subp_lemma_TCC1: OBLIGATION (FORALL (i, j: nat): i >= j);
```

Fortunately, the present example does not contain any construction where this distinction is significant.

### 6.6.3 Looseness in VDM and PVS

Another semantic difference between VDM and our translation to PVS is in the interpretation of expressions whose values are not fully determined.

In VDM, looseness in function definitions is interpreted as underspecification, that is to say, every invocation of a function with the same argument will return the same result; whereas looseness in operation definitions is understood to be genuine non-determinism, so separate invocations of a loosely specified operation can yield different results even if called with the same arguments and in the same state [16]. The motivation for this distinction is that, in an implementation, the result of an operation may depend on some state not being modelled in the abstraction, whereas a function should be declarative however it is implemented.

For example, for functions $f$ and $g$:

$$f(x) = \textsf{let}\ y\ \textsf{be st}\ y > x\ \textsf{in}\ y\ \textsf{end}$$
$$g(x) = \textsf{let}\ y\ \textsf{be st}\ y > x\ \textsf{in}\ y\ \textsf{end}$$

we can always be certain that $f(x) = f(x)$, but $f(x) = g(x)$ may not necessarily hold in a refinement.

PVS, on the other hand, takes a more constrained interpretation of looseness: all occurrences of the same choice expression must yield the same result wherever they

[1]This definition is due to Klaus Havelund

occur. So, if we make the corresponding definitions in PVS,

```
f(x) = choose({y:nat | y > x})
g(x) = choose({y:nat | y > x})
```

then we always have `f(x) = g(x)` also.

The VDM interpretation of loose functions is appropriate in the context of a development employing the "design by contract" paradigm. Underspecification represents the deferral of a design decision concerning the choice of a fully determined implementation. Thus looseness can be removed during refinement and the resulting behaviour will be no worse from the caller's point of view than that of the loose function. However, this interpretation of looseness has severe implications for reasoning as it prohibits the indiscriminate substitution of equals. We cannot make the simple chain of equalities:

$$f(x) = \mathsf{let}\ y\ \mathsf{be\ st}\ y > x\ \mathsf{in}\ y\ \mathsf{end} = g(x)$$

Rather, each occurrence of a loose expression must in some way be tagged in order that it is possible to determine which occurrence is being referred to when it occurs in proofs. It also means that beta-reduction can only be undertaken when the argument is fully determined [16].

The PVS interpretation of looseness yields simpler proofs since we can be sure that identical expressions will have equal value irrespective of how they arise. However, this interpretation of looseness defies compositionality in refinement as if the same choice expression occurs in two separate parts of a specification, they must both be treated similarly in any subsequent refinement.

In the present example, we have interpreted implicit operations by use of PVS choice operator. In the case where there is some genuine choice, as in *slave*, this is not strictly correct. With this interpretation we could prove properties of the specification which are not necessarily preserved by an implementation.

These differences in the semantics raise methodological questions about the use of such constructions in practice. Though partiality and looseness are both extremely useful, they should be used with caution particularly in circumstances where there is disagreement as to their interpretation.

### 6.6.4 Errors in Example Specifications

The translation into PVS did not reveal any errors in the MSMIE specification. However, a number of errors were found in two other realistic VDM-SL specifications (not shown here) which were translated into PVS by Agerholm. A third specification, also translated by Agerholm, was not found to contain any errors.

The errors themselves are not major and should perhaps mainly be read as small and funny, but also worrying, examples of the errors that people make in writing formal specifications (and programs). They may be divided into three categories: (1) those that were pointed out directly during the proof of a type checking condition, (2) those that probably could have been found easily by testing specifications, (3) other errors, some of which are quite subtle, e.g. due to parentheses problems. What the errors teach us is that in specification debugging one can benefit from working with specifications in a formal way. However, other alternatives for validation such as testing could have found some of the errors as well. For detailed discussion of the errors that were found, the reader is referred to [3].

## 6.7 Conclusion

The VDM-SL style of specification and refinement fits well with the PVS specification language. As a result we were able to use a very direct embedding of VDM in PVS (for example, logical formulae in VDM are represented by those of PVS), which means that the proof capabilities of the PVS system are available directly to the VDM user. This is not always the case where a deeper, more indirect embedding is required, forcing the user to navigate through layers of definitions. A shallow embedding is very desirable in a closed system like PVS, though it is less important in open systems such as HOL and Isabelle where a programming language is available to automate the deep embedding process.

At present the translation from VDM-SL to PVS, and the generation of refinement proof obligations must both be done manually. As well as being inconvenient, these manual processes are opportunities for the introduction of errors. It is possible to automate the translation step outside of PVS, generating PVS theories from VDM-SL specifications. However, the closed nature of the PVS system makes it difficult to achieve a close integration with other tools supporting VDM-SL such as can be achieved with other more open systems [4, 3]. This leads to the disadvantage that a VDM-SL user who wishes to use PVS for proofs must master the PVS notation and become, in effect, a PVS user as well.

Because of the difficulties described in the previous paragraph, as well as the semantic differences between PVS and VDM-SL described in Section 6, we do not view PVS as a satisfactory proof tool for VDM-SL. However, the ease with which VDM-SL *style* may be transported to PVS means that this style may be a useful approach for VDM-SL users wishing to experiment with PVS.

The authors are all convinced of the need for and the benefits to be derived from the use of tools to support VDM specification. The extensive type-checking done by the PVS system contributes greatly to our confidence in the correctness of the specifications and refinements. For example, we can be certain that functions are applied only to arguments within their domain.[2] We also derived confidence from the proof process. Although this is somewhat muted by reports of bugs in the PVS prover, the advantages of mechanical support for proof compared to making proofs by hand almost certainly outweighs the possibility of the system constructing an

[2]A similar kind of facility, called a proof obligation generator, is currently being developed for the IFAD VDM-SL Toolbox [6].

erroneous proof.

The PVS prover was sufficiently fast and powerful to make it feasible to do proofs of the size shown in this chapter, though the time required to do this is considerable and so proof for "real-world" applications remains an expensive activity. On the other hand, the proofs which were undertaken, though not mathematically sophisticated, involved such elaborate case analyses that it is unlikely that they would be successfully carried out without the help of tools.

Further work needs to be carried out in order to discover the implications as far as refinements are concerned of the different approaches to looseness taken in VDM-SL and PVS. It would also be interesting to see how the approach scales up to larger, more "real-world" applications which might provide a more exacting test of the capabilities of the PVS system. Finally, the authors are interested in a comparison between PVS and the new VDM proof tool based on Isabelle which is currently under development at IFAD.

## Acknowledgments

Sten Agerholm would like to thank Peter Gorm Larsen for his support and interest in the work reported here. Agerholm's work was funded by the Danish Research Councils. Savi Maharaj wishes to thank the European Research Consortium in Informatics and Mathematics for funding her stay at the Rutherford Appleton Laboratory where her work was carried out.

## 6.8 Bibliography

[1] S. Agerholm. *A HOL Basis for Reasoning about Functional Programs.* PhD thesis, BRICS, Department of Computer Science, University of Aarhus, 1994. Available as Technical Report RS-94-44.

[2] S. Agerholm. LCF examples in HOL. In *The Computer Journal*, 38(2), 1995.

[3] S. Agerholm. Translating Specifications in VDM-SL to PVS. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOL '96)*, Springer-Verlag LNCS 1125, 1996.

[4] S. Agerholm and J. Frost. An Isabelle-based Theorem Prover for VDM-SL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, Springer-Verlag LNCS, 1997.

[5] S. Agerholm and J. Frost. Towards an Integrated CASE and Theorem Proving Tool for VDM-SL. In *FME'97*, Springer-Verlag LNCS, 1997.

[6] B. Aichernig and P. G. Larsen. A Proof Obligation Generator for VDM-SL. In *FME'97*, Springer-Verlag LNCS, 1997.

[7] D.J. Andrews and M. Bruun et al. Information Technology — Programming Languages, their environments and system software interfaces — Vienna Development Method-Specification Language Part 1: Base language. ISO Draft International Standard: 13817-1, 1995.

[8] J.C. Bicarregui. A Model-Oriented Analysis of a Communications Protocol. Technical report RAL-93-099, Rutherford Appleton Laboratory, 1993.

[9] J.C. Bicarregui and B. Ritchie. Invariants, Frames and Postconditions: a comparison of the VDM and B notations. In *Proceeding of Formal Methods Europe '93*, Springer-Verlag LNCS 670, 1993. Also in *IEEE Transaction on Software Engineering*, 21(2), 1995.

[10] R. J. Boulton and A. D. Gordon *et al.* Experience with Embedding Hardware Description Languages in HOL. In *Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP TC10/WG 10.2 International Conference*, North-Holland, IFIP Transactions A-10, 1992.

[11] G. Bruns and S. Anderson. The Formalization and Analysis of a Communications Protocol. In *Formal Aspects of Computing*, 6(1), Springer, 1994.

[12] G. Collins and D. Syme. A theory of finite maps. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Springer-Verlag LNCS 971, September 1995.

[13] J. Crow, S. Owre *et al. A Tutorial Introduction to PVS.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[14] R. Elmstrøm, P.G. Larsen, and P.B. Lassen. The IFAD VDM-SL Toolbox: A practical approach to formal specifications. In *ACM Sigplan Notices* 29(9), 1994

[15] P.G. Larsen. *Towards Proof Rules for VDM-SL.* PhD thesis, Technical University of Denmark, Department of Computer Science, March 1995. ID-TR:1995-160.

[16] Peter G. Larsen and Bo S. Hansen. Semantics of Underdetermined Expressions. In *Formal Aspects of Computing*, 8(1), 1996.

[17] L.L. Santoline *et al.* Multiprocessor Shared-Memory Information Exchange. In *IEEE Transactions on Nuclear Science,* 36(1), 1989.

[18] H. Sondergaard and P. Sestoft. Non-determinism in functional languages. In *The Computer Journal*, 35(5), 1992.

View publication stats