On the Verification of VDM Specification and Refinement with PVS

Savi Maharaj*
Department of Computing and Mathematics
Stirling University
Stirling FK9 4LA, UK
savi@cs.stir.ac.uk

Juan Bicarregui
Rutherford Appleton Laboratory
and
Imperial College, London

Abstract

Although the formal method VDM has been in existence since the 1970's, there are still no satisfactory tools to support verification in VDM. This paper deals with one possible means of approaching this problem by using the PVS theorem-prover. It describes a translation of a VDM-SL specification into the PVS specification language using, essentially, the very transparent translation methods described in [1]. PVS was used to typecheck the specification and to prove some non-trivial validation conditions. Next, a more abstract specification of the same system was also expressed in PVS, and the original specification was shown to be a refinement of this one. 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.

1. Introduction

In the paper [1], Sten Agerholm demonstrates how a large subset of VDM-SL[4] may be translated directly into the specification language of the PVS theorem prover[8]. The translation is a "shallow embedding" which works well in this case because the two languages possess many structures and concepts in common. The result is a simple and direct means of harnessing the power of the PVS proof-checker for verifying results about VDM-SL specifications.

We independently discovered essentially the same representation of VDM-SL in PVS, and explored this by translating a medium sized VDM-SL specification into PVS and proving several theorems about it. Next, we considered VDM-style refinement, and found that the PVS theory mechanism can be used to express this quite satisfactorily. Again, we used our example to explore this approach, and found that we were able to prove all of the refinement proof obligations with the PVS proof-checker.

The VDM-SL example which is used was developed with the help of the IFAD VDM-SL Toolbox [9] which provides syntax-checking and some typechecking for VDM-SL. Other work which deals with refinement within PVS includes [12] which is concerned with providing proof support for Abrial's "B" Abstract Machine Notation.

The main contributions of this paper are a medium-sized case study with non-trivial theorems proved giving evidence to support the feasibility of the approach proposed in [1] and a simple extension of this approach to incorporate refinement. The paper is organized as follows: Section 2 describes the application which serves as our main example. In Section 3 we show how this system was formally specified in VDM-SL, the specification translated into PVS, and a number of proof obligations and validation conditions formally verified using PVS. In Section 4 we present our first specification as a refinement of a more abstract specification, and discuss using PVS to prove the refinement proof obligations. In the final sections, we present some observations about what was learned during this project and our conclusions.

2. The MSMIE protocol

MSMIE (Multiprocessor Shared-Memory Information Exchange) is a protocol for "inter-processor communications in distributed, microprocessor-based nuclear safety systems" [13], 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

*Savi Maharaj would like to thank the European Research Consortium in Informatics and Mathematics for supporting this work.

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 [5] and is used in as a basis of a comparison of VDM and B in [7]. In a simplification considered in [5] 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 realized 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 [6] and [7] is concerned with various "data models" of MSMIE: the state of the device is modeled 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 behavior is purely sequential.

There are three such operations:

slave This operation is executed when a write finishes. It sets the status of the buffer that was being written to "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 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 [13] by the use of timing constraints. The paper [5] 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 modeled in [6] by using non-standard extensions to VDM which provide a more concise means of expression than standard VDM. In this paper we restrict ourselves to standard VDM specifications and do not treat the improved MSMIE protocol.

3. Formal specifications of MSMIE

The paper [6] explores several ways of specifying this 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 translation can be done 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 shown here have been modified to conform with standard VDM-SL so they differ slightly from those in [6].

3.1. 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

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

$count\,(s, ss) \triangleq$
  cases ss :
    $[\,] \rightarrow 0$,
    others $\rightarrow$ 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

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

$$MName = \text{token}$$

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.

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

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

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

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.

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

The release operation takes place when a master process has finished reading. Because of space limitations in this paper, the specification of this operation is not shown.

3.2. The PVS system

The PVS system[8] was developed at SRI International in California. It features an expressive specification language 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. The PVS specification language is supported by a collection of tools including a type checker and an interactive proof checker in which basic proof commands are tightly integrated with powerful decision procedures and tactics. The main attraction of the method exemplified in this paper is that it enables us to use this powerful proof-checker for verification of VDM.

3.3. 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 a nondeterministically chosen valid state which, when paired with the input value, satisfies the postcondition[1].

Typechecking such a definition causes PVS to generate a TCC stating that a suitable value for the non-deterministic choice 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)))
```

[1]The interpretation of looseness in VDM and PVS is discussed in Section 5.2.

The slave operation is then defined using the nondeterministic choice operator "choose".

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

The specifications of acquire and release are done similarly. 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 not shown.

3.4. Typechecking constraints

Typechecking a PVS theory usually results in the creation of proof obligations called *typechecking constraints* (TCCs). 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`, 4 of the TCCs which are generated are too difficult for `tcp` to prove. Interestingly, these TCCs correspond to the satisfiability proof obligations for the specification: in effect, we must show that the initial state satisfies the invariant, and that each of the three operations is feasible.

3.4.1. 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 recognizes to be trivially true.

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

3.4.2. 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?
       ({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 postcondition 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 a powerful 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. (The PVS implementors are aware of this bug, and are working on correcting it.)

3.5. Some validation conditions

In our specification of the slave operation, the postconditions explicitly 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, one can show that, 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 [6]. We have verified it using PVS. We show its statement in both VDM and PVS notations.

$$\forall\, i \in \{1,2,3\} \cdot (\overleftarrow{b}\,(i) \in \{\mathrm{N},\mathrm{I}\} \;\Rightarrow\; b\,(i) \in \{\mathrm{I},\mathrm{S}\})$$

```
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, which does propositional simplification, 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 RECORD 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)))
```

4. A more abstract description: 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).

4.1. The VDM specification

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

types

$$Status1 = \mathrm{SII} \mid \mathrm{SIN} \mid \mathrm{SIM} \mid \mathrm{SNM}$$

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. As before, we do not show the specification of the release operation.

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 $bs = \overleftarrow{bs}$;

4.2. The PVS specification

We have formalized 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)})
```

4.3. The refinement relationship

We can also formalize, as a PVS theory, the statement that sigma2 is a refinement of sigma1. The refinement relationship is modeled 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 then 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 [6]. 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 retrieve function, 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 retrieve function 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 (not shown) for the release operation was proved automatically by `GRIND`. The result rule for this operation required a complex, interactive proof very similar to that of the result rule for acq.

5. Results

Generally, the combination of the PVS specification language with VDM specification and refinement style seems to work well. PVS can express a large, useful subset of VDM-SL, despite its lack of union types and its restriction to total functions only. Indeed, much, but not all, of the expressiveness of partial functions in VDM-SL can be captured in PVS by the use of subtyping and dependent types. Some examples of partial functions which show this are given below in Section 5.1. Section 5.2 discusses the treatment of looseness in VDM and PVS.

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 this development using PVS. This augers well for the usability of the system for further applications.

Like [1], 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 TCCs, 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.

Translating from VDM to PVS must be done by hand. There is also no automatic proof obligation generation for proof obligations arising from refinement. The closed design of the PVS system means that it may be difficult to implement extensions to it which would support these processes. In practice, there is no need for the translation if the user is willing to work directly with the PVS specification language. The lack of automatic translation would make it a time-consuming process to apply our approach to translation to the large body of existing VDM specifications. However, the time to translate and individual specification is small relative to the time taken to prove it and so this lack of support for this should not deter the serious formal developer.

5.1. 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 this 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[2].

[2] This definition is due to Klaus Havelund.

```
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 formalized 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.

5.2. 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[11]. The motivation for this distinction is that, in an implementation, the result of an operation may depend on some state not being modeled in the abstraction, whereas a function should be declarative however it is implemented.

For example, for functions f and g:

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

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 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 behavior 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:

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

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[11].

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 the 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 two 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. Conclusion

The VDM-SL style of specification and refinement fits well with the PVS language. As a result we are 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.

The extensive type-checking done by the PVS system contributes greatly to our confidence in the correctness of specifications and refinements. For example, we can be certain that functions are applied only to arguments within their domain. 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 erroneous proof.

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 [3, 2]. 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.

The PVS prover was sufficiently fast and powerful to make it feasible to do proofs of the size attempted in this example, 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 to discover the implications, as far as refinement is concerned, of the different approaches to looseness taken in VDM-SL and PVS. It would also be interesting to see how well 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.

References

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

[2] 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.

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

[4] D.J. Andrews and H. 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

[5] G. Bruns and S. Anderson. The Formalization and Analysis of a Communications Protocol, Formal Aspects of Computingm Volume 6, Number 1, Springer, 1994

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

[7] J.C. Bicarregui and B. Ritchie. Invariants, Frames and Postconditions: a comparison of the VDM and B notations, Proceeding of Formal Methods Europe '93, Woodcock and Larsen (Eds), LNCS 670, Springer-Verlag. Also in IEEE Transaction on Software Engineering, Vol. 21, No. 2, pp. 79-89, Feb. 1995.

[8] J. Crow, S. Owre, J. Rushby, N. Shankar and M. Srivas. A Tutorial Introduction to PVS, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[9] R. Elmstrøm, P.G. Larsen, and P.B. Lassen. The IFAD VDM-SL Toolbox: A practical approach to formal specifications. *ACM Sigplan Notices*, 29(9):77-80, September 1994.

[10] C.B. Jones. Systematic Software Development using VDM (second edition), Prentice Hall, 1990.

[11] Peter G. Larsen, Bo S. Hansen. Semantics of Underdetermined Expressions, Formal Aspects of Computing, vol. 8, no. 1, 1996.

[12] C.H. Pratten, An Introduction to proving AMN Specifications with PVS and the AMN-Proof Tool, *Z Twenty Years on- What is its Future?,* Proceedings of the 7th International Conference on: Putting into practice methods and tools for information systems design. H. Habrias (ed), ISBN: 2-906082-19-8, Nantes, France, October 1995.

[13] L.L. Santoline *et al.* Multiprocessor Shared-Memory Information Exchange, IEEE Transactions on Nuclear Science. Vol. 36. No. 1, February 1989, pp. 626-633.