九州大学
KYUSHU UNIVERSITY

VDMTools

The Integrity Checking: Using Proof Obligations

ver.1.0

How to contact:

http://fmvdm.org/	VDM information web site(in Japanese)
http://fmvdm.org/tools/vdmtools	VDMTools web site(in Japanese)
inq@fmvdm.org	Mail

The Integrity Checking: Using Proof Obligations 1.0
— Revised for VDMTools v9.0.6

Contents

1 Introduction

A VDM++ model, even one that is type correct, may contain internal inconsistencies. For example, there may be potentially erroneous applications of partial operators, or there may be more subtle defects, such as failure of a function to maintain a data type invariant. Freedom from such defects is called "integrity". The Integrity Examiner in VDMTools helps users to check the integrity of models by automatically generating the checks (called "proof obligations") that have to be performed in order to ensure integrity. If all the proof obligations are shown to be satisfied by the model, then there is a guarantee that the VDM model is internally consistent [6]. Checking proof obligations helps to identify many defects in models. However, it should be stressed that it does not guarantee that the model is fit for purpose; for example, it does not prove that the models is an accurate expression of system requirements.

This document provides insight into the different kinds of proof obligation and into alternative techniques to gain confidence in the correctness of each kind of proof obligation.

The most typical proof obligations involve checking the following:

- data type invariants,
- pre-conditions,
- post-conditions,
- sequence bounds,
- mapping domains and
- in general application of partial operators.

How should proof obligations be checked? A range of techniques is available, including inspection, testing and proof. At the simplest level, the user may simply inspect the obligations and check them manually. In the case of many obligations this can be a very quick process, but may lack reliability. Proof obligations ay also be checked by testing, in which the statement of the obligation itself is used to suggest carefully selected tests that check for potential violations. These tests can form useful additions to the normal test set for the model and may be useful in regression testing.

Perhaps the most obvious way to check a proof obligation is by proving it using a VDM proof support system. Such a proof support system is not yet a standard feature of VDMTools, although experimental systems are under development in the research community. This document explains how users may instead prove parts of the proof obligations by a mapping to using Higher Order Logic (HOL) [11] or alternatively by doing manual proofs in the manner [7].

The user manual for VDMTools [12] contains an explanation of the integrity checker. This manual provides further detail on the types of proof obligation generated by the integrity checker and offers advice on the ways in which they can be checked. For each different kind of proof obligation the manual provides:

- A brief explanation;
- A small example; and
- Guidance on how one may gain more confidence in the correctness is provided.

The guidance here covers inspection, testing and, in some cases, example proofs written by hand using the classical VDM style [7, 13]. This style is used because it is easier to comprehend for human beings than the typical automatic proofs produced by a theorem prover such as HOL [10].

The examples used in this manual are based mainly on functional models. This is because proof obligations for explicit operations with statements in the body are generated by the integrity examiner as predicates that that are required to hold at a particular point in the operation body. This contrasts with the obligations generated for functions and implicit operations; in these cases the context of the predicate is incorporated into the proof obligation. Explicit operations are treated in this different way because it is not possible to derive the appropriate context information automatically because of the presence of the presence of loops etc. For these obligations, the user needs to gain confidence manually that the predicate will not be violated in any possible execution of a body statement.

Section 2 gives an overview of the different categories of proof obligation; the obligations in each category are then decribed in Scetions 3 to 6. Section 7 explains how proofs can be carried out using automated support by mapping to HOL [24].

2 Proof Obligation Categories

The integrity checker in VDMTools distinguishes around 30 types of proof obligation for VDM++ models. Many of the obligations share similarities and can be proved in similar ways. Therefore, a categorization will be followed that was proposed and used in [5] as part of the PROSPER project. The following groups of proof obligations are distinguished:

Domain checking: This group contains the obligations resulting from the use of partial functions and partial operators. These are mainly caused by the use of preconditions on functions or on operators.

Subtype checking: Proof obligations that are required due to the use of subtypes, in particular the use of invariants and union types.

Satisfiability of implicit definitions: Proof obligations that are the result of the use of implicit function/operation definition style. For each of these, a proof has to be given that the implicitly defined function can have a result for any valid input.

Termination: For each recursive function a proof is required that it will always terminate.

3 Domain Checking Proof Obligations

3.1 Function Applications

Explanation: Whenever a function or an operation that has a pre-condition is called it is the caller's responsibility to ensure that the pre-condition is satisfied at the calling point.

Example:

```
values

  c1 : nat = 8

functions

Div: real * real -> real
Div(a,b) ==
  a / b
pre b <> 0;

Use: real -> real
Use(a1) ==
  let b1 = if a1 <= c1 then a1 else 0
  in
    Div(a1,b1)
pre a1 <> 0 and a1 < c1
```

Here the application of the `Div` function will yield a proof obligation:

```
(forall a1 : real &
   a1 < c1 =>
   (let b1 = (if a1 <= c1 then a1 else 0)
    in
      pre_Div(a1, b1)))
```

Guidance:

Inspection: This kind of proof obligation can be inspected by checking manually whether the context of the function application ensures that the pre-condition of the called function is satisfied. In this example if the function `Use` did not have a pre-condition itself `b1` could be defined using the `else` clause and thus be 0. In essense the "`else 0`" part here is "dead code" because it can never be reached if the pre-condition is not violated.

Test: Tests for this kind of proof obligation can improve confidence in the correctness in the calling function if they aim to violate the pre-condition of the function being called. So in this case one would try to find arguments to the `Use` function that satisfy both its type constraint and its pre-condition (if any), but violate the pre-condition of the `Div` function. If in this case the first "`a1 <> 0`" was omitted from the pre-condition of `Use` it would be possible to violate the pre-condition of `Div` by the call `Use(0)`.

Proof: A manual proof of this proof obligation would take the following form:

```
from
1 from a1 : real
1.1 from a1 <> 0 and a1 < c1
1.1.1  a1 <> 0                                          and-E-R(1.1.h)
1.1.2  a1 < c1                                          and-E-L(1.1.h)
1.1.3  (if a1 <= c1 then a1 else 0) =  a1
                                              cond-true(1.h,1.1.3)
1.1.4  (if a1 <= c1 then a1 else 0) <> 0
                                    =-subs-L(b)(1.h,1.1.1,1.1.3)
1.1.5  (if a1 <= c1 then a1 else 0) : real
                                           type-inh-L(1.h,1.1.3)
1.1.6  pre-Div(a1,(if a1 <= c1
                   then a1
                   else 0))                              fold(1.1.4)
    infer (let b1 = if a1 <= c1 then a1 else 0
           in
             pre-Div(a1,b1)                        let(1.1.5,1.1.6)
  infer a1 <> 0 and a1 < c1 =>
        (let b1 = if a1 <= c1 then a1 else 0
         in
           pre-Div(a1,b1)                                  =>-I(1.1)
infer forall a1 : real & a1 <> 0 and a1 < c1 =>
        (let b1 = if a1 <= c1 then a1 else 0
         in
           pre-Div(a1,b1)                                forall-I(1)
```

If, for example, one had forgotten the "`a1 <> 0`" part in the precondition this would be discovered in an automatic proof by this predicate appearing as a new sub-goal (which naturally could not be proved). If an interactive proof approach was used, the proof would come to a failing point as shown below:

```
from
1 from a1 : real
1.1 from a1 < c1
1.1.1  a1 <= c1                                          <to<=(1.1.h)
1.1.2  (if a1 <= c1 then a1 else 0) =  a1
                                              cond-true(1.h,1.1.1)
1.1.3  a1 <> 0                                              /FAILS/
1.1.4  (if a1 <= c1 then a1 else 0) <> 0
                                    =-subs-L(b)(1.h,1.1.1,1.1.3)
1.1.5  (if a1 <= c1 then a1 else 0) : real
                                         type-inh-L(1.h,1.1.3)
1.1.6  pre-Div(a1,(if a1 <= c1
                   then a1
                   else 0))                              fold(1.1.4)
    infer (let b1 = if a1 <= c1 then a1 else 0
           in
             pre-Div(a1,b1)                      let(1.1.5,1.1.6)
  infer a1 < c1 =>
        (let b1 = if a1 <= c1 then a1 else 0
         in
           pre-Div(a1,b1)                                =>-I(1.1)
infer forall a1 : real & a1 < c1 =>
        (let b1 = if a1 <= c1 then a1 else 0
         in
           pre-Div(a1,b1)                              forall-I(1)
```

3.2 Mapping Application

Explanation: Whenever a mapping is applied to a value it must be ensured that the value is present in the domain of the mapping.

Example:

```
functions

MapApply: (map bool to real) * bool -> real
MapApply(m,a) ==
```

```
  m(a)
pre {true,false} = dom m
```

Here the application of the mapping `m` yields the following proof obligation:

```
(forall m : (map bool to real), a : bool &
   {true,false} = dom (m) => a in set dom (m))
```

Note that if the obvious pre-condition "`a in set dom m`" was used the integrity checker would not have generated a proof obligation at all, because it would be trivially found in the context.

Guidance:

Inspection: When inspecting proof obligations for mapping applications one needs to look for assurance of the existence of the applied expression in the domain of the mapping. Typically this is guranteed by either a pre-condition or a check in the context before the mapping application of the form "`a in set dom m`".

Test: In deriving test cases for such proof obligations one must search for cases where the application is not defined in the domain of the mapping.

Proof: Proofs of this kind of obligations follow the same style as presented above for function application. When automatic proof support is available this kind of proof obligation can often be proved entirely automatically.

3.3 Sequence Application

Explanation: Whenever a sequence is applied with an index value it must be ensured that the index is within the length of the sequence.

Example:

```
functions

MapApply: (seq of real) * nat1 -> real
MapApply(s,i) ==
  s(i)
pre i in set {1,..., len s}
```

Here the application of the sequence `s` yield the following proof obligation:

```
(forall s : (seq of real), i : nat1 &
   i in set {1,...,len (s)} => i in set inds (s))
```

Note that if the pre-condition had been expressed in the natural way: "`i in set inds s`" the proof obligation would not be generated at all, because it would be trivially be found in the context.

Guidance:

Inspection: When inspecting proof obligations for sequence applications one needs to look for assurance of the existence of the indexed expression in the indices of the sequence. Typically this is guranteed by either a pre-condition or a check in the context before the sequence application of the form "`i in set inds s`".

Test: In deriving test cases for such proof obligations one must seach for cases where the application is not defined in the indices of the sequence.

Proof: Proofs of this kind of proof obligations follow the same style as presented above for function application. When automatic proof support is available this kind of proof obligations can normally be proved entirely automatically.

3.4 Non-empty Sequence

Explanation: Some operators on sequences (i.e. hd and tl) are partial and when they are not defined for empty sequences this proof obligation will be generated.

Example:

```
functions

NonEmptySeq: seq of nat -> nat
NonEmptySeq(list) ==
  if len list > 1
  then hd list
  else 0
```

Here the application of the `hd` sequence operation on the `list` parameter yield the following proof obligation:

```
(forall list : seq of nat &
   len (list) > 1 => list <> [])
```

Guidance:

Inspection: When inspecting proof obligations for non-empty sequences one needs to check if the sequence potentially could be empty. Typically this is guranteed by either a pre-condition or a check in the context before the sequence application of the form "`list <> []`".

Test: In deriving test cases for such proof obligations one must seach for cases where the sequence is empty.

Proof: Proofs of this kind of proof obligations are typically fairly straightforward. In this case the proof would look like:

```
from
1 from list : seq of nat
1.1 from len (list) > 1
    infer list <> []                                  List-prop(1.1.h)
  infer len (list) > 1 => list <> []                          =>-I(1.1)
infer (forall list : seq of nat &
         len (list) > 1 => list <> [])                    forall-I(1)
```

3.5 Sequence Modification

Explanation: Whenever a sequence modification expression is used there will be a proof obligation that ensures that the indices in the modifying part of the expression are indices of the sequence to be modified.

Example:

```
functions

SeqMod: nat1 * real * seq of real -> seq of real
SeqMod(ind,val,list) ==
  list ++ {ind |-> val}
pre ind in set inds list
```

Here the application of the sequence modification operation on the `list` parameter yield the following proof obligation:

```
(forall ind : nat1, val : real, list : seq of real &
   ind in set inds (list) =>
      dom ({ind |-> val}) subset inds (list))
```

Guidance:

Inspection: When inspecting proof obligations for sequence modifications one needs to check that the index to be modified always will be an existing index in the sequence.

Typically this is guranteed by either a pre-condition or a check in the context before the sequence application of the form "`ind in set inds list`".

Test: In deriving test cases for such proof obligations one must seach for cases where the index does not already exist in the sequence.

Proof: Proofs of this kind of proof obligations are typically fairly straightforward. In this case the proof would look like:

```
from
1 from ind : nat1, val : real, list : seq of real
1.1 from ind in set inds (list)
1.1.1 {ind} = dom ({ind |-> val})                Dom-prop(1.1.h)
    infer dom ({ind |-> val}) subset
          inds (list)                    subset-prop(1.1.h,1.1.1)
  infer ind in set inds (list) =>
        dom ({ind |-> val}) subset inds (list)         =>-I(1.1)
infer (forall ind : nat1, val : real, list : seq of real &
          ind in set inds (list) =>
          dom ({ind |-> val}) subset inds (list))   forall-I(1)
```

3.6 Map Compatibility

Explanation: Some operators on mappings (i.e. `munion` and `merge`) require mappings to be compatible (i.e. whenever they have overlapping domain elements these map uniquely to the same range value). When these are present this proof obligation will be present.

Example:

```
functions

MUnion: (map nat to nat) * (map nat to nat) ->
         map nat to nat
MUnion(m1,m2) ==
  m1 munion m2
pre forall a in set dom m1 inter dom m2 & m1(a) = m2(a)
```

Here the use of the `munion` operation on the `m1` and `m2` parameters yields the following proof obligations:

```
(forall m1 : (map nat to nat), m2 : (map nat to nat) &
   (forall a in set dom (m1) inter dom (m2) &
```

```
      m1(a) = m2(a)) =>
      (forall id_1 in set dom (m1), id_2 in set dom (m2) &
         id_1 = id_2 => m1(id_1) = m2(id_2)))

(forall m1 : (map nat to nat), m2 : (map nat to nat) &
   (forall a in set dom (m1) inter dom (m2) &
      a in set dom (m1)))

(forall m1 : (map nat to nat), m2 : (map nat to nat) &
   (forall a in set dom (m1) inter dom (m2) &
      a in set dom (m2)))
```

where the last two are generated because of the two mapping applications in the pre-condition and both are trivially satisfied. The first one is more complicated because it needs to express that the mappings shall be compatible.

Guidance:

Inspection: When inspecting proof obligations for mapping compatibility one needs to check for overlapping domain elements mapping to the same value in both mappings. Typically this is guranteed by either not having any overlapping domain elements. This is usually expressed in a pre-condition or a check in the context before the merge of two mappings of the form "`dom m1 inter dom m2 =` $\{\}$".

Test: In deriving test cases for such proof obligations one must seach for cases where there are domain elements that overlap in a way such that they are not compatible.

Proof: Proofs of this kind of obligation can be complex. In this example the proof would look like:

```
from
1 from m1 : (map nat to nat), m2 : (map nat to nat)
1.1 from a in set dom (m1) inter dom (m2)
1.1.1 from m1(a) = m2(a)
1.1.2 from id_1 in set dom (m1), id_2 in set dom (m2)
1.1.2.1 from id_1 = id_2
        infer m1(id_1) = m2(id_2)       =-subs(1.1.1h,1.1.2.1h)
      infer id_1 = id_2 =>
            m1(id_1) = m2(id_2)                  =>-I(1.1.2.1)
      infer (forall id_1 in set dom (m1),
                    id_2 in set dom (m2) &
               id_1 = id_2 =>
               m1(id_1) = m2(id_2))            =forall-I(1.1.1)
  infer m1(a) = m2(a) =>
        (forall id_1 in set dom (m1), id_2 in set dom (m2) &
```

```
            id_1 = id_2 =>
            m1(id_1) = m2(id_2)))                    forall-I(1.1)
infer (forall m1 : (map nat to nat), m2 : (map nat to nat) &
          (forall a in set dom (m1) inter dom (m2) &
             m1(a) = m2(a)) =>
             (forall id_1 in set dom (m1),
                     id_2 in set dom (m2) &
                id_1 = id_2 =>
                m1(id_1) = m2(id_2)))                 forall-I(1)
```

3.7 Map Enumeration

Explanation: Whenever a mapping enumeration expression is used with more than one maplet, it must be ensured that if the same domain value is used multiple times that it always maps to the same range value.

Example:

```
functions

MapEnum: nat * bool * nat * bool -> map nat to bool
MapEnum(n1,b1,n2,b2) ==
  {n1 |-> b1, n2 |-> b2}
pre n1 <> n2
```

Here the enumeration of more than one maplet yield the following proof obligation:

```
(forall n1 : nat, b1 : bool, n2 : nat, b2 : bool &
  n1 <> n2 =>
  (forall m_3,m_4 in set {{n1 |-> b1},{n2 |-> b2}} &
    (forall id_5 in set dom (m_3), id_6 in set dom (m_4) &
       id_5 = id_6 => m_3(id_5) = m_4(id_6))))
```

Guidance:

Inspection: Inspecting mapping enumerations is typically not a problem because the domain elements are typically very simple expressions, normally constants.

Test: In deriving test cases for such proof obligations one must seach for cases where there are domain elements that overlap.

Proof: If constants are used these proofs are very straithforward.

3.8 Map Composition

Explanation: Whenever map composition is used there will be a proof obligation ensuring that the range element of one mapping will belong to the domain of the composed mapping.

Example:

```
functions

MCompo: (map nat to nat) * (map nat to nat) ->
         map nat to nat
MCompo(m1,m2) ==
  m1 comp m2
pre rng m2 subset dom m1
```

Here the composition of two mappings yield the following proof obligation:

```
(forall m1 : (map nat to nat), m2 : (map nat to nat) &
   rng (m2) subset dom (m1) =>
   rng (m2) subset dom (m1))
```

Guidance:

Inspection: Mapping composition is not often used in VDM models. However, when it is employed, the user must check that the range of the second mapping is a subset of the domain of the first mapping. In other formalisms where relations are used frequently such elements simply disappear from the resulting relation and thus they do not have a similar limitation to the one from this kind of proof obligation.

Test: In deriving test cases for such proof obligations one must seach for cases where the range of the second mapping is not contained in the domain of the first mapping.

Proof: The proof obligation produced here is trivially proved correct.

3.9 Map Iteration

Explanation: Whenever map iteration is used either the right hand side must be 0 (an identity mapping of the domain of the argument map) or the right hand side is equal to 1 (the mapping unchanged) or the range of the mapping must be a subset of the domain of the mapping.

Example:

```
functions
```

```
MIter: (map nat to nat) * nat -> map nat to nat
MIter(m1,n) ==
  m1 ** n
pre rng m1 subset dom m1
```

Here the composition of the mapping and the natural number yield the following proof obligation:

```
(forall m1 : (map nat to nat), n : nat &
   rng (m1) subset dom (m1) =>
   n = 0 or
   n = 1 or
   rng (m1) subset dom (m1))
```

As stated above if `n` is zero we would simply get an identity mapping over the domain of the argument mapping. If `n` is one then we just get the argument mapping. However, if `n` is larger than one we have the same limitation as presented for mapping composition above.

Guidance:

Inspection: Mapping iteration is not often used in VDM models, but when it is used, and one needs to inspect this kind of proof obligation, it is necessary to check that the range of the mapping is a subset of the domain of the mapping.

Test: In deriving test cases for such proof obligations one must seach for cases where the range of the mapping is not contained in the domain of the mapping.

Proof: In this example the pre-condition is slightly stronger than necessary, so it would be trivially proved.

3.10 Function Composition

Explanation: Whenever function composition is used there will be a proof obligation ensuring that the resulting value of one function will belong to the domain of the composed function (i.e. satisfy the pre-condition).

Example:

```
functions

FunComposition: (nat -> nat) * (nat -> nat) -> (nat -> nat)
FunComposition(fun1,fun2) ==
```

```
  fun1 comp fun2
pre forall x : nat & pre_(fun2,x) => pre_(fun1,fun2(x))
```

Here the composition of the two functions yield the following proof obligation:

```
(forall fun1 : (nat -> nat), fun2 : (nat -> nat) &
   (forall x : nat & pre_(fun2,x) =>
                     pre_(fun1,fun2(x)))
   =>
   (forall xx_13 : nat & pre_(fun2,xx_13) =>
                         pre_(fun1,fun2(xx_13))))
```

Guidance:

Inspection: Function composition is not often used in VDM models, but when they are used and one needs to inspect this kind of proof obligation one needs to check if the range of the second function is a subset of the domain of the first function. In other formalisms where relations are used frequently such elements simply disappear from the resulting relation and thus they do not have a similar limitation as the one from this kind of proof obligation.

Test: In deriving test cases for such proof obligations one must seach for cases where the range of the second function is not contained in the domain of the first function.

Proof: Because of the pre-condition the proof obligatio in the example here is trivially proved.

3.11 Function Iteration

Explanation: Whenever map iteration is used either the right hand side must be 0 (an identity mapping of the domain of the mapping) or the right hand side is equal to 1 (the mapping unchanged) or the range of the mapping must be a subset of the domain of the mapping.

Example:

```
functions

FunIter: (nat -> nat) * nat -> (nat -> nat)
FunIter(fun,n) ==
  fun ** n
pre forall x : nat & pre_(fun,x) => pre_(fun,fun(x))
```

Here the composition of the function and the natural number yield the following proof obligation:

```
(forall fun : (nat -> nat), n : nat &
   (forall x : nat & pre_(fun,x) =>
                     pre_(fun,fun(x)))
   =>
   n > 1 =>
   (forall xx_7 : nat & pre_(fun,xx_7) =>
                        pre_(fun,fun(xx_7))))
```

Guidance:

Inspection: Function iteration is not often used in VDM models, but when they are used, and one needs to inspect this kind of proof obligation, one needs to check if the range of the function is a subset of the domain of the function.

Test: In deriving test cases for such proof obligations one must seach for cases where the range of the function is not contained in the domain of the function.

Proof: In this example with the pre-condition it would be trivially proved.

3.12 Non-empty Set

Explanation: Whenever a distributed set intersection operator is used there is a proof obligation that the argument set is non-empty.

Example:

```
functions

DInter: set of set of nat -> set of nat
DInter(ss) ==
  dinter ss
```

Here the distributed intersection yields the following proof obligation:

```
(forall ss : set of set of nat & ss <> {})
```

Guidance:

Inspection: When inspecting this kind of proof obligation one simply needs to look for the possibility of the set of sets being empty. That should be very easy to see.

Test: In deriving test cases for this kind of proof obligation one simply needs to look for test cases that would yield an empty set as argument to a `dinter` operator.

Proof: In this case the pre-condition guarding `ss` from being empty is missing so it cannot be proved. If that was included it would be trial to prove it.

3.13 Non-zeroness

Explanation: Some operators on numbers require one of the operands to be different from 0 (e.g. division). In these cases this proof obligation is used.

Example:

```
functions

Div: real * real -> real
Div(a,b) ==
  a / b
```

Here a proof obligation will be generated:

```
forall a : real, b : real & b <> 0
```

expressing that division by 0 is not defined. Note that in case a pre-condition with "`b <> 0`" is added, the proof obligation will be skipped entirely by the integrity checker, because it can see that the predicate to be proven is directly present in the context.

Guidance:

Inspection: For this kind of proof obligations it tend to be very easy to simply inspect a VDM model and see whether the context will ensure that divide by zero does not occur.

Test: In order to gain further confidence one can try to select test cases that potentially could give rise to a divide by zero situation. In this case it would be easy for example with a test case like `Div(7,0)`.

Proof: In this case it is trivial to see that a proof would highlight the missing pre-condition right away.

3.14 Tuple Selection

Explanation: Whenever a tuple selection expression is used there will be a proof obligation ensuring that the tuple contains at least as many components as the selected field number. Note that since the selector is a constant number this proof obligation only needs to be generated in the presence of union types of tuples. Otherwise the plain type checker will be able to statically detect violations.

Example:

```
types

T1 = nat * bool;

T2 = nat * bool * real

functions

Select: T1 | T2 -> nat
Select(a) ==
  a.#1
```

With a VDM model like this the tuple select expression will give rise to two proof obligations:

```
(forall a : (T1 | T2) &
   is_(a,(nat * bool * real)) => 1 <= 3)

forall a : (T1 | T2) &
   is_(a,(nat * bool)) => 1 <= 2)
```

in order to indicate that the selector must be smaller than or equal to the length of a possible tuple type.

Guidance:

Inspection: In this case it is easy to see that both proof obligations are satisfied simply by inspection.

Test: With a more complicated context for a tuple select expression it could be worthwhile to try to make special tests that would select an index higher than the length of a particular possible tuple type.

Proof: In this case both proof obligations can be trivially proved.

4 Subtype Checking Proof Obligations

4.1 Subtype

Explanation: In many different situations one type is expected and the type of the value provided overlaps with the expected type. In these cases proof obligations are needed to ensure that the value provided is indeed a subtype of the expected type.

Example:

```
functions

SubType: real -> nat
SubType(r) ==
  r
pre is_nat(r)
```

Since the natural numbers form a subset of the real numbers in VDM this gives rise to the following proof obligation:

```
(forall r : real & is_nat(r) => is_nat(r))
```

Guidance:

Inspection: When inspecting subtype proof obligations one needs to see if it is ensured that only values of the correct subtype will be present. With the use of union types this proof obligation will appear frequently, and it is easy to spot omissions by plain inspection.

Test: In deriving tests for this kind of proof obligation one needs to look for test cases where values outside the desired subtype will reach the relevant point in the VDM model.

Proof: The proof of the proof obligation in the example here is trivial but that will not always be the case with larger union types. However, typically it is possible to use automatic proof support for this kind of proof obligations.

4.2 Invariants

Explanation: Whenever a type has an invariant then all expressions that must belong to this type gives rise to this kind of proof obligation.

Example:

```
types

Even = int
inv e == e mod 2 = 0

functions

Next: Even -> Even
```

```
Next(e) ==
  e + 2
```

With this model the invariant will give rise to the following proof obligation:

```
(forall e : Even & inv_Even(e + 2))
```

Guidance:

Inspection: When inspecting invariant proof obligations one needs to see if it is ensured that only values satisfying the invariant will be present.

Test: In deriving tests for this kind of proof obligation one needs to look for test cases that violates the invariant. Depending upon the complexity of the invariant, and the number of types used in the definition of the type that themselves have invariants, this can require human insight and thus in the general case, it can be hard to automate.

Proof: The proof of the the proof obligation for the example is trivial but depending upon the complexity of the invariants involved this can be a complex task.

4.3 Post Condition

Explanation: When functions or operations are defined explicitly and in addition has a post-condition then the body of the function/operation should terminate in a situation that does not violate the post-condition. This property can easily be formulated for functions by inserting the body expression as a parameter to a call of the implicitly produced post-condition predicate. However, since there is no way to automatically transform a statement to an expression + information about the side effects the actual predicate is left out for operations.

Example:

```
operations

OpPost: nat ==> nat
OpPost(n) ==
  (dcl sum : nat := 0;
   for i = 1 to n do
     sum := sum + i;
   return sum)
post 2 * RESULT = n * (n+1)
```

Here one would need to prove that given any natural number the `OpPost` operation will yield a result that satisfy the predicate in the post-condition. In the general case this would also need to take all possible states into account and the explicit body of the operation will have to live up to the state-changes documented in the post-condition predicate.

Guidance:

Inspection: When inspecting this kind of proof obligation one needs to consider whether the explicit body of the operation in all cases will be able to ensure the post-condition provided. Note that in the presence of concurrency this may be a bit tricky because permission predicates are then necessary to ensure secure access to critical regions.

Test: When tests needs to be derived in relation to this proof obligation one needs to consider situations where the post condition provided may be violated. From a practical perspective the testing of this should be carried out when switching on post-condition checking in the interpreter for VDMTools.

Proof: At present this cannot be proved because the proof theory for the explicit operations is not provided in complete form anywhere.

5 Satisfiability of Implicit Definitions Proof Obligations

5.1 Satisfiability

Explanation: For all functions and operations that are defined implicitly (with a post-condition) a satisfiability proof obligation is generated. This expresses that for all valid inputs from the input domains satisfying the pre-condition there must exist at least one resulting value satisfying the post-condition.

Example:

```
functions

SatFun(n : nat) r : nat
post n = 0 => r = 1 and
     n <> 0 => r = n * (n + 1)
```

With an implicit definition such as this a general satisfiability proof obligation will be produced:

```
(forall n : nat &
   (exists r : nat & post_SatFun(n, r)))
```

Guidance:

Inspection: When inspecting this kind of proof obligation one needs to consider whether it always will be possible to find a result that satisfy the post-condition. In particular in the presence of invariants for the resulting type this can be a complex task.

Test: Testing this kind of proof obligations cannot be done directly since no explicit function definition is present. Thus, one can either gain increased confidence in running the post-condition on its own. Alternatively one can derive an explicit version of the function and test that with respect to its post-condition.

Proof: Proving this kind of proof obligations is complex to automate, because that essentially requires deriving an algorithm for the different possible values that can be given as input for the function. For manual proofs it is recommended that meaningful names are given to different parts of a post-condition in the VDM model in case that it is large (spans over many lines). If this is done it is possible to fold the different parts according to the names of the different predicates and in this way it becomes more human readable. For the example here the proof would look like:

```
from
1 from n : nat
1.1 from n = 0
1.1.1 exists r : nat & n = 0 => r = 1           exists-I(1.1.h)
    infer (exists r : nat & n = 0 => r = 1 and
                            n <> 0 => r = n * (n + 1))
                                  and-true-neutral-lemma(1.1.1)
1.2 from n <> 0
1.2.1 exists r : nat & n <> 0 => r = n * (n + 1) exists-I(1.1.h)
    infer (exists r : nat & n = 0 => r = 1 and
                            n <> 0 => r = n * (n + 1))
                                  and-true-neutral-lemma(1.2.1)
1.3 (exists r : nat & n = 0 => r = = 1 and
                       n <> 0 => r = n * (n + 1))  case(1.1,1.2)
  infer (exists r : nat & post_SatFun(n, r))          fold(1.3)
infer (forall n : nat &
         (exists r : nat & post_SatFun(n, r)))      forall-I(1)
```

5.2 Exhaustive Matching in Cases Expression

Explanation: Whenever a cases expression is used without an `others` clause there is a proof obligation that for all possible case values must match at least one of the pattern matching alternatives.

Example:

```
functions

AllCases: nat -> nat
AllCases(n) ==
  cases {n}:
    {1}            -> 1,
    {e} union {1} -> e
  end
```

The cases expression inside `AllCases` yield the following proof obligation:

```
(forall n : nat & {n} = {(1)} or
   (exists {e} union {(1)} : set of nat &
      {n} = {e} union {(1)}))
```

Guidance:

Inspection: When inspecting this kind of proof obligation one needs to consider whether it always will be possible to match the value against at least of of the case alternatives. If there is any doubt that this may not always be the case it is recommended to introduce an `others` clause.

Test: In deriving tests for this proof obligation one may attempt to look for test cases that potentially may not match any of the case alternatives.

Proof: With large case expressions the proof obligation being generated becomes rather longwinded as well. This means that from a human readability point of view it becomes more complex but otherwise it is not too difficult to prove its correctness, assuming that the case alternatives are not too exotic. In most cases such case alternatives are used to match the different record alternatives for a union type where each element of the union is defined as a record type.

5.3 State Invariants

Explanation: Whenever a class or a superclass of that class has invariants over the instance variables it must be ensured that such invariants are always satisfied. Thus, whenever an assignment is made to an instance variable in such a class a proof obligation about respecting the invariant(s) is produced.

Example:

```
class A
```

```
instance variables

  a : nat := 8;
  inv a < 10;

operations

InstInvOp: () ==> nat
InstInvOp() ==
  (if a < 10
   then a := a + 1;
   return a)

end A
```

Here the assignment to an instance variable in a class that has an invariant as in class `A` a proof obligation stating that the state invariant must be preserved. No predicate expressing this is given.

Guidance:

Inspection: When inspecting proof obligations of this nature one needs to consider whether the context will ensure that the invariants for the instance variables being assigned to will always be satisfied. Note that multiple invariants can be defined over the instance variables and all of these shall be satisfied whenever an assignment is made to one of them.

Test: When tests needs to be derived in relation to this proof obligation one needs to consider situations where the invariant for the instance variables may be violated. From a practical perspective the testing of this should be carried out when switching on both the dynamic type checking option as well as the invariant checking option checking in the interpreter for VDMTools.

Proof: At present this cannot be proved because the proof theory for the explicit operations is not provided in complete form anywhere.

5.4 Exhaustive Function Patterns

Explanation: Whenever a pattern that will not always match potential values is used in the parameter list of a function or an operation, it is a proof obligation that this matching always will succeed by either a pre-condition to the function/operation or a subtype of the argument type.

Example:

```
types

A ::
  a : nat;

B ::
  a : nat;

functions

FunPat: A | B -> nat
FunPat(mk_A(elem)) ==
  elem
```

Here the pattern in the function parameter list yield a proof obligation as:

```
(forall xx_1 : (A | B) &
   (exists mk_A(elem) : (A | B) &
       xx_1 = mk_A(elem)))
```

Guidance:

Inspection: When inspecting this kind of proof obligation one needs to consider whether it always will be possible to match values from the input type(s) against the pattern(s) in the formal parameter declaration of the function/operation in question.

Test: Testing whether this kind of proof obligation is satisfied one needs to look for values from subtypes that will not match the parameter pattern(s).

Proof: For the example here it will not be possible to prove the proof obligation because it will not match values from the `B` type. In case this was taken care of in the precondition (or in the function signature) this would be easy to prove.

5.5 Non-emptiness of Let be such Binding

Explanation: Whenever a let-be-such-that expression or statement is used it is a proof obligation to ensure that there exists at least one element in the collection that the choice must be conducted from.

Example:

```
functions
```

```
Simple: () -> nat
Simple() ==
  let x in set {1,2}
  in
    x;

Choose: (nat -> bool) -> nat
Choose(p) ==
  let r : nat be st pre_(p,r) and p(r)
  in
    r
pre exists x : nat & pre_(p,x) and p(x) = true
```

With a VDM model like this the let-be expressions will give rise to two proof obligations:

```
(forall p : (nat -> bool) &
   (exists x : nat & pre_(p,x) and p(x) = true)
   =>
   (exists r : nat & pre_(p,r) and p(r)))

(exists x in set {1,2} & true)
```

derived from `Choose` and `Simple` respectively.

Guidance:

Inspection: When inspecting this kind of proof obligation one needs to consider whether it always will be possible to choose values from a set or type. This could fail if either the set is empty or the restricting predicate is false for all possible values from the set/type.

Test: When test cases are derived for this kind of proof obligations one need to look for ways of either getting an empty set or how the restricting predicate can yield false for all set/type values.

Proof: Both of the proof obligations provided in the given example are easy to prove. However in general it can be difficult to prove this kind of proof obligations automatically when complex predicates are used.

5.6 Non-emptiness of Binding

Explanation: Whenever a value is bound to a complex pattern it is a proof obligation to ensure that it is possible to match the value against the pattern.

Example:

```
values

[a] ^ l2 = if 6 > 8 then [] else [7,5,2]
```

This value definition gives rise to the following proof obligation:

```
(exists [a] ^ l2 : seq of nat1 &
   [a] ^ l2 = (if 6 > 8 then [] else [7,5,2]))
```

Guidance:

Inspection: When inspecting a VDM model that gives rise to this kind of proof obligation one needs to look for values that cannot match the complex pattern.

Test: When tests needs to be derived in order to gain confidence in the correctness of this kind of proof obligations one needs to try to find possibilities for test values not matching a complex pattern.

Proof: In the example provided here the proof obligation can be proved easily but if the test predicate could yield true it would be impossible to prove it, because it would be invalid (i.e. the pattern cannot match an empty sequence).

5.7 Unique Existence Binding

Explanation: Whenever an iota expression or an exists unique expression is used there is a proof obligation to ensure that exactly one element satisfies the iota predicate.

Example:

```
functions

Unique: set of nat -> nat
Unique(s) ==
  iota e in set s & forall e2 in set s & e <= e2
pre s <> {}
```

The iota expression here will give rise to this proof obligation:

```
(forall s : set of nat &
   s <> {} =>
   (exists1 e in set s &
      (forall e2 in set s & e <= e2)))
```

Guidance:

Inspection: When inspecting a VDM model that gives rise to this kind of proof obligation one needs to look for ways in which no elements or more than one element satisfy a given predicate.

Test: When tests needs to be derived in order to gain confidence in the correctness of this kind of proof obligations one needs to try to find ways in which none or more than one value will satisfy the predicate.

Proof: Proof obligations of this kind can be difficult to prove. In this case the proof looks like:

```
from
1 from  s : set of nat
1.1 from s <> {}
    infer (exists1 e in set s &
              (forall e2 in set s & e <= e2))  total-order(1.1h)
  infer s <> {} =>
         (exists1 e in set s &
            (forall e2 in set s & e <= e2))            =>-I(1.1)
infer (forall s : set of nat &
          s <> {} =>
          (exists1 e in set s &
             (forall e2 in set s & e <= e2)))       forall-I(1)
```

5.8 Finiteness of Set

Explanation: When set comprehensions are defined using a type binding there will be a proof obligation ensuring that the resulting set will be finite.

There are many ways of establishing that a set is finite. The obligation that is generated for a mapping comprehension takes the following approach. For a set comprehension

```
{a | a : A & P(a)}
```

It is necessary to show that there is only a finite number of values of type A that satisfy the predicate P, hence there is only a finite number of values in the set. This is done by showing that the values can be counted, i.e. a mapping can be constructed from the natural numbers on to all those values of type A satisfying P, i.e.

```
exists m: map nat to A &
   forall a:A & P(a) => exists in set dom m & m(i)=a
```

Since mappings in VDM are finite, the existence of a counting mapping means that the set is finite.

Example:

```
functions

FinSet: nat -> set of nat
FinSet(max) ==
  {x | x :nat & x <= max}
```

This would give rise of the following proof obligation:

```
(forall max : nat &
   (exists f_1 : map nat to nat &
      (forall x : nat & x <= max
       =>
      (exists i_2 in set dom (f_1) & f_1(i_2) = x))))
```

Guidance:

Inspection: For this kind of proof obligation one needs to look for ensuring that either type type used only contains a finite collection of elements or the predicate limits the resulting set so it will indeed only contain a finite number of members.

Test: This cannot be tested at all because type bindings cannot be interpreted.

Proof: The proof of finiteness conjectures can be challenging because o fhte need to invent a suitable counting mapping. In the example shown below, the counting mapping is very simple. Each number `i` is simply mapped onto `i` up to the maximum value `max`. The counting mapping is thus:

```
{i |-> i | i:nat & i in set {0,...,max}}
```

```
from
1 from max : nat
1.1 {0,...,max} : set of nat                    interval-form(1.h1)
1.2 {i |-> i | i:nat & i in set {0,...,max}} : map nat to nat
                                        Lemma iden-map-form (1.1)
1.3 from x:nat
1.3.1 from x <= max
1.3.1.1    x in set {1,...,max}
```

```
                                        interval-memb(1.3.h1,1.3.1.h1)
1.3.1.2    dom {i |-> i | i in set {0,...,max}} = {1,...,max}
                                      ident-map-comp-dom-form(1.1)
1.3.1.3    x in set dom {i |-> i | i in set {0,...,max}}
                                 =-subs-left(b)(1.1,1.3.1.2,1.3.1.1)
1.3.1.4     {i |-> i | i in set {0,...,max}}(x) = x
                                       ident-apply(1.2,1.3.1.3)
      infer exists i_2 in set dom {i |-> i
                                      | i in set {0,...,max}} &
              {i |-> i | i in set {0,...,max}}(i_2) = x
                                       exists-I(1.3.h1,1.3.1.4)
    infer x <= max
          =>
          (exists i_2 in set dom (f_1) & f_1(i_2) = x))
                                                    =>-I(1.3.1)
1.4 forall x : nat & x <= max
                     =>
                     (exists i_2 in set dom (f_1) &
                         f_1(i_2) = x))
                                                 forall-I(1.3)
  infer (exists f_1 : map nat to nat &
           (forall x : nat & x <= max
            =>
           (exists i_2 in set dom (f_1) & f_1(i_2) = x)))
                                               exists-I(1.2,1.4)
infer (forall max : nat &
         (exists f_1 : map nat to nat &
            (forall x : nat & x <= max
             =>
            (exists i_2 in set dom (f_1) & f_1(i_2) = x))))
                                                 forall-I(1)
```

5.9 Finiteness of Map

Explanation: When mapping comprehensions are defined using a type binding there will be a proof obligation ensuring that the resulting mapping will be finite. This proof obligation is approached in the same way as that for finiteness of sets; the mapping being analysed is treated as a set of maplets. The proof obligation requires us to show that we can construct a counting mapping for the maplets in the mapping being checked. For a mapping M of type `map A to B`, the type of the counting mapping `c` is therefore:

```
map nat to (map A to B)
```

If m is a mapping of the form:

```
{f(x) |-> g(x) | x:A & P(x) }
```

we have to show that there exists a counting mapping c such that

```
forall x:A & P(x) =>
   exists i in set dom c & c(i) = {f(x) |-> g(x)}
```

Example: The following function constructs a mapping taking a number to its successor:

```
functions

FinMap: nat -> map nat to nat
FinMap(max) ==
  {x |-> x + 1 | x :nat & x <= max}
```

The generated proof obligation requires the existence of a suitable counting mapping f_1:

```
(forall max : nat &
   (exists f_1 : map nat to map nat to nat1 &
      (forall x : nat & x <= max
       =>
      (exists i_2 in set dom (f_1) &
         f_1(i_2) = {x |-> x + 1}))))
```

Guidance:

Inspection: For this kind of proof obligation one needs to look for ensuring that either type type used only contains a finite collection of elements or the predicate limits the resulting set so it will indeed only contain a finite number of members.

Test: This cannot be tested at all because type bindings cannot be interpreted.

Proof: Here the proof structure would be similar to that for the finiteness of a set. A suitable counting mapping for this example would would be the following:

```
{ i |-> {i |-> i+1} | i: nat & i in set {0,...,max}}
```

The structure of a possible proof using this counting mapping is outlined below, where we only show the key lines in the proof.

```
from
1 from max : nat
1.1 {0,...,max} : set of nat                    interval-form(1.h1)
1.2 {i |-> {i |-> i+1} | i:nat & i in set {0,...,max}} :
    map nat to nat1                                        Lemma (1.1)
1.3 from x:nat
1.3.1 from x <= max
...
1.3.1.4 {i |-> {i |-> i+1} | i in set {0,...,max}}(x) =
        {x |-> x+1}
      infer exists i_2 in set
                dom {i |-> i | i in set {0,...,max}} &
               {i |-> {i |-> i+1}
               | i in set {0,...,max}}(i_2) = {x |-> x+1}
                                          exists-I(1.3.h1,1.3.1.4)
    infer x <= max
          =>
          (exists i_2 in set dom (f_1) &
              f_1(i_2) = {x |-> x+1}))                  =>-I(1.3.1)
1.4 forall x : nat & x <= max
                   =>
                   (exists i_2 in set dom (f_1) &
                       f_1(i_2) = {x |-> x+1}))
                                                      forall-I(1.3)
  infer (exists f_1 : map nat to map nat to nat1 &
           (forall x : nat & x <= max
            =>
           (exists i_2 in set dom (f_1) &
               f_1(i_2) = {x |-> x+1}))))
                                                 exists-I(1.2,1.4)
infer (forall max : nat &
         (exists f_1 : map nat to map nat to nat1 &
            (forall x : nat & x <= max
             =>
            (exists i_2 in set dom (f_1) &
               f_1(i_2) = {x |-> x + 1}))))          forall-I(1)
```

6 Termination Proof Obligations

6.1 Terminating While Loop

Explanation: Whenever a while loop is used inside the body of an operation there is the potential that this loop may not be terminating. Thus, in all such cases this proof obligation is used, although it is typically not needed in reactive systems where non-terminating threads are acceptable.

Example:

```
class A

operations

WhileOp: nat ==> nat
WhileOp(n) ==
  (dcl local : nat := n,
       res   : nat := 1;
   while local > 0 do
    (local := local - 1;
     res   := local * res) ;
   return res)

end A
```

This operation gives rise to two proof obligations. The first one is a subtype one (that `local` will still be a natural number when one is subtracted from it). Note that since it is inside an explicit operation the context is not provided explicitly:

```
local - 1 > 0
```

The other one only state that one needs to prove that the while loop will terminate.

```
while loop terminating
```

Guidance:

Inspection: When inspecting a while loop for termination one needs to check that the the code in the body of the loop "gets closer and closer" to making the testing predicate false so the loop will be left. It is also essential that the loop will reach this situation in a finite number of steps.

Test: When tests needs to be derived in relation to this proof obligation one needs to consider situations where infinite loops will appear.

Proof: At present this cannot be proved because the proof theory for the explicit operations is not provided in complete form anywhere.

7 Proof of Proof Obligations

The first tool support for proof of VDM, mural, was developed in the UK at the beginning of the 1990s [8]. Mural was a generic proof support environmant that was instantiated with a range of proof theories for various kinds of proof task. Its VDM instantiation started form the informal rules given in [13] and evolved into the more detailed proof theory given in [7]. Mural was *not* an authomatic proof system, but an assistant for human-guided proofs and, as such it had a comparatively advanced graphical user interface for interactive proof construction. The mural tool is no longer in existence but inspiration from it was later on used in different interactive proof tools.

In the mid-1990s, several steps were made towards the development of automatic proof support for VDMTools, using translations of VDM models to Isabelle, PVS and HOL [1, 3, 2, 4]. This inspired contributions to the European project PROSPER [9, 5]. Here the proof support was based on a hol98 kernel which was improved during the project. The PROSPER version of the VDM-SL Toolbox supported both automatic proof construction using HOL "behind the scenes" as well as interactive proofs using a GUI in Java. This feature was, for example, used by RTRI (the Japanese Railway Technical Research Institute) [20, 21, 23, 22]. Sadly, the sources for the PROSPER extensions of VDMTools were lost in the transfer from IFAD to CSK.

Some years later, a new attempt at providing automating proof support for VDM was made using a more recent version of HOL [24]. This new work is seen as a part of the Overture project [19] and it is reconstructing automatic proof support from PROSPER by an automatic mapping from VDM++ models and proof obligations to HOL4 (the newest version of HOL).

It is, however, worth noting that the nature of automatic theorem provers is to move a proof into very small steps based upon basic axioms for a language. Such small steps are typically combined into higher level tactics that can be used by a user. However, the level of deep understanding that the user gets from a automatic proof is quite limited. This is where the manual proof style shown in this document has an advantage. The style, known as "natural deduction", is intended for human readers. It is easy to follow and check the logical steps of the deduction process. For a machine, the number of small steps is not a problem but if one was to gain understanding from all these small steps the complexity of realistic proofs at that level would be way to complex for a human being to gain understanding from. So when an automatic prover needs assistance to complete a proof the user needs to understand how the different tactics built into the prover work, and this is the reason why a higher level of skill is required to conduct automatic proofs than to carry out tests of a formal abstract model of a system.

Standard VDM proof rules are provided for a large part of the VDM-SL language in [7]. Proof rules for the more advanced areas of VDM-SL in explored in [18]. No coherent and complete proof system that cover the entire VDM++ notation has been described anywhere although work towards this has been underway [14, 17, 16, 15]. Thus, more work is needed to be able to prove all proof obligations generated for VDM++. Initially it is advisable to do automatic proof for the subset of VDM++ supported by Overture. For the parts that are not supported by Overture the only opportunity is to carry out manual proofs. In case the parts are covered by some of the proof rules already published these can be used. However in case the part used is not covered by any of proof rules developed already it is necessary to develop new proof rules.

References

[1] AGERHOLM, S. Translating specifications in VDM-SL to PVS. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)* (1996), J. von Wright, J. Grundy, and J. Harrison, Eds., vol. 1125 of *Lecture Notes of Computer Science*, Springer-Verlag.

[2] AGERHOLM, S., AND FROST, J. An Isabelle-based theorem prover for VDM-SL. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)* (August 1997), LNCS, Springer-Verlag. Also available as technical report IT-TR: 1997-009 from the Department of Information Technology at the Technical University of Denmark.

[3] AGERHOLM, S., AND FROST, J. Towards an integrated CASE and theorem proving tool for VDM-SL. In *FME'97* (September 1997), LNCS, Springer-Verlag.

[4] AGERHOLM, S., AND FROST, J. Towards an integrated case and theorem proving tool for vdm-sl. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)* (September 1997), J. Fitzgerald, C. B. Jones, and P. Lucas, Eds., vol. 1313 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 278–297. ISBN 3-540-63533-5.

[5] AGERHOLM, S., AND SUNESEN, K. Reasoning about VDM-SL Proof Obligations in HOL. Tech. rep., IFAD, 1999.

[6] BERNHARD K. AICHERNIG AND PETER GORM LARSEN. A Proof Obligation Generator for VDM-SL. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)* (September 1997), J. S. Fitzgerald, C. B. Jones, and P. Lucas, Eds., vol. 1313 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 338–357. ISBN 3-540-63533-5.

[7] BICARREGUI, J., FITZGERALD, J., LINDSAY, P., MOORE, R., AND RITCHIE, B. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[8] CLIFF JONES, KEVIN JONES, P. L., AND MOORE, R., Eds. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X.

[9] DENNIS, L. A., COLLINS, G., NORRISH, M., BOULTON, R., SLIND, K., ROBINSON, G., GORDON, M., AND MELHAM, T. The PROSPER Toolkit. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Germany, March/April 2000), Springer-Verlag, Lecture Notes in Computer Science volume 1785.

[10] GORDON, M. Hol: A proof generating system for higher-order logic. In *VLSI Specification, Verification, and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, Eds. Kluwer Academic Publishers, 1987.

[11] *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[12] GROUP, T. V. T. VDM++ Toolbox User Manual. Tech. rep., CSK Systems, January 2008.

[13] JONES, C. B. *Systematic Software Development Using VDM*, second ed. Prentice-Hall International, Englewood Cliffs, New Jersey, 1990. ISBN 0-13-880733-7.

[14] KENT, S., AND MOORE, R. An axiomatic semantics for vdm++: Oo aspects, 1993.

[15] K.LANO, S. Refinement, subtyping and subclassing in vdm++. In *2nd Theory and Formal Methods Workshop, Cambridge* (1995), I. C.Hankin, Ed., IC Press.

[16] LANO, K. Expressing the semantics of vdm++ in rtl.

[17] LANO, K. Reasoning techniques in vdm++.

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

[19] OVERTURE-CORE-TEAM. Overture Web site. http://www.overturetool.org, 2007.

[20] TERADA, N. Formal Integrity Analysis of Digital ATC Database. In *Proceedings of WCRR2001 (World Congress on Railway Research)* (2001).

[21] TERADA, N. Integrity Analysis of Digital ATC Database with Automatic Proofs. In *VDM Workshop 3* (Copenhagen, Denmark, July 2002), J. F. J. Bicarregui and P. Larsen, Eds., Part of the FME 2002 conference.

[22] TERADA, N. Application of formal methods to automatic train control systems. In *Proceedings of Symposium on Formal Methods for Railway Operation and Control Systems (FORMS 2003)* (2003).

[23] TERADA, N., AND FUKUDA, M. Application of Formal Methods to the Railway Signaling Systems. *Quarterly Report of RTRI 43*, 4 (2002), 169–174.

[24] VERMOLEN, S. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department, August 2007.