Towards Operation Proof Obligation Generation for VDM

Nick Battle[1] [0009-0001-1523-4964] and Peter Gorm Larsen[1] [0000-0002-4589-1500]

Department of Electrical and Computer Engineering, Aarhus University, Denmark,
nick.battle@gmail.com, pgl@ece.au.dk

Abstract. All formalisms have the ability to ensure that their models are internally consistent. Potential inconsistencies are generally highlighted by assertions called *proof obligations*, and the generation of these obligations is an important role of the tools that support the method. This capability has been available for VDM tools for many years. However, support for obligation generation for explicit operation bodies has always been limited. This work describes the current state of work to address this, showing the capabilities so far and highlighting the work remaining.

1 Introduction

Tools for working with VDM-SL specifications have been able to produce *proof obligations*[1] for many years. A proof obligation (PO) is a small VDM-SL boolean expression, which should always be true – it should be a tautology. If all obligations can be shown to be true, then the internal consistency of the specification has been verified, e.g., showing that a calculation never divides by zero.

Tool support for proof obligation generation (POG) from VDM *functions* is mature, and recent work even allows POs to be checked and discharged [2,5]. However, support for the generation of obligations from VDM *explicit operations* has always been limited. This is partly due to the fact that proof rules for statements were never fully defined [8]. In addition, the depth of analysis required is greater, since operations have a more complex control flow, possibly involving loops, exceptions and side effects. In this paper, we describe the progress towards generating obligations for explicitly declared VDM operations. This is accomplished with a combination of static analysis and annotations.

Section 2 looks at the background to POG in VDMTools [9]. Afterwards, Section 3 gives an overview of the POG in VDMJ [2,7]. Then Section 4 describes the new POG for operations and Section 5 looks at the results achieved so far. Finally, Section 6 looks at related work and Section 7 considers future work.

2 History

The initial POG for VDM was introduced in a Masters thesis in 1997 [1]. This was incorporated in VDMTools at IFAD, and subsequently the ideas behind it were trans-

[1] VDMTools [9] calls them "integrity properties".

ferred to the Overture VDM tool in 2010 [10]. The POG was then extended with better support for recursive definitions and measures for functions later in 2010 [13].

However, none of this research attempted to generate obligations for statements present inside explicitly defined operations. Originally, the VDM method was focused on refinement for operation validation [6]. This left explicit operation obligations as rudimentary, giving the basic conditions to be proved, but lacking the context needed to prove them.

The current work describes progress in providing the missing context to operation obligations, in order to allow them to be discharged.

3 POG Overview

Proof obligations are generated from VDM specifications that have passed a static type check. They are boolean expressions that express an obligation on the implementer to verify that some condition always holds. For example, a type definition with an invariant creates an obligation to make sure that at least one value passes the invariant. Similarly, within the bodies of functions and operations, obligations arise to make sure that partial operators are not applied outside their domain. For example, keys must exist in the domain of maps to which they are applied, and values must be within the size of sequences that they index.

The general form of obligations within function or operation bodies has three parts:

- An outermost *forall* quantifies over all possible arguments that can be passed. This includes the precondition, if any.
- A series of *context* steps describe the path that the evaluation must take to get to the obligation point.
- Finally, the fundamental obligation itself.

So in plain language, the obligation in the specification below would read "For all possible key arguments to *lookup*, if the value is non-zero and it is a legal key, then the key exists in the domain of the table."

```
functions
    lookup(key:nat) r:set of nat ==
        if key <> 0 and isValid(key)
        then table(key)   -- Obligation here
        else {};

--Proof Obligation 1: (Unproved)
lookup: map apply obligation at line 4:10
(forall key:nat &
  (((key <> 0) and isValid(key)) =>
    key in set dom table))
```

4 Operation POG

4.1 Representing State in POs

One significant difference between POs from operations and functions is that the former also have access to any state data that is defined in the module. So, whereas the top-level quantifier of a function PO can consider all possible arguments passed, the quantifier of an operation has to consider all possible arguments and module states. This is represented by the *Sigma* type of the model[2]. For example:

```
state Sigma of
    sv : nat
    xv : nat
end;

operations
    op(a:nat) r:real ==
        return 1/(sv - a)   -- Obligation here
    pre sv > a;

--Proof Obligation 1: (Unproved)
op: non-zero obligation at line 8:22
(forall a:nat, mk_Sigma(sv, xv):Sigma &
  pre_op(a, mk_Sigma(sv, xv)) =>
    (sv - a) <> 0)
```

Note that the **forall** quantifer includes the parameter "`a`", but it also defines variables "`sv`" and "`xv`" for the state variables, using the Sigma type and a record pattern to match the variable names.

The `pre_op` function is subsequently called to eliminate argument/state combinations that do not match the operation's precondition. This uses a Sigma record to create a specific state vector from the variables.

This approach to state representation means that the context and the base obligation can reason about state variables by name directly, as they do in the specification itself. In VDM-SL an operation can only directly reason about the state in its own module. The position with state representation for VDM++ and VDM-RT is much more complex and is discussed in Section 7.

4.2 Assignments to State

An operation can make assignments to state variables, as well as simply reasoning about them. Therefore, the logic in an obligation that is *after* a state assignment must include this change. This is achieved by using **let** expressions in the obligation context to *hide* the variables bound by the outer quantifier. Thereafter, referring to "`sv`" (say) will refer to the updated definition, rather than the outer definition. For example:

[2] By convention, the state of a model is called "Sigma". But in general, we mean the name of the `state` definition.

```
operations
    op(a:nat) r:real ==
    (
        sv := sv + 1;
        xv := xv + sv;
        return a + 1/xv   -- Obligation here
    );

--Proof Obligation 1: (Unproved)
op: non-zero obligation at line 6:17
(forall a:nat, mk_Sigma(sv, xv):Sigma &
  (let sv : nat = (sv + 1) in
    (let xv : nat = (xv + sv) in
      xv <> 0)))
```

This clearly works for simple variable definitions, but it also works for more complex state designators, with careful use of "++" and **mu** operators to create new state variable values. Note that even complex assignments only ever update one state variable (albeit with a complex new value). For example:

```
state Sigma of
    sv : seq of R
end

types
  R ::
    size : real;

op(z:nat) r:real ==
(
  sv(1).size := 456;
  return 1/len sv   -- Obligation here
);

--Proof Obligation 2: (Unproved)
op: non-zero obligation at line 12:11
(forall z:nat, mk_Sigma(sv):Sigma &
  (let sv : seq of R = sv ++ {1 |-> mu(sv(1), size |-> 456)} in
    (len sv) <> 0))
```

Here, the assignment to "`sv(1).size`" only updates the "`sv`" variable, but it does so by indexing into the sequence value and then updating a record field at that position. This update is included in the obligation, using "++" to update the sequence with a record value that uses **mu** to modify the size field. This approach extends to arbitrary combinations of maps, sequences and records.

4.3 Local State Handling

In addition to module state, operations can define local state using assignment definitions (**dcl**) within block statements. In most respects, these are treated the same way as module state by the POG, but special care has to be taken with scoping if those state variables have been used to update module state. For example:

```
op(z:nat) r:real ==
(
    dcl a:nat := 0;
    a := a + 1;

    ( dcl b:nat := a + 1;
      sv := b );   -- NOTE: updates sv using b

    ( dcl c:nat := a + 2;
      c := c + 1 );

    return 1/sv   --PO depends on a, b but not c
);

--Proof Obligation 1: (Unproved)
op: non-zero obligation at line 12:13
(forall z:nat, mk_Sigma(sv):Sigma &
  (let a : nat = 0 in
    (let a : nat = (a + 1) in
      (let b : nat = (a + 1) in
        (let sv : nat = b in
          sv <> 0)))))
```

Here, three **dcl** definitions are created, two of them in sub-blocks. But note that the middle block uses its local "b" value to update "sv". This means that the "b" value must be retained at the end of its block, so that can appear in the obligation to allow the assignment to "sv" to be well defined. In contrast, the "c" value is not used to update anything and is not included in the PO.

There is considerable scope for confusion here if the specification either hides variable values or re-defines the same name in different blocks. This is discussed in Section 7.

4.4 Alternate Paths

So far, the examples shown have only included a single control flow path from the start of the operation to the obligation point. In obligations for functions, there is only ever one path. However, for explicitly defined operations, there can be arbitrarily many paths to reach a given statement. In these cases, the separate possible paths are treated independently, generating multiple obligations for a single location.

The example below is a simple illustration. Notice that there are three obligations, corresponding to the three possible paths to reach the obligation location. Each obligation includes context that covers its particular path.

```
op(z:nat) r:real ==
(
  if z > 10 then
    if z > 100 then
      sv := 999
    else
      sv := 888
  else
    sv := z+1;

  return 1/sv     --PO#1,2,3
);

--Proof Obligation 1: (Unproved)
op: non-zero obligation at line 11:11
(forall z:nat, mk_Sigma(sv):Sigma &
  ((z > 10) =>
    ((z > 100) =>
      (let sv : nat = 999 in
        sv <> 0))))

--Proof Obligation 2: (Unproved)
op: non-zero obligation at line 11:11
(forall z:nat, mk_Sigma(sv):Sigma &
  ((z > 10) =>
    (not (z > 100) =>
      (let sv : nat = 888 in
        sv <> 0))))

--Proof Obligation 3: (Unproved)
op: non-zero obligation at line 11:11
(forall z:nat, mk_Sigma(sv):Sigma &
  (not (z > 10) =>
    (let sv : nat = (z + 1) in
      sv <> 0)))
```

4.5 Ambiguous States

The examples given above consider direct updates to state variables by statements in an operation. However, an operation can make calls to other operations, which in turn may update the state of the system, affecting statements after the call. In general, we cannot know what an operation call will do without analyzing the entire specification. So the POG uses the concept of *ambiguous states*, which record the names of variables whose actual state is not known.

A simple operation call is assumed to put every module state variable into an ambiguous state (local **dcl** state is not affected). If the operation has an **ext wr** clause that identifies named variables, then just those are marked as ambiguous, whereas if the operation is **pure** is it known not to update anything. The return value from an operation is generally unknown, so any variable assigned a value using the result from an operation call is marked as ambiguous.

If any variables in the context of an obligation are ambiguous, the PO is produced but marked as "Unchecked".

Note that ambiguous variables can subsequently be disambiguated by being assigned with values that are calculated from unambiguous values.

4.6 Atomic Updates

The **atomic** statement in operations has to have special handling because of its semantics. These statements define a collection of assignments which all happen *atomically* – that is, the effect of each assignment is not visible to the others. In practice, this is equivalent to calculating the RHS values for them all, then making the assignments with any state invariant disabled, and finally checking the invariant after all the assignments have completed. This process is therefore reflected in the obligation context for atomic statements. For example:

```
state Sigma of
    sv : real
    xv : real
inv s == s.sv <> s.xv
end

op(a:nat) ==
    atomic ( sv := xv; xv := sv );  -- Obligation here

--Proof Obligation 1: (Unproved)
(forall a:nat, mk_Sigma(sv, xv):Sigma &
  (let $atomic1 : real = xv in
    (let $atomic2 : real = sv in
      (let sv : real = $atomic1 in
        (let xv : real = $atomic2 in
          let s = mk_Sigma!(sv, xv) in ((s.sv) <> (s.xv)))))))
```

Note the use of the maximal operator in the final `mk_Sigma!`. This allows the state record to be created without the invariant, for the purpose of explicitly checking the invariant on the resulting value.

4.7 Post-conditions

Post-conditions in operations are able to reason about the "old" value of state variables, using a tilde syntax, like `var~`. This poses a problem for the obligation generator, partly because the original value of variables must be represented, and partly because the tilde syntax can only be used legally in a post-condition clause.

Old variables and return values are therefore handled as follows:

```
state Sigma of
    sv : nat
end

op(z:nat) r:real ==
(
  sv := z;
  sv := sv * 2;
  return sv + 1
)
post r > 0 and sv > sv~;

-- Proof Obligation 1: (Unproved)
(forall z:nat, mk_Sigma(sv):Sigma &
  (let sv$ = sv in                <-- Old state
    (let sv : nat = z in
      (let sv : nat = (sv * 2) in
        (let r = (sv + 1) in    <-- return sets "r"
          ((r > 0) and (sv > sv$)))))))
```

If an operation has a post-condition which reasons about the original value of state variables, these are captured at the start of the obligation, using a substitution for "$" in place of the tilde. Return statements create a context that defines the returning variable, or "**`RESULT`**" if none is defined. This can then be used in the postcondition assertion at the end.

4.8 Loop Invariants

The analysis of loop statements requires a *loop invariant* to be specified for each loop. Proof obligations can then be generated for points before and during the loop, and the invariant also appears in the context of obligations after the loop.

Loop invariants are defined using a `@LoopInvariant` annotation [3], which takes a single expression as an argument. The expression is obliged to reason about the relationship between all of the variables that the loop modifies. It is then used in obligations.

In the example below, we use *inline functions* within the obligation to allow the loop to be represented as a recursive function. The `body` performs the changes to the updatable variables in the loop; the `invariant` is the expression from the annotation; and the `loop` checks the while condition, then checks the invariant before and after the updates, before recursing to process the next loop iteration.

This method of creating loop obligations should be regarded as experimental and the tools do not currently produce the PO shown[3]. However, we believe that this is a possible route to handling loop obligations.

[3] Currently, valid obligations are produced for the point before the loop, and at the start and end of the first loop iteration.

```
state Sigma of
  s : seq of int
end

op(data:seq of int) ==
(
  dcl count : int := 0;
  s := data;

  -- @LoopInvariant(count + len s = len data);
  while s <> [] do
  (
    s := tl s;
    count := count + 1
  )

  -- Here, invariant holds and s = []
  ...
);

--Proof Obligation 1: (Unproved)
(forall data:seq of int, mk_Sigma(s):Sigma &
    let body: seq of int * int +> seq of int * int
        body(s, count) ==
            (let s : seq of int = (tl s) in
                (let count : int = (count + 1) in
                    mk_(s, count))),

        invariant: seq of int * int * seq of int +> bool
        invariant(s, count, data) ==
            ((count + (len s)) = (len data)),

        loop: seq of int * int * seq of int +> bool
        loop(s, count, data) ==
            s <> [] =>
                invariant(s, count, data) and
                let mk_(s, count) = body(s, count) in
                    invariant(s, count, data)
                    and loop(s, count, data)
    in
        (let count : int = 0 in
            (let s : seq of int = data in
                (loop(s, count, data)))))
```

The example above is a **while** loop, but a similar approach works for **for** loops if the `loop` function is passed the variable(s) to modify and test as the loop progresses.

If the annotation is missing from a loop[4], the POG cannot produce the corresponding obligations. In this case, the variables updated by the loop are marked as ambiguous and any obligations resulting from the body of the loop are marked as "Unchecked".

Note that the type-checker verifies that the `@LoopInvariant` expression reasons about all the modified variables in the loop. The invariant is also checked by the interpreter runtime, so ad-hoc testing or combinatorial testing may also pick up failures.

4.9 Obligation Correctness

Obligations are intended for discharge by the proof process, but the development of the obligation generator *itself* must be validated to ensure that the POs produced are sufficient and correctly describe the obligations placed on the specifier.

This process is hampered by the lack of a proof theory for explicit operations in [8]. But the general approach we adopt is to break down the flow of control in operations to give a set of possible paths. Relevant actions that the operation makes along a path are represented by clauses in the PO context, as described earlier. If the sequential composition of these clauses always results in a semantic equivalence to the operation itself, then the complete obligation will correctly represent the state of the system on that one path.

This approach seems promising for the cases considered so far, as demonstrated in this section. However, it may not be sufficient for more complex cases (see Section 7). The authors hope that the technique of using in-line functions (see 4.8) can be expanded to deal with more complex cases.

5 Preliminary Evaluation

The new POG has been tested on the VDM-SL example suite that comes with Overture [4]. The 50 example specifications generate 5500+ obligations. These are still being checked, but the POG does not crash, and tools such as QuickCheck [12] can attempt to discharge the obligations generated.

Before the changes described in this work, about 21% of the obligations generated by the example suite were marked as "Unchecked"; with the new POG, about 9.6% of them remain "Unchecked". This indicates that, although the techniques deployed so far do not tackle complex cases, they still result in a significant increase in the number of obligations that can be checked by proof tools.

With the new POG, around 14% (811) of the obligations are *failed* by QuickCheck. Most failures are cases where operations are missing obvious constraints. But in at least one case, the new POG caught a very subtle error in the "`TicTacToe`" model, where the "move" operation did not have a sufficiently strong precondition. These problems were previously missed because the corresponding obligations were incomplete and marked "Unchecked".

[4] Obviously they are missing from historical specifications.

6 Related Work

The process of calculating proof obligations for imperative VDM operations is similar to other formal languages, most notably *Dafny* [11]. Some of the work here was guided by Dafny's *Verification Conditions* (VCs).

7 Future Work

The work outlined above is not complete. The following areas or issues remain to be completed or solved:

– Loop invariants are the right direction for dealing with loops, but the generated POs are currently insufficient. For example, although the simple example in Section 4.8 works, it does not account for obligations within the loop body or the effect of loops on subsequent obligations on the path. We also probably need a `@LoopTermination` annotation, which would be similar to a recursive measure, raising POs to verify that each loop gets closer to termination.

– Statements that handle exceptions (**`always`**, **`trap`** and **`tixe`**) cause control flows that are currently too complex to handle. Every statement within the body of these statements that could raise an exception effectively creates a set of new paths, one for each type of exception that could be thrown. Since VDM operations do not explicitly declare the exceptions that they can raise (i.e. there is no equivalent of Java's "throws" clause), this requires a deep analysis of the specification, and probably requires a pessimistic assumption about whether a given exception can actually occur.

– Specifications that include variable hiding can easily confuse the POG and produce invalid POs. Currently, a few cases of variable hiding are explicitly checked by the POG and POs are subsequently marked "Unchecked". But this approach is not generalized. A more robust approach would understand the variable hiding in the specification and perhaps rename variables in the obligation to compensate. The best advice is to avoid variable hiding in specifications and respect the type checker warnings if you get them.

– A deeper analysis of a specification's call graph and variable assignments may allow a more sophisticated technique than marking state as ambiguous after operation calls. Unfortunately, analyzing the call graph can become uncertain, given that we cannot predict which exceptions will be thrown. So such analysis would have to be pessimistic and *assume* that any path that could modify the state would actually do so. The result may produce obligations for paths that are actually not reachable.

– As mentioned in Section 4.9, the correct operation of the POG itself must be determined. This should ultimately be linked to a proof theory for VDM operations.

– VDM++ and VDM-RT cause a host of problems that have yet to be solved. Unlike the *mk_Sigma* record of VDM-SL, the state of these dialects cannot be easily represented, or passed to precondition functions. Furthermore, the state includes all accessible `static` fields in the model. VDM++ models can include multiple threads with synchronization clauses, which massively complicates the POG analysis. And VDM-RT introduces distributed behaviour with CPUs over connecting busses, which is even more complicated.

Acknowledgements We are grateful for the support of the European Union, Aarhus University, Newcastle University and the Grundfos Foundation. We also thank the reviewers for their valuable feedback on the original version of this paper.

References

1. Aichernig, B.K., Larsen, P.G.: A Proof Obligation Generator for VDM-SL. In: Fitzgerald, J.S., Jones, C.B., Lucas, P. (eds.) FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997). Lecture Notes in Computer Science, vol. 1313, pp. 338–357. Springer-Verlag (September 1997). https://doi.org/10.1007/3-540-63533-5_18
2. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
3. Battle, N.: VDMJ Annotations Guide. Tech. rep., Aarhus University (2025)
4. Coleman, J.W., Malmos, A.K., Nielsen, C.B., Larsen, P.G.: Evolution of the Overture Tool Platform. In: Proceedings of the 10th Overture Workshop 2012. School of Computing Science, Newcastle University (2012)
5. Freitas, L., Jones, C.B., Velykis, A., Whiteside, I.: How to Say Why. Tech. Rep. CS-TR-1398, Newcastle University, www.ai4fm.org/tr (November 2013)
6. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, second edn. (1990), iSBN 0-13-880733-7
7. J.Rask, F.Madsen, N.Battle, P.G.Larsen: Visual Studio Code VDM Support. In: Proceedings of the 18th Overture Workshop (2020)
8. Larsen, P.G.: Towards Proof Rules for VDM-SL. Ph.D. thesis, Technical University of Denmark, Department of Computer Science (March 1995), iD-TR:1995-160
9. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science **7**(8), 692–709 (2001). https://doi.org/10.3217/jucs-007-08-0692
10. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes **35**(1), 1–6 (January 2010). https://doi.org/10.1145/1668862.1668864
11. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
12. N.Battle, M.Ellyton: QuickCheck for VDM. In: Proceedings of the 22nd Overture Workshop (2024). https://doi.org/10.48550/arXiv.2410.02046
13. Ribeiro, A., Larsen, P.G.: Proof Obligation Generation and Discharging for Recursive Definitions in VDM. In: Song, J., Huibiao (eds.) The 12th International Conference on Formal Engineering Methods (ICFEM 2010). Springer-Verlag (November 2010)