Releasing VDM Proof Obligations with SMT Solvers

Hsin-Hung Lin
Institute of Information Science, Academia Sinica
Taipei, Taiwan
hlin@iis.sinica.edu.tw

Bow-Yaw Wang
Institute of Information Science, Academia Sinica
Taipei, Taiwan
bywang@iis.sinica.edu.tw

ABSTRACT

The Vienna Development Method (VDM) is a formal method that supports modeling and analysis of software systems at various levels of abstractions. For a model specified by the VDM specification language (VDM-SL), the correctness of the model relies on discharging the proof obligations (POs), especially in the case of implicit specifications. In this paper, we propose an approach that encodes and discharges POs of VDM-SL models using SMT solvers. More specifically, POs generated by the Overture tool are encoded and discharged in the Z3 SMT solver. Our case studies showed that the approach can discharge significant part of proof obligations of a VDM-SL model efficiently.

CCS CONCEPTS

• **Software and its engineering** → **Consistency**; **Software verification**; **Specification languages**; • **Hardware** → *Theorem proving and SAT solving*;

KEYWORDS

Vienna Development Method, Proof Obligation, Satisfiability Module Theories, Z3

ACM Reference Format:
Hsin-Hung Lin and Bow-Yaw Wang. 2017. Releasing VDM Proof Obligations with SMT Solvers. In *Proceedings of MEMOCODE '17, Vienna, Austria, September 29-October 2, 2017,* 4 pages.
https://doi.org/10.1145/3127041.3127066

1 INTRODUCTION

The Vienna Development Method (VDM) [5] is a formal method which supports modeling and analysis of software systems at various levels of abstraction. A VDM specification can be explicit, implicit, or both. An explicit specification is operational; it describes *how* system components work. An implicit specification, on the other hand, is declarative; it describes *what* system components should work. Software specifications in VDM models define functionalities explicitly or implicitly according to software requirements.

Discharging proof obligations (POs) [2] of VDM models is essential to guarantee the consistency of the formal specifications. For a VDM model, the number of POs is usually large compared to the size of the model. Consequently, it is not uncommon to have hundreds of POs for large VDM models. All of them have to be discharged to guarantee consistency of software specifications. Manually discharging so many POs with theorem provers, however, can be a daunting task in practice. Furthermore, both erroneous VDM specifications and bad proof strategies induce un-dischargeable POs. Theorem provers provide little information when they fail to discharge POs. It is unclear whether a VDM specification is flawed or a better proof strategy is possible when POs cannot be discharged by theorem provers.

Satisfiability Modulo Theories (SMT) solvers extend SAT solvers by incorporating decision procedures for first-order theories such as linear arithmetic. Given a quantifier-free first-order logic formula in mixed theories, SMT solvers will try to find a model satisfying the given formula or report that the formula is unsatisfiable. Thanks to their efficiency, SMT solvers have found a wide range of applications from software verification to software testing. Recent SMT solvers can even check satisfiability of general first-order formulas. It is therefore interesting to see if one can take advantages of recent developments of SMT solvers to discharge first-order POs in VDM models.

In this paper, we apply SMT solvers to discharge POs of VDM specifications. We focus on VDM-SL models and the objective is to discharge POs of a VDM-SL model as many as possible through SMT solvers. In our approach, we encode each POs with its context information such as type and state constraints as a first-order formula, and then prove the PO by sending the formula to SMT solvers. The advantages of our approach are (1) the encoding involves only a few segments of a VDM specification; (2) the proof process is automated with SMT solvers; (3) If a PO's proof fails, a counterexample model is returned for further examination.

We chose the Overture tool [7] as the PO generator. For the POs generated by the Overture tool, we encode and prove POs of VDM-SL models by solving SMT formulas with Python API of Z3 [10]. Z3 is a popular SMT solver widely used in software verification and its Python API provides flexibility in constructing SMT formulas [3]. To encode POs, built-in VDM types such as sequences and maps must be characterized by theories in Z3. We conducted case studies with manual encoding, and the results showed that our approach can efficiently discharge the significant part of POs of a VDM-SL model.

2 VDM PROOF OBLIGATIONS

For a VDM model, a proof obligation (PO) is a statement that must be proved to ensure the consistency of the model. A PO contains a predicate with its context information as shown below. The context information is from code segments of the VDM model that relates to the predicate to be proved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMOCODE '17, September 29-October 2, 2017, Vienna, Austria
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5093-8/17/09…$15.00
https://doi.org/10.1145/3127041.3127066

```
PO: context information ==> predicate
```

POs of VDM-SL can be classified type compatibility (subtype checking), domain checking, and satisfiability [2, 13]. Type compatibility relates to types with invariant or subtypes; domain checking relates to the use of partial functions and partial operators; satisfiability relates postconditions of functions and operations.

Below is a segment of VDM-SL model and two of the corresponding POs generated by the Overture tool. The VDM-SL specification is a state definition which consists of a sequence of commands <R> or <L> or nil. The state invariant says that for every element of commands, its next element must be assigned different value. PO1 is a state invariant satisfiable obligation which is a subtype checking for confirming that there exist values of state S satisfying the state invariant. PO2 is a legal sequence application obligation which is a domain checking for confirming that in the state invariant, (s.commands)(k), the k-th element of commands has to be defined. Though looked obvious, this kind of POs is important for VDM because of the use of partial functions in VDM.

```
-- //spec. segment
state S of
  commands : seq of [<R> | <L>]
  inv s == forall k in set
    {1,...,len s.commands - 1} &
    s.commands(k) <> s.commands(k+1)
  init p == p = mk_S([])
end

-- //PO1: state invariant satisfiable obligation
-- //     (subtype check)
(exists commands:seq of [<R> | <L>] &
  (forall k in set {1, ... ,((len (s.commands)) - 1)} &
    ((s.commands)(k) <> (s.commands)((k + 1)))
))

-- //PO2: legal sequence application obligation
-- //    (domain check)
(forall s:CMDS`S &
  (forall k in set
    {1, ... ,((len (s.commands)) - 1)} &
    (k in set (inds (s.commands)))))
```

3 ENCODING STRATEGY

In this work, we encode and prove VDM POs with the Z3 SMT solver. Z3 is one of the popular SMT solver wildly used in verification of programs and software systems. Z3 has APIs for major programming languages such as C, Java, and Python. The APIs provide a better way of constructing SMT formulas than SMT-LIB [3] because one can adopt the flow control characteristics like loop and if-then-else of programming languages to build and solve formulas in a flexible and smart way. In our approach, we chose Z3's Python API (Z3py) as the encoding environment.

Currently, the encoding is manually performed based on the obligation to be discharged and below are the steps of encoding and solving an obligation with SMT solvers.

(1) Determine the context information of the PO.
(2) Encode context information.
   (a) For a VDM type in the context information, both the type itself and its invariant have to be encoded.
   (b) If pre/post-conditions of functions/operations are used in a PO in the form of pre_<name> or post_<name>, encoding of pre/post-conditions is needed.
   (c) If a PO only considers a part of the specification of a type, a function, or an operation, it is only needed to encode the expressions that are directly used in the PO.
(3) Obligation negation and quantifier elimination.
   (a) Obligation negation: In model checking with SMT solvers, the assertion to be checked is usually negated, and the result of unsat means the proof of the assertion succeeds. For the result of sat which means the assertion can be violated, SMT solvers return a model as the counterexample that shows an example of how the assertion is violated. In this work, we also negate the PO to be discharged. However, the elimination of universal quantifier should be considered together when applying the negation to the PO.
   (b) Elimination of universal quantifiers: POs of VDM are usually with universal quantifiers. Though Z3 accepts formulas with universal quantifiers, there are limitations applying universal quantifier in Z3. Therefore, we eliminate universal quantifiers in the predicate of a PO before encoding. Obligation negation described above also achieves the elimination of universal quantifiers. However, for POs with both universal and existential quantifiers, or multiple universal quantifiers, it should be avoid introducing troublesome universal quantifiers, i.e., universal quantifiers on sequences, sets, and maps, from exists quantifiers.
(4) Encode the predicate of obligation and check for satisfiability.

VDM has rich types from natural number and quotes to sequences and maps. There are no direct matching for all VDM types to the types in Z3. For example, to encode a natural number, we need to declare an integer and then add a constraint of greater and equal to zero because there is no natural number type in Z3. The enumeration type in Z3 can be used to encode simple quote types. For complex types like sequences, maps, or records, we may use uninterpreted functions or arrays as the base type and capture the characteristic and operators of the VDM types by adding corresponding constraints. At this point, there is not yet a complete and systematic way of encoding VDM types in Z3. What we can do is to choose the encoding carefully based on the types and expressions involved in a PO. Based on our case studies, there are patterns of encoding that are frequently used for corresponding expression styles in VDM-SL models.

4 CASE STUDIES

We have conduct two case studies[1]: an abstract pacemaker and a telephone exchange system.

Abstract Pacemaker. The first case is a small VDM-SL model specifies an abstract pacemaker. The model is not a full specification and focuses on the part of definitions and operations related to heartbeat signals detected and recorded in a pacemaker. For this

[1]The VDM-SL models are from the Overture repository: http://overturetool.org/download/examples/VDMSL/. The encoding of POs done by hand and their checking results can be founded in our repository: https://github.com/fmlab-iis/VDM-SMT

model of 58 lines, the Overture tool generated nine POs. The encoding and checking of POs follow the encoding steps described in Section 3. In this section, we demonstrate some of the encoding patterns for encoding lists with arrays.

As the code shown below, in the abstract pacemaker model, the core types are `Trace` and `Event` since they show in almost all POs of the model. `Trace` is of type sequence of `Event`, where `Event` is of quote type with two values `<A>` and `<V>` that stand for the A-pace and V-pace of heartbeat signals.

```
Trace = seq of [Event];
Event = <A> | <V>;
```

The code below shows the encoding of `Event` and `Trace`. Since `Trace` is not exactly the sequence of `Event` but `[Event]` (a union with `nil`), we did not define `Event` but define `[Event]` directly. Furthermore, a type in VDM can be undefined because of VDM's logic of partial functions. This means that besides predefined values as in the specification, we have to consider the case that the value is undefined. In our work, we used `NDF` to represent the undefined value of `Event`, and defined `Event_lift` in Z3. The postfixed `_lift` indicates that we have considered the undefined value.

```
Event_lift, (A, V, nil, NDF) =
  EnumSort('Event_lift',
          ['A', 'V', 'nil', 'NDF'])

Trace = ArraySort(IntSort(),Event_lift)
```

In the above encoding, `Trace` is encoded as an array type in Z3 and this array type is by default indexed through the integer domain. An array type can not be a sequence type unless we add constraints regarding sequences to limit an array to a sequence. Below shows the encoding of the constraint to `tr`, the instance of `Trace` as an array. Note that in Z3, we cannot add constraints to types but instances. The code describes three constraints to reduce an array (`tr`) to a sequence that starts from index 1 to its last element.

```
1  tr = Const('tr',Trace)
2  [i,j] = Ints('i j')
3
4  ForAll(i, Implies( i<=0, tr[i]==NDF ) )
5
6  ForAll(i,
7    Implies(
8      And(i>=1,tr[i]!=NDF),
9      ForAll(j,
10       Implies(And(j>=1,j<=i),tr[j]!=NDF)
11     )
12   )
13 )
14
15 ForAll(i,
16   Implies(
17     And(i>=1,tr[i]==NDF),
18     ForAll(j,
19       Implies(j>=i,tr[j]==NDF)
20     )
21   )
22 )
```

- The first constraint (line 4) says that all indices of `tr` less than 1 should be undefined because a sequence's index starts from 1 in VDM.
- The second constraint (lines 6-13) says that if an index is defined, i.e., not undefined, all indices lower than it should be defined.
- The third constraint (line 15-22) says tat if an index is undefined, all indices greater than it should be undefined.

With another encoding for `len` operator of sequences. We were able to prove all nine POs of the abstracted pacemaker model. We also applied some other encoding patterns to encode operators `hd`, `tl`, `in set inds`. These operators can be encoded as constraints on the range of a sequence so that we do not have to declare corresponding instances of sequences or sets.

Telephone Exchange. The second case study is a VDM-SL model specifies an abstracted telephone exchange system. In this model, the operations specify the events which can be initiated either by the system or by a subscriber (user) with implicit style. The system state monitors the calling status of users and the connecting status among users. The Overture tool generated 27 POs, and we managed to encode and prove 16 POs. The main reason of leaving some POs unproved is because of nested universal quantifiers.

Instead of using arrays, we tried using uninterpreted functions to encode maps. We also applied encoding patterns mentioned in the abstract pacemaker case. In the checking, we found an inconsistency about the constraint regarding `calls`, the map specifying relations between subscribers. The counterexamples generated by Z3 were helpful for suggesting how to correct the model.

Table 1 shows the results of the two case studies. The checks are conducted on a machine with Intel Core i7 2.4GHz CPU and 8GB RAM.

Abstract Pacemaker				Telephone Exchange			
PO#	neg.	res.	time(s)	PO#	neg.	res.	time(s)
1	+	sat	0.031	1	+	unsat	0.032
2	+	unsat	0.031	2	+	unsat	0.031
3	-	sat	15.109	3	+	unsat	0.031
4	+	unsat	0.032	4	+	unsat	0.032
5	+	unsat	0.046	5	+	unsat	0.03
6	+	unsat	0.048	6	+	unsat	0.047
7	+	unsat	0.062	7	-	sat	0.031
8	+	unsat	0.031	9	+	unsat	0.047
9	+	sat	0.047	12	+	sat	0.015
				14	+	sat	0.063
				17	+	sat	0.016
				18	+	unsat	0.047
				20	+	sat	0.016
				21	+	unsat	0.046
				23	+	sat	0.015
				25	+	sat	0.063
POs checked: 9 / 9 (100%)				POs checked: 16 / 27 (59%)			

Table 1: Results of Case Studies

5 DISCUSSION

As an overall result of our case studies, the results showed that our approach can efficiently discharge the significant part of POs of a VDM-SL model. In the first case study (Abstracted Pacemaker), we discharged all nine POs; in the second case study, we discharged 16 POs our of total 27 POs (about 59%). To increase the percentage of discharged POs, we may apply different encoding, even limited to finite types such as BitVector, to solve remaining POs in the Telephone Exchange case study.

From our experiences, the cost and complexity of encoding varies from specifications in a VDM-SL model. Usually, the complexity of encoded formulas depends on the complexity of VDM-SL types. For example, natural number based types such as `seq of nat` or `map from nat to nat` have less complexity, and one may encode and solve this kind of types with SMT solvers without difficulty. On the contrary, for VDM types such as a map of sequences or a set of maps, the encoding may be difficult, and the complexity of the encoded formulas may grow a lot so that Z3 may fail to check the satisfiability.

We have found some encoding patterns through the case studies. For example, the encoding of union type and quote type with user-defined types or enumeration type, and the encoding of maps and sequences with uninterpreted functions or arrays, are useful to encoding POs of VDM-SL models. Also, the encoding patterns like the constraints on arrays to represent a sequence, set inclusion of indices of a sequence, or set inclusion of domain/range of a map, are also useful. In our case studies, these encoding patterns are applied multiple times.

Though the encoding patterns are useful, applying the encoding patterns still depends on the specification of VDM models because the encoding patterns are not systemic or syntax-based at this point. Considering automated encoding of POs with related context information, the encoding of types may be fine, but encoding expressions remains difficult and rely on manual assistances may be needed.

As initial results, it is recognized that discharging POs using SMT solvers is efficient, and the counterexample generation of SMT solvers is useful for correcting inconsistency of VDM models. Currently, the encoding is done by hand. As for future work, we are working on systematic encoding patterns and automated encoding for conducting more case studies.

6 RELATED WORK

Proof obligation generation of VDM is firstly proposed by B. K. Aichernig and P. G. Larsen [2]. Later, A. Ribeiro and P. G. Larsen worked on proof obligation generation and discharging related to the termination of recursive functions [11]. The mainstream of discharging POs of VDM models rely on translation from VDM to theorem provers. The generation and discharging of POs are conducted on theorem provers such as PVS and Isabella/HOL. S. Agerholm [1] proposed a translation from a subset of VDM-SL to PVS for type checking. S. Maharaj and J. Bicarregui [8] used Agerholm's translation to assist the verification of VDM-SL models and their refinements. Later, S. D. Vermolen et al. [12, 13] utilized the parsing mechanism of the Overture tool and developed automated translation from a subset of VDM-SL to HOL theorem prover. Recently, work on improvement of translating VDM-SL to Isabella/HOL was proposed [4].

In general, the theorem prover approach has two major issues: (1) The proof process is usually complicated and requires an expert even with the help of proof tactics; (2) Theorem provers provide little information when they fail to discharge POs. Compared to our approach, we adopt the efficiency of SMT solvers, and the counterexample generation of SMT solvers is helpful for locating the problem. However, encoding in our approach is manual at this point and still need improvements.

Theorem proving with SMT solvers is not a new idea. S. Merz and H. Vanzetto [9] has proposed a backend using SMT solvers to assist discharging proof obligations of TLA+. D. Kröning et al. [6] has proposed a theory of finit sets, lists, and maps for SMT-LIB. The proposed theory is general but limited in basic types of `Int` and `Real`. The authors only discussed how to realize the theory by reduction to existing SMT theories with a report of initial benchmarks[2] on a Event-B case study. The idea of adopting existing theories of SMT is the same as our approach. However, we argue that a strategic encoding like our approach might is necessary for discharging VDM POs than introducing general theories because the infinite and undecidable nature of sets, lists, and maps in VDM.

REFERENCES

[1] Sten Agerholm. 1996. *Translating specifications in VDM-SL to PVS.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[2] Bernhard K. Aichernig and Peter Gorm Larsen. 1997. A Proof Obligation Generator for VDM-SL. In *Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods (FME '97).* Verlag, London, UK, UK, 338–357.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`. (2016).

[4] Luís Diogo Couto and Peter W. V. Tran-Jørgensen. 2015. Extending the Overture code generator towards Isabelle syntax. In *Proceedings of the 13th Overture Workshop.* Center for Global Research in Advanced Software Science and Engineering, 48–59.

[5] Cliff B. Jones. 1990. *Systematic Software Development Using VDM, 2nd Ed.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[6] Daniel Kröning, Philipp Rümmer, and Georg Weissenbacher. 2009. A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard. In *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22.*

[7] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. 2010. The Overture Initiative Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes* 35, 1 (jan 2010), 1–6.

[8] Savi Maharaj and J. Bicarregui. 1997. On the verification of VDM specification and refinement with PVS. In *Proceedings 12th IEEE International Conference Automated Software Engineering.* 280–289.

[9] Hernán Merz, Stephanand Vanzetto. 2012. *Automatic Verification of TLA + Proof Obligations with SMT Solvers.* Springer Berlin Heidelberg, Berlin, Heidelberg, 289–303.

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS.* Springer, 337–340.

[11] Augusto Ribeiro and Peter Gorm Larsen. 2010. *Proof Obligation Generation and Discharging for Recursive Definitions in VDM.* Springer Berlin Heidelberg, Berlin, Heidelberg, 40–55.

[12] Sander D. Vermolen. 2007. *Automatically Discharging VDM Proof Obligations using HOL.* Master's thesis. Computing Science Department, Radboud University Nijmegen.

[13] Sander D. Vermolen, Jozef Hooman, and Peter Gorm Larsen. 2010. Proving Consistency of VDM Models Using HOL. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10).* ACM, New York, NY, USA, 2503–2510.

[2] http://www.cprover.org/SMT-LIB-LSM/