Releasing VDM Proof Obligations with SMT Solvers

Hsin-Hung Lin, Bow-Yaw Wang
Institute of Information Science, Academia Sinica
MEMOCODE'17@Vienna
Sep. 30, 2017

Good afternoon, everyone. My name is Hsin-Hung. It is my pleasure to present my work-in-progress here.

Contents

- Introduction
  - VDM and Proof Obligations
  - Motivation and Objective
- Approach
  - Strategy and Steps
- Preliminary Case Studies
- Summary

The contents today will be firstly an introduction about VDM proof obligations and the objective of this research.
Then I will talk about the approach and preliminary case studies.
Finally, I will summarize this talk and you may ask questions or kindly give me some advices or comments.

Introduction

Vienna Development Method (VDM)

- Developed by IBM Vienna Lab. in the 70's
- Specification Language of VDM (VDM-SL)
- Types, (state) variables, operations, functions
  - Primitive types: bool, int, nat, token, quote
  - Compound types: set, seq, map, record
- Constraints
  - functions, operations: pre/post-condition
  - types, variables: invariant
  - proof obligation generation

At first. I would like to talk about the Vienna Development Method, VDM.
VDM is a formal method firstly developed in the 70's, at Vienna.

And we are in Vienna. What a coincidence!

Generally, VDM is a methodology of software development from software specification to implementation.
VDM has a specification language called VDM-SL
VDM-SL is like a programming language with more abstraction.
VDM-SL has basic types such as Boolean, integers, natural numbers which are commonly used in programming languages.
Some basics types like token and quote are not. They can be used to describe software in an abstracted way.
VDM-SL has compound types like sets, sequences, maps that are recently started being used in programming languages.

Besides types, another important feature of VDM-SL is specifying constraints.
Constraints include pre and post conditions, type constraints.
From these constraints, we may generate proof obligations that should be proved to ensure the consistency of a VDM-SL model.

Vienna Development Method (VDM)

- Dialects of VDM specification languages
  - VDM++ (Object-Oriented)
  - VDM-RT (Real-Time Scheduling)
- Animation by interpreter
  - Specifications need to be specified explicitly
  - Verification/Validation by Testing
- Tools support editing, type checking, and animation
  - Overture Tool (http://overturetool.org/)
  - VDMTools (http://fmvdm.org/)

Because VDM is developed very early, with the advances of software and programming languages, there are dialects of VDM.
VDM++ is an extension of VDM-SL for object-oriented software development.
VDM-RT further extends for real-time feature for modeling Cyber-Physical-Systems.
Today we will focus on VDM-SL.

Within a subset of VDM-SL, a VDM-SL model can be executed by the interpreter.
This is a feature welcomed by the industries because they can test software specifications described in VDM-SL models.
There are two tools support modeling with VDM. In this work we will use Overture tool.

VDM-SL model CMDS(1/2)

```
module CMDS
definitions

types
CMD = <R> | <L>;
CMD_series = seq of [CMD];
CMD_times = map CMD to nat;

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


For some of you not familiar with VDM. This is a simple VDM-SL model:
In tis model, CMD is a quote type, which is usually used to represent enumeration.
CDM_series is a sequence of CMD with optional nil.
CMD_times is a map from CDM to natural numbers
In VDM-SL, state is defined as a special type, “inv” part specifies the state invariant.
In this model, the state is a sequence of commands, and the state invariant says that:
For every element in the sequence, its next element has to be different.

VDM-SL model CMDS(2/2)

```
operations
  push_cmd(a:[CMD])
  pre commands = [] or hd commands <> a
  post hd commands = a and tl commands = commands~;

functions
times_count : CMD_series -> CMD_times
times_count(a) == {
       <R> |-> len [ i | i in set inds a & a(i)=<R> ],
       <L> |-> len [ i | i in set inds a & a(i)=<L> ]
  };

end CMDS
```


In VDM-SL, we may define operations which may change the value of state.
In this model, we have defined an operation push command
This operation has pre and post condition
The precondition says the command sequence in the state has to be empty or different to the command to be pushed
The postcondition says the new command sequence after the operation execution shall be the pushed command concatenated with the old command sequence

You may notice the tild symbol post-fixed to “commands”, it represents the oldstate in postcondition of an operation.
This operation is defined implicitly where no body is specified, only pre and post conditions.

This model also has a function to count the occurrences of commands. The function is defined explicitly. This way the function is executable by the interpreter.

VDM Proof Obligations (PO)

- PO: Statements must be proved to ensure consistency

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

- Three Classes of PO
  - Type compatibility: type invariant or subtype
  - Domain checking: partial functions/operators
  - Satisfiability: postconditions

VDM proof obligation is in the form: context information implies a specific predicate
In VDM, there are three classes of proof obligations
First one is type compatibility, which is relates to type invariant, or subtype.
Second one is domain checking, which Is regarding partial function and operators
Simply speaking, this is very like checking divide by zero in a division.
The last one is satisfiability which concerns the satisfiability of postconditions

POs of CMDS (1/2)

```
-- //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
```


Let us recall the state definition of the simple VDM-SL model introduced.

POs of CMDS (2/2)

```
-- //PO1: state invariant satisfiable obligation
-- //      (subtype check: state invariant)
(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: apply operator of sequence)
(forall s:CMDS`S &
  (forall k in set
    { 1, ... ,((len (s.commands)) - 1) } &
    (k in set (inds (s.commands)))))
```


From the state invariant, there are two proof obligations
The upper one is a subtype check: is there exists a sequence of commands that satisfies the state invariant
The lower one is a domain check regarding the sequence application operator (check previous slide)
This proof obligation says that the index “k” shall be defined in the command sequence.

Motivation and Objective

- A VDM-SL model may have a large number of POs compared to its LOC
  - For example, CMDS: 24 LOC with 10 POs
  - Too many to prove manually or interactively with theorem provers
- Existing Approach
  - Prove with HOL/Isabella theorem prover
    - Translate VDM-SL to Isabella
    - Generate and prove POs in Isabella
  - Specialists of theorem provers are required
  - Automation relies on proof tactics
  - Get little hint from unsuccessful proofs

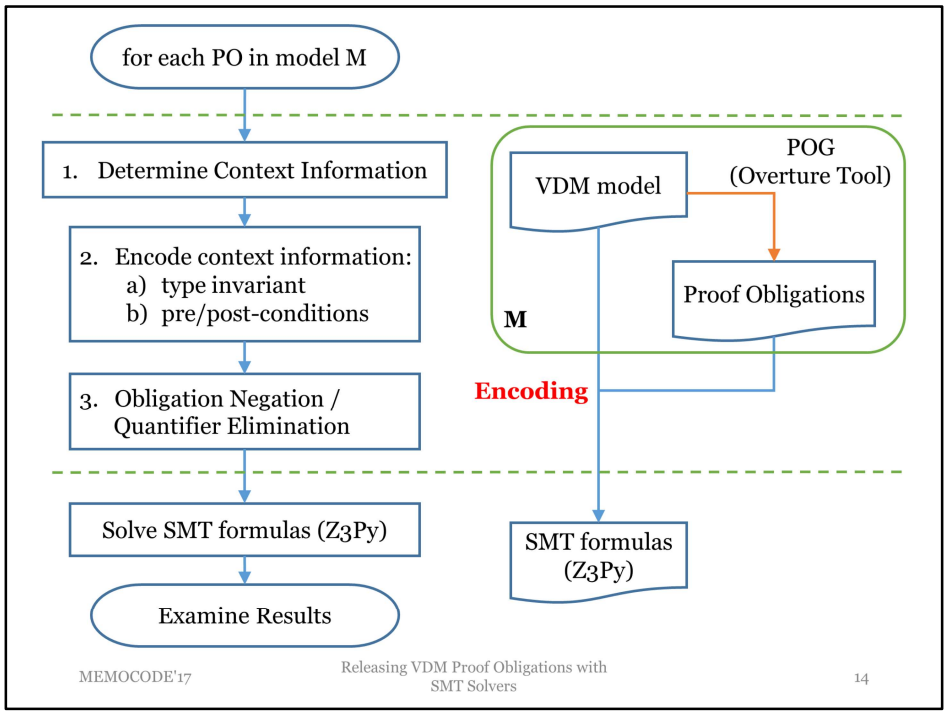
As explained shortly in previous slides, a few lines of VDM-SL code may produce about same number of proof obligations.
In the VDM-SL model CMDS, 24 lines of code results in 10 proof obligations.
So, to prove them all with theorem provers, even interactively with machine support, will consume much time and effort.
Currently, an approach that translate VDM-SL code to Isabella was proposed.
This approach generates proof obligations in Isabella and prove them with tactics
But this still need well trained specialists to conduct the process and the tactics are not always work.
Also, if a proof obligation is not proved, there is little information whether the model has bug or just lack of lemma.

Motivation and Objective

- Satisfiability Modulo Theories (SMT)
  - State-of-the-art technology for HW/SW verification
  - Efficient solving
  - Automated proof
  - Counterexample
- Prove VDM POs as many as possible with SMT solvers
  - Encoding VDM POs to SMT formulas
  - Difficulty: higher order logic, partial function

Here we think about the satisfiability modulo theories.
SMT has become a powerful technique for software verification.
SMT can efficiently solve hundreds or thousands of formulas automatically.
If unsatisfied, SMT solver will return a counterexample model, which is very convenient for debug
So, our objective is to apply SMT solvers for discharging VDM proof obligations as many as possible.
Our approach is to encode VDM proof obligations to SMT formulas and prove the proof obligations by SMT solving.
However, this is not a easy task because VDM is generally higher older logic with partial function, while SMT is good at solving quantifier free FOL.

Encoding Strategy



The right side shows the VDM-SL model and proof obligations generated by the Overture tool.
The encoding results in a set of SMT formulas in Z3Py, a Python API of Z3 SMT solver.
The encoding process is simplified as several steps as the left diagram shows.
For each proof obligation, firstly, we determine its context information, then encode the context informations
The context information mainly contains type invariants and pre/post conditions.
Then we encode the predicate of the proof obligation.
Before solve the obligation, we need to perform the obligation negation and quantifier elimination.
Finally we solve the SMT formals with Z3Py and examine the result.

Encoding Strategy

- Determine context information
  - Types and constraints involved (invariants, pre/post-conditions)
- Encode context information
  - Encode type/state invariant only when the type/state (`a:typename,oldstate,newstate`) is mentioned
  - Encode pre/post-condition if `pre_<name>` or `post_<name>` is mentioned
- Obligation negation / Quantifier elimination
  - Negation also does quantifier elimination
- Solve formulas with SMT solver (Z3Py)

For more details, when encoding the context information, we encode type invariants only when the type is explicitly mentioned.
For states, oldstate and newstate in an obligation means the state before and after the execution of an operation.
For pre and post conditions, they are mentioned in the form of pre_ function or operation name, post_ function or operation name.
Currently, the encoding process is done by hand.

Preliminary Case Studies

Abstract Pacemaker

- A simplified pacemaker
  - Definitions about heartbeat signals detected and recorded
  - Primary types: Quote, Sequence
- Encoding in Z3Py
  - Sequence: array sort with constraints
  - `len` operator: function

We have done two preliminary case studies. The two models were collected from the repository of Overture tool site, and the encoding was manually done.
The first case is an abstract pacemaker.
This model defines a pacemaker partly, only focus on heartbeat signals
The types used in this model are quote and sequence.
To encode in SMT, we represent sequence as array sort with constraints in Z3
In this model, we need to also encode the “len” operator that returns the length of a sequence.

PO 8: legal sequence application

```
-- PO8
(forall tr:Trace, aperi:nat1, vdel:nat1,
  oldstate &
    (forall i in set (inds (tl tr)) &
      (((i mod aperi) = (vdel + 1)) =>
        (i in set (inds tr)))
))
```

VDM type definition (contex information)

```
-- type: sequence of quotes
Trace = seq of [Event];
Event = <A> | <V>;
```

Encoding in Z3Py

```
-- VDM types encoded in z3py
Event_lift, (A, V, nil, NDF) =
    EnumSort('Event_lift',
              ['A', 'V', 'nil', 'NDF'])

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


For example, look at the obligation 8 here, tr:Trace is explicitly mentioned so we will need to encode the type “Trace” which is a sequence of Events, and an event is the enumeration of [A] or [V]
We encode Trace as array sort.
Here we encode Event type in “lifted” form because VDM uses partial functions, this means there is not always value defined.
For the undefined case we use “NDF” to represent so we have full function in Z3.

Constraints of seq (Z3Py):

1. Undefined below index one
2. If defined, indice, all defined for indice between 1 and itself
3. If undefined, all undefined for higher indice

Note: Constraints are applied on instance variable, not type

```
tr = Const('tr',Trace)
[i,j] = Ints('i j')

ForAll(i,
       Implies( i<=0, tr[i]==NDF ) )

ForAll(i,
  Implies(
    And(i>=1,tr[i]!=NDF),
    ForAll(j,
      Implies(And(j>=1,j<=i),
      tr[j]!=NDF)
    )) )

ForAll(i,
  Implies(
    And(i>=1,tr[i]==NDF),
    ForAll(j,
      Implies(j>=i,tr[j]==NDF)
    )) )
```


Also, array is not sequence, so we added 3 constraints to restrict array to sequence.
The first constraint is that below index zero, all elements are undefined since a sequence in VDM starts from index 1
The second constraints is that if we find an element of index i is defined, then all indexes below i and greater than 1 shall be defined.
The third constraint is that if we find an element of index i is undefined, then all indexes above i shall be undefined.

Operator **`len`** (Z3Py):

1. A uninterpreted accepts a Trace and returns Integer.
2. The return value must be zero, where undefined at index one, or
3. A number greater then zero, where undefined next to the index of lenfth.

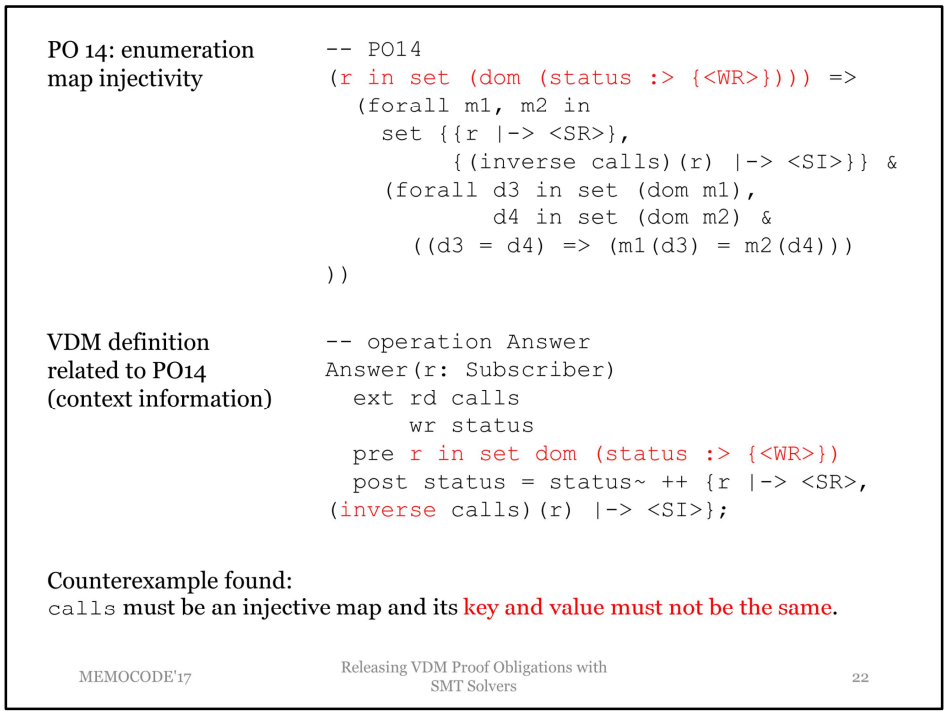
```
len_tr=Function('len_tr',Trace,IntSort())

Or(
  And( len_tr(tr)==0, tr[1]==NDF ),
  And(
    len_tr(tr)>0,
    tr[len_tr(tr)]!=NDF,
    tr[len_tr(tr)+1]==NDF )
)
```


For operator of length, we use uninterpreted function to encode it.
This formula says that the 'len' function must be zero or the length is the biggest index of defined elements.

Telephone Exchange

- An abstracted telephone exchange system
  - Specifying operations (events) of a telephone exchange system
    - Ex: lift, connect, answer, etc.
  - Operations changes system state
    - Subscriber (user)
    - Connecting status between subscribers
  - Types used: Quote, Map

The other case study is a telephone exchange model.
This model specified events in the system like lift a phone, connect users, and so on.
This model has types of quote and map.



To save time, I will go fast with this case study. So the code will not be explained in detail.
This slide shows obligation 14 that relates to map injectivity from the definition of Answer operation.
The obligation is to prove the consistency of applying inverse to “calls”

**Back from slide 25:
Actually we found a counterexample to this obligation.
The counterexample suggested that “calls” is not allow to have its key and value the same.
This means in this system, a subscriber shall not connect to him or her self.

VDM type definition

```
-- type: map of quotes
Subscriber = token;
Initiator =  <AI> | <WI> | <SI>;
Recipient = <WR> | <SR>;
Status = <fr> | <un> | Initiator | Recipient;

state Exchange of
  status: map Subscriber to Status
  calls:  inmap Subscriber to Subscriber
inv mk_Exchange(status, calls) ==
  forall i in set dom calls &
    (status(i) = <WI> and
     status(calls(i)) = <WR>) or
    (status(i) = <SI> and
     status(calls(i)) = <SR>)
init s == s = mk_Exchange({|->},{|->})
end
```


The state definition in this model has two maps:
Here we only interest to “calls”, which is an injective map that maps subscribers to subscribers. This represents the connection relation of subscribers.

Encoding in Z3Py

```
Recipient = Datatype('Recipient')
Recipient.declare('WR')
Recipient.declare('SR')
Recipient = Recipient.create()

Status = Datatype('Status')
Status.declare('fr')
Status.declare('un')
Status.declare('INITIATOR',('get_initiator', Initiator))
Status.declare('RECIPIENT',('get_recipient', Recipient))
Status = Status.create()

Status_lift = Datatype('Status_lift')
Status_lift.declare('STATUS',('get_status', Status))
Status_lift.declare('NDF')
Status_lift = Status_lift.create()

status = Function('status',Subscriber,Status_lift)
calls = Function('calls',Subscriber,Subscriber_lift)
```

token is treated as enumeration type with predefined values

In this case, we encode map with uninterpreted function.
Similar to pacemaker case study, we also define VDM types with lift.

Encoding injective map

```
k  = Const('k',Subscriber)
l  = Const('l',Subscriber)
ForAll([k,l],
  If(
    k==l,
    calls(k)==calls(l),
    Or(
      And(
        calls(k)==Subscriber_lift.NDF,
        calls(l)==Subscriber_lift.NDF
      ),
      calls(k)!=calls(l)
    )
  )
)
```

Note: Constraints are applied on instance variable, not type

"calls" is a injective type, which means its key and value are one-to-one.

Some frequently used encoding patterns (in both case studies)

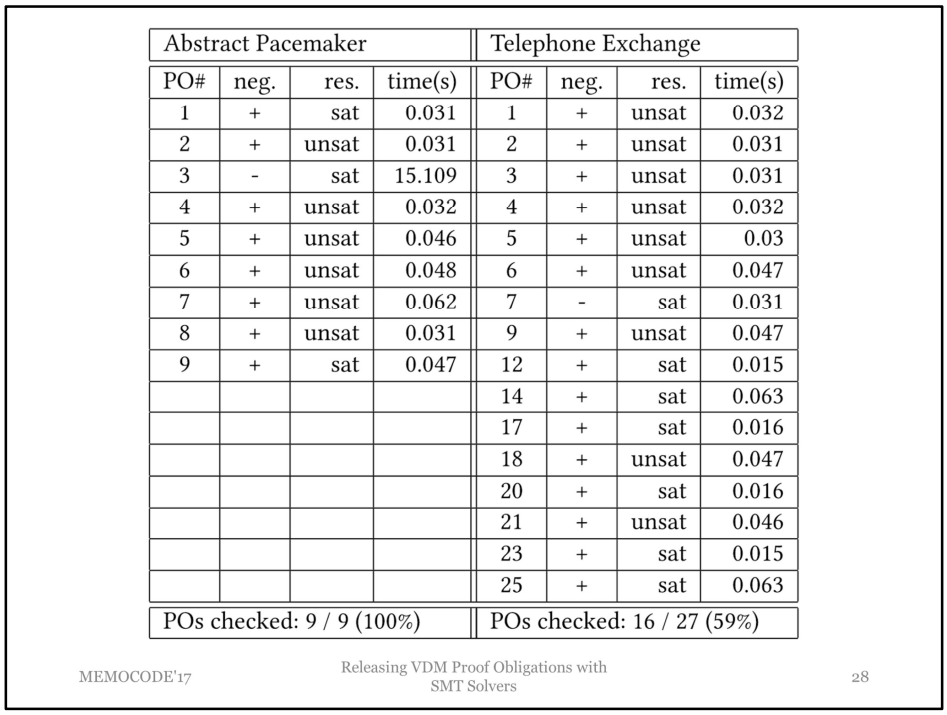
```
-- VDM code
i in set (inds (tl tr)

-- encoding in z3py
And(i>=1, i<=len_tr(tr)-1)
```

```
-- VDM code
i in set (dom calls)

-- encoding in z3py
calls(i) != Subscriber_lift.NDF
```


One more thing.
In the two case studies, some encoding patterns are used frequently.
For example, the set inclusion operator “in set” is encoded as the formula that the index is within the corresponding range.
For maps, “in set dom” can be encoded as the formula that the value of the key is defined.

Summary

- What types/patterns are encoded
  - quote type
  - seq/map of quotes
    - array sort with constraints
    - uninterpreter function with constraints
    - Introducing undefined value (NDF) and lift for map tpyes
  - Encoding pattern
    - seq: hd, tl, in set inds
    - map: in set dom, in set rng



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%)			

The results of the two case studies.
The abstract pacemaker, we managed to prove all proof obligations
The telephone exchange, we managed to prove 59% of the proof obligations.
From the time consumed, the efficiency is quite good.

PO3 of the pacemaker model is an operation postcondition satisfiability obligation.
But the postcondition/obligation does not have oldstate in its predicate.

Discussions: Some Issues

- Encoding set using array sort
  - Encoding cardinality: frequently used in VDM
- Function/Operation satisfiability obligations in the telephone case study
  - In the form with multiple quantifiers

```
forall a:type_a, oldstate:STATE &
    pre_f(a,oldstate) =>
        exists newstate:STATE &
            post_f(a,oldstate,newstate)
```

  - For set, seq, map types, Z3 returns unknown

We still have issues to be solved.
In the two case studies, we did not encode set type.
This is because the set cardinality is difficult but it is frequently used in VDM.
Also, we encountered proof obligations with multiple quantifiers.
This usually happens in postcondition satisfiability obligation.
For the above issues, though we can still encode with array and uninterpreted function, Z3 returns unknow and fails to prove nor gives counterexample.

Summary

Summary

- An approach to releasing VDM proof obligations with SMT solvers
  - Objective: discharge as many as possible
  - Pros: automated proof, efficiency, counterexample
  - Actually found bugs not found before (telephone exchange case)
- Issues
  - Array sort or uninterpreted function can not be applied to every Types and POs
- Work on going
  - Define a subset of VDM-SL appropriate to encoding
    - Must have: VDM-like (set, seq, map must be included)
    - Finite vs. Infinite

So, as the summary of this talk, we proposed an approach to releasing VDM proof obligations with SMT solvers.
The preliminary case studies showed good results but we still have issues to be solved.
Now we are working on these issues.

That's all.
It is great if I can have some advices or comments form you.
Thank you very much.