Topologically sorting VDM-SL definitions for Isabelle/HOL translation

Leo Freitas[1] and Nick Battle

School of Computing, Newcastle University,
leo.freitas@newcastle.ac.uk

Abstract. There is an ecosystem of VDM libraries and extensions that includes a translation and proof environment for VDM in Isabelle [1]. Translation works for a large subset of VDM-SL and further constructs are being added on demand. A key impediment for novice users is the fact Isabelle/HOL requires all definitions to be declared before they are used, where (mutually) recursive definitions **must** be defined in tandem. In this paper, we describe a solution to this problem, which will enable wider access to the translator plugin to novice users as well real models.

Keywords: VSCode, VDM, Sorting

1 Introduction

The Vienna Development Method (VDM) has has been widely used both in industrial contexts and academic ones covering several domains of the fields of Security [3, 4], Fault-Tolerance [7], Medical Devices [5], among others. We extend VDM specification support with a suite of tools and mathematical libraries [2].

VDM also has an Visual Studio Code (VS Code) IDE with multiple features, including a translation strategy to Isabelle/HOL [8]. Nevertheless, the user needs to follow a strict and specific specification style. Otherwise, translation will either fail or be impossible. For example, every Isabelle/HOL theory **must** be written within a file of the same name, whereas VDM files might be moduleless or have multiple modules. Translation of such VDM modules will have to impose the required Isabelle style. Thus, users have to be aware of such issues, or else translation will fail.

There are various such issues, which are listed in various example files in the distribution, and explained in detail in [2]. These VDM "idioms" are important not only to enable translation, but also to ensure proofs will be manageable and proof strategies will be easier to identify.

Nevertheless, a major impediment for use is the fact VDM specification **must** be ordered (*i. e.,*have declarations defined before they are used). This is rather unnatural to most VDM users, which tend to write specifications top-down, from larger/complex concepts to simpler/easier definitions. Also, it makes translation of legacy models hard because they were not written with such order requirement in mind. Manual

[1] https://github.com/leouk/VDM_Toolkit/
[2] https://github.com/leouk/VDM_Toolkit/

rearrangement is not difficult, yet it is laborious and error prone, hence creating a barrier to entry for the translator.

Solving this specific order requirement is the key contribution of this paper. We present the solution following the style by Naur's N-Queens algorithm [6]: we provide a historical account to how various developments within the VDM tools eventually led to the possibility of module sorting.

2 Background

VDM translation is performed by traversing its abstract syntax tree (AST) and issuing corresponding Isabelle/HOL, for every part of the VDM concrete syntax that are compatible for translation. For example, VDM union types are quite expressive. They even allow some wacky definitions like this cross union selection example.

```
types
    TUnion1 = int | nat
    inv u == (is_nat(u) => u > 0) and (is_int(u) => u < 0);

    TUnion2 = seq of nat | set of real;

functions
    f: TUnion2 * TUnion1 -> bool
  f(u2, u1) ==
    (is_int(u2) =>
      ((is_(u1, seq of nat) => u2 in set elems u1)
       and
       (is_(u1, set of real) => u2 in set u1)
      )
    )
    and
    (is_nat(u2) =>
      ((is_(u1, seq of nat) => not u2 in set elems u1)
       and
       (is_(u1, set of real) => not u2 in set u1)
      )
    );
```

VDM union types will create a maximal type set over the united types, where invariants are preserved. This can create quite complex (and confusing) selection and invariant checking processes. Of course, rarely such specifications are written. Yet, from a translation point of view, these are some of the challenges around VDM's expressivity with respect to how they can be translated to Isabelle.

That means the translator caters for the subset of VDM that is "tamer" (and mostly used). The translator has a VDM analysis tool (named `exu`[3]), which checks for various such syntactic conondrums and limitations, telling users what to do and where problems are. Fixing such errors and warnings is important to ensure that translations are possible and without type errors for the user to solve on their own.

`exu` also provides support to prepare users for translation (*e. g.,* there are over 50 kinds of "error" and 20 "warnings" the translator can issue). For example, it checks that

[3] The name stems from the Yoruba divinity *Èšù* that was the gate keeper (and messenger) between mortals and deities.

any function call within a function definition ought to include that function's precondition, if one exist. Yet, one key hindrance remained: ordering of module definitions and their dependencies. This hindrance will be addressed in this paper and have recently been implemented in the latest versoin of the translator.

3 Important Historical Developments

Before getting to our solution, it is important to identify key (unrelated) extensions to the main VDM tools. They were the stepping stones that eventually enabled `exu` to sort definitions for translation. We highlight the motivation for such extensions, as well as how they are implemented in the main VDM tools below.

3.1 Load time problems

When working with large VDM models [**?**] (*e. g.,* 150+ modules, 60+ KLOC), typecheck time (*e. g.,* $2-4$ minutes) can become a hindrance, whereas initialisation time (*e. g.,* 15+ minutes) can compromise further development.

Such long loading times occured because the VDM typechecker performs multiple passes through the AST in order to establishes whether all modules are consistent. The number of passes depends on the number of module inter dependencies (*i. e.,* `module A` imports `module B` and vice-versa), order of declarations within modules (*i. e.,* using definitions before defining them), *etc.*. LF: Nick, any more here?.

Users of large models would have to keep track of dependencies by carefully checking import chains. For example, we had to carefully craft dependencies and keep their documentation explicitly. This was error prone and hard to maintain.

Verbose output. To address this, an extension was added to VDMJ that would track top-level type dependencies across modules. A simple and easy solution was for the typechecker to issue verbose output about every top-level definition dependencies as it traversed the VDM AST. This required a tree visitor looking for dependencies and a mechanism for discovering free variables throughout the AST. This verbose output enabled users to know how many passes have taken place, and whether cyclic dependencies existed and where. Users would then go and fix them manually. This meant one could use the verbose output to minimise passes by lowering dependencies, hence minimising load time.

Dependency and free variable visitors. In VDMJ [1], visitors are used to make specialised traversals over the AST. For identifying module ordering, there are two visitors. The dependency visitor traverses the tree LF: Nick, explain here what the dependencies and free variable visitors do?.

Topological sort of module dependencies. With the information gathered about dependencies, it was possible to create a topological sort of module names, where the result would be a list of module names with the fewest passes possible. This effectively solved (in most cases) the dependency warnings from the verbose output, hence giving the end user the ordered list of module names to load. This could then ge given to VDMJ at load time directly. This was enabled output of a acyclic directed graph view of module dependencies as a `dot` file.

3.2 Recursive cycles detection

LF: Nick, maybe comment here the issues about mutual recursion and recursive type dependencies? And the tools / functionality they created?

4 Exu ordering

5 Applications and Examples

Other examples of use are provided in the `ISQ.vdmsl` distribution [4]. They show various ISQ measurement systems in practical use.

6 Results and discussion

In this paper, we presented .

Future work. Future.

Acknowledgements. Acks.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Freitas, L.: A VDM translation strategy to isabelle (Nill Full game) (2016)
3. Kulik, T., Macedo, H.D., Talasila, P., Larsen, P.G.: Modelling the HUBCAP Sandbox Architecture In VDM – a Study In Security. In: Fitzgerald, J.S., Oda, T., Macedo, H.D. (eds.) Proceedings of the 18th International Overture Workshop. pp. 20–34. Overture (December 2020)
4. Kulik, T., Talasila, P., Greco, P., Veneziano, G., Marguglio, A., Sutton, L.F., Larsen, P.G., Macedo, H.D.: Extending the formal security analysis of the hubcap sandbox. In: Macedo, H.D., Thule, C., Pierce, K. (eds.) Proceedings of the 19th International Overture Workshop. Overture (10 2021)
5. Macedo, H.D., Larsen, P.G., Fitzgerald, J.: Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods, 15th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 5014, pp. 181–197. Springer-Verlag (2008)
6. Naur, P.: An Experiment in Program Development. BIT 12, 347–365 (1972)
7. Nilsson, R., Lausdahl, K., Macedo, H.D., Larsen, P.G.: Transforming an industrial case study from VDM++ to VDM-SL. In: Pierce, K., Verhoef, M. (eds.) The 16th Overture Workshop. pp. 107–122. Newcastle University, School of Computing, Oxford (July 2018), TR-1524
8. Rask, J.K., et al.: Advanced VDM Support in Visual Studio Code. In: 20th International Overture Workshop. pp. 35–50 (2022)

[4] https://github.com/leouk/VDM_Toolkit/