vdm2dafny: An Automated Translation Tool for VDM-SL

Adam Winstanley and Leo Freitas[1 [0000−0001−9661−563X]]

School of Computing, Newcastle University, Newcastle-upon-Tyne, United Kingdom

Abstract. This paper presents a VDMJ plugin that automatically translates VDM-SL specifications into Dafny modules. Some VDM specifications are executable, yet there is limited verification support. We translate a significant subset of VDM-SL to Dafny, where users can work to discharge proof obligations. This follows the work from a VDM to Isabelle/HOL translator. This allows VDM access to modern satisfiability modulo theory solvers through Dafny. This paper also uses the StringTemplate4 Java library to produce and format translations. This allows for a minimal code-base and a more directly readable translation template compared to in-code approaches to translation.

Keywords: VDM, Dafny, StringTemplate4, proof support, automation, translation

1 Introduction

This paper describes the translation tool for a subset of the VDM-SL specification language [8] to the Dafny formal verification language [3]. The translation strategies used are syntactically driven, taking into account the VDM-SL corresponding formal grammar. Beyond these translation strategies, this paper aims to identify the points at which automated translation would fail, including situations where manual translation would be better.

Our tool (`vdm2dafny`) is based upon the VDM toolkit `vdm2isa` tool [5]. That is, `vdm2dafny` works similarly and operates within the VDMJ plugin environment [2].

Similarly to VDM, Dafny is a language designed for specification. VDM-SL and Dafny contain specification features, such as type invariants, pre/post conditions, sub-typing, methods, and program statements *etc*. Dafny also includes programming features like objects, and naming scopes — these concepts exist in other VDM dialects but are not the focus of this work, and as such other VDM dialects are left to future extensions.

On the other hand, Dafny is also geared for verification. It includes verification features, where program correctness is guaranteed through Hoare-logic style reasoning, where verification conditions are discharged using SAT/SMT solvers through encodings within the Boogie [9] and Z3 [4] verifiers . Finally, verified Dafny code is directly translatable to C#. For example, Dafny code can be used as a library in C#, which allows for a much more direct path to implementation from a Dafny specification. Dafny is also being used in production by industry like AWS Security services [14].

The combination of these factors lead to the choice of Dafny as the target for this project, as it combines many desirable features for VDM and is a useful target for translation. Given Dafny's object-oriented features, we also anticipate translating VDM++ to Dafny in the future.

1.1 Structure of the paper

In Section 2, we present a background of other translation tools that inspired this work, as well as how string template tools can effect the approach taken to translation. Next, in Section 3, we discuss the approach taken to translation in more detail and outline the adaptations needed to the approach for different strategies and where translations were difficult/impossible. After this in Section 4, we discuss the limitations and capabilities of the tool, as well as how the use of string templates affected the results of the work. Finally, we conclude in Section 5 with a short section of future work and possible improvements to the work that could be made.

2 Background

There are a number of other translators with different translators available within the VDM community. These include UML [11], Isabelle [5], C, and Java [7]. The Java tool specifically has additional functionality built on top of it through the work in [13], which adds semantically correct JML annotations to the Java code, which allows for static verification of the conditions set in VDM-SL. Both the JML work and this aim to produce a statically verifiable translation to a VDM module. However, this work is based on and is an extension of the VDM to Isabelle/HOL translator (`github.com/leouk/VDM_Toolkit`). That translator uses VDMJ's AST infrastructure to generate Isabelle theory files and proof scripts with different proof strategies. The main advantage to using the approach taken when creating the Isabelle translator is that this work is a complete translation of the language rather than an extension of an existing tool.

2.1 String Templates

Differently from the VDM to Isabelle/HOL translator, this work uses the StringTemplate (ST) engine (`stringtemplate.org`)[1] as the output mechanism of our Dafny translation strategy. This is important because it enables a minimal code base (with great reuse) and separation of concerns. For example, there is no need to use VDMJ's class mapping mechanism [1], given all translation templates encode what would otherwise have been Java code over VDMJ's VDM-SL AST.

Our approach follows VDM's BNF grammar in [8, Appendix A]. That is, every covered VDM-SL construct had a corresponding ST encoding in Dafny. This syntactic-driven approach makes the overall translation (and any future extension) modular and easily extensible.

The VDM abstract syntax [8, Appendix A] is formed by modules (or a default nameless module) possibly with imports/exports sections. Each module then contains various definitions consisting of each of the VDM specification constructs, namely: types, values, functions, state, operations, *etc*.

Correspondingly, the Dafny syntax has an almost 1-1 relationship between each of these concepts. For example, Dafny alias types correspond to VDM (re-)named types, whereas Dafny methods/functions correspond to VDM operations/functions. Nevertheless, some VDM constructs are more expressive than Dafny's. For instance, VDM values and case expressions allow for complex patterns, some of which are not available in Dafny.

Finally, ST templates allow for: free-text with placeholder "wildcards" to be instantiated by the source language AST traversal; template functions, where such wildcards can be generalised through template parameters; template comprehensions, where AST structures can be iterated over; and so on. Thus, templates are a mix between target language output and inline "template programs", which enables access to source language AST structures linked with the template. Because the template spacing/newlines will influence the resulting output, they can often be hard to read at first. That is, if you indent the template for readability, the resulting code output is unlikely to be properly formatted.

For example, Listing 1.1 shows the (3) template functions used for translating VDM `if-then-else` expressions to Dafny. Template functions have a name (possibly with parameters), where its definition within `::= « ... »` brackets. The template function `ifthenelse` (lines 1–10) expects an `iexp` object, which corresponds to the VDM AST for `if-then-else` expressions (*e.g.* VDMJ's `TCIfExpression`). First (on line 2), we check whether there are any user-defined comments through a call to the `comment` template function. This also shows how user-defined template programs can be written: a template function call with the same input parameters (line 2).

```
1 comment(obj) ::= <<
2 <obj.comments:{comment|<trim(comment.message)>};separator="\n">
3 >>
4
5 ifthenelse(iexp) ::= <<
```

[1] StringTemplate4 uses the BSD license and only requires reproduction of the copyright notice upon code/binary redistribution.

```
 6 <comment(iexp)>
 7 if <iexp.condition> then
 8     <iexp.then>
 9 <if(iexp.hasElseIfs)>
10 <iexp.elseIfs:{eif|<elseIf(eif)>}; separator="\n">
11 <endif>
12 else
13     <iexp.elseCase>
14 >>
15
16 elseIf(eif) ::= <<
17 <comment(eif)>
18 else if (<eif.condition>) then
19     <eif.then>
20 >>
```

Listing 1.1. VDM to Dafny `if-then-else` expression template snippet

Next, the first piece of Dafny code output appears as the `if-then` keywords, where we output the if-condition from the AST structure given as a parameter, followed by the if-then expressions (lines 3–4). After that, lines 5–7 define a conditional template program: if the VDM AST input (`iexp`) contains any elseif-expressions (*i.e.* VDMJ's `TCElseIfExpressionList`), then process each of them (line 6) using a template comprehension, where each output will be separated by a newline. Template comprehension works just like set comprehension: `eif` is bound to each element within the elseif-expressions list (*i.e.* VDMJ's `TCElseIfExpression`), which is processed by another template function (lines 12–16). Finally, the final (mandatory) else-expression is output directly (lines 8–9). The remainder of the template is similar. The `elseIf` template function (lines 12–16) issues Dafny code for each of the elseif-expressions, whereas the `comment` template function (lines 18–20) defines a template comprehension expressions issuing line-separated Dafny comments for each VDM defined comment (*i.e.* VDMJ's `LexCommentList`). The complete set of template groups for translating VDM to Dafny is available in the VDM Toolkit [6].

3 Translation strategies

Since ST allows for template programs (as opposed to parameterised templates), ST provide a powerful engine to generate target language code without the need to mix AST traversal code with translation-strategy decisions. Effectively, ST enables the encoding of a general translation engine, which provides VDMJ plumbing and AST traversal code, where each template group encodes different translation strategies.

From experience with both languages, we envisaged that a translation strategy between VDM and Dafny will be quite close/direct, given both languages share several similar concepts/constructs.

Our decision to use ST has a significant impact on translation strategies. That is because implicit VDM checks (*e.g.* type invariant checks on entry/exit of functions) can be made explicit as part of the templates. This is different to the VDM to Isabelle translator, which has such checks dealt with within the code base itself. Thus, the use of ST parameterised the Dafny translator by templates, which simplifies the overall code base considerably.

Furthermore, this separation of concerns between the VDM AST visitation/search and the translator output via ST enables change in translation strategy with minimal (or no) change in the code base. This push to use ST started within `vdm2isa`, which already uses ST for top-level module translation. On the other hand, Isabelle is not as close to VDM in features as Dafny, hence full use of ST is considered for future work.

The translation strategies are divided in three categories: syntax-driven, partial, missing, which are described in the next subsections. Each strategy is encoded through various templates groups, according to the VDM AST structure. That is, we have ST template (group `.stg`) files for each VDM top-level syntactic categories (*e.g.* modules, definitions, types, values, functions, expressions, *etc.*). These correspond to VDMJ's AST directly. Each ST template group file corresponding to a

VDM top-level syntax category defines various template functions encoding the expected translation between VDM and Dafny. For example, the `expressions.stg` template contains template functions encoding differences between VDM's and Dafny's expression languages.

3.1 Syntax-driven translation strategy

The syntax-driven translation strategy caters for the majority of the AST, where VDM and Dafny match directly, perhaps with minor adjustments. For instance, Dafny map application expression is just as VDM with uncurried parameters passing, whereas the sequence application has to take into account Dafny's starting from index 0 (instead of VDM's index 1).

For example, VDMJ expressions AST contains just over 100 specific productions. Thus, within the `expressions.stg` template group file we have corresponding template functions for all those expression ASTs which have a direct translation. In practice, that corresponds to 35 template functions, which caters for about 90% of the VDM expressions AST, with total coverage being documented in [15, Appendix A]. The number of template functions is fewer because we parameterise unary/binary expressions with inputs and their corresponding operators. . The same strategy applies to other VDM AST parts like imports/exports, functions, types, *etc*.

VDM types (and their invariants) have a direct correspondence in Dafny, except around ordering predicates and other examples discussed below. . Like VDM, Dafny already implicitly check type invariants everywhere (*e.g.* value assignments, function inputs/outputs, *etc.*).

Another interesting example of syntactic-driven translation that is greatly simplified through the use of templates is that of types. A snippet of alias type declaration template is defined in Listing 1.2

```
alias(type) ::= <<
//@vdm.type(<type.name>)
type <type.name> = <if(type.hasInv)><type.var>: <type.alias>
|   <type.invCall>
<else><type.alias><endif>

<if(type.hasInv)><invariant(<type.inv>)><endif>
...
>>

invariant(inv) ::= <<
//@vdm.function(<inv.name>)
predicate <inv.name>(<inv.param.name>: <inv.implType>) { <inv.body> }
>>
...
```

Listing 1.2. VDM to Dafny function declaration template snippet

To define the string template for an alias type shown in 1.2, there are some unique considerations that need to be made. In this case, two templates are defined, where the invariant is defined separately as a predicate function. In this case, two Dafny language constructs are condensed into the same template using the `if` tag; this checks if the type has an invariant, and if it does it uses the syntax for a Dafny subset type. Otherwise, it uses the syntax for an alias type. It is worth noting that the invariant template rendering is repeated for all of the implicitly defined type methods (i.e. `ord`, `eq`, `min`, and `max`).

```
T = nat
inv t == t < 10;
```

Listing 1.3. VDM input example

```
//@vdm.type(T)
type T = varT: nat
|   inv_T(varT)

//@vdm.function(inv_T)
predicate inv_T(t: nat) { (t < 10) }
```

Listing 1.4. Dafny output

Concretely, for the VDM input in Listing 1.3, we get the Dafny output in Listing 1.4. The VDM explicit function is straightforward. The corresponding Dafny function implicitly checks input/result type invariants (in this case just ≥ 0 for `nat`) and issues corresponding pre/post (as requires/ensures) predicates. For the post condition, the template ensures that variable `RESULT` is available in Dafny.

The template ensures that pre/post conditions (*e.g.* `pre/post_f`) are issued as Dafny ghost predicates (*i.e.* a boolean valued side-effect free function that is specification-only and will not produce any C# code).

3.2 Partial translation strategy

For some VDM constructs we only have a partial translation strategy. This is due to differences between the languages, where some form of context environment is required to decide what the target translation should look like. That is, they relate to syntactic categories requiring more elaborate translation strategies.

3.2.1 Alias Types VDM named types have a Dafny equivalent in synonym types; this makes the translation of this construct simple. Despite this, some additional features are included in the translation. The adopted strategy translates these types in phases. Firstly, the VDM named type is translated as an unbounded Dafny synonym type. This is then bounded with a subset type — which can define the bounds for the type using an invariant predicate. The biggest limitation from this is the requirement for witness values, which are not automatically required/generated by VDM. Currently, there is no support for a custom annotation for witnesses in this work; but this would be a good way to improve the translation strategies for invariant types.

Furthermore, VDM named types allow definition of three custom predicates: i) a type invariant, which limits the space of values for a type (*e.g.* type `nat` is the same as `int`, where values are ≥ 0); ii) a type ordering predicate, which establishes a strict/total order relationship between values of that type (*e.g.* akin to `int` "$<$"-order for any type); and iii) a type equality predicate, which is used in conjunction with type ordering for total order definition.

```
1  LangManOrderingProof(name) ::= <<
2  // Prove correct irreflexive, transitive ordering of the ord and eq clauses.
3  lemma <name>__Ordering()
4  {
5      assert vdm.Transitive(ord_<name>)
6      assert vdm.Irreflexive(ord_<name>)
7      assert vdm.EqualityOperationsCorrect(ord_<name>, eq_<name>)
8  }
9  ...
10 >>
```

Listing 1.5. By defining predicates in the helper Dafny module. A lemma can be easily defined in the template to check correctness for all translations.

When these predicates are defined by the user, additional functions are implicitly defined within the VDM specification — these can be called from other areas of code. Dafny does not natively support calling another type's bounds; and does not natively support the custom ordering or equality that VDM has. To address this, these custom methods are translated separately and called when they would be used in the code (*e.g.* $A < B$ for A, B of type $Type$ is replaced with a call to $ord\_Type(A, B)$). These specification predicates are defined implicitly in Dafny as ghost predicates.

By translating these clauses into Dafny, additional specifications can be included automatically to aid the Dafny automatic prover. As shown in Listing 1.5, Dafny lemmas can be used to automatically prove whether the defined order and equality clauses satisfy the requirements defined by VDM [8, pg. 36]. Further, this could aid in the debugging complicated ordering cases which allows for an increased ease of debugging issues in the expression.

3.2.2 Union Types VDM was one of the first languages to introduce union types and pattern matching. Thus, these constructs in VDM are the hardest to translate to any language. Dafny represents a union type as a set of uniquely identified constructors, each of which hold a different set of named parameters, whereas VDM represents a union type similarly to an alias for multiple targets.

For the Dafny translation of VDM union types, we follow the same approach (and suffer similar limitations) taken by the `vdm2isa` translation strategy for Isabelle/HOL. That is, complex union types have to be traversed prior to translation in order to identify the necessary constructors needed for Dafny. For example, Listings 1.6, and 1.7 shows how we translate complex VDM union types to Dafny. It is worth noting that the translations of the types `T1, T2` are simplified. These would use their respective strategies in actual implementation.

```
types
 //Witness annotations have name + value
 T1 = nat inv n == n < 10;
 T2 = real inv r == r > 5;

 Union = T1 | T2 | <Literal>;
```

Listing 1.6. VDM complex union type

```
//@vdm.type('T1')
newtype T1 = n: nat | n < 10
    // witness *
//@vdm.type('T2')
newtype T2 = r: real | r < 10
    // witness *

datatype Union = T1(t1: T1) |
    T2(t2: T2) | Literal
```

Listing 1.7. Dafny union type output

There were a few strategies that were initially considered for this feature, but eventually, we settled on the strategy shown in Listings 1.6, and 1.7. However this does have some downsides; for example, it would require a transformation on base type names to allow them to be valid constructor identifiers in Dafny. Another key downside that is unique to this kind of project is that it produces significantly more verbose code than a hand-written example. Normally, this would be no issue as the results of a translation are not normally read by a human — but considering this is translating a specification, it is very likely that it would need to be read and modified at some point. This poses a disadvantage in the current implementation of translation strategies that would need to be improved in future work.

Regardless, there still exists a key issue in the translation of union types in the project in how VDMJ will automatically narrow a union type once it discovers which type it is meant to be. As this project relies upon the VDMJ compiler's type-checking, and the context provided by these type-checking objects, this automatic narrowing loses some of the required information for translation. The proposed fix for this is to produce a system to track the context of variables and their defined type; which will allow for more accurate, and concise translations.

3.2.3 Functions and Operations While functions and operations do warrant differing implementations, there is a significant overlap in the structure and syntax when creating a function and operation. These both define a signature with a number of parameters and a return type, optional additional specification bodies for preconditions and postconditions, and a body. The only significant difference between the two definitions is the body, as an operation's body will be a statement; while a function's body will be a function. There are other differences between the two in practice, but for the task of translation, these are the only meaningful features.

Dafny also has the same separation between functions and operations — which are called methods in Dafny; this makes a basic translation strategy for these simple as there is a one-to-one relationship between each component of VDM and Dafny. 'Preconditions' and 'postconditions' are translated to 'requires' and 'ensures' clauses; the 'measure' clause to the 'decreases' clause; 'externals' clauses are translated to both the 'reads' and 'modifies' clauses in Dafny depending upon context.

Initially, this seemed to be a sufficient strategy to translate a single function/operation. However, there are cases in VDM where a function/operation's constraints are called outside of the definition's scope. This is typically done using the 'pre_' and 'post_' prefixes; Dafny does not permit this natively, as such a change to the translation strategy had to be made to independently define preconditions and postconditions as a separate definition in Dafny which are then called from the re-

quires/ensures clauses in the main strategy. This extension to the base strategy allows us to preserve this part of VDM which is not natively accepted in Dafny.

Finally, another slight difference between the VDM and Dafny is that VDM allows for a parameter to be any type of pattern. The strategy employed to translate these cases is to assign a dummy identifier to the pattern, and then attempt to reconcile the parameter in the body via a pattern assignment — this is not a perfect strategy, as there are some patterns which are not available to use in Dafny pattern assignments. Currently, this subset of translations is not possible; and an error is raised in the VDMJ console when this translation is attempted.

3.2.4 Cases/Match Expressions To translate the body of functions and operations, there were a number of challenges that needed to be overcome concerning specific expressions and statements.

There are a collection of language features that cause issues in a number of locations. The first of these are patterns, Dafny has much more constrained usage of patterns compared to VDM, which allows for any pattern to be used anywhere — for example, as previously mentioned in the parameters of a top level definition; in the 'cases' expression/statement; and in variable assignments. On the other hand, Dafny does not allow patterns at all in top level definitions. It is also much stricter in which patterns can be used in its 'match' expression/statement; and in variable definitions.

Specifically, matches in Dafny only allow for the Dafny equivalent of the identifier pattern, the record pattern, as well as literal patterns. This leaves out a considerable amount of the use cases for patterns in VDM and is a considerable limitation when translating a VDM specification using this tool.

3.2.5 Set Range Expressions There were also cases in which a translation strategy had to be devised for a language feature that does not exist in Dafny natively, but is implementable using a combination of other language features. One of these features was set ranges, which, in VDM allow the lower and upper bound of a numeric set to be defined which is then filled with all values in between the two bounds. This is implemented in Dafny as a set comprehension which is able to produce an ideal result.

```
1 { x * 2 | x in set {1, ..., 5}}
```

Listing 1.8. VDM set comprehension with a set range

```
1  // Automatic translation
2  set x
3      |   x in (set dummy
4          | 1 <= dummy <= 5 :: dummy)
5      ::  x * 2
6
7  // Better translation
8  set x
9      |   1 <= x <= 5
10     ::  x * 2
```

Listing 1.9. Dafny set comprehension with a set range as a bound

However, this implementation does carry unforeseen problems as it can obfuscate the actual meaning of the specification. Even in simple examples as shown in Listings 1.8, and 1.9, the automatically translated specification produces an unnecessary comprehension which is detrimental to both the readability of the specification and the performance. Ideally, the translation tool would be able to make this kind of observation and dynamically switch the translation strategy if necessary, but that is far beyond the scope of this project.

3.2.6 Quantified Expressions While some quantified expressions exist in both VDM and Dafny, there are expressions which only exist in VDM. These being the exists1 expression, and the iota expression. Translations for both of these expressions have been encoded into Dafny. However, the method to translate these quantified expressions is unique compared to the previously mentioned translation strategies, as it was the first expression which could be encoded easily in to Dafny, but would require an additional helper function in Dafny to encode. This lead to the creation of a VDM tooling library in Dafny which would be imported automatically at the top of translated VDM files

using an include directive. Overall, this makes the translation strategy succinct as it only requires calling the defined predicate with the required bind and predicate as parameters.

```
exists1 i in set {1, 2, 3, 4} & i > 3
```

Listing 1.10. VDM set comprehension with a set enumeration

```
Exists1.Set({1, 2, 3, 4}, i => i > 3).
    check()
```

Listing 1.11. Dafny output from the translation

One important note for this specific translation that can be noticed is that it does not completely follow the approach of syntax-driven translation strategies as it discounts any Dafny grammar by producing a Dafny helper function and calling it. In any case, it cannot be expected that it will be possible to find analogous parts for 100% of both languages, and there may be some constructs which can only be translated in a one-sided manner by looking at the source grammar.

3.2.7 User-defined Comparisons There are cases in VDM where the default functionality for an operator is overridden by a user-defined predicate. This occurs when the 'ord', or 'eq' clause is included when creating a type. Similarly to the approach for quantified expressions, it is impossible to fully implement the syntax-driven approach as Dafny does not support overriding the functionality of its symbols as VDM does. However, the final approach for translating these comparisons is simple, as the binary expressions are encoded as predicates in Dafny which are then called when the translator detects that the comparison expression refers to a type which overrides the native comparison methods.

3.2.8 Quotes Quote types in VDM are used similarly to enumerated types. In Dafny, these can be represented through a blank constructor in a datatype definition. Using this implementation, an amount of the native VDM capabilities can be translated directly into Dafny, and the differences in how the common uses are translated are shown in Listings 1.12, and 1.13. However, there are some limitations in the Dafny implementation of quotes which cannot be avoided. The first of these is that a quote type must be assigned to a type before it is referenced — though this is a limitation that would be rarely encountered, as it only limits examples where a quote type has not been defined properly — which would be uncommon.

```
f(variable: Quote) r:bool ==
    variable = <QUOTE> or
    variable in set {<QUOTE1>, <QUOTE2>};
```

Listing 1.12. Common VDM expressions on a quote type

```
predicate f(variable: Quote) {
    ((variable.QUOTE?) ||
    (variable in {QUOTE1, QUOTE2}))
}
```

Listing 1.13. Dafny output from the translation

However, the other problem that comes with defining the quotes as constructors in the datatypes is that it removes the possibility of comparing across datatypes. This limitation is demonstrated in Listings 1.14, and 1.15

```
Nation = <England> | <France>;
Flag = <England> | <France>
...
isNationFlag: Nation * Flag -> bool
isNationFlag(n, f) == n = f;
```

Listing 1.14. Issues that arise in translation

```
datatype Nation = England | France
datatype Flag = England | France

predicate isNationFlag(n: Nation, f: Flag)
      {
  n == f // Error raised as Nation and
       Flag are incomparable.
}
```

Listing 1.15. Dafny output from the translation

3.2.9 Automated Conversion of Types This project heavily relies on the VDMJ compiler's type-checking capabilities to make its translations, and for the most part this approach works. However,

there are some situations where this reliance works against the translation. One of these situations is shown in Listings 1.16, and 1.17. This issue arises because when VDMJ evaluates the else branch of the statement it has already narrowed the union type to be an integer. At the same time, Dafny requires this narrowing to be done manually; but since VDMJ does not provide the context that the variable is a union type the translator will struggle to produce a fully valid translation for these cases.

The most promising proposed solution to this problem is to produce an additional system within the translator that can determine the context in which a variable is referenced. This would allow the translator to determine the maximal type of the variable and produce accurate field expressions to narrow towards the proper translation. This approach will likely be the best way to move forward with resolving these issues, but will come with its own challenges that will need to be overcome, as such it was not feasible to implement this in the timeframe allotted for this project.

```
AliasType = int;
Union = <Quote> | AliasType
functions
example: Union -> int
example(u) ==
    if u = <Quote> then 0
    else u;
```

Listing 1.16. Situation in which VDMJ automatically narrows an expression

```
type AliasType = int
datatype Union = Quote | AliasType(
     AliasType: AliasType)

function example(u: Union): int {
  if u.Quote? then 0
  // This should be 'else u.AliasType'
  else u
}
```

Listing 1.17. Dafny output from the translation

3.2.10 Indexed For Statements While strategies for 'indexed for' statements have been devised in this project, there are still features which Dafny cannot fully implement. The most prevalent of these being that for loops in VDM have an optional '`step`' parameter which controls the steps that the for loop takes — Dafny does not have this, instead it increments or decrements by one depending on whether the '`to`' or '`downto`' tokens are used in the loop statements. Currently, the step parameter is discounted completely; but as part of further work only '1' and '-1' could be accepted as possible values which would dictate whether to use '`to`' or '`downto`'.

3.3 Missing translations

3.3.1 State This project leaves state to further work, but as part of investigations, several initially promising implementations were attempted. As there is no native implementation of a global state in Dafny, there was significant challenges in discovering strategies that could be used. The first strategy attempted was to produce a global variable which was bound to a custom state type which would handle the invariants. This approach would work if Dafny allowed for global variables. However, global values in Dafny are immutable, and as such this approach was not feasible.

```
state MyState of
    idx: int
init s == s = mk_MyState(0)
end;

operations

main() == (
    idx := idx + 1
)
ext wr idx
```

Listing 1.18. VDM implementation of a simple state

```
class MyState {
    var idx: int
    constructor Init() {
        this.idx := 0;
    }
    method main()
        modifies this`idx
    {
        idx := idx + 1;
    }
}

method main() {
    var s: MyState := new MyState.Init();
    s.main();
}
```

Listing 1.19. A possible Dafny translation of VDM state

Overall, there was no good implementation for state discovered in

3.3.2 Set & Sequence Comprehensions The syntax for VDM set comprehensions is significantly different than the Dafny syntax. This makes these language features, despite having similar purposes, tricky to implement. VDM allows for a defined lower and upper bound for the comprehension through a set range expression, while Dafny only allows for the upper bound to be defined — which determines the size of the output sequence. This would mean that expressions which reference the index varaible in VDM would need to be modified correctly in the Dafny code. This project does not tackle this, but it could be implemented by adding the lower bound value to the translation result of the expression. Similarly to the possible improvements for field expressions discussed previously; this would likely require for additional context to be passed to the body expression of the sequence comprehension — as such, this could be implemented in tandem with the field expression improvements.

3.3.3 Non-deterministic & Error Statement VDM has some special statements which are not compatible with Dafny's language philosophy. The first of these is the non-deterministic statement, which operates similarly to a block statement, but the order of execution of the statements is randomly determined — as Dafny does not allow for randomness, this is not possible to translate.

Another of these statements is the VDM error statement — while Dafny does have fail cases, they operate in a completely different way as they require the return type of a function to be 'failure compatible', which is not a concept that is present in VDM which makes this statement impossible to directly translate.

3.3.4 Imports and exports Both languages handle importing from external libraries slightly differently. Dafny requires the external module to be referenced in an include directive before including. Whereas, VDM-SL only requires the module to be within the compiler environment. This can be handled by pointing to the module in the `.generated/dafny/*` directory, which is where the tool will save the generated Dafny modules.

While the previous problem is easy to solve, the larger issue is how Dafny handles exports. Dafny requires exports to be defined as part of a named set. VDM, on the other hand, only requires the exported item to be named. The approach that was settled on in this tool was to create a new export set for each exported function; this keeps the same form of imports for VDM which makes the import translation consistent between the languages. However, this is likely not the best implementation of export sets, as such if the result will be modified it is recommended that export sets and imports are properly defined for the translated module.

4 Discussion

As part of our strategy for translation, we initially looked to encode Dafny examples into VDM, this helped give an idea of what kinds of patterns were typically used in properly written Dafny, as well as best practices for translation. There were a few preliminary rules to discover strategies in this way. First, each translation made must require as little thought as possible — this would remove a lot of complexity from the task and allow for a greater understanding of what constitutes an easily translatable AST node. This exercise was completed using the examples in the Program Proofs book [10], which was of great help in providing well-written, simple, and well-documented Dafny examples.

Based on this preliminary approach, we selected a pragmatic view of what to focus on translating. For this exercise, translating functions, and expressions was deemed to be the most important — as VDM operations and Dafny methods are used primarily to encode functionality rather than specification, and the best approach for this can vary greatly between the two languages. This lead to a lot more effort being put into encoding difficult expressions rather than difficult statements. This limitation of scope was necessary considering this project was done as an MComp research project [15] which had a very limited time allowance. A greater scope could have been applied if more time was allowed.

4.1 Proof Capabilities

With Dafny's SMT solvers, the proof process of code can be significantly expedited. An extension to this tool could allow it to pull from VDMJ's generated proof obligations for automated proofs in Dafny. This is one of the key reasons that Dafny was chosen as a target for this tool. On top of this, using string templates can prove to be powerful when implementing automatic proof capabilities into the translation tool. For example, the VDM language manual states conditions for ordering functions [8, 3.5 Order], these conditions can be used to automatically produce a lemma in the translation to discharge these conditions. Furthermore, the Dafny verifier describes exactly which property is violated when verifying the ordering functions.

Overall, the proof capabilities of the vdm2dafny tool are promising — however, the completeness of the translation is directly linked to these capabilities. With the limitations of patterns in constructs such as cases and assignments which are used considerably in VDM, there can be concern regarding how well this tool can be used over a larger project, considering these points specifically would require a rewrite of what could be some important functions/operations.

4.2 String Templates

Through the use of string templates, there was a lot more freedom to change the translation strategies. Compared to other approaches where the program strings would be built entirely in code, this approach allowed for a rapid iteration of the translations. This specific choice was very useful when implementing many rules through the syntax-driven approach, as the grammar rules could be written in a one-to-one manner through the templating language, and then the back-end only needed to supply data in the proper shape for the template. However, this wasn't only useful in terms of ease of production of the translation strategies. There were also cases when VDM had additional rules that could not be qualified in the BNF syntax rules. For example, the invariant function of a type could be called anywhere using the `'inv_Type(...)'` function call, which was not a native feature in Dafny. To allow for this, additional predicates could be defined in the translation strategy for types which would allow for the invariants to be called from anywhere in Dafny.

This quirk of VDM changed some of the translation strategies away from being completely transcribed BNF rules, as such the template files produced in this project would likely not all be useful for generically producing a Dafny translation. Despite this, as mentioned previously the nature of using template files for the translation strategies allows for changes to be made to the translation quickly and easily — which should reduce the impact of this limitation.

4.3 Class Mapping

One of the primary drawbacks of the overall approach taken in this project was the lack of usage of the native class mapping capabilities of VDMJ, which would have allowed for simple mappings to be made between type checker objects and translation objects. This would have helped to make the tool more general, and ensure complete coverage over all language features. The justification against using it for this project specifically was the worries regarding the time constraints — starting this project there was a need to learn more about the target languages, the VDMJ compiler which was used to interface with VDM, and string templates which were used to produce the translation strategies. Ultimately, class mapping was deemed to be a lower priority than these aspects of the tool; but definitely would have been a good inclusion if more time was allowed.

5 Conclusion

This extends VDM with further proof/verification support through Dafny's use of program verifiers (*i.e.* Boogie [9] and Z3 [4]) with Boogie supporting various SMT solvers [12]. Given Dafny is already close enough to VDM, translation strategies are relatively simple. This also enables certain executable VDM specifications to be translated to C# via Dafny.

Future work. Due to the constraints of this project as a short individual dissertation project, some features are only partially completed. We envisage first extending the translation strategy to cater for key VDM-SL constructs like state. It is also possible to translate some aspects of VDM++. To resolve some of the partial translation issues like union types, a context-building system that understands what an AST will look like before traversing would be useful — this could be included as an extension to the `exu` tool, which warns/errs on such situations for the `vdm2isa` plugin already.

To further make use of the proof capabilities of Dafny, it would be good to produce an additional Dafny file to contain automatically produced lemmas. This would use the proof obligation generation from VDMJ to determine what needs proving. This could significantly aid the proof process as there are cases where Dafny can automatically discharge the proofs without needing any additional help.

References

1. Nick Battle. Analysis Separation without Visitors. https://www.researchgate.net/publication/321018670_Analysis_Separation_without_Visitors, Sep 2017.
2. Nick Battle. VDMJ Compiler for VDM. https://github.com/nickbattle/vdmj, Feb 2024.
3. Dafny-lang Community. Dafny reference manual. https://dafny.org/latest/DafnyRef/DafnyRef, 2024. Chapter 17.
4. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
5. Leo Freitas. Topologically sorting vdm-sl definitions for isabelle/hol translation. https://arxiv.org/abs/2304.15006, 2023.
6. Leo Freitas. VDM Toolkit. https://github.com/leouk/VDM_Toolkit, 2024.
7. Peter Gorm Larsen, Kenneth Lausdahl, Peter Tran-Jørgensen, Joey Coleman, Sune Wolff, Luıs Diogo Couto, and Victor Bandur. Overture VDM-10 Tool Support: User Guide. https://raw.githubusercontent.com/overturetool/documentation/editing/documentation/UserGuideOvertureIDE/OvertureIDEUserGuide.pdf, May 2019.
8. Peter Gorm Larsen, Kenneth Guldbrandt Lausdahl, and Nick Battle. VDM-10 Language Manual. https://raw.githubusercontent.com/overturetool/documentation/editing/documentation/VDM10LangMan/VDM10_lang_man.pdf, 2022.
9. K. Rustan M. Leino. This is Boogie 2. https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/, June 2008.
10. K. Rustan M. Leino. *Program Proofs*. The MIT Press, Mar 2023.
11. Jonas Lund, Lucas Jensen, Nick Battle, Peter Larsen, and Hugo Macedo. Bidirectional UML Visualisation of VDM Models, Apr 2023.
12. RiSE Group, Microsoft Research. Boogie documentation on supported SMT sovlers. https://github.com/boogie-org/boogie/blob/master/README.md#backend-smt-solver.
13. Peter W. Tran-Jørgensen, Peter Gorm Larsen, and Gary T. Leavens. Automated translation of VDM to JML-annotated Java. *Int. J. Softw. Tools Technol. Transf.*, 20(2):211–235, apr 2018.
14. Lucas Wagner and Sean McLauglin. Proving the correctness of AWS authorization. https://www.youtube.com/watch?v=oshxAJGrwMU, 2024. AWS re:Inforce.
15. Adam Winstanley. VDM2Dafny: An Automated Translation Tool for VDM-SL to Dafny, 2024.