VDM2Dafny: An Automated Translation Tool for VDMSL to Dafny

Adam Winstanley

School of Computing Science, Newcastle University, UK

Abstract

The aim of this project is to produce a plugin for VDMJ which will automatically translate loaded VDMSL modules into Dafny modules. This project tackles a subset of both VDM and Dafny, and will miss certain aspects of both languages due to incompatibilities in the languages. This will detail the decisions made in the translations and give a rationale for more subjective design decisions.

Keywords: Translation, VDMSL, Dafny, VDMJ plugin.

AW: I think that the main thing that I want to focus on is the overall approach, with more specific detail being given on the problems that I encountered.

1 Introduction

Why VDM and why Dafny?

- VDM is an old language, and it has many powerful features. But due to it being an old language, users may want to translate to more modern tools which have richer features in language translation. Dafny can be translated directly into C#, which is an incredibly powerful modern language.

- Production of nice VDM features in Dafny. As mentioned, VDM has many powerful features, and in translating this project. A toolkit will be produced to aid in the translation between languages — this toolkit will contain implementations of these powerful features in Dafny. These may be useful to someone who is modelling in native Dafny.

- The main purpose of this exercise is to extend VDM to Dafny, which will by proxy aid in the implementations of VDM models into newer, more commonplace languages.

[1] Email: a.winstanley2@newcastle.ac.uk

2 Background

2.1 VDM

- General background on VDM, what it is, when/why it is used.
  - The need for translation in a language like VDM.
  - General challenges in translation

2.2 Dafny

- General background on Dafny, what it is, when/why it is used.
  - Existing translation tools from Dafny to other languages, this is the main reason that Dafny is the targeted language for this project.
  - Identify missing language features from Dafny that are in VDM. Identify missing features in VDM that are in Dafny. This will be used to define the subset of features that can be translated fully, those that are translated partially, and those that are not translatable.
  - Use Programming Proofs as a source for this.

2.3 String Templates

- Overview of the need for string templating, why is it better than producing recipes in code?
  - Overview of the syntax of StringTemplates4 and whether or not it is suitable enough for the goal.

2.4 VDM2Isa

- Existing translation tool from VDM to another language.
  - Analysis of the approach taken.
  - Analysis of the challenges faced in this, are these challenges expected to be the same in this project?

3 What you did and how

Define the approach. First develop a VDM implementation of a Dafny program; this will be provided from the book on program proofs.

3.1 Manual Translation

3.1.1 Selecting an example

- Philosophy in selecting a manual translation example. Must be a valid and well-written piece of Dafny code; must be able to be translated in a ”dumb” way. I.E. no clever tricks to make it easier. Everything must be consistently reversible.

3.1.2 Thoughts on translation

- Talk about types, specifically how the general ideas behind type derivation in Dafny and VDM programs seems to differ.

- Talk about how the lemmas from dafny were translated. Evaluate whether this is a viable strategy for the project.

3.2 Possible problems and approaches

3.2.1 VDM features

There are numerous VDM features that are unavailable in Dafny. This subsection will focus on the specific functions within VDM that are unavailable in Dafny; and how this was remedied.

To start, I went through both the VDM language manual [4] and the Dafny reference manual [2] to find the disparities in the language. These can be generally sorted into two main categories; the first are features which are not compatible at all, which includes problems in pattern matching — which are detailed in 3.2.2, as well as the lack of certain language features such as sequence comprehensions.

The other category, which can be tackled, are certain language features which are not available in Dafny, but can be encoded. The generic sequence and set types can be tricky to encode into Dafny however. As the types that need implementing are the "non-empty" types (seq1, set1); Dafny requires a witness value which is not possible to provide when encoding these generally, these encoded types are shown in Figure 1.

```
1 //@vdm.type('nat1')
2 newtype nat1 = n: nat | n > 0 witness 1
3 //@vdm.type('seq1 of T')
4 type seq1<T> = s: seq<T> | |s| > 0 witness *
5 //@vdm.type('set1 of T')
6 type set1<T(==)> = s: set<T> | |s| > 0 witness *
7
8 // Recommendation
9 type set1ofint = s: set<int> | |s| > 0 witness {1}
```

Fig. 1. This figure shows the types in VDM as they are encoded into the Dafny helper file. It can clearly be seen that the witness for nat1 is trivial, but providing a witness for the other types is not possible. As such, they have been given the wildcard as their witness. This approach is poor for verification, and it is recommended that this type should be extended with a proper witness value if heavy use is made of either of these collection types.

A few of the language features that initially seem possible to encode are not as simple as they seem when attempting to encode them into Dafny. The most used features which fall under this category are the distributed set expressions (union, intersection), as well as VDM's iota expression. In the case of distributed union and intersection; Dafny is unable to determine bounds for the distribution; and thus cannot prove that the result is not infinite, two initially promising translations of the "duniuon" expression are listed in Figure 2, as well as the VDMJ implementation of the distributed union.

3.2.2 Patterns

One of the main strengths of VDM as a language is it's powerful patterning capabilities; patterns can be placed into any definition and the pattern matcher will match the pattern automatically. Dafny does not have the same level of expression that VDM does with it's pattern matching — this poses a great problem in translation strategies. As pattern matching expressions will be incredibly tricky to translate directly, and in some cases may not be translatable at all.

One approach that I have taken to reduce the cases wherein translation is difficult is to attempt to 'reconcile' the pattern matching at a later point in the called function/operation.

```
// Excerpt from the dunion interpretation code in VDMJ
ValueSet setset = this.exp.eval(ctxt).setValue(ctxt);
ValueSet result = new ValueSet();
Iterator iter = setset.iterator();

while(iter.hasNext()) {
    Value v = (Value)iter.next();
    result.addAll(v.setValue(ctxt));
}

return new SetValue(result);
```

```
// The VDMJ code seems like it would translate nicely into this recursive Dafny
    function.
function dunion<Value>(setset: set<set<Value>>): set<Value> {
    if s == {} then {}
    else
        var v: set<Value> :| v in setset;
        v + dunion(setset - v)
}
```

```
// A seperate approach using set comprehensions
function dunion<Value>(setset: set<set<Value>>): set<Value> {
    set v: Value | (exists s: set<Value> :: v in s) :: v
}
```

Fig. 2. This figure shows the underlying implementation of the distributed union expression in Java. The iteration approach cannot work in a Dafny function as a function must be deterministic. As such, the specification must determine which value to get from the set deterministically, which is not possible to do over the generic type $Value$. Another attempted approach was to use set comprehensions, however, this is also not possible due to Dafny guarding against possibly infinite sets. As Dafny cannot prove that the result is non-infinite, it cannot produce the distributed union in a generic way.

```
patternsInParameters: seq of (int * int) -> ?
patternsInParameters([mk_(a, b)] ^ c) == is not yet specified;
```

Fig. 3. This shows the unique way that definitions can be made to include patterns in VDM, this would be difficult to translate directly into Dafny.

This approach is valid when the patterns are defined in the definition of the function; as shown in Figure 3. The approach of reconcilation would replace the pattern in the definition with a dummy argument, and then attempt a pattern matching expression as the first part of the function.

```
function patternsInParameters(arg0: seq<(int, int)>): ? {
    var a:int, b:int, c:seq<(int, int)> :| arg0 == [(a, b)] + c;
    ?
}
```

Fig. 4. This shows the process of "reconciliation" of the patterns, using the built-in dafny pattern matching expression. Unfortunately, the produced Dafny code in the example is not valid as Dafny is unable to determine valid values for a, b, and c. Therefore, when the program reconciles a pattern in the parameters of a function, it will warn the user that it may not produce valid Dafny code. But the strategy is kept as this type of hinting at what each variable needs to be set to is very useful in manually fixing the translation to a proper translation.

This approach meets the criteria of being easily reproducible on any provided VDM function — as the patterns themselves should be able to be translated 1-1 into Dafny patterns.

3.2.3 Cases

This problem is related in part to the patterning issue mentioned in Section 3.2.2. In

```
sum: seq of int -> int
sum(s) ==
cases s:
    [v] -> v,
    [v] ^ rest -> v + sum(rest)
measure len s;
```

Fig. 5. Simple sum function written in VDM using switch case expressions with sequence pattern matching. When the sequence has a single element, it will return that element, otherwise it will recursively add each element to a sum.

```
1  // direct translation
2  function sum(s: seq<int>): int
3      decreases |s|
4  {
5      //@vdm.warning(This type of pattern is not supported in Dafny, you will need
           to manually translate the problematic cases into proper Dafny.)
6      match s
7          case [v] => v
8          case [v] + rest  => v + sum(rest)
9  }
10
11 // proper translation
12 function sum(s: seq<int>): int
13     decreases |s|
14 {
15     if (s == []) then 0
16     else s[0] + sum(s[1..])
17 }
```

Fig. 6. A direct translation from VDM to Dafny, this does not work as Dafny does not support direct sequence match expressions. The proper translation of the function to VDM requires considerably more thought than a computer is capable of — this should become clear as more complex case statements are considered, as this statement is comparably simple.

VDM, cases can check for any valid pattern; this is vastly more powerful than Dafny's match expressions, which can only check for literal values, simple identifiers, or type constructors. This is significantly less powerful than the VDM implementation of case patterns; and poses a problem for the feasibility of a full translation with no requirement for human intervention.

To this end, rather than attempting a proper translation for case statements as shown in line 11 of Figure 6; it would likely be best to directly translate invalid patterns with an error that it will not produce valid Dafny code, and will need human intervention (shown in Figure 6). This would ensure that when using the tool, the user would have to opt-in to receiving possibly broken translations to see this issue.

3.2.4 Types

```
1  type TypeName = varName: AliasType | inv_TypeName(varName)
2
3  predicate inv_TypeName(varName: AliasType) {
4      // predicate body
5  }
6  // ... and so on for each (eq, ord, max, min) definition in the type.
```

Fig. 7. This is an example of an alias type pattern, this should be a simple. This is also the same pattern for product types, where the 'AliasType' is replaced with a tuple type.

As can be seen in Figures 7, and 8; three of the four cases for VDM types are trivial to translate — which are synonym, product, and record types.

However, union types provide a much more difficult challenge — there is no catering for

```
// Bounded record type
type RecordName = varName: ImplRecordName'
    | inv_RecordName(varName) // Apply bind to record
// Unbounded record type
datatype ImplRecordName' = mk_RecordName(...)

predicate inv_RecordName(v: ImplRecordName') {
    // predicate body
}
// ... and so on for each (eq, ord, max, min) definition in the type.
```

Fig. 8. Record types are produced using a single constructor of an unbounded datatype. This is then attached to a bound subset type where the type invariant is then called over the invariant type. One possible issue in the automated translation is that necessary witness values for subset types are not produced; this means that for each subset type that is produced by the translation, the user will need to produce a witness to prove that it is not an invalid subset.

the possibility of union types in Dafny; with the only real possibility being using multiple constructors in a datatype. A lot of thought is needed when implementing these translation strategies to ensure that a maximal amount of the language is captured. Ultimately, the approach that I took captures a considerable amount of the uses of the language; but it does fail to capture a number of possible cases.

```
-- Cannot translate literal unions well, so they are left to future work, however,
     if these are aliased into another type then they will work.
LiteralUnion = char | int;

Char = char;
Integer = int;
SupportedLiteralUnion = Char | Integer;

-- Field expressions become difficult with this, these are still translated, but a
     warning is produced for field expressions
Record::     v: int;
Record2::    v: int;
RecordUnion = Record | Record2;

-- Quote unions are supported, but comparisons with other quotes are not. I.E.
    only 'quoteVariable = <QUOTE>' is supported, not 'quoteVariable1 =
    quoteVariable2'.
QuoteUnion = <QUOTE1> | <QUOTE2>;

-- Combinations of valid union types are supported.
ComboUnion = <QUOTE> | Record | QuoteUnion | SupportedLiteralUnion;
```

```
// Dafny translation strategy for unions and quotes.
// These can be any type other than literals.
type UnionType = ?
type UnionType2 = ?

datatype Union = UnionType(UnionType: UnionType) | UnionType2(UnionType2:
    UnionType2) | QuoteLiteral
```

Fig. 9. This figure shows different general styles of producing union types, and explains whether or not they are supported by the tool. This also shows the selected dafny implementations of the supported union types.

The selected translation strategy allows for a limited implementation of the possibilities that are available in VDM. These narrower available cases are shown in Figure 9.

The decision to leave out unions of literals could likely be mitigated by transforming the literal name when producing the union translations in Dafny. Though this would require a lot more context on the type of each expression that involves these union types as an additional field expression would need to be added to access the correct typing of the expression. This issue exists for the unions of record types as well, and the reasons behind the problem and possible resolutions are discussed in further detail in Section 3.4.6.

Literals were not the only part of VDM that needs to be narrowed in this situation. Since quote literals do not exist in Dafny, they are implemented in the translation as blank constructors; and they are checked for by checking if the union types was produced with that constructor — this is the standard for Dafny quotes, and is used repeatedly throughout Program Proofs [6] to identify the nil case of an inductive type.

Improving the implementation of union types should be a top priority when moving forward with this project. Currently, this is one of the key mitigatable limitations when producing a dafny translation of a VDMSL specification. This would require solving some of the difficult problems that have been left for future work. The first of which would be the issue regarding field expressions and extracting the values from literal types which was described earlier.

3.2.5 Lemmas

While the previous sections describe language features that VDM has and Dafny lacks. This will describe a key feature of Dafny that is missing from VDM. Dafny code requires certain properties to be proven before compilation, these are generally proven through lemmas or are automatically proven by Dafny. VDM does not have lemmas, this causes a big problem when trying to translate into Dafny, as there may be some properties that Dafny requires proof for that VDM does not.

There are possible strategies that could be used to encode these lemmas into VDM. Namely, I initially tried to encode them as total boolean functions which should always return true. However, this does not syntactically match with Dafny lemmas, which are more akin to VDM operations, in that they use a list of proof statements, rather than being a single evaluated expression.

Other translator tools, such as VDM2Isa describe a lemma using the '@Lemma' annotation. These add additional proof obligations to the output and are useful in concept. However, this does not solve the fact that these lemmas need to be produced at some point. Moving these to the VDM specification does not solve the issue of needing to write out the lemmas — though it may be helpful in producing consistent formatting for the file structure.

In this case, it was decided that lemmas will not be included in the subset of the languages that are handled by the tool. However, this does not mean that lemmas themselves are impossible. The framework for the lemma production is available as part of VDMJ's 'pog' command, which produces a list of all proof obligations. In VDM2Isa, these obligations are produced under a seperate proof file which are then discharged either automatically or by the user's proofs. This approach would be an ideal extension for the project, as a seperate module for the lemmas would be produced that extends upon the initial module; or a seperate section of the file would be produced to house the lemma definitions.

However, it is very likely that simply producing the framework for the user to complete the lemmas is as far as the project could reasonably go; producing an automated proof for each obligation is an excessively ambitious goal that is not feasible in reality; as such it is not expected that this will become integrated into the project at any point.

3.3 Deriving string templates

3.3.1 Approaches

AW: I think it may be good here to include a figure to show what a grammar rule looks like, and maybe how this would be translated in to a string template? The first considered approach to producing string templates for this project was to tackle the language in a grammatical manner. This would mean translating the grammatical rules on the Dafny reference manual [3] into StringTemplate4 templates. This approach is ideal as it ensures that all possible parts of the supported language features are accounted for.

Another approach would be to produce Dafny rules from individual expression and interpretations from the VDM code that it is based upon. This approach is significantly easier compared to the grammar-based approach and would be signficantly quicker to implement and test. These specific advantages are very attractive for this project due to the time constraints; and the ambitious nature of the alternative approach.

Ultimately, considering the advantages and disadvantages of both, the best option for this project was deemed to be a hybrid approach, where only the templates that were relevant to VDM would be produced, but these would be produced with the grammar rules of Dafny in mind. Ultimately, this saved a lot of time in producing the project, at the cost of not having a complete representation of Dafny in the grammar rules.

However, by taking this approach, it could mean that certain features that are possible to translate are missed, or that the grammar rules need to be slightly modified to account for VDM's language features later in the project. One example of this is variable declarations, in VDM the left-hand side is significantly more expressive compared to Dafny, and the problems that this caused when translating are described in greater detail in Section 3.4.4.

3.4 AST traversal

3.4.1 Identifiers

Identifiers, in this project, are simply transformed rather than translated. This is done in a deterministic manner to ensure that any mention of the same identifier is translated in a predictable way. These rules ensure that any identifier from VDM does not compromise the grammar rules set out by Dafny in Section 17.1.2 of [2]. This includes guarding against reserved words, replacing illegal characters, and prefixing identifiers with illegal prefixes. Every time an identifier is declared, or referenced in any way, it passes through this transformation.

3.4.2 Modules

Modules are very similarly defined in VDM as they are in Dafny. Namely, both of these require a name and a list of declarations at the minimum. The translation strategies for these declarations are unimportant at this level. This is because it is assumed that when the declarations are passed through to the module template, they have already been parsed and translated. For this reason, discussing these declarations is left for Section 3.4.3.

Despite, at the basic level, these modules being very similar, there are some features which are both different and unique in Dafny. Namely, Dafny allows for modules to refine another, and in that manner, they allow for abstract definitions of it's declarations. In this project, module refinement is not discussed fully, but possible use-cases for this language feature are mentioned in Section 5.3.

AW: Dafny include directives

The first key difference between Dafny and VDM module definitions is that Dafny has 'include directives', which define an external Dafny file to be included when interpreting the Dafny program. In VDM, these are implicitly defined as every loaded VDM module. A direct translation approach would simply load each translated Dafny module into each Dafny include directive. This is inefficient, and it is much better to only add an include directive if the Dafny file references the external file via an import definition.

AW: Differences between VDM and Dafny export definitions

```
1
2  imports from ModuleName
3      types
4          -- Renamed type import
5          TypeName renamed OtherName;
6          -- Non-renamed import
7          OtherType
8          -- Type import with declarations are not treated differently, only the
               name matters for Dafny
9
10 definitions
11 ...
12 -- This shows how access to each type differs depending on the type of import used
       .
13 SomeFunction: ModuleName`OtherType * OtherName -> ?
14 end VDM
```

```
1 // Renamed imports
2 import OtherName = ModuleName`TypeName
3 // Non-renamed imports
4 import ModuleName`{OtherType}
5 // Imports with declarations are not supported, so they are transformed to one of
      these types.
6
7 // Shows the similarity between type access when renaming and not renaming the
      import.
8 predicate SomeFunction(dummy: ModuleName.OtherType, dummy2: OtherName)
```

Fig. 10. Importing a type in Dafny syntactically is similar to how it is done in VDM, the order of operations is reversed, with the rename being first and the module export being after. There are some key differences however. Namely, in VDM, when a type is exported, all of the definitions within that type (order, equivalence, and it's invariants) are exported alongside it. This is not the case in Dafny, as they are defined as seperate functions in the translation. This means that one VDM export can produce many Dafny exports.

On top of this, Dafny's import and export definitions work in a slightly different way to VDM that prevents a direct translation. Thankfully, importing from a module can be done in a way that preserves as much of the VDM nature of the program as possible. Though in a translation strategy, a distinction needs to be made between non-renamed and renamed imports. This is primarily due to how references to the types are handled. In the case of renamed types, only the renamed is needed in both Dafny and VDM.

However, when accessing these types through the VDMJ type checker, we cannot access the renamed definition easily. Therefore, upon defining the imports of a module, it is important to keep track of what each type is renamed as in the module before translating the rest of the definitions. This ensures that types are properly referenced in the generated code. We only need to keep track of renamed types in this manner however, as non-renamed types can be accessed properly through the available information the compiler.

Another issue with exporting a VDM module to Dafny is incompatible export sets. This issue arises specifically when exporting all of a VDM module, as Dafny does not allow for a mix of reveal types (provides and reveals) within an export set. This means that it is not

feasible to produce an export set which exports all of a module. The approach that has been taken in this project is to produce a singleton export set for each export definition which will allow for cherry-picking of imports, as well as importing all of a module — though this will produce a signficantly more verbose import statement compared to VDM's 'import all' definition.

```
exports
    -- no struct export is 'provides' in Dafny, otherwise export as 'reveals'
    types A

types

A = int
inv a == a > 10
ord a > b == a > b
eq a = b == a = b;
```

```
// Care needs to be taken when translating an export to export all required
    implicit definitions, the same goes for functions and their preconditions.
export A provides A
export inv_a provides inv_A
export ord_A provides ord_A
export eq_A provides eq_A
export min_A provides min_A
export max_A provides max_A

type A = int
...
```

Fig. 11. In this case, all of the exports listed in the Dafny example are implicitly exported in VDM as they are defined as part of the A type; and are thus exported alongside it.

Furthermore, when exporting a type, specific care has to be given to additionally exporting any implicit definitions that the type has. Due to the implementation of these, which is described in Section 3.4.3, each implicit definition needs to be exported separately. In VDM, the export definition for these does not need to be given if the definition that they belong to is exported. As such, when translating these exports, a lot more care needs to be given, as such an export for each invariant, order, equivalence, precondition, and postcondition definition is provided when they are defined in an exported definition.

3.4.3 Top level declarations

At the top level of the module, there are a number of different possible declaration types that can be made, the separation of definitions is a clear difference between how VDM handles the top level declarations compared to Dafny; in VDM, it is required to define the type of the declarations within a region of the program before beginning any definition. Dafny, on the other hand, does not require regions of the program text to be defined for specific types of definitions; and instead uses keywords to denote the type of definition that follows — this approach is more standard of modern programming languages; and is definitely compatible with VDM's syntax for definitions.

Types

To this end, the strategies for type translations discussed in Section 3.2.4 could be employed without much issue, there are obviously still the aforementioned concerns on the

```
function max_<type_.name>(arg0: <type_.name>, arg1: <type_.name>): <type_.name> {
    if ((ord_<type_.name>(arg0, arg1)) || (arg0 == arg1)) then arg1 else arg0
}

function min_<type_.name>(arg0: <type_.name>, arg1: <type_.name>): <type_.name> {
    if ((ord_<type_.name>(arg0, arg1)) || (arg0 == arg1)) then arg0 else arg1
}
```

Fig. 12. These functions are the minimum and maximum functions that are prescribed in the VDM language manual [5]. These are translated into Dafny functions and are included in the translation of a type only if it has a defined ordering function.

viability of VDM union types within Dafny, and these would be addressed by a change of strategy for both field, and literal expressions or the union types themselves.

Values

As mentioned consistently throughout this project, patterns are the key limiting factor in the translation; namely their use in places that are not supported in Dafny. Value definitions in this translation tool are translated as constant values in Dafny. The key difference in this translation is that only definitions which use the identifier pattern will be supported for translation to Dafny — though it may be possible to extend this to other patterns. For example, the record pattern could be extended to multiple definitions for each defined field as shown in Figure 13.

```
/*
Record ::
    one: char
    two: int  */
datatype Record = mk_Record(one: char, two: int)

// mk_Record(a, b) := mk_Record("a", 1);
const a: int := mk_Record("a", 1).one
const b: int := mk_Record("a", 1).two
```

Fig. 13. Shows one possible strategy for translating a record pattern into Dafny const declarations. Though this is still limited by the internal patterns of the record pattern — as each of these would have to be checked to see if they are themselves a valid pattern, and this could become convoluted quickly when considering deeply nested patterns.

AW: State definitions

State

Functions and Operations

The definitions of VDM functions can be translated in a one-to-one manner. with boolean-valued functions being transformed into Dafny predicates, and all others into functions. Initially, the approach to preconditions and postconditions was to make use of the built-in requires and ensures clauses in Dafny. However, this is insufficient for all uses of VDM; as the preconditions and postconditions can be called from other locations in the code — which is only available in Dafny for the requires clause. To fix this, these conditions surrounding the function are translated as seperate functions which are then called in the requisite requires/ensures clause. This approach conserves the functionality of VDM's preconditions and postconditions as they are translated into Dafny.

Other than these, the VDM 'measures' clause is translated directly into Dafny's 'decreases' clause. While the majority of the function specification can be translated simply, Dafny has the 'reads' clause, which declares which objects need reading from the current

heap — this is helpful for determining the least-privilege required to complete a task, but it is not supported. Currently, the implementation of this assumes that a function can read all of the heap. But in the future, this could be extended to automatically generate minimal reads clauses for a function, which would greatly aid in parsing whether or not the generated code is accessing data that it shouldn't need to.

Operations are handled in a similar way to VDM functions, these are translated to Dafny methods, and the specification of a method follows the exact same grammar rules as functions — this makes the translation of operation definitions a simple task given that a lot of the work had already been done for functions. The only difference between these is the keyword, and the type of body. While functions have expression bodies (Section 3.4.4); methods have statement bodies (Section 3.4.5) which both operate by different rules and require different templates to translate.

As mentioned in Section 3.2.2, there are a number of issues in how VDM uses patterns when producing a translation strategy to Dafny. When discussing the definitions of functions and operations, this problem arises in that patterns can appear in when defining parameters. VDM allows any pattern to be used in place of an identifier for a parameter. This issue could be easily 'solved' by limiting the use of patterns for translations. However, the 'reconciliation' strategy mentioned earlier is used to minimise the cases in which translation has to be aborted due to the use of unsupported patterns.

AW: Maybe a diagram showing a possible approach to generating reads clauses?

3.4.4 Function and Predicate Bodies

When defining the bodies of functions and predicates, a strategy is needed for every type of expression in VDM, and this means that the translations become less involved in terms of matching together the parts of VDM's grammar with Dafny's, and the decision on whether or not something is translatable is usually a binary decision. For example, the distributed union expression in VDM is difficult to translate into Dafny for the reasons laid out in Section 3.2.1.

Variable assignments

These issues in patterns are consistent throughout the translation of expressions, and variable assignment expressions face issues because of VDM's expressive pattern matching. While Dafny does have a pattern assignment expression, it is significantly more limited than VDM's. Because of this, there are certain assignments that can be made easily in VDM that are not possible in Dafny. In the case that one of the blacklisted patterns/expressions are found in an assignment, an error will be shown to the user, and if they allow for a non-strict translation, it will be translated literally into Dafny as an invalid assignment — as shown in Figure 14.

Variable assignments are just a single type of expression that needs to be catered for in this project; at this stage of the project, a lot of the work is simple in nature. When comparing with the need to produce a middleground between VDM and Dafny grammar rules when translating top level declarations and modules; expressions are much simpler, but they still require a lot of work to produce due to how many need to be translated. As an example of this, when parsing expressions, VDMJ has seperate classes for each binary expression, unary expression, and every other special case. In this way, VDMJ makes it very easy to determine what type of expression an object is, and this is very useful when

```
f: int * int -> int
f(a, b) ==
    let c = a + b in c;

f2: seq of int * int -> int
f2(a, b) ==
    let c ^ [d] = a in d
```

```
f(a: int, b: int): int {
    var c := a + b;
    c
}

f2(a: seq<int>, b: int): int {
    // No patterns on left hand side, but pattern matching is allowed, so you can write:
    var c: seq<int>, d: int :| c + [d] == a;
    d
    // Which is a direct translation of the VDM, but this does not work in Dafny, as it cannot prove that it will return the same thing every single time, so the translation of this would be...
    var c := a[..|a|-1];
    var d := a[|a|-1];
    d
}
```

Fig. 14. This shows a simple VDM code to split a sequence into a head sequence and to return the very end of the tail. VDM's pattern matching can handle this situation, but Dafny's fail in this. While in this case, an automatic translation would be possible, it is easy to imagine that this type of translation is not feasible for every possible failed case. Therefore, the translation will return the first example with a failed Dafny pattern assignment and warn that pattern assignment definitions are not fully supported in Dafny.

parsing what translation strategy needs to be used.

Simple Expressions

```
public enum DafnyToken {
...
    //   VDM token | Dafny Token | String template to load
    PLUS(Token.PLUS,     "+",      TranslationStrategy.BINARY_EXP)
...

    public String render(String... args) {
        // Define the strategies for each token here.
        // args is checked so that it contains the correct number of args.
        // each is added alongside the token in a specific order, e.g.
        ...
        case PLUS:
            assert args.length == 2;
            return strategy.render(args[0], dafnyToken, args[1]);
        ...
    }
}

public enum TranslationStrategy {
    BINARY_EXP("<arg0> <arg1> <arg2>")
    ...
    public String render(String... args) {
        // Adds each arg in args to the given template as "argN" where N is it's position in the 'args' array
        ...
    }
}
```

Fig. 15. This shows the code used to handle expression simple expressions in a quick way. This allows for new templates to be added to enumerated values in the 'TranslationStrategies' class, and quickly integrated with any expression that may use them, for example, the 'BINARY_EXP' enumerated value could be used for both binary expressions like addition, as well as encasing sequences, sets, and map enumerations with the corresponding tokens. This assumes that any arguments passed are pre-translated, but checks that the number of arguments passed is as expected.

To minimise the number of locations that need to be check for translation issues, it was decided that a 'DafnyToken' file was needed to map VDM tokens to their Dafny token equivalent and the translation strategy that is needed when this token is encountered. This approach was inspired by the 'IsaToken' class in VDM2Isa; and this file was modified to change the Isabelle token to Dafny tokens; and the capability to render the translation strategy for the token was added. These strategies are basic in nature. As an example, one strategy for binary expressions can be seen in Figure 15.

Cases Expressions

There were a number of expressions that required specific handling however, for example, the VDM 'cases' expression allows for any pattern to be used in the check. This is not allowed in Dafny — and the strategy to translate these was documented upon first inspection of the language in Section 3.2.3. As mentioned, in the case that an invalid pattern is used in a case expression, an error describing this will be shown to the user; and a direct, but incorrect translation will be returned.

Enumeration Expressions

Enumeration expresions are the simplest way to produce collection types in both VDM and Dafny, these are both handled in the same way in both VDM and Dafny. The translation strategy is to translate the internal expressions/maplets by all other translation rules, and list them in the same order as they are given in VDM. This strategy is identical for all of set, sequence, map, tuple, and record enumeration — with the only changes being the type of parentheses to match the Dafny syntax.

VDM does allow the user to produce a set using a range of values; giving only the lower and upper bounds. To translate this into Dafny, a set comprehension expression is used to produce a set between the two bounds — which functions identically to the VDM expression.

Comprehension Expressions

Set comprehension expressions operate in the same way in VDM as they do in Dafny — there are minor syntactic differences, but both operate with a bind, predicate, and expression — as with other translations into Dafny, the bind and predicate are merged into the same boolean expression. And the expression is evaluated for all values for which the bind holds.

There are cases when translating these comprehensions that Dafny cannot prove that the resulting collection is not infinite. These instances are accepted in the translation tool as an artefact of the differences between how Dafny and VDM handle comprehensions.

The final issue in the translation of comprehensions is simply dismissed as sequence comprehensions are not supported in Dafny in the same way — namely they use only the index number of the sequence rather than using a bind, predicate and expression like VDM does. As a result of this difference, this type of expression is not supported in the tool.

Quantified Expressions

AW: Forall and exists exist Quantified expressions also exist in Dafny. Obviously there is a different syntax regarding them, but this is easily reconcilable. The main difference

between VDM and Dafny in this regard is that Dafny does not seperate the bind and the predicate; and it is expected that the variables are bound within the predicate. This is simply done by connecting the translated VDM bind and predicate with an and expression.

AW: Same problem as comprehension expressions, infinite sets

Compared to comprehension expressions, quantified expressions do not generate the complaints about possible infinite sets, this is because of the required binding expression — which eliminates the majority of cases in which Dafny cannot prove that the expression is finite.

AW: Iota expressions Another type of quantified expression in VDM is the iota expression, this has a similar structure to the forall and exists expressions, but instead of returning a boolean value; it returns a value of the bind's type if and only if there exists one value in the binding that satisfies the given predicate.

At first, this seemed trivial to implement, as it is similar in function to the 'exists1' expression that was simple to implement into Dafny. However, this expression was not possible to implement into a translation strategy. This is because Dafny does not allow for sets to be generically accessed within a function. This is because functions must be deterministic, and as such a proof must be provided that access to a set will produce the same result every single time. As a result of this difficulty, the iota expression has been left for future work.

Comparisons using user-defined expressions

VDM allows for the user to define certain operations for their types. These are the equality, and ordering expressions. These allow the user to define their own curried expressions, replacing the in-built operators with custom functions. While these cannot be directly translated into Dafny, in their curried form, the ordering and equality definitions can be translated and called whenever the curried code appears.

Quotes

In this project, quotes are only handled when they are defined as part of a union type; and comparisons are only allowed when testing a given type against a quote literal. Compared to what is allowed in the entirety of VDM, this is a harsh criteria — and it limits some of the uses that are shown in the language manual as examples. This is unavoidable for Dafny due to it's lack of quote types. As mentioned in Section **??**, these are represented as blank constructors when producing a type, which is why these specific limitations are necessary.

Automated Conversion of Subset Types

Another issue arises from VDM's automated conversion of subset types. For example, when negating a natural number, the expression expects the result to be an integer (as the negation of a positive number is negative); in VDM, this is automatically done; however, in Dafny it is expected that whenever an expression would change types, the 'as' expression is used. I.E. in this case, you would have to negate the natural number after casting it to an integer.

This is partially mitigated by testing the result type of an expression or function against the expression's actual type; and then adding a Dafny 'as' expression if these do not match.

While this mitigation does not cover all possible problems, it does hint towards the solution in many of the remaining issues. This technique is shown at work in Section 4.3.1, where it does not completely cover all possible problems, but does provide the afforementioned hints that would help the user.

3.4.5 Operation Bodies

AW: Talk on the VDM structure of operations, how this is similar to functions except instead of a single expression it is made of a single statement (which could be a block statement)

Basic statements

VDM has many statements which have a similar structure in Dafny, these translations are generally primitive and simple in nature, and so the translation strategy is equally simple. For instance, translating a return statement into Dafny is simple; as the word 'return' is prepended to an expression which is translated with the strategies mentioned in Section 3.4.4.

If statements also have a similar structure to previously mentioned if expressions — the only difference between the two is that they use curly braces to denote the start and end of the if/else body, and that the body contains a statement rather than an expression.

Another easily translatable statement is the call statement; which simply calls another operation — this takes in a name, and an expression list as arguments. This obviously is possible in Dafny, and is translated directly, with the argument expressions being translated by previously mentioned strategies, and the name being transformed by the deterministic rules stated in Section 3.4.1.

Further, the identity statement ('skip' in VDM) is translated as a blank statement — this is because Dafny allows for empty program blocks for when a skip would be used in VDM.

Block Statements

Another simple statement to translate to Dafny is the block statement. It has been mentioned that both the definitions and bodies of if statements only contain a single statement. This is because in a majority of examples, this statement would be a block statement; which defines a list of statements to be executed in order. This statement is simple to translate, as it only needs to call the requisite translation strategies for each internal statement, then output them as a semi-colon seperated list.

Non-deterministic Statements

While this feature is not commonly used in VDM or in any modern language, non-deterministic statements are nonetheless apart of VDM. These are similar in structure to block statements, except that the order of operation of the internal statements is non-deterministic. At first glances, this is not possible to translate in to a language like Dafny since Dafny is an entirely deterministic language. It may be possible to produce a translation strategy that is

AW: Similar in structure and idea to block statements, just weird.

```
||(
    BubbleMin(),
    BubbleMax()
)
```

```
// shuffles a sequence of size 2, i.e. [0, 1] or [1, 0]
var orderOfStatements: seq<int> := randomGen.shuffledSeq(2);

//@vdm.non-deterministic
while (|orderOfStatements| > 0)
    decreases orderOfStatements
{
    //@vdm.non-deterministic.1
    if (orderOfStatements[0] == 0) {
        BubbleMin();
    }
    //@vdm.non-deterministic.2
    if (orderOfStatements[0] == 1) {
        BubbleMax();
    }
    // Get tail of seq
    orderOfStatements := orderOfStatements[1..];
}
```

Fig. 16. This shows one possible implementation of the non-determinstic statement. Which in the current implementation would allow for a seeded pseudorandom number generator to be used to generate a 'non-deterministic' order of operations; which would then be iterated through.

Assignment Statements

There is a lot of overlap between translating statements and expressions. This makes sense as a language designer would want as much overlap between functionally similar parts of their language as possible. To this end, assignment statements in Dafny operate identically to assignment expressions. While VDM has multiple places in which assignments can be declared ('dcl', 'def', 'let', and 'let be'); these are all combined into a single type of assignment expression for Dafny.

Due to the nature of VDM assignments, which was mentioned when talking about assignment expressions; a pattern matching assignment statement is used for all cases, this allows for more cases to be catered for the left-hand side with a single strategy. A more ideal solution would be to check the type of pattern that the left-hand side of the assignment is, and then assign a strategy from there. This is an obvious target for improvement, but it does not impact the functionality of the code at all.

Iteration statements

For the most part, iteration statements can be easily translated in to Dafny, though Dafny does handle loops with stricter rules than VDM, in that a specification can be outlined for each loop statement — these specifications can describe conditions that should never be broken within the loop; expressions which should always decrease after each iteration, and which parts of the heap the loop can access. While VDM does not define these, they are incredibly useful for formally verifying loop conditions in Dafny, as such, the translation tool will leave space and hints to include each of these conditions within the translated Dafny code.

There are three main types of iteration statements in both VDM and Dafny, these are the for, indexed for, and while loops. Both of the 'for' loop types have direct translations into Dafny. With the pattern bind and expressions from VDM becoming the quantifier domain

in Dafny in the same way as quantified expressions are translated. Indexed for loops operate in a similar manner in both VDM and Dafny; with the key difference being in the handling of of the iteration. Dafny only allows for 'to' or 'downto' to be used, which increment of decrement the index respectively; while VDM allows for a 'by' expression, whose value is used to determine the next iteration. Due to this inconsistency between the languages, an error is raised whenever the 'by' expression is specified. Though this could be relaxed slightly to only raise an error when it is specified as anything other than one or negative one — which translate to 'to' and 'downto'.

As mentioned previously, loops in Dafny allow for a loop specification which can be extremely useful. In for loops, this is generally unneeded, as they will never lead to infinite loops due to having well-defined bounds. While loops, on the other hand, may often require invariant and decreases clauses in their specification to prove that the loop will terminate.

Unsupported Statements

While most statements are supported in the translation tool, there are some that remain challenging to translate. These primarily relate to VDM's error handling statements, which are mentioned in Sections 12-13 of the VDM language manual [4]. While the mentioned error statement may be possible to translate by making use of Dafny's failure returns, though there is a lot more tricky syntactic sugar used in Dafny's handling of failure types that VDM does not have that would make it more tricky to handle the error and exception handling statements.

The main point of failure in this would come from the fact that a failure type in Dafny is not properly defined, and is instead a type of user-defined type that is produced in a manner that is 'failure-compatible'. Therefore, despite this likely being possible to produce a translation strategy for, it has been left to further work so that priority could be given to a greater subset of the language translation.

3.4.6 Issues with VDMJ's handling of expressions

This project is dependent on the use of the open-source VDMJ compiler for VDM [1]; and as such, it is dependent upon the design choices and various abstractions made in the compiler. There are certain instances where the VDMJ compiler abstracts an expression using an implication that is made using VDM. An example of this is shown in Figure 17. Which shows a field expression in the f function. VDMJ interprets the expected type of ut at this point to be $Triple$, rather than $UnionType$.

This obviously causes problems when translating to Dafny. As the required context to determine whether a field expression originates from an object of a union type does not exist. This issue, with the current system, cannot be resolved. As such, the approach that has been taken is to warn the user of the tool when they are attempting to translate a union type with records that there will be issues when attempting to access fields from a union type.

One approach to solve this issue would be to keep track of the initial definitions of each variable as it is initialised — and then use this context to determine if there needs to be modifications to the current field expression translation.

```
types
Pair::
    k: int
    v: int;

Triple
    v0: int
    v1: int
    v2: int;

UnionType = Pair | Triple;

functions

f: UnionType -> bool
f(ut) == is_Triple(ut) => ut.v0 > 0;
```

Fig. 17. An expression involving a union type. With the combination of how VDMJ parses this, and how the translator handles union types, this becomes tricky as no context is given with regard to the original type of ut when the field $v0$ is accessed.

```
type Pair = ImplPair'
datatype ImplPair = mk_Pair(k: int, v: int)

type Triple = ImplTriple'
datatype ImplTriple' = mk_Triple(v0: int, v1: int, v2: int)

UnionType = ImplUnionType'
datatype ImplUnionType' =
    |   Pair(Pair: Pair)
    |   Triple(Triple: Triple)

// A correct translation
predicate f(ut: UnionType) {
    ut.Triple? ==> ut.Triple.v0 > 0
}

// What the translator will give
predicate f(ut: UnionType) {
    ut.Triple? ==> ut.v0 > 0
}
```

Fig. 18. Dafny implementation of Figure 17. The translator has issues with the field expression to get $v0$, this is because the VDMJ compiler has already deduced that ut is a $Triple$, and does not provide corresponding context that ut is passed as a $UnionType$. This problem could be resolved by keeping track of what each type is defined as upon entry and definition of a function, but this would add significant difficulty to the project, and has been delayed for a future improvement to the tool. Currently, it will just warn the user of this possibility when creating a union type of records.

3.5 Bringing it all together

- How well do the string templates do with the VDMJ code.

  Do the templates produce valid code. In what situations does that fail?

  - Do the templates remain readable, and is the code produced neatly organised.

4 Evaluation

4.1 The subset of the languages

AW: Failure cases, syntactic [syntax is not compatible, still possible to translate parts properly, but it is an incomplete translation], semantic [the meaning of a feature is completely different across the two languages, this is not possible to translate], and complete incompatibility [a feature is missing in one of the two languages, most commonly missing from VDM.]

AW: Maybe a list in the appendix of what features meet each of these criteria would be

nice? Will only do that if time permits though. This would be nice alongside any future mitigations that could be implemented to change it's classification.

This project tackled two very similar languages, with some very similar capabilities; and the points at which translation failed can be split into three distinct categories.

The first of these cases is a syntactic incompatibility; which means that a full translation is difficult, or impossible, from VDM to Dafny given the differing grammar specifications for each language. This case encompasses situations in which there is an available translation strategy, but it does not fully cover all possible cases. And includes examples like the VDM cases expression — which failed because Dafny has more limited pattern matching rules in case expressions compared to VDM; and while statements, which in some cases require additional context in Dafny compared to VDM. These cases are acceptable to include in a translation, as they generally cover all of VDM's use-cases, but may require warning against in the tool to ensure that the transition to Dafny is smooth.

The most severe case is a complete incompatibility, where a language feature cannot be translated properly at all. The most severe of these cases in this project is likely the definition of quote types. While there was effort taken to ensure that quotes can be used as part of union types, it is not possible to make use of quote literals or compare quotes of different types. This has also occurred on several expression types in VDM, namely the distributed types; which are tricky to encode into Dafny and have still not fully been encoded, the issues surrounding these are mentioned in Section 3.2.1.

The final case is a semantic incompatibility. This is where a feature is in both VDM and Dafny, but operates in a significantly different manner, This makes any reconciliation between both of the grammar rules impossible. The most commonly used expression that falls under this case is the sequence comprehension expression. In VDM, this type of expression operates in a similar manner to every other comprehension expression, as it follows a bind expression, predicate, and another expression which is evaluated as the sequence item's value. On the other hand, in Dafny, a sequence comprehension takes only the sequence index and a lambda function, with no possibility of allowing a bind or predicate check. Due to this difference in meaning between two similar features, it was not possible to implement either as a translation strategy within this project.

4.2 Program Proofs Examples

4.3 VDM Toolkit Examples

4.3.1 Conway's Game of Life

AW: Conway's game of life

AW: Chosen because we expect a certain type of error in the translation; namely Conway's game of life is defined on an infinite/unbounded grid, which Dafny does not allow — if we see this error then we know that Dafny is picking up on infeasible issues that VDM does not pick up on.

4.4 Comparison of translation approaches

- Grammar based translation is the best available strategy, but was not feasible to tackle in the time that I had. This strategy is compatible with the built system however. As it would just require rewriting templates to be fully representative of the grammar rules.

- Class mapping is better for implicit strategies, but would require learning how to use the specific class mapping language and setting that up, so I avoided it in this project. In hindsight, this may have been a nicer approach to take.

5 Conclusions and further work

5.1 Alternative approaches

- Grammar based approach
  - VDMJ class mapping approach

5.2 Improved translation strategies

- Translation strategies for union types could use some work, especially with respect to field expressions.
  - Automated ”as” expressions in Dafny to cover some additional points of failure
  - Error/warn for infinite sets.
  - Automated witnesses? Or inclusion of witness annotations.
  - Improvements to error handling.

5.3 Proof obligations

As mentioned in Section 3.2.5, integrating further with VDMJ's existing commands would be ideal. In this case, the 'pog' command would be integrated with the tool to produce the required structure for lemmas to be written so that each proof obligation can be quickly discharged, either using Dafny's automatic testing, or by having the user produce a proof lemma manually.

AW: Cases for module refinement in lemma declarations, I.E. translator will produce a non-refined module, which is then refined through lemmas/discharging proof obligations in another module.

5.4 Integrations with existing tools

- VDM Toolkit

6 Acknowledgements

References

[1] Nick Battle. VDMJ Compiler for VDM. https://github.com/nickbattle/vdmj, Feb 2024.

[2] Dafny-lang Community. Dafny reference manual. https://dafny.org/latest/DafnyRef/DafnyRef, 2024.

[3] Dafny-lang Community. Dafny reference manual. https://dafny.org/latest/DafnyRef/DafnyRef, 2024. Chapter 17.

[4] Peter Gorm Larsen, Kenneth Guldbrandt Lausdahl, and Nick Battle. VDM-10 Language Manual. 2010.

[5] Peter Gorm Larsen, Kenneth Guldbrandt Lausdahl, and Nick Battle. VDM-10 Language Manual, pg. 35. 2022.

[6] K. Rustan M. Leino. *Program Proofs*. The MIT Press, Mar 2023.