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 the VDMJ compiler which will automatically translate loaded VDM specifications into Dafny modules. Formal specification with the VDM notation is executable through the VDMJ compiler, but not automatically proven. This project aims to produce a translation for a subset of VDM to the Dafny language, which can automatically discharge proofs. We will follow the translation tool from VDM to Isabelle/HOL theorem prover to provide proof support within the Dafny notation. This is important as it allows VDM to access modern satisfiability modulo theorem provers through the Dafny notation. Which provide a level of automated proof that would not be accessible otherwise.

Keywords: Translation, VDMSL, Dafny, VDMJ plugin.

1 Introduction

The Vienna Development Method (VDM) is an old formally specified language which has been standardised in ISO [22]. There exist a number of translation tools, which have been created throughout it's history to provide various advantages [10] [19]. This project derives new translation strategies for translating VDM to Dafny [7]; these will be implemented as an extension to the VDMJ compiler [5].

Dafny was selected as the target for this project due to it's similar expressivity to VDMSL and because it has considerable proof support, which VDMSL lacks. As a formal specification language, it aims to produce a set of provable specification through types, functions and methods. Furthermore, Dafny is supported by Microsoft, and has integrated capabilities with the .NET framework. This makes it a lucrative target, as it will allow for VDMSL code to reap the same benefits that Dafny can through a number of translations. Furthermore, since Dafny is also a formally specified language, with grammar rules that are openly available; it is possible to approach the task of translation in a very methodical manner.

Since VDM is an old language, there will likely be a tangible gap between the capabilities of the translation targets. Because of this, it is impossible to expect a complete translation. Nevertheless, attempts will be made to reconcile the difference between the two

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

languages through the production of an auxiliary Dafny file, which will serve as a library for VDM features which Dafny does not natively support. Furthermore, the features provided in this library may also prove to be useful to someone who is natively programming in Dafny.

This report aims to document the process of producing a translation tool between two formally specified languages — as well as the specific struggles and issues that were encountered during this process.

1.1 Aims and Objectives

This project aims to produce an automated translation tool for VDMSL to Dafny translations. This should tackle a subset of both VDM that is relevant to Dafny; and omissions should produce errors in the tool; and warnings for when additional context is required by Dafny for a proper translation. As part of the completion of this project, a report should be produced to document both the approach of producing a translation tool, and the issues that were encountered during the project. Translation strategies produced as part of this project should be derived from the grammar rules of both target languages [8], [15]; these strategies should be easily modifiable and extensible — it should not require knowledge on how the tool specifically works to write the translation strategies.

Omissions from this project should be justified as part of this document. If these features are possible to implement but omitted from this project, then a possible implementation strategy should be provided — as well as a justification for why the strategy was not implemented in the tool.

Ultimately, due to time constraints, it is expected that this project will be a prototype for a more complete translation tool. As such, additional emphasis should be placed on deriving possible strategies for unimplemented features; as they will likely be integrated in the future. Despite being a prototype, it should aim to produce a translation strategy for all trivial cases of translation, and then work to extend coverage to more complicated translation strategies. This report documents the coverage of the translation tool in Appendix A.

Further, to aid in the production of a viable prototype, the plugin and command registration code of the VDM2Isa tool will be used to simplify the bootstrapping process of producing a VDMJ plugin.

1.2 Structure of the Paper

In Section 2, we present a background of both targeted languages as well and their tools. Next, in Section 3, we address the approach taken to translate between the two languages, and demonstrates the intricacies encountered in this process. Further, in Section 4, we evaluate the tool by translating multiple examples from both Program Proofs [13], and the publicly available VDM Toolkit [11]. Finally, in Section 5, we summarise the project, and extensions of the project which would prove to be useful.

2 Background

2.1 VDM

VDM is a formal specification language that was initially created in 1970. Since then, it has been adopted into international standards as one of the earliest languages for formal specifications. VDM was one of the first languages to have every single aspect formally

defined and specified; this sets it apart from many languages of it's time, and many useful algorithms are specified in VDM [22].

In modern times, the usage of VDM has declined due to more modern languages providing some more powerful features; to address this issue in VDM, language translations have been created previously to mitigate some of the issues that arise within typical use of VDM. These existing projects can either aim to extend on VDM's proof capabilities by translating to a formal proof aid tool such as Isabelle [10], or aid in the visualisation of VDM modules and their dependencies through the translation to UML [19].

2.2 Dafny

Dafny, is a modern formal specification language which was built originally as a tool for object-oriented software. This was created at Microsoft Research in 2008 [18], and has been receiving updates and incremental improvements since. Dafny is used widely in educative circumstances [21] due to it's integrations the .NET framework; as well as it's simplistic introduction to program proofs and proof tools [17].

The Dafny verifier is built on top of Boogie, which is a commonly used intermediate verification language — and other verifiers have been built on top of this, such as HAVOC, VCC, Chalice, and Spec# [20]. Since the verifier is built on top of Boogie, it primarily uses the underlying theorem solvers provided by Z3. Understanding how these theorem provers work is out of specification for this project, but it is useful when anticipating what sort of issues we can expect when translating VDM to Dafny.

Dafny is designed to compile into a DLL file, which can be directly integrated with C# as a library. Dafny is designed to be compiled easily into these DLLs, as well as executable files, which will execute the 'Main' method if it exists. This feature is the key reason that Dafny was chosen as the translation target for this project; as it provides a two-step translation approach from a VDM specification to a valid C# library.

As previously mentioned, Dafny is widely used in education as a tool in formal modelling. This means that there is a great array of available resources to use to learn the best practices, as well as how to properly produce a Dafny module. For this project, Programming Proofs [17] was used; which was both an excellent resource for learning Dafny, and for providing well-written modules which will be used in manual translations to both aid in the discovery of translation strategies (Section 3.1), and to evaluate the results of this project (Section 4.2).

There were other languages considered for translation. The leading contender in this was Why3 [9], though this was chosen against due to it's level in the abstraction chain. Dafny translates to Boogie [20], which is an intermediate language between Dafny and SMT solvers. Why3 is at a similar level in the chain to Boogie — which means that it provides the same proof aid as Dafny, but misses on the automated integrations to the .NET framework which Dafny provides.

2.3 VDMJ

In order to translate VDM to any language, there is obviously a need to understand and parse through a VDM file so that a proper translation can be provided, and the finished product will simply be an additional runnable command in the VDMJ command line interface [6]. For this reason, this project will take the shape of a plugin written for the open-source VDMJ compiler. This compiler will dictate many of the design decisions made throughout the project. Firstly, since the compiler is written in Java, the project will

also be written in Java.

This requires an basic understanding of how the compiler operates, and will require identification of where this project can extract information from. The VDM compilation process maps classes of VDM grammar into separate package which each handle different purposes. Initially, the data is in the abstract syntax tree format (AST), which is a large tree describing the syntax of the entire program code. Next, VDMJ maps these AST classes to the type checking classes — these are then verified and passed to both proof obligations, and interpretation classes [4] [3].

For the purposes of producing Dafny code, the type checking classes will be used. This is because they carry all the required information to produce translation strategies; and also because it is in keeping with the current convention of producing extensions on the type checker classes. There are a number of possible approaches that would be valid to take in this case, the most promising of these would be to produce a class mapping file. Though this would require learning and making full use of the class mapping syntax which may prove to be an issue when implementing. Due to the limited time allocated for this project, the best decision will likely be to manually produce the mappings by traversing the syntax tree from the type-checked module class — as this will minimise the learning, use, and integration of unknown technology during this project.

Despite this decision, it would likely be best to revisit this approach in future work, and refine the classes to be compatible with the class mapping approach.

2.4 String Templates

This project will make use of the Antlr StringTemplate4 library for Java [1]. This is a templating library that allows for significantly more expressive templates to be created compared to using the baked-in Java string formatting. The approaches that will be taken to interface with this library will be described, as well as the specific reasons for which this library was chosen.

This library satisfies our objective of producing translation strategies with a method that allows them to be easily modifiable and extensible, as knowledge over the string templating language is the only required knowledge to both create new strategies, and modify existing ones.

Useful syntax in StringTemplate4 for this project

As shown in Figure 1, Dafny grammar rules can be encoded into a string template using this framework; and there are a number of useful expressions which can be used within the template. These include field expressions, where an object field is fetched similarly to Java's field expressions. These field expressions use Java reflection to first get the 'getField', 'isField', 'hasField', and then if none of the previous terms are defined, it will attempt to get a field of the provided object with the name 'field'. This is useful, as it allows for class files to be made for each type of expression which only need to define these 'get' expressions.

In combination with these field expressions, there are a number of other useful constructs which are used throughout this project.

- Collection comprehension: If the result of a field returns a collection object, it can be iterated through with an internal template. This is shown in Figure 1 in the translation of the 'IncludeDirective_' rule.
- Conditional expressions: If the result of an expression is evaluated to be true, or present

```
Dafny = { IncludeDirective_ } { TopDecl(isTopLevel:true, isAbstract: false) } EOF

IncludeDirective_ = "include" stringToken

TopDecl(isTopLevel, isAbstract) =
  { DeclModifier }
  ( SubModuleDecl(isTopLevel)
  | ClassDecl
  | DatatypeDecl
  | NewtypeDecl
  | SynonymTypeDecl  // includes abstract types
  | IteratorDecl
  | TraitDecl
  | ClassMemberDecl(allowConstructors: false, isValueType: true, moduleLevelDecl:
      true)
  )

SubModuleDecl(isTopLevel) = ( ModuleDefinition | ModuleImport | ModuleExport )

ModuleDefinition(isTopLevel) =
  "module" { Attribute } ModuleQualifiedName
  [ "refines" ModuleQualifiedName ]
  "{" { TopDecl(isTopLevel:false, isAbstract) } "}"

...
```

```
Dafny(mod) ::= <<
// IncludeDirective_
<includeDirective(mod.includes)>

// ModuleDefinition
<ModuleDefinition(mod)> {
    // ModuleExports, ModuleImports, and other declarations are abstracted, the
        get calls for these will produce a translated object
    // ModuleExports
    <mod.exports:{export|<export>};separator="\n">


    // ModuleImports
    <mod.imports:{import|<import>};separator="\n">

    // ClassDecl | DataTypeDecl | NewtypeDecl | SynyonymTypeDecl | IteratorDecl |
        TraitDecl | ClassMemberDecl
    <mod.definitions>
}
>>

IncludeDirective(includes) ::= <<
<includes:{include|include "<include>"}; separator="\n">
>>

ModuleDefinition(mod) ::= <<
<if(mod.modifier)><mod.modifier> <endif>module <if(mod.attribute)>{<mod.attribute
    >} <endif><mod.name><if(mod.refines)><mod.refines><endif>
>>
```

Fig. 1. The grammar rules in this figure are taken directly from the Dafny reference manual [8]. This is an example of a grammar rule translation strategy that can be used. In this example, there are abstractions made to avoid producing the entire language at once. In this case, it is assumed that '¡mod.definitions¿' will call a translator function for all the different types of declarations that can be made. It is not likely that all of this translation will be needed. For example, the refinement, attribute, and modifier clauses may not need to exist in this project due to the lack of these in VDM. This will help in making clearer the purpose of the string template, as in areas of the grammar rules where there are a number of optional fields, it can be difficult to parse the template.

if a non-boolean value is returned. Then the internal template is rendered. These also contain an 'else' rule, which renders the internal template in all other cases.

- Template calls: Throughout the proposed translation strategy for the first level of the 'Dafny' grammar rule, a number of template calls were made use of to make the top level definition easier to read. It can be difficult to parse the rule for 'ModuleDefinition' without close inspection, and as such it can be moved into it's own template so that the overall template can be made clearer.

How time constraints will affect the approach of creating templates

Ideally, this project would first aim to produce a Java implementation of the grammar rules for Dafny, and then it would move on to implementing the produced interface with the VDMJ compiler. Unfortunately, due to the time-constrained nature of this project, it will not be feasible to produce an in-depth set of templates for Dafny. And as such, only the parts of the grammar rules which are analogous to VDM will be tackled. Fortunately, these templates are designed to be dynamically changable. As such, the templates could easily be extended in the future to include a more complete view of Dafny; which would help improve the overall quality of the translated code.

2.5 VDM2Isa

There are a number of existing translations for VDM to other languages. These include UML [19], Isabelle [10], C, and Java [14]. This project will specifically take the Isabelle translation into consideration — this is because it is an example of a language with somewhat similar purposes as Dafny as a proof aid; and the translation is produced using the VDMJ compiler, which this project will make use of.

The Isabelle translation makes use of the VDMJ class mapping to produce a new type of 'TR' class to aid in the translation process. These translation classes have an additional 'translate' method which builds up a translation using a coded Java strategy. This can be difficult to parse when the translation methods. As such, this project will place additional emphasis on the string templates used for translation; and there should be a minimisation of translation strategies that occur completely in the Java code.

There were some unique challenges faced in the translation of VDM to Isabelle, there are some that this project will almost certainly face, such as issues with VDM's expressiveness in patterns and typing — which are detailed with respect to this project in Section 3.2. Though, some of the problems which necessitated the use of class mapping in the Isabelle translation do not exist within Dafny. This is primarily due to subset types; which require additional checks in Isabelle compared to Dafny. Overall, this makes the derivation of translation strategies considerably easier for this project, as less thought needs to be given to ensuring the defined bounds of types are held.

3 What you did and how

This section focuses on the approach that was taken to produce the translation tool. This includes: the discovery of problem areas; steps taken to mitigate issues; and the overall approach taken to both discovering and implementing the translation rules. First, a manual translation will be taken from Dafny to VDM; this is so that there is a tangible final goal for the translation — as well as to aid in discovering the problem areas where common patterns in Dafny may not be feasible to implement in VDM.

Then this will approach the areas and constructs of the language which are identified as problematic prior to beginning the translation work — this aims to highlight the most obvious issues that the project will face, and propose countermeasures to protect against them.

Next, the approach to producing templates is briefly touched upon. This section will weigh different viable approaches against each other, and ultimately determine the strategy for producing the translation templates.

Finally, this section will discuss the AST traversal and translation, which builds up

the translation of VDMSL to Dafny code. This subsection is structured similarly to how the project will parse the VDM. With the top level overview being described at first, and gradually decreasing the level of abstraction to individual expressions, and statements.

3.1 Manual Translation

3.1.1 Selecting an example

When selecting an example to manually translate from Dafny to VDM, the most important aspect to ensure is the quality of the original Dafny. The exercise would not be as fruitful as it could be if a low-quality Dafny example is selected. As a result, code examples from the Program Proofs book [17] were selected, these are provided online [13] for every chapter of the book, and will be used within this project to demonstrate the realistic use of Dafny; and to provide an idealistic baseline for the project to translate towards.

The philosophy regarding the translation process is simple. Under no circumstance during the manual translation should a 'smart' decision be made; that is, there should be a definable rule for each choice that is made within the translation — these should be both reproducible, and importantly reversible.

```
ColourTree = Leaf | Node(left: ColourTree, right: ColourTree)
```

```
Node::
    left: ColourTree
    right: ColourTree;
ColourTree = <Leaf> | Node;

-- While this looks similar to the Dafny example, it is functionally identical to
    the above VDM, and difficult to read, so this is avoided.
ColourTree = <Leaf> | compose Node of left: ColourTree right:ColourTree end;
```

Fig. 2. This is a common type construction for recursive types in Dafny, which utilises the parameterless constructor 'Leaf' as an exit condition, and the 'Node' constructor to create a tree structure. To translate this into VDM, it was decided to split this into two types. This is the only real way to preserve the meaning of these constructors in VDM, but may make the translation back more difficult.

To this end, translations from Dafny to VDM are generally simple to manually compute — however, throughout the process of translating and discovering translation rules, there were some issues which consistently appeared. Namely, when translating types with multiple constructors into VDM. These constructs are common when producing Dafny code, and are used in the majority of examples from the Program Proofs book [17]. An example of this is shown in Figure 2, alongside an explanation of why the lack of constructors in VDM types make these difficult to simply translate.

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, both the VDM language manual [15] and the Dafny reference manual [7] were compared 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 deceitfully tricky to encode into Dafny, however. As the types that need additional implementation 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 3.

```
//@vdm.type('nat1')
newtype nat1 = n: nat | n > 0 witness 1
//@vdm.type('seq1 of T')
type seq1<T> = s: seq<T> | |s| > 0 witness *
//@vdm.type('set1 of T')
type set1<T(==)> = s: set<T> | |s| > 0 witness *

// Recommendation
type set1ofint = s: set<int> | |s| > 0 witness {1}
```

Fig. 3. This figure shows the types in VDM as they are encoded into the Dafny helper file. The '@vdm' annotation comments show where in VDM the translation targets. This is kept in the automated translations to allow for traceability during debugging. 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.

```
// 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. 4. 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.

A few of the language features that initially seem possible to encode are not as simple as they seem when actually taking the effort to encode them. 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 "dunion" expression are listed in Figure 4, 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.

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

Fig. 5. This shows the unique way that parameter definitions can be written to include patterns in VDM, this would be difficult to translate directly into Dafny.

VDM patterns are significantly more expressive than Dafny patterns, and can, in some cases impose invariants implicitly. Which can lead to much more complicated proofs. For example, in Figure 5, the passes sequence is implicitly checked to be non-empty, since the pattern asserts that there must be an item to assign 'a', and 'b' to.

One approach that has been 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. This approach is valid when the patterns are defined in the definition of the function; as shown in Figure 5. 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.

```
1 function patternsInParameters(arg0: seq<(int, int)>): ? {
2     var a:int, b:int, c:seq<(int, int)> :| arg0 == [(a, b)] + c;
3     ?
4 }
```

Fig. 6. This shows the process of "reconciliation" of the patterns, using the built-in Dafny pattern matching expression.

This 'reconciliation' approach is shown in Figure 6, where the VDM pattern is replaced by a dummy parameter, and the pattern assignment statement is used to assign the values of 'a', 'b', and 'c'. This approach will not always work, and the plugin warns against using this type of pattern when translating, as it will often produce invalid Dafny code.

This approach is easily reproducible on any provided VDM function definition — as the patterns themselves should be able to be translated 1-1 into Dafny patterns. Though there are some issues which may arise as described in Figure 6.

3.2.3 *Cases*

Cases expressions in VDM also contain the problematic expressive patterns which are present in parameters. However, the reconciliation process which aids in the process of function definitions is not possible within a case expression.

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

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

Fig. 7. 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. 8. In 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.

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 8; 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 8). 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

The vast majority of type translations between VDM and Dafny are simple to encode — alias types are translated simply as a synonym type, or a subset type if an invariant is provided. Record types are translated as a synonym type which points to a datatype with a single constructor. It is required to implement both as a synonym type, as these are the only types which can manually define additional bounds.

However, union types provide a much more difficult challenge — there is no catering for 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 was taken captures a considerable amount of the uses of the language; but it does fail to capture a number of possible cases.

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 re-

```
-- 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 they are supported by the tool. This also shows the selected Dafny implementations of the supported union types.

quire 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 type object was produced with the constructor that has the same name as the quote literal — this mirrors how blank constructors are used within Dafny, and is used repeatedly throughout Program Proofs [17] primarily 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 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.

Initially they were encoded 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 separate 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 separate module for the lemmas would be produced that extends upon the initial module; or a separate 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

The first considered approach to producing string templates for this project was to tackle the language in a formal-syntax driven manner. This would mean translating the grammatical rules on the Dafny reference manual [8] into StringTemplate4 templates; an example of how this can be done was shown in Figure 1. 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 significantly 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.

3.3.2 Benefits and Difficulties of String Templating

String templating is undoubtably powerful for this purpose, and allows for a significant amount of the benefits of VDMJ's class mapping to be accounted for — that is implicit VDM checks can be added directly into the template; and this saves a considerable amount of effort in debugging translation errors, as tricky cases which are implicitly handled by VDM can be automatically added — which can include automatically adding relevant missing constructors to functions with a union return type.

However, there is a lot of ambiguity in the library, which can make producing well-formatted code challenging. These issues are primarily associated with line-breaks after a conditional expression. This is an issue, especially when considering that the addition of spurious new lines in code could severely degrade how well a user could parse the result. Further, as mentioned in Section 2.4, a field expression in the template is accessed in a number of ways which are specified in the documentation. However, the order of access is not specified. And is the same regardless of the context of the expression. An ideal improvement to the templating framework would be to make the field expressions context-dependent. With 'isField' or 'hasField' being given priority within conditional expressions; which would be very beneficial within this project.

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 predictably. These rules ensure that any identifier from VDM does not compromise the grammar rules set out by Dafny in Section 17.1.2 of [7]. 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. In both languages, they 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. This pattern of assumptions follows for all other translation constructs mentioned in this report. Wherein the body translation is left for other objects to handle, and the total translation that any individual object does is minimal.

Despite, the structure of 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 due to the lack of VDM equivalent, but possible use-cases for this language feature are mentioned in Section 5.2.

Another 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.

```
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 new name is needed in both Dafny and VDM, while the module name is required if it was not renamed.

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 information that the compiler makes available.

There can be issues with default import and export sets; within Dafny, these are the export sets which do not provide a name. These are used in this project to define the 'exports all' which is translated directly as 'export reveals *'. This can cause issues when not all parts of the exported module are imported. This is because all individual exports need to be specified for them to be individually imported. Initially, the approach to producing this export set was to combine exported definitions and re-export them as part of the default set. This is not a valid appraoch however, as Dafny cannot mix types of exports in the same set; I.E. an export set cannot 'provide' one definition, but 'reveal' another. Currently, this means that importing all of a module is unsupported, and requires the user to cherry-pick

the required definitions in the provided VDM code.

```
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 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, an export for each invariant, order, equivalence, precondition, and postcondition definition is provided whenever they are defined in an exported type.

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.

```
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 [16]. These are translated into Dafny functions and are included in the translation of a type only if it has a defined ordering function.

Types

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 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 a 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.

State

The state declaration in VDM provides a global, modifiable object which can be accessed by every single operation. There is no direct analogy to this in Dafny. The first approach taken to attempt to introduce a global state implementation into Dafny was to make use of a singleton pattern in a Dafny class. However, this design pattern is not possible as top level static variables are disallowed [12].

```
method method1(a: int, b: int) returns (c: int) {
    return method2(a, b);
}
method method2(a: int, b: int) returns (c: int) {
    return a + b;
}
// becomes
method method1(a: int, b: int, VDMState: State) returns (c: int, VDMState': State)
    {
    var method2result, VDMState' := method2(a, b);
    return method2result, VDMState';
}
method method2(a: int, b: int, VDMState: State) returns (c: int, VDMState': State)
    {
    return a + b, VDMState;
}
```

Fig. 14. This demonstrates the difference between a Dafny implementation without state and one that contains state.

This makes the implementation of state challenging in Dafny, as regardless of it's implementation, it would likely operate similarly to a singleton object. If a singleton-like object is used within Dafny, it is generally expected to be passed around through the parameters of methods. While this is possible, it would require significant modifications to the underlying functionality, and it would be incredibly likely that the translation strategy would significantly obfuscate the meaning of the code. This can be seen in Figure 14, which compares how a very simple pair of methods, which contain a single statement each could become difficult to parse if state is included.

Numerous other strategies were attempted to bring state to Dafny, but these all failed, primarily due to contradictions in how state was aiming to be set up. These attempts aimed to set up the state as a constant value of a class type, this class type would have mutable fields, and as such, would be able to be modified from anywhere in the code. These attempts failed as the constant value needs to be initialised immediately, and could not cause any side effects. This means that a class constructor is illegal in constant declarations, which effectively ensures that the only value that the state could have would be 'null'. Which is obviously unhelpful.

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 using the 'pre_' and 'post_' prefixes — which is only available in Dafny for the 'requires' clause. To fix this, these conditions surrounding the function are translated as separate functions, which are then called in the requisite requires/ensures clause. This approach preserves 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 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 the generated code is accessing data that it shouldn't need to.

Operations are handled similarly 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.

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.

```
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. 15. 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.

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 15.

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 middle ground 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 separate 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 parsing what translation strategy needs to be used.

Simple Expressions

To minimise the number of locations that need to be checked 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

```
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. 16. 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, and 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 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 16.

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 expressions 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 the set, sequence, map, tuple, and record enumeration expressions — 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

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 separate 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.

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.

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.

```
predicate exists1Set<T>(bind: set<T>, pred: T -> bool)
    ensures var RESULT := exists1( bind, pred );
            RESULT ==> exists x: T :: x in bind
                &&  forall y: T :: y in bind - {x} ==> pred(x) && !pred(y)
{
    // Check that there is only one result
    |(set x: T | x in bind && pred(x) :: x)| == 1
    &&
    // Ensure that there exists a value such that pred(x) and there is no other
        value which meets this condition
    exists x: T :: x in bind
        &&  forall y: T :: y in bind - {x} ==> (pred(x) && !pred(y))
}

function iotaSet<T>(bind: set<T>, pred: T -> bool): T
    requires exists1( bind, pred )
{
    var value: T :| value in bind && pred(v);
    v
}
```

Fig. 17. This shows the conditions required to allow

At first, this seemed trivial to implement, as it is similar in function to the 'exists1' expression that was possible to implement into Dafny. However, this expression was much more difficult to implement into a translation strategy. This is because Dafny does not allow for sets to be generically accessed within a function. Further, as Dafny does not allow for the existence of errors, there must be guarding against possible error cases within the iota expression. To address this, additional conditions on the 'exists1' expression had to be specified. These conditions allowed Dafny to ensure that the result of the getting from the set is allowed.

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, these are 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 4.4, 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 aforementioned hints that would help the user.

3.4.5 Operation Bodies

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 semicolon separated list.

Non-deterministic Statements

While this feature is not commonly used in VDM or in any modern language, non-deterministic statements are nonetheless a part 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 operates using pseudo-random principles, but this would require a pseudo-random number generator to be implemented into Dafny, and would largely morph the structure of the code to be less readable, as the structure of a non-deterministic block would become similar to what is shown in Figure 18.

This also does not produce a completely non-deterministic result, as it will produce the same result with the same initial conditions every time; this means that the user would have to modify the initial conditions after every execution to simulate the non-deterministic behaviour that is desired from the original VDM.

```
||(
    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. 18. This shows one possible implementation of the non-deterministic 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 considerable amount of overlap between translating statements and expressions. This makes sense, as a language designer would want as much overlap between semantically 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.

Initially, due to the nature of VDM assignments, a pattern matching assignment statement was used for both assignment expressions and statements. While this is functionally correct, it is not in keeping with the proper coding conventions of Dafny. As such, check are made prior for each assignment’s left-hand side. As only certain patterns are allowed on the left-hand side of Dafny assignments, certain VDM patterns need to be transformed into Dafny pattern assignments — while some can remain as regular assignments.

Within VDM, assignment statements can also occur in a list within an ‘atomic’ statement, which states that all assignments within the block are done at the same time; this is achieved in Dafny by placing the ‘atomic’ attribute on an update statement; this has same functionality as the VDM atomic statement.

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 similarly in both VDM and Dafny; with the key difference being in the handling of the iteration. Dafny only allows for ‘to’ or ‘downto’ to be used, which increment or 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. Due to this addition in Dafny, it is not possible to natively support all translations into Dafny while loops. However, one approach that would resolve this would be to allow for an annotation in the VDM code to define these loop invariants. This has not been finalised in this project, but it would be an ideal extension.

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 [15]. 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 style for a 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 [5]; 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 19. 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. 19. 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.

4 Evaluation

4.1 The subset of the languages

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

```
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. 20. Dafny implementation of Figure 19. 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 the 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.

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 similarly 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.

A coverage list of VDM language features that are translated into Dafny is provided as part of Appendix A.

4.2 Program Proofs Examples

To evaluate our translation work, we used the Program Proofs book examples as a key source [13]. To use these, we fist encoded the known Dafny examples into VDM and then

applied our translator to them. This was important so that we had a concrete example to evaluate our strategies against. This also provided meaningful help in the development of translation strategies, as we knew why the Dafny outcomes worked with clear explanations. We envisage expanding the example sets to other native VDMSL examples [2] as well.

4.2.1 Examples from Program Proofs Chapter Four

This chapter of the program proofs book focuses on the introduction of inductive datatypes. These were mentioned in the initial translations as being difficult to translate literally to VDM. Therefore, type implementations are not expected to be as efficient as they were in the original code.

This example was selected as it introduces simple challenging constructors and destructors for datatypes. As expected, when translating the VDM implementations back to Dafny, the errors produced by the translator in this example are due to invalid field expressions and destructors; which as mentioned stems from a combination of a lack of provided context for proper typing from the VDMJ compiler, and the changes made to union type implementations.

Other than these errors, which are trivial to fix manually, there were no other issues in the translation of this chapter's modules — which should be expected due to the minimal nature of all other expressions in the module.

4.2.2 Examples from Program Proofs Chapter Seven

Chapter seven of the programming proofs book provides a more complete Dafny example which was translated into VDM, this example makes considerable use of lemmas in the original Dafny, and relies upon their integration within Dafny to prove it's correctness. Since the translation of lemmas is outside the scope of this project, it is expected that the errors within this translation will be entirely field expressions, and unproven proof obligations.

Within the manual translation, the lemmas were translated as total boolean functions — which, as mentioned in Section 3.2.5, does not fully capture their properties. But these implementations do aid in a simple implementation of the corresponding Dafny lemmas, as the 'requires' and 'ensures' clauses can be written as calls to the generated 'pre' and 'post' conditions from the translation tool.

As expected for this example, the only issues were the malformed field, and deconstructor expressions; as well as the unproven proof obligations.

4.3 VDM Toolkit Examples

For completeness, some examples have been taken from the VDM Toolkit [11], which is a collection of useful VDM plugins and experiments. These have also been used to validate the VDM2Isa tool, and as such, provide a neat comparison between the completeness of the Isabelle tool and this project.

4.3.1 Conway's Game of Life

Since the usage of union types which is common in the Dafny translations is uncommon when writing VDM, it is expected that the VDM toolkit definitions will produce less field expression errors, but may produce more errors that are specific to Dafny that VDM does not cater for.

To this end, this specific VDM module was selected due to it's specification involving

the use of an infinitely sized board. This is obviously a problem for Dafny as Dafny guards strictly against infinite sets. This means that it is expected that there are some instances where set comprehension expression will produce errors within Dafny. This would obviously be resolved by the user after use of the tool by introducing strict bounds for the size of the game board.

After translating the module, there are additional unexpected errors that were not picked up by the tool. These are the result of the automated expression casting that VDM provides (wherein a 'nat1' is converted to an 'int' type implicitly using a unary minus expression). Dafny does not support this automated casting, and requires the use of an as expression.

This issue would need to be addressed alongside the field expression issue by implementing a context system to determine the type and any necessary 'as' expressions that are needed.

4.3.2 Basics

To give a sense of how the basic expressions of VDM are translated in isolation, the basics directory of the VDM toolkit's GitHub experiments directory was translated in its entirety. This includes 18 modules of basic VDM, which each demonstrate a different subset of the language. This should allow specific issues with certain translation strategies to be identified in a significantly easier way.

While the results on these initially seemed disappointing. Some limitations of Dafny statements were uncovered in this process. Namely, VDM allows for statements to be called within statements. Whereas in Dafny, if an operation has a return type, it must be assigned to a value, and will raise an error if it is not assigned; this limits some direct translations with the tool. And this was important information to uncover.

This discovery, in addition with an incomplete state implementation, rules out a lot of the toolkit for translation use; and while the vast majority of rules are translated properly, there is still manual intervention needed in all tested cases. This is within the acceptance criteria for this project, but it would be ideal if the tool could independently produce translations.

Looking positively at the translations produced for the basics of VDM, all the issues that the project runs into in the translation are issues that have been identified as part of this report. And since the basics directory aims to give a beginners view of what the most commonly used constructs and types of expressions are used in VDM. Therefore, while the tool never does produce an ideal translation, the fact that the majority of the legwork is completed quickly and easily is a great success.

4.4 Improved translation strategies

There are a number of different improvements that can be made to the translation strategies, in some cases, these are purely aesthetic changes that either make the code neater, or ensure that it adheres to Dafny coding conventions. While in other cases, certain strategies may need to be modified for a complete translation, or an additional tricky difference between VDM and Dafny needs to be reconciled. For example, automated type casting, error-compliant types all could be included

Coding Conventions

While the code produced largely follows proper coding conventions due to the use of string templates, there are still some issues that can arise as a result of ambiguity in the template

files. This issue mainly presents itself when nesting quantified statements, and using comprehensions. To fix this, additional parentheses would need to be added in some places, but these need to be limited as Dafny may, in certain circumstances, mistake additional use of parentheses for a tuple enumeration.

Union types

Additionally, there are some templates which could use some work; this primarily applies to the union type definition template, and an ideal extension of this work would include a much broader translation strategy for union types. However, there are a number of challenges that need to be addressed to bring union types to an ideal translation. The main challenges that need to be faced when producing new union type templates is automated constructors, destructors, and field expressions. Implementing these will likely require the project to keep track of the context in which each expression takes place; which would be a significant undertaking for the project.

Automatic type casting

```
1 function add100(c: real): real {
2     // c + 100 // Invalid code, as '100' is an int, and c is a real
3     c + 100.0 // valid code, '100.0' is a real number
4     // c + (100 as real) // also valid code, '100' is denoted as a real number
5 }
```

Fig. 21. In VDM, an integer is a subtype of a real number. This means that it can used as a real value without any additional casting required. The lack of this automation in Dafny means that expressions would need additional context on both how Dafny will interpret a literal value, and on the intentions of an expression. The tool is currently not capable of either of these.

There are some cases when using Dafny that type casting is required. This is automatically done in VDMSL, as any type can automatically be cast into any of it's super-types. That is, a bounded synonym type can become it's synonym; a natural number is automatically cast into integers or reals; etc. This can happen for any subtype, and is a useful language feature for producing easy to read expressions. However, this does not translate well to Dafny. As shown in Figure 21, both sides of a binary expression need to have the same type for it to properly be interpreted.

This is addressed within unary expressions in the project, as the expected type of the expression is checked against the argument, if there is a discrepancy there, then it will include an 'as' expression in the translation strategy to mitigate the problems caused by this issue.

Improved errors and warnings

While the errors and warning system of the project makes use of the VDMJ compiler's inbuilt console system, and builds on top of the system used in the VDM2Isa project, the list of errors is not exhaustive, and some Dafny errors can slip through. For this, an ideal extension of the project would be to improve the error/warning detection system to warn against possible Dafny errors, such as infinite sets, lack of witness values, insufficient loop specifications, etc.

Use of other plugins and modules

Another feature that would be useful to add for both the VDM and Dafny code, would be to make use of the witness annotation from the 'annotations' plugin in the VDM toolkit. This would allow for witness values to be provided and automatically translated into the Dafny witness clause for subset types. Further, special cases could be added for certain

modules — for instance the print statements in the publicly distributed IO module could be translated as Dafny's inbuilt print statements.

5 Conclusions and further work

To conclude, while the project does tackle a majority of VDM features, there are some commonly used grammar constructs that are not supported by the tool. This can produce some disappointing results, especially when considering that several common VDM patterns are among these unsupported expressions. Some of these issues will be resolvable with future work, and partial progress has been made on the majority of these resolvable issues. But the lack of expressive patterns in Dafny is a major limiting factor that could prove fatal to the majority of translation cases.

5.1 Alternative approaches

As mentioned in Section 3.3.1, there are numerous strategies that can be used to produce a translation tool. For this project, a grammar-based approach was taken, where the grammar rules present in the language manuals for both target languages [8] [15] were used to inform the templates that would be used for translation. However, due to time constraints, a limited view of the grammar rules was taken — and these were written with the project in mind, rather than the target language. As a result of this, it would likely be beneficial for the future work on this project to properly write out templates for all grammar rules in Dafny from a Dafny-first perspective. This would produce a more complete template of the Dafny language, which could possibly lead to higher quality translations, as well as improve the context in which the templates could be used.

Regarding the AST traversal section of the project, currently, the AST is traversed by percolating translate calls from the top level module to each internal definition, which then call the 'translate' method of the expression types. While this approach is valid and works to produce valid Dafny code, it would likely be better to make use of VDMJ's in-built class mapping, as mentioned in Section 2.3 — which could have been used to produce proper mappings of each VDMJ class. This approach would have ensured that no classes are erroneously missed out, but would have required learning how to use an additional framework, on top of the existing learning that needed to be done for this project. For this reason, it was decided to produce the classes in a way that would work for the purposes of this project, but would also be able to be extended to make use of this class mapping in the future.

5.2 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.

Stylistically, this extension of the project could benefit greatly from Dafny's module refinement capabilities. In the VDM2Isa project, these proof obligations are generated and discharged in a separate module. This pattern would be useful in Dafny as well, as it allows for proofs to be discharged in a separate file; and would not overwhelm the original translation with additional proof obligations scattered throughout the program text.

5.3 Further work

Due to the nature of this project, there are several things that have been left for further work, and these have been mentioned throughout this report. As a result, this section will simply describe the primary targets for future improvements to the project. A top priority should be implementing a context system that allows for the proper integration of field, constructor and deconstructor expressions for union types; this will bring the project one step closer to fully implementing VDM union types to the best ability of the Dafny language.

On top of this, implementing a translation strategy for state definitions would be ideal, as currently these are left for the user to implement. There are a lot of options to explore in regard to this, and as such it will likely take some time to settle on a strategy that best captures the VDM specification. And finally, it would be ideal to improve the helper Dafny module to include all distributed expressions, the iota expression, among other VDM features which do not appear in native Dafny.

References

[1] ANTLR Team. StringTemplate4: Github Repository. https://github.com/antlr/stringtemplate4, 2024.

[2] Various Authors. Overture Tool: VDMSL Examples. https://www.overturetool.org/download/examples/VDMSL/, 2009.

[3] Nick Battle. Analysis separation without visitors. Sep 2017.

[4] Nick Battle. VDMJ Design Specification. https://github.com/nickbattle/vdmj/blob/master/vdmj/documentation/DesignSpec.pdf, Nov 2023.

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

[6] Nick Battle. VDMJ LSP Plugin Writer's Guide. 2024.

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

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

[9] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In *European Symposium on Programming*, 2013.

[10] Leo Freitas. Topologically sorting VDM-SL definitions for Isabelle/HOL translation, Apr 2023.

[11] Leo Freitas. VDM Toolkit. https://github.com/leouk/VDM_Toolkit, 2024.

[12] K. Rustan M. Leino. Can static variables exist in dafny. https://stackoverflow.com/a/55032477, 2019.

[13] K. Rustan M. Leino. Program proofs code examples. https://www.program-proofs.com/code.html, 2023.

[14] 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.

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

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

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

[18] Rustan Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School 2008*, June 2008.

[19] Jonas Lund, Lucas Jensen, Nick Battle, Peter Larsen, and Hugo Macedo. Bidirectional UML Visualisation of VDM Models, Apr 2023.

[20] Microsoft RiSE Group. Boogie: An intermediate verification language. https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/, 2008.

[21] Microsoft RiSE Group. Dafny: A language and program verifier for functional correctness. https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/, 2008.

[22] Nico Plat and Peter Larsen. An overview of the ISO/VDM-SL standard. *ACM SIGPLAN Notices*, 27, 09 1994.

A Language coverage

A.1 Complete coverage

- Literal expressions.
- Boolean binary expressions.
- Arithmetic binary expressions.
- Arithmetic unary expressions.
- Function call expressions.
- Map and sequence access expressions.
- Function composition and iteration.
- Subsequence and sequence specific expressions.
- Collection size expressions (cardinality, length...).
- Set comprehensions.
- Map comprehensions.
- Collection enumerations.
- Domain/range restriction by/to.
- Conditional expressions.
- Quantifier expressions.
- Is expressions.
- Lambda expressions.
- Mu expressions.
- Narrow Expressions
- Implicit/explicit function and operation definitions
- Synonym type definitions.
- Invariant type definitions.
- Record/compose type definitions.
- Product type definitions.
- Imports and exports.
- Value definitions.
- Module structure.
- Atomic statements.
- Block statements.
- Call statements.
- Conditional statements.
- Case statements.
- Forall loops.
- Return statement.
- External clauses.

A.2 Partial coverage

- Assignment expressions/statements — some work needs to be done to implement a proper selection of strategy. This would involve checking the patterns against a list of compatible patterns with each strategy; and selecting the most applicable.
- Field expressions — possible to implement and is planned in future work.
- Cases expressions — impossible to implement due to pattern limitations in Dafny.
- State — read and write permissions are covered, but state definitions are tricky. It would likely require a lot more work focused on this to produce a valid translation strategy for state — and unfortunately time constraints for the project do not permit such an in-detail look at this.
- Make expressions — these are implemented fully for all types, barring specific instances for union types. This would be implemented alongside field expressions.
- Patterns — while all patterns are covered and translated in this project, there are instances where the patterns are incompatible with Dafny, and there are no suggestions made to rectify this other than rewriting the expression manually. For this reason, patterns are left as partially complete.
- Indexed for loops — while there is a translation strategy in place, Dafny cannot fully support all of VDM's capabilities in this; as the VDM 'by' clause is omitted from Dafny's specification.
- While loops — these are translated to the best of the tool's ability to Dafny, however, these will almost certainly be incomplete translations since Dafny's loop specification requirements can be too strict for the direct VDM translation.
- Iota expressions — current implementation passes syntax, semantic and proof checks but cannot compile due to internal C# errors with the bind definitions.
- Exists1 expressions — current implementation is compilable, properly evaluates, and ensures correct properties. But it cannot be used in assertions currently, which is a slight limitation that should be addressed.

A.3 No coverage

- Sequence comprehensions — these work in a significantly different way in Dafny compared to VDM, and as such are off-limits for this project.
- Distributed expressions — These are not possible to encode into Dafny functions.
- Error statement — Dafny does not have error statements in the same way, and would require significant rewriting of the codebase to make it error-compliant.
- Non-deterministic statement — Dafny does not support non-determinism.