

Bidirectional UML Visualisation of VDM Models

Jonas Lund¹, Lucas Bjarke Jensen¹, Nick Battle², Peter Gorm Larsen¹, and
Hugo Daniel Macedo¹

¹ DIGIT, Aarhus University, Department of Engineering,
Finlandsgade 22, 8200 Aarhus N, Denmark
{201906201,201907355}@post.au.dk, {pgl,hdm}@ece.au.dk
² Independent nick.battle@acm.org

Abstract. The VDM-PlantUML Plugin enables translations between the text based UML tool PlantUML and VDM++ and has been released as a part of the VDM VSCode extension. This enhances already extensive feature-set of VDM VSCode with support for UML. The link between VDM and UML is thoroughly described with a set of translation rules that serve as the base of the implementation of the translation plugin. This is however still an early rendition of the plugin with limited usability due to the loss of information between translations and a lack of workflow optimisations, which we plan to solve in the future.

1 Introduction

As computer-based systems grow in complexity, the necessity for applying formal methods increase as well. One such method is the Vienna Development Method (VDM), which has found success in both industrial and academic environments. The object-oriented (OO) dialect of VDM, VDM++, extends it with classes, objects, inheritance, among other object-oriented features allowing for the modeling of complex software systems.

An abstract visual representation of the architecture of a system can be provided using Unified Modeling Language (UML) [9] class diagrams. This visualisation is useful both when communicating the structure of a system and when designing the architecture itself. Because of this, multiple couplings between various VDM and UML tools have been made. However, the open-source IDE for VDM, the Overture tool, previously being based on the Eclipse platform, is now taking steps to becoming a Visual Studio Code (VS Code) extension [12,11], where UML support is yet to be implemented.

This presents the opportunity to develop a coupling with a faster workflow and support for a more thorough representation of VDM components. This coupling was proposed in a paper for the Overture Workshop 2022 [8] as a VS Code plugin that would enable a bidirectional translation between the open-source text-based diagram tool PlantUML and VDM VSCode.

Since then, the VDM-PlantUML plugin has been developed and released on VDM VSCode. Some changes were made to the proposed architecture, such as

omitting the use of class mapping and instead opting for a simpler architecture using a set of visitor patterns.

Progress has been made in the long term goals mentioned in [8], one of these being the implementation of continual updates on the bidirectional mapping between VDM and UML tools. It is now possible to have the VDMJ language server listen for events, such as a change to a model file, and thus perform the translations continually. To circumvent the information loss that happens between translations, in cooperation with the people at PlantUML, progress is being made on an abstract representation that encapsulates both the specifications from the VDM model as well as the PlantUML model. This would allow users to write complex VDM specifications and translate them back and forth between UML and VDM, without losing any information included in the original VDM specification.

Another goal mentioned was static analysis of models that would warn users when a VDM construct lacks a compatible UML counterpart, or ensuring that UML models are properly formatted for translation to VDM. The need for the former is somewhat alleviated since associative VDM constructs that are too complex to be shown in a UML model are still included in the model, now as class members instead of associations. This is a part of the inherent abstraction in UML.

The need for the latter is somewhat enhanced since PlantUML has a very flexible syntax. It is therefore relatively easy to make choices that do not properly map onto VDM and static analysis would therefore be helpful. Progress has been made in this area by defining the subset of PlantUML that describes VDM models formally, called PlantUML-for-VDM, which can be found in this paper's appendix in section 6.1.

2 Background

2.1 VDM++

VDM++ is an object-oriented dialect in the VDM language family [6]. It builds upon VDM-SL while introducing OO concepts such as classes, making it suitable for visualisation with class diagrams.

Many tools that add to the functionality of VDM have been developed. One of these is the command line tool VDMJ which is built on Java [5]. Some of the features VDMJ provides are type checking, proof obligations, and combinatorial test generation, among others.

For VDM VSCode, the features of VDMJ are accessed through the Language Server Protocol [10], letting VS Code act as an interface for VDMJ. One way of implementing a new plugin for VDMJ is by creating an analysis plugin. Analysis plugins make use of the VDMJ interpreter to perform analyses on whatever VDM specification is open in VS Code. It also happens to be how other translation features are implemented on VDM VSCode.

2.2 UML

UML is an industry standard visual tool for designing and modeling software-based systems [9]. It is specified by the Object Management Group (OMG) to provide a semi-formal visual language to give an abstract representation of OO systems. UML is therefore very relevant as an aid in VDM as a tool for designing and communicating the architecture of a system. UML class diagrams in particular help enhance the utility that VDM++ offers, since VDM++ provides an OO feature-set, and class diagrams can visualise how the different objects in a VDM model relate to each other.

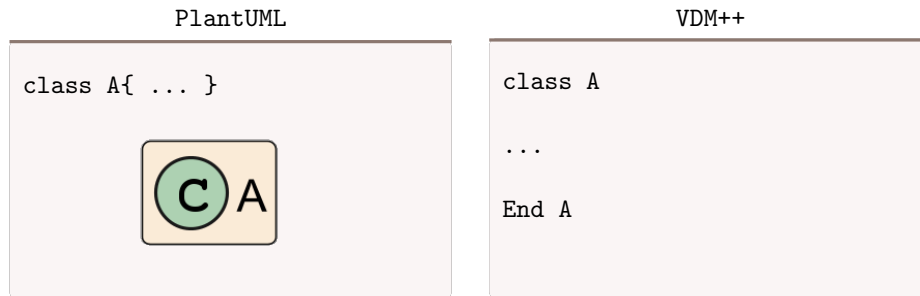
The OMG also maintain their own interchange methods with the XML Metadata Interchange (XMI) standard being used to exchange information using the Extensible Markup Language (XML). XMI is often used as a diagram interchange method for UML models but can be used for any meta model that is expressed in the OMG Meta-Object Facility (MOF).

3 Transformation Rules

In the VDM++ book [7], mapping rules between UML class diagrams and VDM++ are presented. These have been updated to describe the new link between PlantUML and VDM++. The rules are generalised enough to describe any link between VDM++ and UML class diagrams. After a transformation rule is stated, a table showing how the rule occurs in both PlantUML and VDM++ is presented, along with the visual representation of the PlantUML code. In the transformation rule examples, the classes **A**, **B** and **C** are used when the name of the class is relevant, and **Class** is used otherwise. **Type** is used to represent any basic type and any uncapitalised string followed by a number, like **string1**, is the name of a class member, the details of which will be provided in the explanatory text following the rule. For an overview of the subset of PlantUML syntax that describes VDM models, a language manual for is provided in this paper's appendix.

Rule 1: Class Declarations

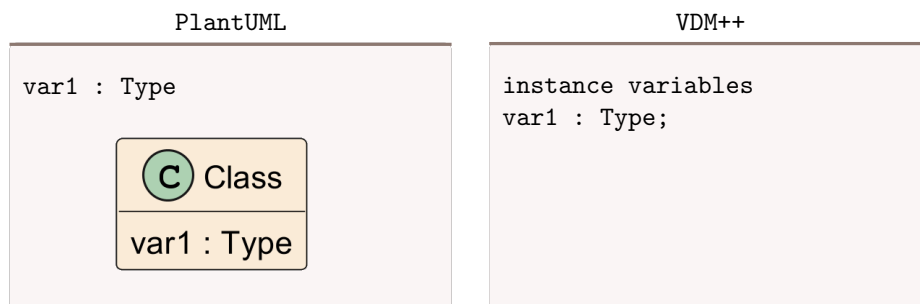
There is a one-to-one relationship between classes in UML and classes in VDM++.



Class members are encapsulated inside class declarations for both the PlantUML and VDM++. However, in PlantUML associations between classes are defined outside class declarations. This means that a VDM++ class member will be defined outside its class in PlantUML, if it is associative. The resulting class diagram shows a single class.

Rule 2: Attributes

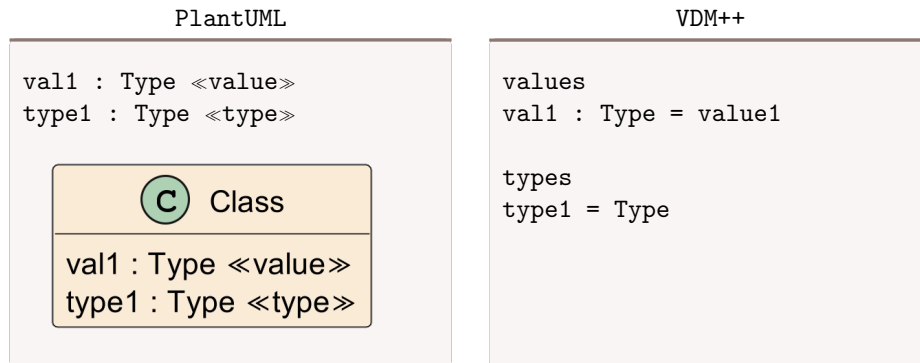
Attributes inside a UML class are represented as instance variables, types or values inside the corresponding VDM++ class.



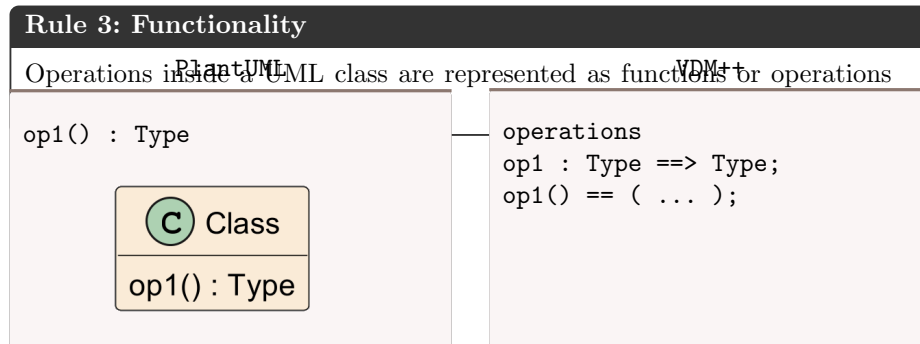
Where **Type** is the type of the attribute and **var1** is its identifier. The resulting class diagram shows a class with a single class attribute.

2.1: Attribute Stereotypes

Instance variables, types and values are differentiated from each other using stereotypes. If no stereotype is used, the attribute is considered an instance variable.



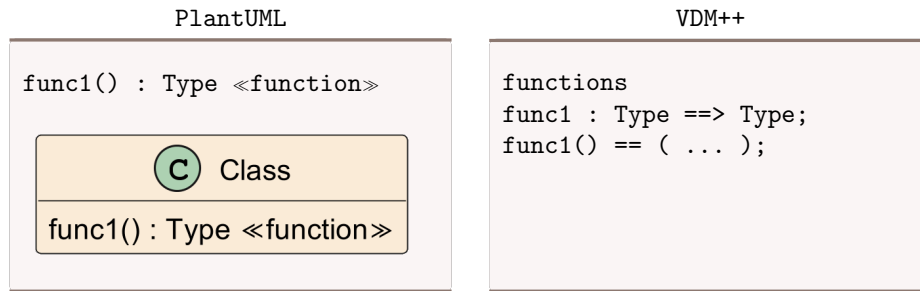
Where **val1** and **type1** are the respective identifiers of the two class members. The stereotype used is the **attribute stereotype** defined in section 6.3. The resulting class diagram shows a class containing two attributes with a stereotype denoting a value and a type, respectively.



Where **op1** is the identifier of the operation. The resulting class diagram shows a class with an operation.

Rule 3.1: Functional Stereotypes

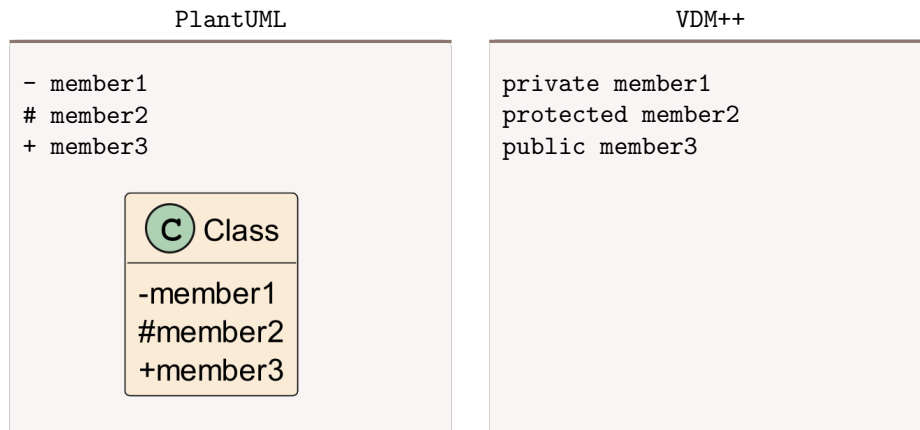
Operations and functions are differentiated from each other using stereotypes. If no stereotype is used, the attribute is considered an operation.



Where `func1` is the identifier of the operation. The stereotype used is the `operation` stereotype defined in section 6.4. The resulting class diagram shows a class containing an operation with an operation stereotype, denoting a function.

Rule 4: Access modifiers

All member declarations can have access modifiers at both the UML class diagram; the VDM++ level and the mapping rules for these are one to one, with the exception that undefined and package at the UML class diagram level are considered as private at the VDM++ level.



Where `member1-3` are the identifiers of the three class members. The resulting class diagram shows three class members, each with a different access modifier.

Rule 5: Static definitions

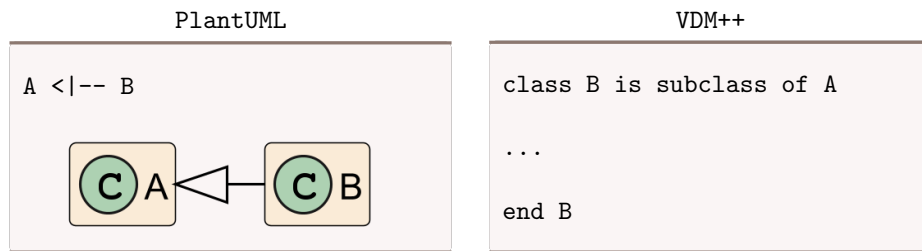
There is a one-to-one relationship between Static member declarations at the VDM++ level and static member declarations at the UML class diagram level.



Where **member1** is the identifier the class members. The resulting class diagram shows the class member underlined to show that it is static.

Rule 6: Inheritance

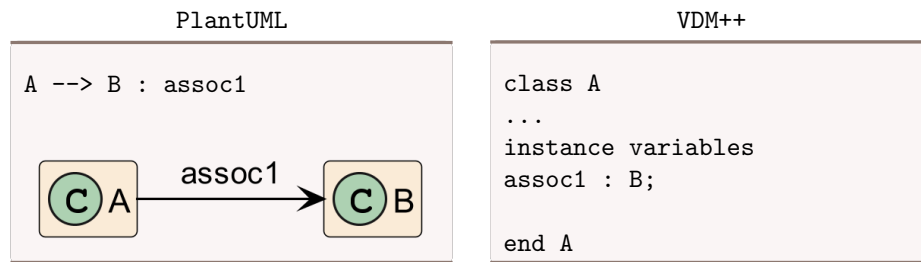
There is a one-to-one relationship between inheritance in UML class diagrams and inheritance in VDM++.



Here, the resulting class diagram shows how class B inherits from class A using the inheritance arrow head.

Rule 7: Associations

Associations between UML classes must be given a role name and a direction. The association is represented in VDM++ as an instance variable in the class at the start of the association, with the role name as its identifier and a value containing a reference to the class at the end of the association.



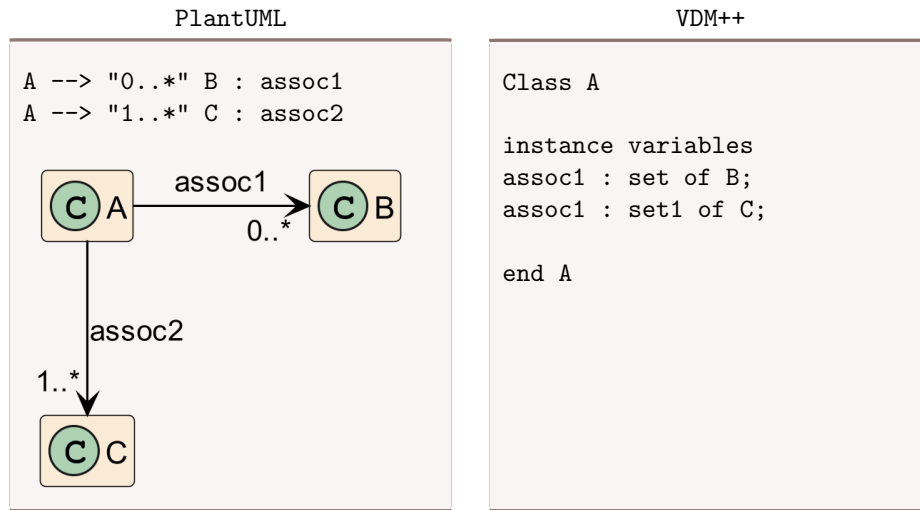
Where **assoc1** is the role name of the association and the identifier of the corresponding instance variable. The resulting class diagram shows an association with the direction denoted by the association arrow, with the role name displayed along the link. For a simple association like this, the value of the instance variable is an object reference type.

Rule 8: Association Multiplicity

The multiplicity and ordering of UML relations determine whether the reference to the class at the end of the association is an object reference type or a compound type with an object reference as its sub-type.

Rule 8.1: Unordered Collection

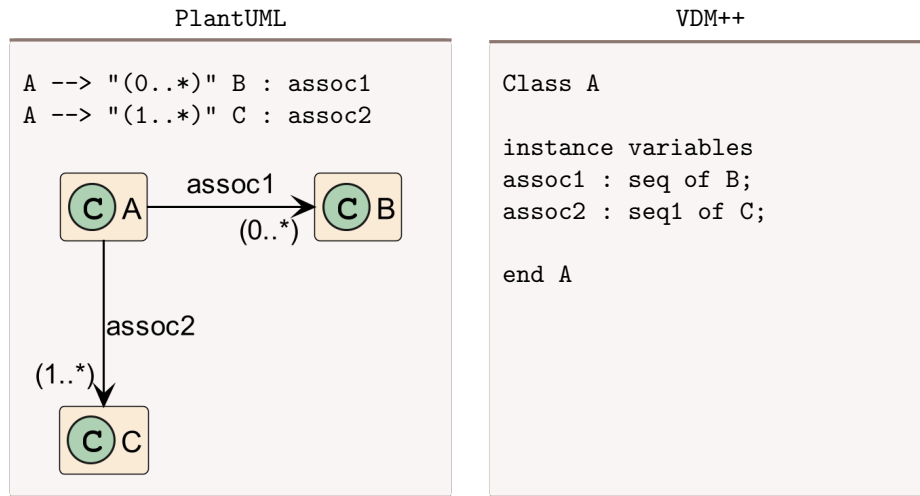
An unordered association with a multiplicity of zero-to-many is modelled in VDM++ as a set of object reference types, using the set type. If the multiplicity is one-to-many, the set1 type is used instead.



Where `assoc1` and `assoc2` are the role names of the associations between the classes. The diagram shows how the association from `A` to `B` has a multiplicity of zero-to-many and the association from `A` to `C` has a multiplicity of one-to-many. In the VDM++ table, we can see how the `set` and `set1` types are used to represent the multiplicity and ordering of the association.

Rule 8.2: Ordered Collection

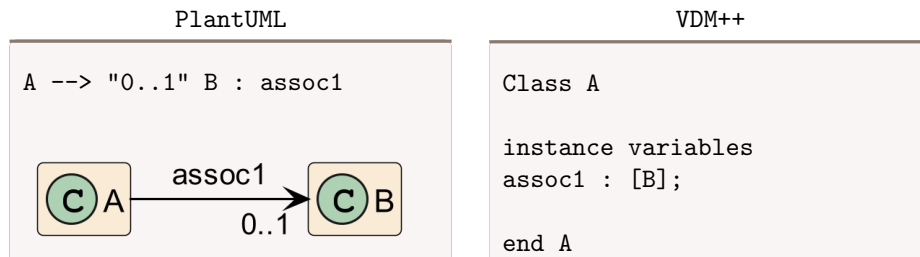
An ordered association with a multiplicity of zero-to-many is modelled in VDM++ as a sequence of object reference types, using the `seq` type. If the multiplicity is one-to-many, the `seq1` type is used instead.



As in rule 8.1, **assoc1-2** are the role names of associations between classes, but here the **seq** and **seq1** type is used to represent the zero-to-many and one-to-many multiplicity instead, since the collection is now ordered. In PlantUML-for-VDM, an ordered collection is specified by parenthesising the multiplicity.

Rule 8.3: Optional Association

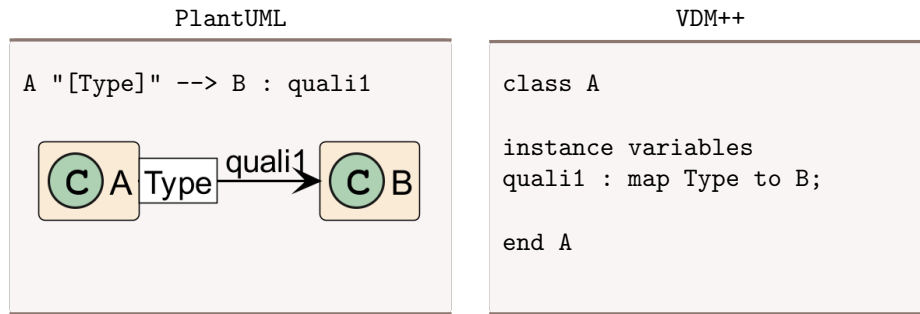
Any association with a multiplicity of "0..1" is modelled in VDM++ as an optional type with an object reference type as the sub-type.



Since the optional association is at most a single element and therefore not a collection, ordering is not considered.

Rule 9: Qualified Associations

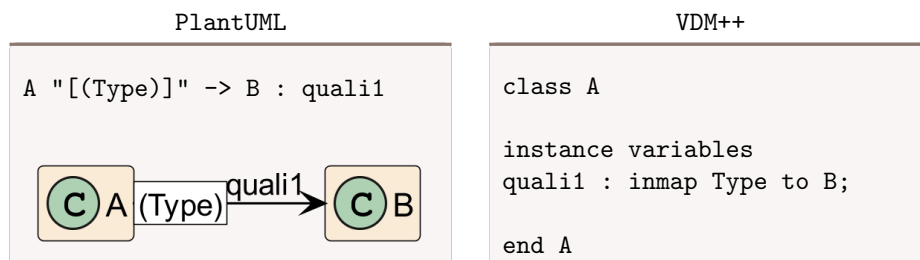
Qualified associations use a type (or a class name) as the qualifier and are modelled as an instance variable containing an association from the qualifier type to an end sub-type using the map type.



Where **quali1** is the role name of the association and **Type** is the qualifier for the association. The resulting class diagram shows a qualified association indicated by showing the qualifier in a white box outside the class at the start of the association.

Rule 9.1: Unique Association Ends

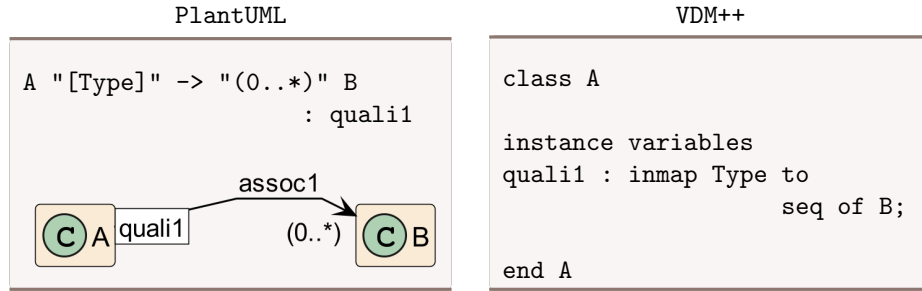
If the mapping from the qualifier to the end sub-type is unique, the association is modelled as an injective mapping instead, using the 'inmap' type.



A qualified association with unique association ends is differentiated from a regular qualified association in PlantUML by parenthesizing the qualifier.

Rule 9.2: Multiplicity in qualified Associations

For qualified associations, the reference to the class at the end of the association is still subject to rule 8, however the reference in the resulting instance variable in VDM++ is located in the second sub-type of the map type.



4 Implementation

The plugin that translates to and from PlantUML is implemented as an analysis plugin [4] which operates using the language server of VDMJ [10]. Implementing the translation as an analysis plugin allows for reusing the `TranslateFeature` which is how other translations such as VDM-to-Word or VDM-to-LaTeX are defined on the client side. It is then as simple as adding the plugin jar to the client's resources and adding VDM-to-UML and UML-to-VDM to the list of possible translations of the `TranslateFeature`.

Another possible implementation using the VDMJ framework would have been as a command plugin. The translation would then only be callable from a VDM debug console which is more hectic to use than simply pressing a button.

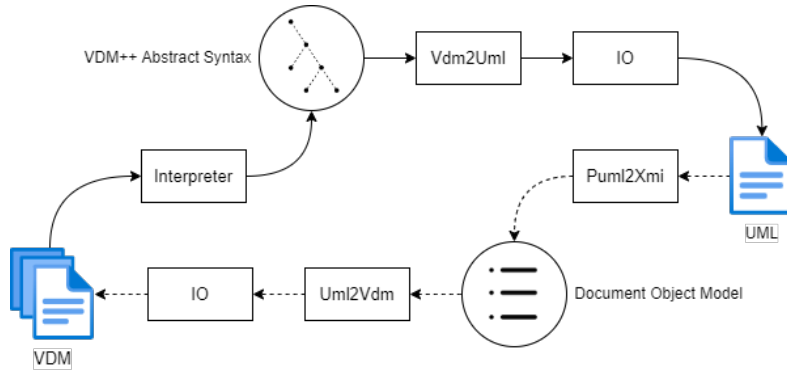


Fig. 1. Architecture of the translations. The dashed line shows the direction from UML to VDM and the solid line is VDM to UML.

4.1 VDM-to-UML

The translation to PlantUML can process either an entire workspace folder or a class from a single file. Since the TC interpreter is running for the currently active workspace, a single file can only be handled by checking whether the selected URI is a path or not, then assume that the **UMLGenerator** should be applied to a class with the same name as the file, meaning that single files can only be handled under the condition that a class it contains must have the exact name as the file containing it.

The translation from VDM to UML is implemented using a couple of different TC visitors. One is for traversing classes and their definitions, these being the operations, variables, etc. of the class. What cases are needed is chosen based on what concepts are desired to be shown in the UML output.

The types of the definitions, such as the type of a variable or the return type of an operation, do exist in the nodes of the definition visitor. However, when using these, the types are printed in a way that is undesirable for UML, being either too long or formatted incorrectly for the case. For types another visitor is then used called the **UMLTypeVisitor** which extends the functionality of the **TCLeafTypeVisitor** class. This visitor pattern can traverse the nodes of a VDM type and process them to our liking.

Following the PlantUML formatting details described in section 3, classes, the attributes within the classes, and associations between classes are written to string buffers and used as output for the final `.puml` file. Since the associations are written separately from the classes and their attributes, two string buffers are used to distinguish these definitions.

4.2 UML-to-VDM

The translation from UML to VDM uses XMI as an intermediate format between PlantUML and the VDMJ analysis plugin. This is however a tighter link than previous VDM-UML couplings and is not meant to receive XMI from any other source than PlantUML.

The contents of the XML file are parsed using a DOM Parser, and the three main elements, the **UML:Class** element, the **UML:Generalisation** element, and the **UML:Association** element are extracted from the XML file. The **UML:Class** element has two kinds of XML attributes, namely **UML:Attribute** and **UML:Operation**. These map directly on to the **VDM:Attribute**, and **VDM:Operation** components, with stereotypes being used to differentiate between the sub-components. The **UML:Generalisation** element has two XML attributes that contain a unique class-ID's of the child and parent class, respectively. The XMI information is mapped onto the **VDM:Class**, **VDM:Attribute**, and **VDM:Operation** components following the mapping rules described in section 3. An overview of the mapping of XMI to VDM constructs, can be seen in the figure below.

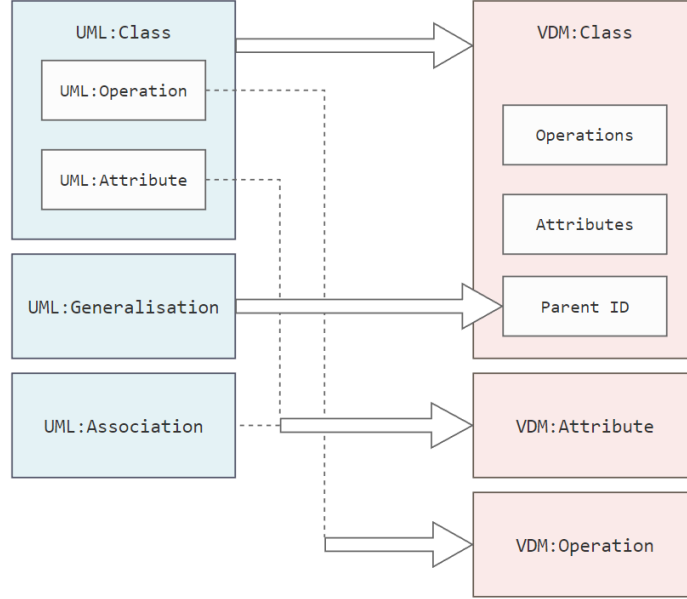


Fig. 2. Overview of UML2VDM mapping from XMI to VDM constructs

5 Conclusion

As of 12 October 2022, the `vdm-plantuml-plugin` has been offered to users in the newest VDM VSCode release and has already found a use in VDM-specific university courses. The plugin now enables VDM VSCode users to swiftly jump between VDM++ models and visually appealing UML models without dealing with difficult-to-manage file formats and the fuller support of VDM specific components in UML, such as types and functions, enriches the overall VDM experience.

The coupling to PlantUML also increases the interoperability of VDM since PlantUML, as a popular diagram tool, is itself integrated into a large number of tools [2]. Furthermore, the door have been opened to further collaboration with the PlantUML team, which may lead to new ways of using UML in a VDM context 5.1.

The nine transformation rules between UML and VDM presented in this paper abstracts the PlantUML link to be applicable to any number of UML tools that support the sufficient VDM components. The PlantUML-for-VDM language manual serves as a guide to writing PlantUML that correctly maps to VDM, but is also as a starting point for constructing a syntax checker for creating VDM models in PlantUML.

One significant drawback of using PlantUML is that the graphical visualisation software GraphViz [1], which PlantUML uses for class diagrams, does not support manually manipulating the positions of the placed classes or the

connections between them by clicking and dragging. There are limited ways of changing the layout of a PlantUML diagram, however it would be more ideal to have something more granular to make the diagrams more custom and allow for better workflow and user experience.

5.1 Future Work

VDM-RT/VDM-SL Support The plugin was developed and tested with the VDM++ as the main dialect. This is because the object-oriented nature of VDM++ lends itself well to visualisation of the classes a VDM++ project consists of. While VDM-RT transformations are possible, VDM-RT specific constructs such as threads are yet to have their own representation in the plugin. Furthermore, there are plans to allow support for VDM-SL transformations, as a VDM-SL specification can be represented as a singular class.

Implicit Stereotypes Strides have been made to hide the use of stereotypes from the user, and instead use a definition block structure, mirroring what is done in VDM to group and denote definition statements. A successful implementation of the definition block structure in PlantUML-for-VDM would still utilise stereotypes in the XML exchange between PlantUML and VDM VSCode, so no change in the transformation rules will be necessary.

Dynamic bidirectionally In its current the state, the plugin is suited for presenting the structure of large VDM projects or giving a project a head start by first doing a visual UML model and converting it VDM afterwards. These use cases are mostly one time uses per project and does not bring UML translations to their full potential, workflow-wise. The next step for the link between VDM and UML is the ability to have dynamic bidirectional transformations, meaning that you can work on either the VDM or the UML model and see the effects of those changes immediately take place in the other project. This requires significant changes to the architecture of the link. Progress however is already being made on VDMJ, where a functionality for reacting to certain events using `EventHub` [3] is coming along.

This functionality will also require a new representation of VDM that can help decide what should and should not be translated to UML and retain the parts that are not translated. Doing so will allow the user to write more complicated models while keeping the simplicity of the UML representation.

Static analysis of VDM-UML Static analysis in PlantUML-for-VDM would increase the ease of use when creating a model in PlantUML helping the user with creating a PlantUml model that is compliant with the PlantUML-for-VDM language manual and therefore fit for translation to VDM.

Acknowledgements We acknowledge the Poul Due Jensen Foundation that funded our basic research for engineering of digital twins.

References

1. Graphviz documentation, <https://graphviz.org/documentation/>
2. Running, <https://plantuml.com/running>
3. Battle, N.: Eventhub class on the vdmj github, <https://github.com/nickbattle/vdmj/blob/master/lsp/src/main/java/workspace/EventHub.java>
4. Battle, N.: Lsp plugin writers guide, <https://github.com/nickbattle/vdmj/wiki/LSP-Plugin-Writers-Guide>
5. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
6. Dürr, E., Katwijk, J.: VDM++, A Formal Specification Language for Object Oriented Designs. In: COMP EURO 92. pp. 214–219. IEEE (May 1992)
7. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer, New York (2005), <http://overturetool.org/publications/books/vdoos/>
8. Lund, J., Jensen, L.B., Macedo, H.D., Larsen, P.G.: Towards UML and VDM Support in the VS Code Environment. In: Macedo, H.D., Pierce, K. (eds.) Proceedings of the 20th International Overture Workshop. pp. 51–66. Overture (7 2022)
9. OMG: About the unified modeling language specification version 2.5.1 (december 2017), <https://www.omg.org/spec/UML/2.5.1>
10. Rask, J., Madsen, F., Battle, N., Macedo, H., Larsen, P.: The specification language server protocol: A proposal for standardised lsp extensions (05 2021)
11. Rask, J.K., Madsen, F.P., Battle, N., Macedo, H.D., Larsen, P.G.: The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions. In: Proença, J., Paskevich, A. (eds.) Proceedings of the 6th Workshop on Formal Integrated Development Environment, Held online, 24-25th May 2021. Electronic Proceedings in Theoretical Computer Science, vol. 338, pp. 3–18. Open Publishing Association
12. Rask, J.K., Madsen, F.P., Battle, N., Macedo, H.D., Larsen, P.G.: Visual Studio Code VDM Support. In: Fitzgerald, J.S., Oda, T. (eds.) Proceedings of the 18th International Overture Workshop. pp. 35–49. Overture (December 2020), <https://arxiv.org/abs/2101.07261>

6 Appendix

6.1 PlantUML-for-VDM Language Manual

This appendix contains formal definitions for the subset of PlantUML that describes VDM++ models, called PlantUML-for-VDM. At the moment, only the VDM++ is covered, but this will be expanded upon, as the other dialects are supported in the VDM-PlantUML plugin. The purpose of the manual is to serve as a guide for writing PlantUML that can be successfully converted into a VDM model and to establish the proper syntax, for future developments.

Where **class** is the identifier of the associating object, **class'** is the identifier of the associated object and **variable** is the identifier of the instance variable that is defined by the association.

```

qualification = general map type
                | injective map type ;

general map type  = "["', type, "]"' ;
injective map type = "["(' , type, ')"]' ;

multiplicity = set type
                | seq type
                | optional type ;

    set type = set0 type
                | set1 type ;

        set0 type = "*" ;
        set1 type = "1..*" ;

```

```

    seq type = seq0 type
                | seq1 type ;

        set0 type = "(*)" ;
        set1 type = "(1..*)" ;

    optional type = "(0..1)" ;

```

6.6 Access Definitions

Access is defined as:

```

access member definition = [ visibility ], [ 'static' ]
                            | [ 'static' ], [ visibility ] ;

```

```

visibility = '+'
            | '-'
            | '#' ;

```

For public, private and protected, respectively. The default visibility to any component is private.

7 VDM2UML Type Abstraction

The VDM2UML type abstraction affects how compound types are represented in UML and can prevent class diagrams from becoming cluttered and verbose. The downside to this is that the translation is no longer bidirectional since information about types may be lost. The VDM type abstraction splits VDM compound types into two groups. The groups are the primary compound types, C_0 and the secondary compound types, C_1 .

$C_0 = \text{set, seq, map, optional}$
 $C_1 = \text{product, union}.$

Each group has a different capacity determined by $\gamma_0, \gamma_1 \in \mathbb{Z}^*$, for C_0, C_1 respectively. The capacity determines how many compound types any given type can compose, before it is deemed too complicated for UML and therefore in need of abstraction. A compound type with multiple compound types within it, will belong to the group of the outer compound type. All non-basic types in the inner type count towards the capacity. If the capacity is reached, abstraction will be done in accordance to the group the type belongs to.

```
abstraction = C_0 abstraction
             | C_1 abstraction ;

C_0 abstraction = 'seq of', type_a
                 | 'set of', type_a
                 | '[' type_a ']'
                 | 'map', type_a
                 | basic type, 'to', type_a
                 | basic type ;
```

```

type_a = c_0'
        | c_1' ;

c_0' = set...
      | seq...
      | [...] ;

c_1' = '*', {'*'}
      | '|', {'|'} ;

C_1 abstraction = c_1' ;

```

The capacity for a map type is $2\gamma_0$, since the map type has a minimum of two sub-types. This is also why a map type can have a basic type as one of its sub-types and still be abstracted, if the other sub-type consists of enough compound sub-types to exceed the capacity.

For c'_1 , the number of symbols used is given by $n - 1$ where n is the number of sub-types in the non-abstracted compound type.