Towards UML and VDM Support in the VS Code Environment

Jonas Lund, Lucas Bjarke Jensen, Hugo Daniel Macedo, and Peter Gorm Larsen

DIGIT, Aarhus University, Department of Engineering,
Finlandsgade 22, 8200 Aarhus N, Denmark
{201906201,201907355}@post.au.dk, {hdm,pgl}@ece.au.dk

Abstract. The coupling between the object-oriented dialects of VDM (VDM++ and VDM-RT) and UML, found on The Overture Tool has been left behind due to the shift in focus onto VDM VSCode. The paper first presents how the coupling is reestablished in VS Code. However, it is subsequently proposed that UML visualisations of VDM models can be further improved by using the text-based diagram tool called PlantUML, as it is supported as a VS Code extension and offers several advantages compared to other UML tools. The best integration of PlantUML requires a direct translation between VDM and PlantUML, which is made possible by class mapping introduced in VDMJ 4.

1 Introduction

The modelling of critical and embedded systems is a complex task with many parts. One of the most popular and mature languages designed for modelling such systems is the Vienna Development Method (VDM) which is extended by the two object-oriented (OO) dialects, VDM++ and VDM Real-Time (VDM-RT) [6]. Some advantages of modelling using the OO dialects are that encapsulation of methods and data along with abstraction of the internal state of these classes allows for higher degrees of model complexity.

The task of modelling these systems is complex and requires a deep understanding of the modelling tools in use. The widely used Overture Tool built on top of the Eclipse IDE allows for easier modelling integrating several debugging tools and plugins. One of these plugins is the transformation from and to Unified Modelling Language (UML) files [9]. This connection was developed because UML class diagrams are a fitting and intuitive way of visualising OO models.

The Eclipse version of Overture is slowly growing obsolete as a new contender rises, the VDM VS Code extension, letting Visual Studio Code (VS Code) function as an IDE for modelling in languages from the VDM language family. Many of the features from Eclipse have been ported to the extension already, but the UML connection has yet to be incorporated.

The UML transformations for The Overture Tool culminated in the coupling with the visual modelling tool, Modelio. However, the coupling has its flaws and shortcomings with regards to visualising the generated UML models and exporting new ones. New tools that integrate more closely with modeling tools have been developed and it may be worthwhile looking into whether the connection between VDM and UML can be repurposed for such tools.

This paper will be tackling two tasks. The first is covering the work that has gone into porting the already existing UML connection from the Eclipse version of Overture to the new VDM VSCode extension. The second is discussing other tools for visualising UML and designing a new transformation plugin for this tool.

2 Background

2.1 Vienna Development Method and Unified modelling Language

The Vienna Development Method (VDM) is a leading formal method, focused on the modelling of computer-based systems while ensuring safety and security in the software before deployment [8]. There exist two dialects that extend the original VDM language, the VDM Specification Language (VDM-SL). These are the object-oriented extensions called VDM++ and VDM Real-Time (VDM-RT), which support concurrency and real time modelling as well. Both VDM++ and VDM-RT use classes to structure their models. A visual representation of a model helps in designing and communicating the architecture of the system. This is at its most effective via class diagrams using the Unified modelling Language (UML) [10]. UML is specified by the Object Management Group (OMG) and uses a semi-formal visual language to give an abstract representation of object-oriented systems.

A set of transformations between VDM and version 2 of UML exist on the Overture Project as a plugin for the Overture Tool. [8] These transformations were made with Modelio in mind as the UML modelling tool, both for import of generated UML files from VDM and for export of class diagrams to be translated to VDM.

2.2 From The Overture Tool to Visual Studio Code

The Overture Tool (Overture) is the most mature and popular platform for VDM modelling. However, the focus of the Overture project has shifted to the Visual Studio Code (VS Code) platform in recent years with the development of the VDM VS Code extension developed by Jonas Rask and Frederik Palludan Madsen [12].

This extension establishes a connection to a language server, letting VS Code function as an Integrated Development Environment (IDE), exactly like its Eclipse counterpart, while also having the advantage of being extendable to other languages thanks to the Language Server Protocol (LSP). [11]

The Eclipse UML transformation plugin has been reimplemented as a command on VDM VS Code. This was achieved by packaging the main functions `Vdm2UmlMain` and `Uml2VdmMain` into a JAR file that can be run from VS Code. Currently, this JAR file is located on the client side of VDM VS Code, but will be moved to the VDMJ language server in the future to decouple the specification language functions from the VS Code extension.

2.3 VDMJ

VDMJ is a command line tool that provides basic tool support for Overture. Some of the features it provides are type checking, proof obligations, and combinatorial test

generation among many others. In Overture for Eclipse, VDMJ and its tools are executed through the Eclipse IDE interface. For VDM VS Code however, the VDMJ features are accessed through the LSP, letting VS Code act as an interface for VDMJ. The latest version of the tool, VDMJ 4, lets the user easily reconstruct the VDM Abstract Syntax Trees (AST) for their desired tool related purposes [3].

2.4 PlantUML

PlantUML [13] is a text-based diagram tool that exists as a VS Code extension [7]. Any model made in PlantUML can be opened in a separate tab in VS Code and continually updates as the model is changed. The models can be worked on by multiple team members as it integrates with versioning tools like GitHub. This is possible since the PlantUML diagrams are generated from text files and can therefore be located in the software repository alongside the program or model which the diagram represents.

PlantUML also seeks to allow interoperability between tool vendors as many export formats are supported (including XMI). It also attempts automatic placement of objects in the diagram and allows for potential adjustments from the user, a feature which was not available in Modelio.

One caveat in representing VDM models using PlantUML is that qualified associations, which are commonly used in VDM class diagrams, but alternatives are available such as using square brackets to indicate the qualifier. A typical class diagram that represents a VDM++ model is presented on Fig. 1.

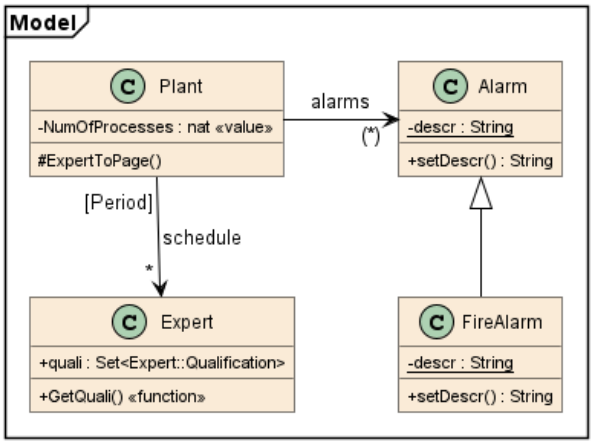

Fig. 1: Resulting PlantUML

PlantUML is even cited in books pertaining to critical [5] and cyber-physical [4] systems. These features make the PlantUML extension one of the more relevant options for fulfilling the goals of this project.

3 Translation Between VDM and UML

This section gives an overview on the migration of the UML connection from Overture to VDM VSCode. To implement the connection, the existing Overture code for converting VDM to UML is packaged as a JAR file and called within VS Code. This method does not involve changing the code which is executed to perform the conversions. It involves the creation of a handler that reads the files, checks the validity of these files, and passes the expected arguments to the converting methods.

This is the first part of the two that this paper will cover. The second part regarding PlantUML is presented in section 4.

3.1 Overture UML Migration to VDM VS Code

In order to port the existing UML connection to VDM VSCode, the connection has to first be decoupled from the Eclipse IDE. This is a necessary step in allowing Maven to package the command in a JAR file. After the decoupling, the main handlers for the UML transformation can be developed.

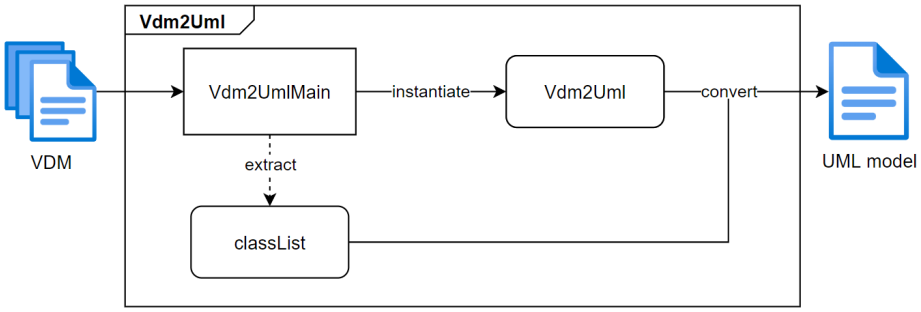

Fig. 2: VDM to UML Architecture Overview

`Vdm2UmlMain` is the transformation handler in the direction of VDM to UML. As illustrated in Fig. 2, it receives the path to the folder containing the VDM files as input. An instance of the `Vdm2Uml` class is constructed, which is the class containing the methods for the transformation. `Vdm2UmlMain` then extracts a list of classes from the VDM files, called `classList`. This list is given as input to the `Vdm2Uml` method called `convert`.

`Uml2VdmMain` is the main function in the direction of UML to VDM. As illustrated in Fig. 3, it receives the path to the UML file as an input. An instance of the `Uml2Vdm` class is constructed, after which the class is initialised using the `initialize` method with the path to the UML file as an argument. The `convert` method is then called, and the corresponding VDM files are created.

The UML transformations are packaged in a JAR file using Maven. The JAR file is integrated into the VDM VS Code environment where it is accessed through a handler

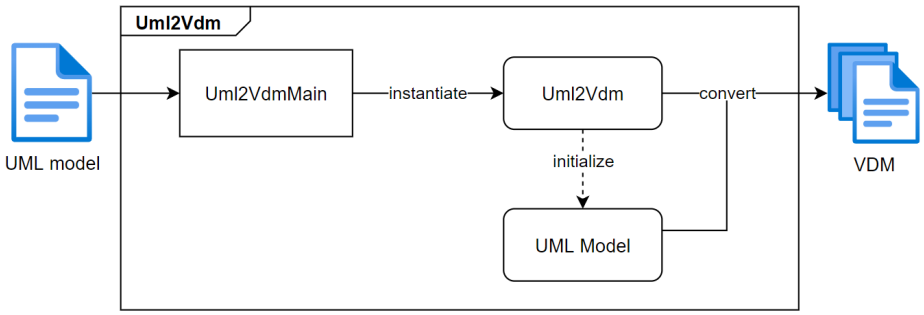

Fig. 3: UML to VDM Architecture Overview

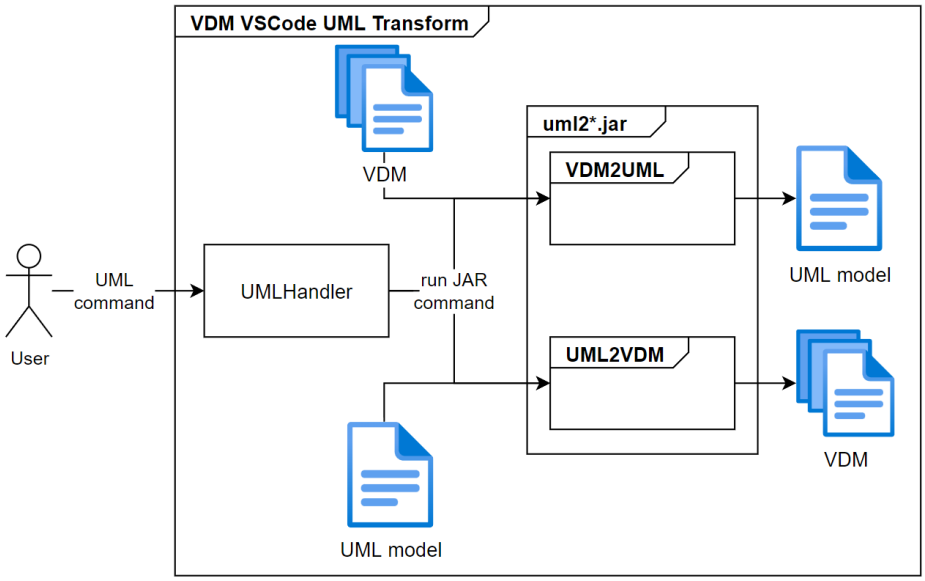

Fig. 4: VDM VS Code Architecture Overview

called `UMLHandler.ts`. This handler is responsible for executing the JAR file with appropriate arguments. Both main functions are contained in the same JAR. The function executed is determined by which command is issued by the VDM VSCode user. The architecture of the JAR file integration is illustrated in Fig. 4.

3.2 Repurposing the Overture Code for UML Transformations

To understand how the VDM VSCode UML transformation handlers will fill the role of the Overture handlers, while enabling VS Code integration, this section will describe what resources the Overture implementation handlers used in the UML transformations, and whether these resources are fit for reuse.

Analysis of the Overture UML Handlers To extract the information needed to perform the UML conversions, the handler in the direction of VDM to UML, called `Vdm2UmlCommand`, relies heavily on the `IVdmProject` class.

The `IVdmProject` class is inherited from the `IProject` class, which is an Eclipse-based interface that grants the notion of a project as an implementation resource. A project encapsulates information about a group of files that share a directory. Furthermore, the need for any file handling is resolved as this is also encapsulated by the Eclipse interface.

The `IVdmProject` class is modified to suit VDM and it therefore provides further abilities that are specific to a group of VDM files. In a similar manner, as `IVdmProject` is derived form the `IProject` class, the `IVdmProject` class contains a method for creating an instance of the `IVdmModel` class, which in turn encapsulate the abstract structure of the VDM model. This model is then used to examine whether the model is syntax and parse correct and therefore fit for UML transformation. The code snippet below shows how the `IVdmModel` class is instantiated and used to check the correctness of the model.

```
final IVdmModel model = vdmProject.getModel();
  if (model.isParseCorrect())
  {
    if (model == null || !model.isTypeCorrect())
    {...}
```

The Eclipse project classes and the functionality they offer are illustrated in Fig. 5.

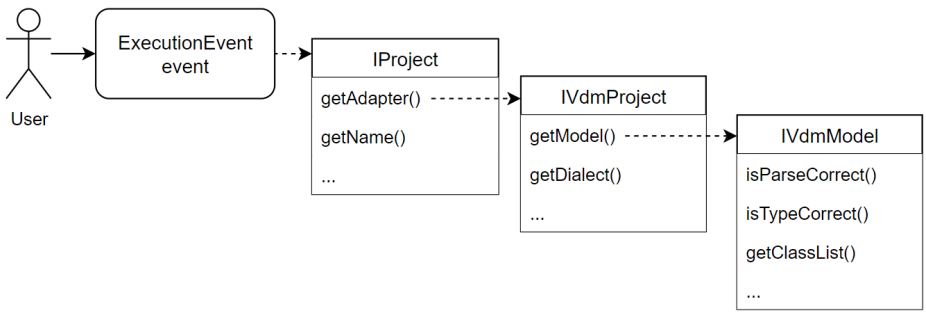

Fig. 5: Overview of IProject its derived classes, and the methods used in the Overture VDM to UML handler

Likewise, the handler in the direction of UML to VDM, called `Uml2VdmCommand` also use Eclipse dependent classes to fulfil its tasks. The class `Uml2Vdm`, which contains the method that performs the conversion needs to be instantiated with the path to the UML file along with information about what dialect the to be generated VDM files should be in. The UML file path is extracted from an instance of the `iFile` class, which encapsulates

the Eclipse notion of a file. This class can then be cast to an `IVdmProject` class, which in turn can be used to discover the dialect of the to be generated VDM files.

Considering the goal of migrating the UML connection to VDM VS Code, the handlers' use of the project classes poses a problem, since their usefulness is dependent on the Eclipse IDE. This stems from the fact that the input to the handler has changed from an `IVdmProject` in Overture, to simply being a list of VDM files in VDM VSCode.

One way to solve this could be to extract information about the current VS Code project, create an intermediate Eclipse project, and then perform the transformation like the Overture UML handlers do. However, this runs into the problem of having to depend on the IDE side. To circumvent this, one might also simply remove the notion of an `IProject` and its derived types. It would then require finding new ways of providing the same functionality as these classes provide.

The VDM VSCode UML Handlers The VDM VSCode handler achieves the same functionality as the Overture handler, without utilising the notion of an Eclipse project. In a lot of cases, this is done simply by accessing the methods used by the `IProject` and its derived classes. This results in the same functionality, but the methods used are accessed at a lower level. This section shows examples of how this strategy is applied.

In the implementation of the Overture handler, the file handling is done implicitly by Eclipse. However, as this is no longer a possibility, the file handling is done internally by `Vdm2UmlMain`. This is achieved by the `Vdm2UmlMain` method '`main`', taking a set of arguments which denote the directory containing the files to be transformed and the directory where the resulting UML file should be placed. The arguments are distinguished using `-folder` and `-output` flags before the path arguments.

On Overture, the handler read the VDM dialect using a method from the `IVdmProject` class. The VDM VSCode handler is instead passed the dialect as a flag, along with the command to execute the JAR file.

While the `IVdmModel` class has an attribute informing whether the model is syntax and type correct, the methods for performing the actual checks are not dependent on the Eclipse project classes, and are therefore fit for reuse. The class list is also provided by the type checker, `typeCheckPp` or `typeCheckRt`, depending on the dialect of the VDM files. The two methods of the class `TypeCheckerUtil` initialise an instance of the `TypeCheckResult` class, which has an attribute containing the class list needed for the conversion.

In the direction of UML to VDM, instead of the path to the UML file being extracted from the `iFile` instance, the path is instead passed as an argument in the command to execute the transformation on the JAR file. Currently the dialect of the resulting VDM files are assumed to be VDM++. The consequence of this is that a UML file containing a VDM-RT model will be changed to a VDM++ model. This short-coming should be trivial to fix in the future.

With the handlers working and ported to the VDM VSCode extension, the UML transformation commands can be run and tested in VS Code. A screenshot demonstrating how the feature is used is shown in Fig. 6.

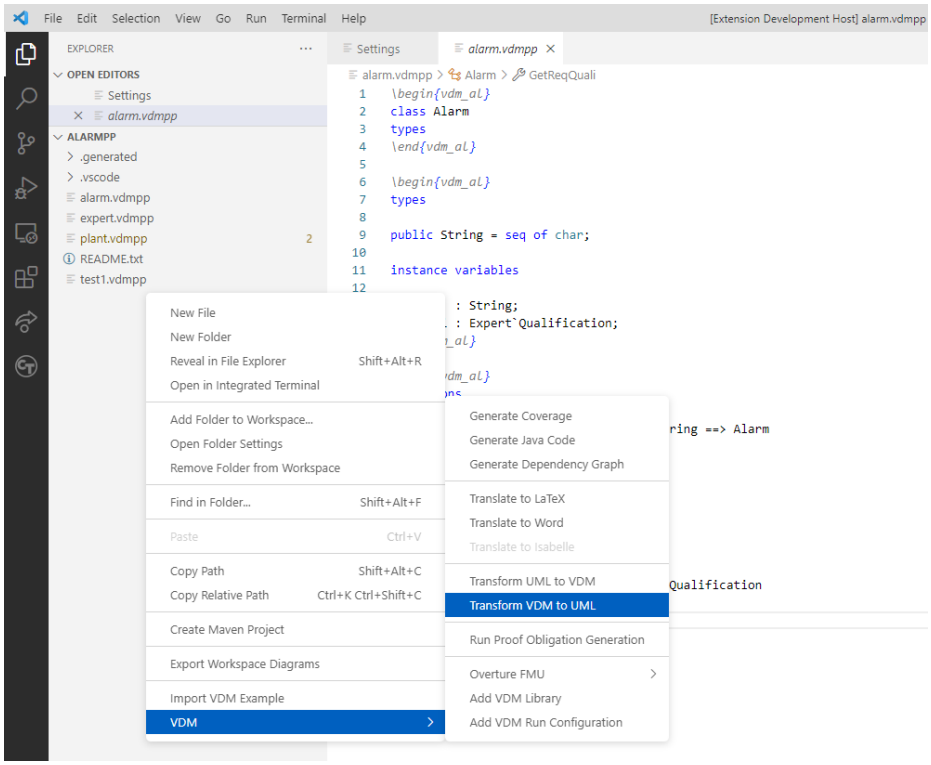

Fig. 6: Screenshot of using the command in VS Code.

4 PlantUML Implementation Planning

This section covers the second part of the paper, which involves describing the motivation behind choosing PlantUML as the primary UML visualisation tool, as well as determining the best way to couple PlantUML with VDM VSCode.

This is almost entirely separate from the porting of the existing UML connection, and will instead dive into the possibility of using the new architecture of the VDM VSCode environment to create a new UML connection.

4.1 Motivation

Currently, the UML transformations are coupled with Modelio version 2.2.1, as the original Overture UML connection was specifically tailored to this tool. This poses some issues:

- The specific version of XMI used is compatible with only some versions of Modelio, and brings little compatibility with any other tools, restricting interoperability.
- There is no spatial information in the standard and Modelio makes no attempt at guessing it, meaning that classes and links must be positioned manually.
- Employing a JAR file that uses Eclipse IDE resources is undesirable, since the Overture project is trying to distance itself from Eclipse with the migration to VDM VSCode. Furthermore, having to rely on Eclipse results in making it significantly more difficult for the UML transformations to achieve maintainability and extendibility.

It would therefore be optimal to use a VS Code extension to visualise UML models that can import and export UML files that are XMI compliant, since this would allow for developing models directly in VS Code. Unfortunately, no such tool seems to exist.

VDMJ version 4, however, introduces class mapping [3], allowing developers to easily reconstruct the VDM ASTs for whichever tool-related purposes needed. It may well be possible to utilise class mapping to develop an AST for a textually based UML tool, that is supported in a third party VS Code extension. This would, in turn, allow a direct translator between UML and this tool to be created, solving the problems laid out and thus achieving a direct connection between VDM and UML on the language server, without relying on outdated Eclipse resources.

4.2 Transformation Overview and Considerations

In order to begin development on the PlantUML transformation using ASTs on the VDMJ language server, some notions must be considered.

The previous UML transformations also use ASTs, specifically for the object-oriented VDM dialects and a UML AST that would be used for converting to the XMI format [9]. It is worth considering how much of the implementation of this previous transformation would be reusable for the PlantUML transformation. Since a UML AST does seem to exist, would it be possible to repurpose it for the new VDMJ, and develop translators between UML and PlantUML? Or is it better to take inspiration from the previously used ASTs and instead create a new AST for PlantUML?

The methods for developing ASTs have improved greatly since the Overture UML implementation with the introduction of VDMJ 4 [3]. This has been achieved with class mapping, and it is because of this progress that it seems viable to produce an AST ourselves for the purpose of UML transformations. An overview of how the transformation between VDM++ and PlantUML could work is illustrated in Fig. 7. The dashed arrows denote the PlantUML to VDM++ direction and the solid lines denote the VDM++ to PlantUML direction.

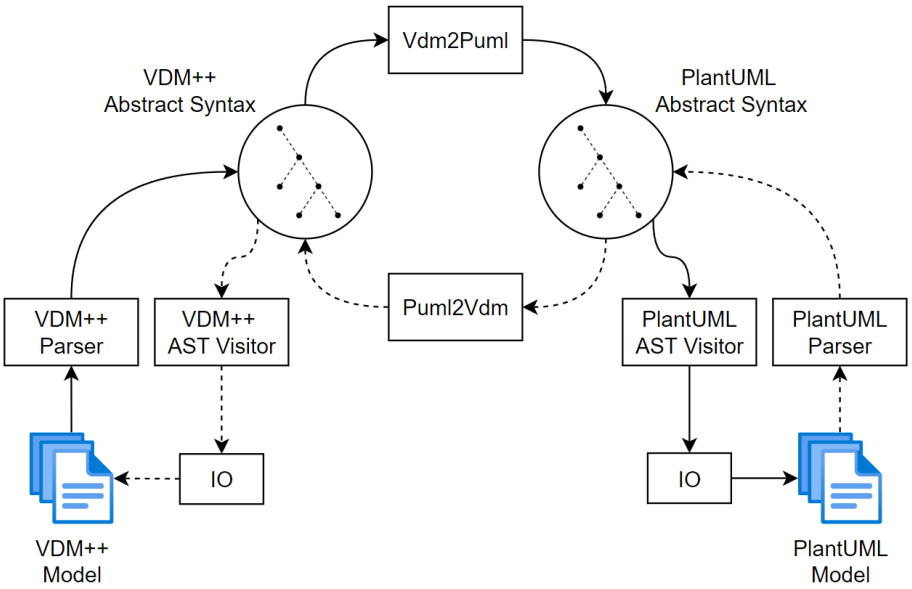

Fig. 7: Overview of the PlantUML transformations using ASTs.

Note that there are several uncertainties as for how the transformations will actually work, since we have yet to properly examine what the process of creating ASTs entails. One uncertainty being how the nodes of the trees are traversed to output the code for both the VDM model and the PlantUML model. It is expected to be done through some

visitor method, similar to how it was done for the previous VDM++ AST [9]. Whether or not the same can be done for the PlantUML AST is also uncertain.

The conversion between the two ASTs is expected to make use of the class mapping mechanism. An example of the class mapping in use can be seen in a simple VDM to C translator [2]. The translator makes use of mapping objects from the type checker (TC) AST to a new translator (TR) AST through a .mappings file. This process of reconstructing existing ASTs for the purposes of a new plugin will make the base for the functions that transform between VDM and PlantUML. However, in this example, it seems that one AST is made for the feature that is translating VDM to C. This poses the question of whether it will be necessary to create ASTs corresponding to the languages involved, or corresponding to the translations that happen between the languages.

4.3 Transformation Rules

In the VDM++ book called Validated Designs for Object-oriented Systems, 12 mapping rules between UML class diagrams and VDM++ are presented [6]. These serve as requirements for the translator between the OO VDM dialects and PlantUML. In Tab. 1, 2, and 3, an example in VDM++ and PlantUML are presented along with the transformation rules.

#	**VDM++**
	PlantUML
1	`Class object1 ... End object1`
	`class object1{ ... }`

2	`class object2`
	`instance variables`
	`asoc1 : object1;`

	`object2 --> object1 : asoc1`
3a	`instance variables` `var1 : Type;`
	`var1 : Type`
3b	`values` `val1 : Type = value1;`
	`val1 : Type «value»`

Table 1: VDM++ to PlantUML Transformation Mapping Table #1-3

1 — Class Declarations in both views are very simple, since "there is a one-to-one relationship between classes in UML and classes in VDM++", as stated in the original mapping rules [6]. The ellipses in the PlantUML view surrounds the contents of the class and all class members are declared within them.

2 — Associations must each be given a role name that denotes the direction of the association. This is represented in VDM++ as an instance variable with the type defined by the type of the destination class of the association. In PlantUML, the direction of an association is denoted by an arrow, followed by the destination. Associations in PlantUML are declared after class declarations.

3a — Instance Variables in the VDM++ view are equivalent to attributes in UML classes, and share a similar notation.

3b — Value Definitions can be differentiated from instance variables using a stereotype to indicate that they should be treated as constants.

#	**VDM++**
	PlantUML

4a	`operations`
	`op1 : Type ==> Type;`
	`op1() == ( ... );`

	`op1() : Type`

4b	`functions`
	`func1 : Type ==> Type;`
	`func1() == ( ... );`

	`func1() : Type «function»`

5	`private member1`
	`protected member2`
	`public member3`

	`- member1`
	`# member2`
	`+ member3`

6	`static member1 ...`
	`{static} : member1`
7	`types` `type1 = Type`
	`{}`

Table 2: VDM++ to PlantUML Transformation Mapping Table #4-7

4a — Operations are defined in both views using a type and a name.

4b — Functions are differentiated from operations using a stereotype in the UML view. The type of function (total, partial, injection, etc.) is also specified in the stereotype.

5 — Access Modifiers are available for all members in both views and are denoted in the member declaration.

6 — The Static keyword is used in both views, with a slight notational change.

7 — Types have no equivalent in UML, and are therefore omitted. However, it may be possible to use stereotypes to declare a type in PlantUML.

#	VDM++
	PlantUML
8	`object1 is subclass of object2`
	`object2 <|-- object1`

9	`class object1`
	`...`
	`asoc1 : seq of object2 := [];`

	`object1 --> " (*) " object2 : asoc1`

10	`class object1`
	`instance variables`
	`quali1 : map Type to object2 := |->;`

	`object1 "[Type]" -> object2 : quali1`
11	`objects : set of object1 := {};`
	`Plant --> " * " Object1 : objects`
12	`asoc1: [object1] := nil;`
	`Object2 --> "0..1" Object1 : objects`

Table 3: VDM++ to PlantUML Transformation Mapping Table #8-12

8 — Inheritance between the two views have a one-to-one mapping. It is defined in PlantUML using an arrow head notation. Inheritance in PlantUML is declared after class declarations.

9 — Association Multiplicity in UML define the type of the corresponding VDM++ instance variable. An ordered zero to n multiplicity corresponds to a VDM++ sequence with the type of the association destination object. An ordered multiplicity is denoted using parentheses.

10 — Qualified Associations in VDM++ are instance variables that encapsulate the mapping from a qualifier type or qualifier class, to a class. In PlantUML the qualifier appears in square brackets on the side of the associating class, similar to how it is shown in Modelio using a smaller box bordering the class.

11 — Sets in VDM++ come as a result of an unordered zero to n multiplicity using the set type constructor.

12 — The Optional Type constructor is used when a multiplicity of zero or one is used.

We hereby see how PlantUML has the capability to fulfil these transformation rules, and is therefore a suitable host for UML visualisations of VDM models. Note that there may very well be edge cases, where some construct exists in VDM but is not defined in UML. It is difficult to consider all such cases, but one way is to do static analysis of the VDM model and warn the user, if they are about to transform some VDM that will not be available in the UML model.

5 Future Work

Further progress is required before a UML visualisation tool is seamlessly integrated into VDM VS Code. Achieving this involves developing a prototype of the PlantUML translator by following the architecture laid out in this paper. This development will hopefully allow for a smooth bidirectional mapping between the OO dialects of VDM and PlantUML. The current overview only considers the VDM++ dialect, but it will also have to incorporate the VDM-RT dialect in the future.

The creation of the direct translator involves the use of class mapping to reconstruct the ASTs present on the VDMJ tool. The process involved is still alien to us, but help from and collaboration with Nick Battle and the rest of the team is expected in the future.

At the moment, a challenge in representing VDM models using PlantUML is the lack of a qualified association definition, even if workarounds exist [1]. A solution to this may involve collaborating with the creators of PlantUML allowing OO VDM models to be fully compatible with PlantUML. The current connection does not support VDM types in its class diagrams, but as PlantUML is malleable to specify custom members, it may be possible to represent VDM types in PlantUML class diagrams.

Furthermore, there is still work to be done on the current implementation of the connection between VDM VSCode and UML. Transforming VDM files to UML is not supported for models that include libraries. Similarly, when transforming a UML file representing a VDM-RT model to VDM files, the files are converted to the VDM++ dialect. However both these problems are relatively simple to correct.

In the long term, work can be done to enable continual updates on the bidirectional mapping between VDM and PlantUML. Another area for potential development is to work on the inherent information loss that occurs when transforming. Solving this would involve developing some way of encapsulating the functionality of the classes present in the model and pass this information along, together with the UML diagram information. It is also planned to perform static analysis of the models to make the tool more user friendly by warning whenever some VDM construct does not have a compatible UML counterpart.

6 Conclusion

This paper has described how the coupling between VDM and UML was established on VS Code. This was performed by repurposing the already existing connection on the Eclipse version of Overture through decoupling it from the Eclipse IDE and packaging the transformation functionality, along with all its dependencies, in a JAR file and

integrating it into VS Code. Steps have been taken to adapt the textually based open-source diagram tool, PlantUML, for UML visualisations. Together with its VS code extension, PlantUML offers a continually updating view of UML diagrams next to the natural language based code. Progress was then made to create a translator that can transform between VDM++ and PlantUML by deciding on the architecture of the translator and a prototype was made.

Acknowledgments We would like to thank all the stakeholders that have contributed to the Overture/VDM development but in particular Kenneth Lausdahl for the original UML mapping and Nick Battle for the establishment of VDMJ.

References

1. Qualified associations, https://forum.plantuml.net/349/qualified-associations
2. Battle, N.: The v2c translation example, https://github.com/nickbattle/vdmj/tree/master/examples/v2c
3. Battle, N.: Analysis Separation without Visitors. In: Fitzgerald, Tran-Jørgensen, Oda (ed.) The 15th Overture Workshop: New Capabilities and Applications for Model-based Systems Engineering. pp. 104–115. Newcastle University, Computing Science. Technical Report Series. CS-TR-1513, Newcastle, UK (September 2017)
4. Bondavalli, A., Bouchenak, S., Kopetz, H.: Cyber-Physical Systems of Systems: Foundations – A Conceptual Model and Some Derivations: The AMADEOS Legacy, vol. 10099 (12 2016)
5. Bondavalli, A., Brancati, F.: Certifications of Critical Systems - The CECRIS Experience, pp. 1–316 (01 2017)
6. 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/
7. jebbs: Rich plantuml support for visual studio code, https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml
8. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The overture initiative integrating tools for vdm. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (jan 2010), https://doi.org/10.1145/1668862.1668864
9. Lausdahl, K., Lintrup, H.K.: Coupling Overture to MDA and UML. Master's thesis, Aarhus University/Engineering College of Aarhus (December 2008)
10. OMG: About the unified modeling language specification version 2.5.1 (december 2017), https://www.omg.org/spec/UML/2.5.1
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
13. Roques, A.: Plantuml language reference guide, https://plantuml.com/guide