Advanced VDM Support in Visual Studio Code

Jonas Kjær Rask[1], Frederik Palludan Madsen[1], Nick Battle[2], Leo Freitas[3], Hugo Daniel Macedo[1], and Peter Gorm Larsen[1]

[1] DIGIT, Aarhus University, Department of Engineering,
Finlandsgade 22, 8200 Aarhus N, Denmark
{jkr, fpm, hdm, pgl}@ece.au.dk
[2] Independent, nick.battle@acm.org
[3] Newcastle University, leo.freitas@newcastle.ac.uk

Abstract. The VDM VSCode extension has made its was to the daily practice of several engineers and students. Recent development leveraged the extension with many of the features previously only available in the Eclipse based tools. With the new additions, we are able to label the VDM VSCode extension to be the most up to date, preferred and recommended tool support for the VDM practitioner. In addition to keeping up with the previous features, recent developments brought new ones available in VSCode only (e.g.: Code Lenses), advances in proof support, and advances on the interpreter that equip the user with a modern tool suite. This paper provides an update on the current features and lays down the discussion of the future developments.

Keywords: Overture, IDEs, VDM, Language Server Protocol, Debug Adapter Protocol

1 Introduction

The Vienna Development Method (VDM) is one of the approaches to follow, when applying formal methods during the development of computer systems. The method has been widely used both in industrial contexts and academic ones covering several domains of the field: Security [6, 7], Fault-Tolerance [10], Medical Devices [9], among others, and its application /acceptance depends on usable and extendable tool support.

The standard open-source tool support for VDM, the Overture tool, has migrated from an Eclipse rich-client platform application to a Visual Studio Code (VS Code) extension, which supported basic editor features as reported in [13]. As described in [12], VSCode is an editor, not an Integrated Development Environment (IDE), yet with the introduction of the Language Server Protocol and the Debug Adapter Protocol, the editor becomes indistinguishable from an IDE. Our implementation of those protocols is thoroughly described in [11] with a comparison of legacy features in the Eclipse based tool and the ones in the new VSCode extension.

The prototype matured into a stable tool which was enriched with both the missing legacy features, proof-support and modern features. We report on the new features, and claim that, with the additions, the VDM VSCode extension has become a fully fledged VDM tool support. In addition to the new features, the claim is based on the fact that the

extension transitioned from prototype to classroom and daily practice, which is reflect on its usage metrics (e.g. the extension has over a thousand installs).

In this paper we report on the recent addition of previously existing features in the Eclipse based tooling and missing in the VDM VSCode extension (legacy features), namely:

- Import Examples - Allowing quick demos and facilitating training (e.g.: classroom exercises).
- Java Code Generation - The conversion of a VDM object oriented dialects specification into a Java project.
- Add Library - The possibility to select and add libraries to a VDM project.
- Remote Control - Extending the console based running and debugging with GUIs.
- Test Coverage - Highlights the parts of the specification that has been exercised.
- Proof Support - The option to generate the boilerplate Isabelle theory and spec definitions to be used in proofs.

Moreover, we have also augmented the features available to VDM practitioners, namely:

- Code snippets - Provide a template/skeleton of a specification primitive by typing a prefix and a triggering the features.
- Code Lenses - A popular feature in Visual Studio Code that we use to allow the launching and debugging of specification components by clicking on links that are interspersed throughout the specification.
- Dependency Graph Generation - Providing the visualisation of the class dependency of a object oriented VDM dialect in graphical form.

Given the completion of added legacy features and its transitioning into practice, we are now able to declare the VDM VSCode extension as a de facto tool support to VDM.

The outline of this paper is as follows: Section 2 provides an overview of the current VSCode extension components and features. Section 3 describes the advanced features added to the initial extension as reported in [11–13]. Section 4 contains the details of the recently added proof support. Section 5 provides insight on the changes made to the previous language server to support the new/modern editing features. Finally, in Section 6, we report on the insights and work done so far, while laying out the features under implementation.

2 Background

2.1 The VDM VSCode Extension

The extension implements a client-server architecture that uses the SLSP, LSP, and DAP protocols for communication. The client implementation is tightly coupled to VS Code, whereas the server implementation is fully decoupled from the client. The VDM language support is provided by the server as an implementation of VDMJ with extended capabilities to support the protocols. As a result, the server can be reused for any clients that support the protocols, and vice versa. Below is an overview of the components in the extension architecture:

Short description of the components:

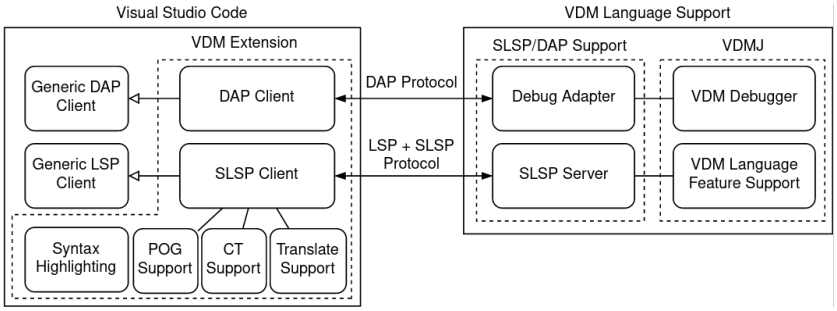

Fig. 1: Architecture of the VDM VSCode extension components.

- VS Code Components
  - Generic DAP Client: Handles the integration of the features enabled by the DAP protocol and provides the messaging infrastructure to handle the protocol on the client side.
  - DAP Client: Extends the Generic DAP Client to connect to the Debug Adapter on the server.
  - Generic LSP Client: Handles the integration of the features enabled by the LSP protocol and provides the messaging infrastructure to handle the protocol on the client side.
  - SLSP Client: Extends the Generic LSP Client with the messaging infrastructure to support the SLSP protocol.
  - Syntax Highlighting: Uses TextMate grammars to highlight VDM keywords.
  - POG Functionality Support: Implements the functionality to support the POG language feature enabled by the SLSP protocol.
  - CT Functionality Support: Implements the functionality to support the CT language feature enabled by the SLSP protocol.
  - Translate Functionality Support: Implements the functionality to support the translation language features enabled by the SLSP protocol.
- VDM Language Support Modules
  - Debug Adapter: Wraps the VDM Debugger to provide the debug functionality using the DAP protocol.
  - VDM Debugger: Provides the VDM debugger functionality.
  - SLSP Server: Wraps the VDMJ language support to provide it using the SLSP and LSP protocols.
  - VDM Language Feature Support: Provides the VDM language support functionality.

2.2 SLSP Client Features

The client side architecture of the extension is designed such that features which rely on the Specification Language Server Protocol (SLSP) protocol are decoupled from the protocol itself. Thus, the interface elements (buttons, web views, tree views) and their supporting logic in the client are not coupled to a client-server architecture.

A concrete example of the decoupling is provided in Figure 2 which shows a class diagram for the Proof Obligation Generation (POG) feature, with the other SLSP features of the extension following the same design.

Following is a description of key elements of the class diagram relevant for understanding the decoupling and also the integration of SLSP features in the client in more technical detail. As seen on the diagram the `extension` class has multiple instances of the class `SpecificationLanguageClient` which is an extension of the VS Code class `LangaugeClient` to enable the use of the functionality that is provided by the VS Code API. The multiplicity is needed since a client and server instance is created for each workspace folder that is opened in the editor. In the `LangaugeClient` class it is possible to register features that implement one of the interfaces `StaticFeature` or `DynamicFeature`. Both of these interfaces describe the functions that must be implemented to handle the initialisation of the feature between client and server. However, the features that are supported by the SLSP protocol currently only supports static registration, hence they implement the `StaticFeature` interface. For the POG feature this is implemented in the class `ProofobligationGenerationFeature`, which is responsible for providing a data provider for the Proof Obligation (PO) view that gets its data from the server. The `ProofObligationPanel` class provides the Graphical User Interface (GUI) elements to be displayed in the view. Only one instance of this class is created by the extension, as all the workspace folders share the same PO view. To be able to support all the workspace folders the panel can have multiple `ProofObligationProvider`'s (one for each workspace folder). They provide the data that is shown in the PO view and subscribe to events that are associated with the feature. In addition, the `ProofObligationPanel` is also responsible for registering the handlers for the commands relating to POG that the VDM VScode extension provides.

2.3 Tracking Features Completeness

VDM VSCode is now labelled as the most up to date, preferred and recommended tool support for the VDM practitioner. To provide a brief overview of the features of VDM VSCode a comparison to Overture, which can then be considered as the prior state of the art tool, is made in Table 1. As illustrated, some features found in Overture is still missing in VDM VSCode, however these are actively being developed. In addition, VDM VScode has support for features like ‘Translate to Word‘ and ‘VDM to Isabelle‘ which are not supported by Overture. The features ‘VDM to LLVM‘ and ‘VDM to Python‘ are not supported by either Overture or VDM VSCode. But, they are being considered for future development and implementation in VDM VSCode.

3 Adding Legacy Features

Import Examples This addition makes it possible to automatically import project from a large collection of existing examples. As it was the case in the previous Eclipse based IDE, users can experiment with a running example project.

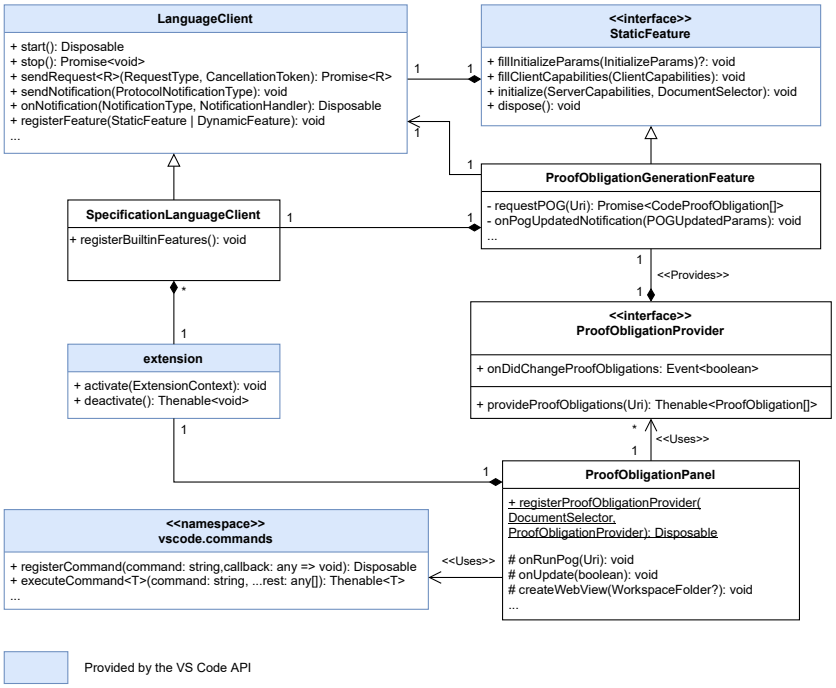

Fig. 2: Class diagram for the POG feature. For several of the classes, only the functions that are mainly used are included in the class description.

Table 1: Feature comparison between VDM VSCode and Overture.

Features	**Overture**	**VDM VSCode**
Syntax highlighting	X	X
Syntax check	X	X
Type check	X	X
Evaluation	X	X
Debugging	X	X
Proof obligation generation	X	X
Combinatorial testing	X	X
Translate to LaTex	X	X
Translate to Word		X
Extract VDM from Word	X	X
Code completion	(X)	(X)
Template expansion	X	X
Standard library import	X	X
Go-to definition	(X)	X
Coverage display	X	X
VDM to Java	X	(X)
VDM to XMI (UML)	X	
VDM to Isabelle		(X)
VDM Example Import	X	X
Real-time log viewer for VDM-RT	X	
Launch in VDMTools	X	X
FMU wrapper	X	
VDM to C	(X)	
VDM to LLVM		
VDM to Python		

Java Code Generation It is possible to generate Java code for a large subset of VDM-SL and VDM++ models. In addition to Java, C and C++ code generators are currently being developed. Both these code generators are in the early stages of development. For comparison, code generation of VDM-SL and VDM++ specifications to both Java and C++ is a feature that is available in VDMTools [Java2VDMMan, CGMan, CGManPP]. The majority of this chapter focuses solely on the Java code generator avail- able in VDM VSCode Extension.

Add Library Support It is possible to add existing standard libraries. This can be done by right-clicking on the Explorer view where the library is to be added and then selecting Add VDM Library. That will make a new window as shown in Figure 4.2. Here the different standard libraries provide different standard functionalities.

Remote Control In some situations, it may be valuable to establish a front end (for example a GUI or a test har- ness) for calling a VDM model. This feature corresponds roughly to the CORBA based API from VDMTools [APIMan]. Remote control should be understood as a delegation of control of the interpreter, which means that the remote controller is in charge of the execution or debug session and is responsible for taking action and executing parts of the VDM model when needed. When finished, it should return and the session will stop. When a Remote controller is used, the Overture debugger continues working normally, so for example breakpoints can be used in debug mode. Moreover, all dialects (VDM-SL, VDM++ and VDM-RT) support Remote Control. A new configuration with the use of a remote controller can be started by (see Figure 13.1 for more details): 1. Clicking on the button ”Add Configuration...” 2. Selecting ”VDM Debug: Remote Control (VDM-SL/++/RT) Then a new snippet (see Figure 13.2) will be created with the remoteControl option. And you simply have to write the full package/class name of the Remote Control.

Code snippets VSCode templates can be particularly useful when you are new to writing VDM models. If you press CTRL+space after typing the first few characters of a template name, Overture will offer a proposal. For example, if you type ”fun” followed by CTRL+space, the IDE will propose the use of an implicit or explicit function template as shown in Figure 5.1. The IDE includes several templates: cases, quantifications, functions (explicit/implicit), operations (explicit/implicit) and many more. The use of templates makes it much easier for users to create models, even if they are not deeply familiar with the VDM syntax

Code Lenses To make execution and debugging easy using the extension we have added code lenses for all public operations and functions. These are shown above their respective definitions. When pressing Launch or Debug a launch configuration is generated and launched immediately. If the operation/function has parameters you will be prompted to input these, the same applies if the class that the operation belongs to has a constructor that takes parameters.

Example of a lens-generated launch configuration

{

```
  "name": "Lens config: Debug Test1`Run",
  "type": "vdm",
  "request": "launch",
  "noDebug": false,
  "defaultName": "Test1",
  "command": "p new Test1().Run()"
}
```

Notice that the name starts with Lens config: if you remove this the launch configuration will not be overwritten the next time you activate a lens.

4 Proof Support within the VSCode Extension

Original proof support for VDM existed within the Mural [5] theorem proving system. Since then, theoretical development demonstrated that it was possible to soundly prove VDM theorems in other logical systems [15, 16]. Moreover, a translation strategy from VDM to Isabelle/HOL[1] was developed as part of a deliverable in the AI4FM project led by Cliff Jones [3]. This includes extended proof support for VDM-style expressions.

The current VDM proof support stems from the combination of these theoretical results and Isabelle translation strategy[2]. It comprises four parts: 1) VDM-VSCode plugin integration; 2) VDMJ plugins extension; 3) VDMJ proof annotations; and 4) Isabelle VDM toolkit.

4.1 VDM-VSCode Proof Support Plugin

VSCode proof support plugin hooks with the translation feature described in Section 2.3. It provides access to VDMJ's (native) plugins, through the language server console as an additional user-command, as well as through the IDE menu. The VSCode proof plugin also provides translation and proof support configuration information available for users to choose. Practically, it is a wrapping interface to the native VDMJ plugin described next. This separation of concerns is crucial for maintenance and stability of the tool chain.

4.2 VDMJ Plugins Extension

Through the LSP, users can access the VDMJ native plugins interface. This gives access to the VDM AST, typechecker, POG and other tools. The VDM to Isabelle/HOL translation plugin is divided in four parts:

1. exu. This plugin analyses a type checked VDM AST looking for: a) unsupported features (e. g., Isabelle requires declaration before use); b) specification consistency (e. g., function call graphs ought to also participate in the corresponding function specification); and c) "proof-friendly" VDM style (e. g., keep type invariants compartmentalised

[1] https://isabelle.in.tum.de
[2] https://github.com/leouk/VDM_Toolkit

and as close to their defining type as possible), which enables higher automation in predicting how proof unfolding will take place. These checks ensure that there will be no Isabelle type errors as a result of translation. They also help improve proof script automation. For example, if a function f calls g, then its good practice that f's precondition also call g's precondition. Of course, this might be spurious, yet can increase proof automation, if present. Similarly, instead of defining type for a numeric sequence ensuring all elements are negative, it is preferable to define a type for negative numbers then make a sequence of that.

2. vdm2isa. This plugin compiles a type checked VDM AST into a Isabelle/HOL translation tree, which is then executed to generate the corresponding Isabelle theory file. Its execution does not depend on exu, yet lingering exu errors/warnings will entail vdm2isa errors and likely Isabelle/HOL type errors. This stage of the translation strategy transform each VDM top-level definition (e. g., types, functions, etc.) to their corresponding Isabelle construct. That is, for every VDM module M, a corresponding Isabelle theory file M.thy is generated. Moreover, it also ensures that the (implicit) VDM rules are checked within the translated Isabelle specification. For example, type invariants are implicitly checked as part of the precondition of translated functions.

3. isapog. This plugin compiles the POG POs into corresponding Isabelle theorems to be proved. Beyond translating the PO predicate itself, the compilation also groups POs within Isabelle local context principle (i. e., locale) to ensure that all proofs within that context **must** be discharged, and to compartmentalise/stratify their proof scripts. That is, for every translated Isabelle theory file M.thy from VDM module M, a further Isabelle theory file M_PO.thy is generated. The plugin also enables the user to extend the POG with lemmas that might be useful for discharging POs. This is achieved through VDM annotations, described in the next subsection.

4. isaproof. This plugin analyses the VDM translation tree to generate tentative proof scripts for each of the isapog-generated POs. That is, for every VDM POs Isabelle theory file M_PO.thy, a proof strategy theory file M_PS.thy is generated containing tentative proof scripts for each of the POG POs. This obviously imply the plugin depends on successful execution of both the vdm2isa and isapog plugins to access translated VDM definitions and POs.

4.3 VDM Annotations

VDMJ allows the user-defined annotations. These can be processed by either the parser, type-checker, interpreter or POG. For proof support, we include two new annotations:

- @Theorem and @Lemma.
  These annotations enable the user to extend the VDMJ POG with new user-defined proof obligations about the model being developed. There is no semantic difference between lemmas and theorems (i. e., they are both boolean expressions that must be true). Theorem expressions must have a unique (global) name and be type correct. If the expression is executable, the interpreter will determine whether the theorem is

true (or not); otherwise, the POG will generate the named PO associated with the theorem.
Beyond documenting specific properties wanted of the model, user-defined lemmas can also be useful to document stepping stones in the later proofs associated with either POG POs or user-defined theorems. These new POs are processed by the isapog and isaproof plugins, as described above.

– @Witness.
This annotation enables explicit user-defined values for types, or specific function/-operation calls. The type checker ensures that witness expressions are correct, and the interpreter evaluate them to ascertain whether the witness is valid (e. g., a positive number as a witness for a new nat sub-type). That is, the witness chosen is executable and satisfy any associated specification [4].
These witness expressions can then be used in proofs involving existential introduction. That is, a witness to a record type will be type checked and interpreted; if that succeeds, this is akin to an existentially quantified variable witness, which will be useful for later generation of translated POs proof scripts.

5 Going Beyond The Previously Existing Tools

5.1 Analysis Plugins

The primary purpose of the VDM VSCode language server is to respond to Client requests via the Language Server Protocol or the Debug Adapter Protocol. The Client facing components of the server accept RPC message requests and dispatch these to the a workspace manager for each protocol. The workspace managers maintain the current state of the specification, for example by applying edits when received from the Client. But to perform more advanced language processing, like checking for syntax or type errors, the workspace managers delegate processing to *Analysis Plugins*:

– Analysis plugins isolate the language specific processing from the workspace managers. In particular, they hide the difference between modular (VDM-SL) and class based processing (VDM++ and VDM-RT).
– Plugins encapsulate the AST specialised for one particular analysis type. VDMJ maps the original AST from the parser into specialised ASTs comprised of classes that perform one analysis, such as type checking, PO generation or execution [?].
– Plugins register themselves and may involve other plugin services while processing, though this is usually coordinated via the workspace managers.
– Plugins can be added to the system via configuration, and allow it to be extended with new analyses (such as the Isabelle translator 4).

The server has three analysis plugins that *must* be available to open any VDM project. These are built into the server:

– AST: This is the plugin for the basic parse of the specification text. It supplies the AST tree to the other plugins for further processing and it offers "on the fly" syntax error reporting to the workspace manager as text is entered. The plugin also has basic symbol location services that can be invoked if there are type errors and the TC plugin cannot do this.

- TC: The type checker plugin manages the TC specialised tree and its lifecycle. It returns type related warning and error messages, as well as building the "outline" of the symbols defined by the specification. It also allows definitions to be located (e.g., for "Go-to definition" requests).
- IN: The interpreter plugin manages the IN specialised tree and its lifecycle. It enables the DAP workspace manager to obtain an interpreter for executing expressions in the specification.

In addition to these essential plugins, two further plugins are shipped with the VSCode extension, since they support Client-side UI enhancements:

- PO: The proof obligation analysis plugin manages the PO specialized tree. It offers a service to build a list of proof obligations, either for the entire specification or for just one file. It is complemented by a Client panel to display the list of obligations and navigate from an obligation to its location in the specification.
- CT: The combinatorial test analysis plugin manages the CT specialised tree and, in combination with the IN plugin, enables the server to generate and execute combinatorial tests. The server feature is complemented by a UI panel to allow the generated tests and results to be explored.

Further analysis plugins can be added by implementing the *AnalysisPlugin* interface. These are instantiated by the LSPX workspace manager (which handles extensions to LSP), by reading a system property that defines the classes to load. Such dynamic plugins register themselves like any other plugin. They typically create and manage their own tree of specialised classes, using the VDMJ *ClassMapper*[?].

The LSPX workspace manager handles non-standard LSP method requests by asking the registered plugins whether they can process the request. The expectation is that a new Client feature will send new SLSP requests that are handled by a new user plugin.

5.2 Code Lenses

The LSP protocol offers language servers the ability to label parts of the source text with a clickable tag that then invokes an arbitrary command on the Client. These are called "code lenses". This request is offered to each of the analysis plugins when a Client code lens request is received.

Only one code lens is currently generated, by either the AST or TC plugins (depending on whether there are type errors). The lens offers "Launch" and "Debug" tags for executable functions or operations and links these with a Client side command that allows the user to quickly build and launch or debug the definition.

But this is a powerful feature and we expect new plugins to add their own lenses. For example, the Isabelle translation plugin may be able to link VDM definitions to the equivalent in the Isabelle environment.

5.3 User Libraries

A library is comprised of one or more VDM sources, in a variety of dialects, with any associated native Java code. These are packaged into a regular Java "jar" file, with one

additional file in the "META" folder, called "library.json". The meta-file defines the source files that should be included for a given VDM dialect as well as dependencies on other libraries. The VDMJ "stdlib" meta-file is a good example (below).

The extension allows new libraries to be included in a workspace and selectively added to projects.

```
{
    "vdmsl": [
        {
            "name": "IO",
            "description": "The IO library",
            "files": ["IO.vdmsl"],
            "depends": []
        },
        {
            "name": "MATH",
            "description": "The MATH library",
            "files": ["MATH.vdmsl"],
            "depends": []
        },
        ...
    ],
    "vdmpp": [
        {
            "name": "IO",
            "description": "The IO library",
            "files": ["IO.vdmpp"],
            "depends": []
        },
        {
            "name": "VDMUnit",
            "description": "The VDMUnit library",
            "files": ["VDMUnit.vdmpp"],
            "depends": ["IO"]
        },
        ...
    ],
    "vdmrt" : [
        ...
    ]
}
```

Dependency Graph Generation The VDM VScode extension allows the generation of a dependency graph that represents the dependencies of the classes/objects in a specification. Thus, it is considered as a directed graph where each node points to the node on which it depends. In some cases you can add some conditions set on the different

connections between the nodes. Moreover, each shape represents a node (usually ellipses or circles) and each connector, composed of one or two arrow heads, indicates the direction of the dependencies. You have also the possibility to add labels on connectors to specify the relation between two nodes. To finish, the main usage of a dependency graph consists of describing processes to make it easier for the developer to understand, reuse and maintain his code. An example of such a dependency graph is shown on Figure 3

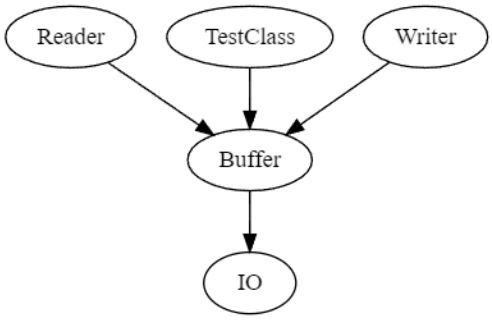

Fig. 3: Example of a visualisation of a dependency graph file generated by the extension. The visualisation is performed by an external tool.

6 Concluding Remarks and Future Work

In this paper, we report on the recent additions to the VDM VSCode extension taking the current support of VDM to a stage where most of the legacy features are supported and new ones are added. The latter are specially important, as they allow the VDM practitioner to use features, which are available in the integrated development environments they are used to, but were not available while writing VDM specifications.

6.1 Future Work

VDM-RT Log Viewer. As described by Fitzgerald et al. [1] real-time traces, which can be generated when executing a VDM-RT model, can provide insight into the ordering and timing of exchange of messages, activation of threads and invocation of operations and combined with validation conjectures [1] enables explicit consideration of system-level properties during the modelling process. As such, a tool for displaying the traces and validation conjectures can provide valuable information. There is therefore ongoing work on implementing a log viewer tool in the VDM VSCode extension similar to the log viewer tool found in Overture today.

Behaviour Driven Development (BDD). VDM modelling involves the transformation of a set of elicited requirements listed in a natural language into an executable model. the practice is to transform the requirements into the specification as a jump which consists of a mental exercise including the iteration of two steps: first understand the requirements, then formulate them as a formal model. It is possible to aid professionals in this jump by adopting the approach of BDD, where requirements are translated into behaviours. The behaviours are written in a constrained subset of natural language with the goal of becoming more understandable by stakeholders. Tool support has been developed in both [2] showing its feasibility and recent efforts are in place to develop a VSCode extension supporting BDD in VDM [14].

UML/VDM translation. The UML connection, allowing the bidirectional translation between object oriented VDM models and UML diagrams, a legacy feature supported in the Eclipse based tools is still not supported in the VSCode extension. The legacy UML transformations were developed with the informal modelling tool, Modelio, in mind, which require the installation of external tools. New and modern UML tools have been developed and are integrated in the VSCode marketplace since the Modelio based translations, and we are working to re-purpose the connection inside the ecosystem. A preliminary account of the new connection can be found in [8].

FMI Support. Though, currently, the VSCode extension allows the generation of Function Mock-Up Units (FMUs), its implementation is still dependent on the source code from the existing Eclipse based solution, given that the generated FMUs wrap the Eclipse based interpreter. This solution allows the usage of tested and reliable code, but introduces the potential for discrepancies between the results when the model is launched/debugged using the VDMJ based LSP-server and the results produced by executing the FMU, given that two different interpreters are used. To simplify codebase maintenance and make sure results are ran with the same interpreter, we plan to overall the VDMJ scheduler and create a new FMI implementation directly in it, thus completely deprecating the Eclipse-based solution.

Acknowledgements We acknowledge the Poul Due Jensen Foundation for funding the project Digital Twins for Cyber-Physical Systems (DiT4CPS).

References

1. Fitzgerald, J.S., Larsen, P.G., Tjell, S., Verhoef, M.: Validation Support for Real-Time Embedded Systems in VDM++. In: Cukic, B., Dong, J. (eds.) Proc. HASE 2007: 10th IEEE High Assurance Systems Engineering Symposium. pp. 331–340. IEEE (November 2007)
2. Fraser, S., Pezzoni, A.: Behaviour driven specification. In: Macedo, H.D., Thule, C., Pierce, K. (eds.) Proceedings of the 19th International Overture Workshop. Overture (10 2021)
3. Freitas, L., Whiteside, I.: Proof patterns for formal methods. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8442, pp. 279–295. Springer (2014), https://doi.org/10.1007/978-3-319-06410-9_20

4. Jacobs, E.: Implementing Witness Annotations for VDMJ. Master's thesis, School of Computing (May 2018)
5. Jones, C.B., Jones, K.D., Lindsay, P.A., Moore, R.: mural: A Formal Development Support System. Springer-Verlag (1991), http://homepages.cs.ncl.ac.uk/cliff.jones/publications/Books/mural.pdf
6. Kulik, T., Macedo, H.D., Talasila, P., Larsen, P.G.: Modelling the HUBCAP Sandbox Architecture In VDM – a Study In Security. In: Fitzgerald, J.S., Oda, T., Macedo, H.D. (eds.) Proceedings of the 18th International Overture Workshop. pp. 20–34. Overture (December 2020)
7. Kulik, T., Talasila, P., Greco, P., Veneziano, G., Marguglio, A., Sutton, L.F., Larsen, P.G., Macedo, H.D.: Extending the formal security analysis of the hubcap sandbox. In: Macedo, H.D., Thule, C., Pierce, K. (eds.) Proceedings of the 19th International Overture Workshop. Overture (10 2021)
8. Lund, J., Jensen, L.B., Macedo, H.D., Larsen, P.G.: Towards UML and VDM support in the vs code environment (submitted). In: Macedo, H.D., Pierce, K. (eds.) Proceedings of the 20th International Overture Workshop. Overture (2022)
9. Macedo, H.D., Larsen, P.G., Fitzgerald, J.: Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods, 15th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 5014, pp. 181–197. Springer-Verlag (2008)
10. Nilsson, R., Lausdahl, K., Macedo, H.D., Larsen, P.G.: Transforming an industrial case study from VDM++ to VDM-SL. In: Pierce, K., Verhoef, M. (eds.) The 16th Overture Workshop. pp. 107–122. Newcastle University, School of Computing, Oxford (July 2018), TR-1524
11. Rask, J.K., Madsen, F.P.: Decoupling of Core Analysis Support for Specification Languages from User Interfaces in Integrated Development Environments. Master's thesis, Aarhus University, Department of Engineering (December 2020)
12. 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
13. 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
14. Villadsen, K.S., Jensen, M.D., Larsen, P.G., Macedo, H.D.: Bridging the requirements-specification gap using behaviour-driven development (submitted). In: Macedo, H.D., Pierce, K. (eds.) Proceedings of the 20th International Overture Workshop. Overture (2022)
15. Woodcock, J., Freitas, L.: Linking VDM and Z. In: 13th International Conference on Engineering of Complex Computer Systems (ICECCS 2008), March 31 2008 - April 3 2008, Belfast, Northern Ireland. pp. 143–152. IEEE Computer Society (2008), https://doi.org/10.1109/ICECCS.2008.36
16. Woodcock, J., Saaltink, M., Freitas, L.: Unifying theories of undefinedness. In: NATO Series D: Information and Communication Security (Marktoberdorf). vol. 22, pp. 311–330. IOS Press (Aug 2009)