Visual Studio Code VDM Support

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

[1] DIGIT, Aarhus University, Department of Engineering, Finlandsgade 22, 8200 Aarhus N, Denmark
{201507306, 201504477}@post.au.dk, {hdm, pgl}@eng.au.dk
[2] Independent, nick.battle@acm.org

Abstract. How is it possible to significantly improve the Integrated Development Environment (IDE) for VDM from the existing Eclipse-based IDE? The proposal made in this paper is to use language-agnostic protocols such as the Language Server Protocol (LSP) and the Debug Adapter Protocol (DAP) connecting a general editor such as Visual Studio Code with core server functionality. This is demonstrated for editor related features, debugging, and Proof Obligation Generation and Combinatorial Testing support. We also believe that the extension of LSP and DAP will be useful for extending other IDEs for similar specification languages, since using such standard protocols will require less effort to upgrade to modern front-ends for their IDEs.

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 prescribes the development of digital/computer models of the system under development in one of the the VDM specification languages and dialects. If the models are described in an executable subset it is then possible to execute and analyse them to reach a high-fidelity system description, which is then subsequently used to produce code in the programming languages used to operate the system implementation.

To support all the steps involved in the development of a VDM model, there are several tools and Integrated Development Environments (IDEs) supporting the variety of VDM specification languages and dialects to different levels [9]. VDMTools were the first available commercial tool developed in the mid-1990s [7]. Then Overture [8] brought free and open-source support to VDM in 2005, and many others are now available [11, 14].

The Overture tool is one of the most complete and popular IDEs for VDM. It consists of multiple plugins that extend the Eclipse IDE, which appeared in the 2000s. Eclipse is the most popular[1] in its class, but it is being challenged by a new contender, the Visual Studio Code (VS Code)[2] editor, which is based on Electron[3] and fully leverages

[1] See https://pypl.github.io/IDE.html
[2] See https://code.visualstudio.com/
[3] See https://www.electronjs.org/.

web technologies. Although not an IDE, VS Code modernized the IDE world, with the introduction of the Language Server Protocol (LSP)[4] and the Debug Adapter Protocol (DAP)[5], which enable the development of IDE features in a general manner. With the two protocols the editor becomes indistinguishable of an IDE, and the implementation code becomes reusable and better maintainable. However, the Overture language core does not implement the LSP and DAP protocols, so the effort required to move towards VS Code is significant, but we expect it to pay off in the long run.

In this paper we investigate the efforts needed to support VDM in VS Code with as much as possible of the support using the standardised protocols LSP and DAP. To support specification lanuage features not found in programming languages we propose an extension to LSP, Specification Language Server Protocol (SLSP). As part of the investigation we have extended VDMJ such that it can be used as a language server that supports both these protocols. The server supports syntax-checking, type-checking and go-to functionality using the LSP protocol, debugging by using the DAP protocol, and Proof Obligation Generation (POG) and Combinatorial Testing (CT) using the SLSP protocol. We have also developed a VS Code extension which connects to the language server in order to provide the language features in the IDE. In addition, we have investigated which features are required to provide full language support for specification languages and which of these are supported by the protocols.

Our work departs from a previous proof of concept, and adds further support for the LSP protocol, support for all the VDM dialects, support for the DAP protocol and support for the SLSP protocol. In addition, this is the first research paper on the topic, and we foresee more publications to emerge from the works towards fully supporting VDM development using VS Code. We believe the resulting extensions will become the next in line supporting VDM development. A modern and robust IDE, which development starts now.

The remaining parts of this paper starts with an overview of the background necessary to understand this paper in Section 2. Afterwards Section 3 explains how the standard protocols LSP and DAP have been used to implement the core of the VS Code support for VDM. This is followed by Section 4 which is evaluating the efforts that has been conducted. In Section 5 we describe related work. Finally, Section 6 provides concluding remarks and future work.

2 Background

In this section, we introduce VDMJ which is a tool that provides language features for VDM. We further describe the development tool VS Code, which is used as the Graphical User Interface (GUI) for integration with the language server. Finally, we describe the standardised protocols LSP and DAP used to decouple the language features from the development tool.

[4] See https://microsoft.github.io/language-server-protocol/.
[5] See https://microsoft.github.io/debug-adapter-protocol/.

2.1 VDMJ

VDMJ [1] is a command-line tool written in Java, that provides basic language support for the VDM dialects VDM-SL, VDM++ and VDM-RT. It includes a parser, a type checker, an interpreter, a debugger, a proof obligation generator and a combinatorial test generator with coverage recording, as well as VDMUnit support for automatic testing and user definable annotations. These are implemented using an extensible Abstract Syntax Tree (AST) analysed using a visitor framework [4].

Using VDMJ for the language server allows parts of the language support to be reused. However, many features of VDMJ are not supported by standardised protocols, thus only a subset of the functionality is used in the language server.

2.2 Visual Studio Code

VS Code[6] is a free source code editor. It has built-in support for the programming languages JavaScript and TypeScript and further enables support for other languages through a rich ecosystem of extensions. VS Code uses a folder or workspace system for interacting with a project and a document system for handling the source code files in the project. This allows VS Code to be language-agnostic and delegate language specific functionality to an extension. Given this design reasoning, a need for the standardisation of decoupling between editor and extension has been identified. At the time of writing, three different protocols have been developed for this purpose. Namely LSP[7] (described in Section 2.3), DAP[8] (described in Section 2.4) and Language Server Index Format (LSIF)[9].

2.3 Language Server Protocol

Since most IDEs support a variety of programming languages, IDE developers face the problem of keeping up with ongoing programming language evolution [2]. At the same time language providers are interested in providing as many IDE integrations as possible to serve a broad audience. Consequently, integrating every language, m, in every IDE, n, leads to a $m \times n$ complexity.

The LSP protocol defines a standardised protocol to be used to decouple a language-agnostic development tool (client) and a language-specific server that provides language features like syntax-checking, hover information and code completion. This is illustrated in Figure 1. The client is responsible for managing editing actions without any knowledge of the language and the server validates the correctness of the source code and reports issues and language-specific information to the client. To facilitate this the LSP protocol communicates using language neutral data types such as document references and document positions.

[6] See https://code.visualstudio.com/
[7] See https://microsoft.github.io/language-server-protocol/
[8] See https://microsoft.github.io/debug-adapter-protocol/
[9] See https://microsoft.github.io/language-server-protocol/overviews/lsif/overview/

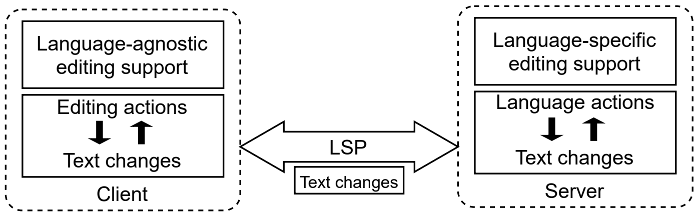


Fig. 1: LSP approach to language support. Borrowed from [13].

Many tools support the LSP protocol which reduces the time needed to create a client implementation[10]. Furthermore, new development tools only have to support the protocol, which can be done with little effort compared to native integration [3]. Additionally, as the server is separated from the development tool it can be used for multiple tools. This allows tools to easily support multiple languages and features. Thus, by decoupling the language implementation from the editor integration the complexity is reduced to $m + n$.

2.4 Debug Adapter Protocol

The DAP protocol is a standardised protocol for decoupling IDEs, editors and other development tools from the implementation of a language-specific debugger. The DAP protocol uses language neutral data types, which makes the protocol possible to use for any text-based language. The debug features supported by the protocol includes: different types of breakpoints, variable values, multi-process and thread support, navigation through data structures and more.

To be compatible with existing debugger components, the protocol relies on an intermediary debug adapter component. It is used to wrap one or multiple debuggers, to allow communication using the DAP protocol. The adapter is then part of a two-way communication with a generic debugger component, which is integrated in a given development environment as illustrated in Figure 2. Thus, the protocol reduces a $m \times n$ problem of implementing each language debugger for each development tool into a $m + n$ problem.

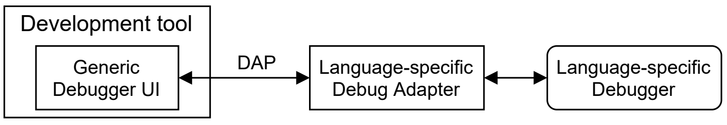


Fig. 2: The decoupled architecture where the DAP protocol is used.

[10] See https://microsoft.github.io/language-server-protocol/implementors/tools/

3 Implementing Language Features

The support for VDM in VS Code is accomplished using an extension that communicates with a language server based on the language support found in VDMJ, as illustrated in Figure 3. The standard protocols LSP and DAP are developed with programming languages in mind. This means that the specification language features that are common for these languages are not supported, e. g., POG and CT, as discussed in Section 4.2. To compensate for this we have developed the SLSP protocol, which is introduced in the section Section 3.1. Followed by a description of the components that the VS Code extensions[11] are comprised of and how they have been implemented. Finally, we describe the work put into VDMJ to provide support for the protocols.

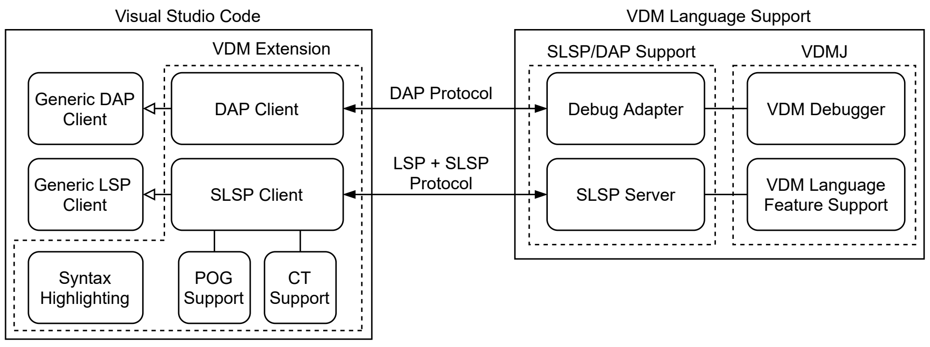


Fig. 3: Overview for the suggested extension architecture

3.1 Specification Language Server Protocol

The purpose of the SLSP protocol is to facilitate support for specification language specific features using the same architecture as for LSP and DAP. The SLSP protocol is an extension to the LSP protocol. This means that it uses the same base protocol as LSP and relies on functionality from LSP such as the synchronisation between client and server. The protocol is developed with the intention that it should also be usable by other languages than VDM and possibly be included in the LSP specification. Hence, the SLSP protocol uses language neutral data types, which allows the client to be language-agnostic enabling it to be used with any server that supports the SLSP protocol. At the time of writing the SLSP protocol defines messages to support POG and CT, and it is used in the VS Code extension for those features.

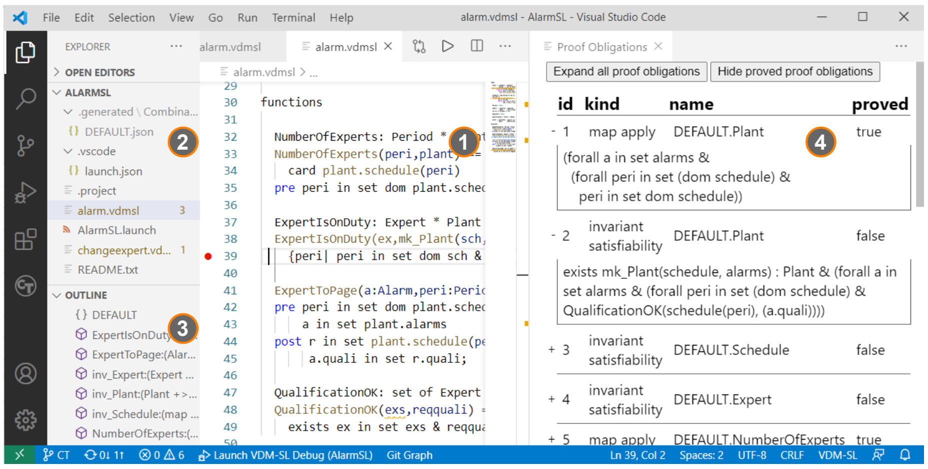
3.2 Visual Studio Code Extension

VS Code operates with a rich extensibility model that enables users to include support for different programming languages, debuggers and various tools to support the develop-

[11] The extensions can be found at: https://github.com/jonaskrask/vdm-vscode

ment workflow, by downloading extensions from the VS Code extensions marketplace[12]. This also enables developers to make extensions that they find missing from the marketplace, which is how VDM is supported in VS Code; by creating an extension for each VDM dialect. The extensions include a SLSP client and a DAP client. Besides communicating with the server the extension also has the responsibility of doing syntax highlighting as this is not supported in any of the protocols.

As illustrated in Figure 3, VS Code includes generic support for both the LSP and DAP protocols which allows the client implementation to be carried out with little effort. The generic protocol support provides the extension with handlers for all the LSP and DAP messages which include code to synchronise the editor and the server, handling of the editor navigation and displaying messages from the server. Thus, the extension provides configuration of the generic protocol support, launching and connecting to the server, syntax highlighting, and the GUI to support the SLSP features. Figure 4 depicts the VS Code environment with the VDM-SL extension active.



```

functions

  NumberOfExperts: Period * 
  NumberOfExperts(peri,plant) ==
    card plant.schedule(peri)
  pre peri in set dom plant.schec

  ExpertIsOnDuty: Expert * Plant
  ExpertIsOnDuty(ex,mk_Plant(sch,
    {peri| peri in set dom sch &

  ExpertToPage(a:Alarm,peri:Peric
  pre peri in set dom plant.schec
      a in set plant.alarms
  post r in set plant.schedule(pe
       a.quali in set r.quali;

  QualificationOK: set of Expert
  QualificationOK(exs,reqquali) =
    exists ex in set exs & reqqua
```

id	kind	name	proved
- 1	map apply	DEFAULT.Plant	true
(forall a in set alarms & (forall peri in set (dom schedule) & peri in set dom schedule))			
- 2	invariant satisfiability	DEFAULT.Plant	false
exists mk_Plant(schedule, alarms) : Plant & (forall a in set alarms & (forall peri in set (dom schedule) & QualificationOK(schedule(peri), (a.quali))))			
+ 3	invariant satisfiability	DEFAULT.Schedule	false
+ 4	invariant satisfiability	DEFAULT.Expert	false
+ 5	map apply	DEFAULT.NumberOfExperts	true

Fig. 4: VDM-SL VS Code Extension: (1) Main Editor; (2) Project Explorer; (3) File Outline; (4) Proof Obligation View

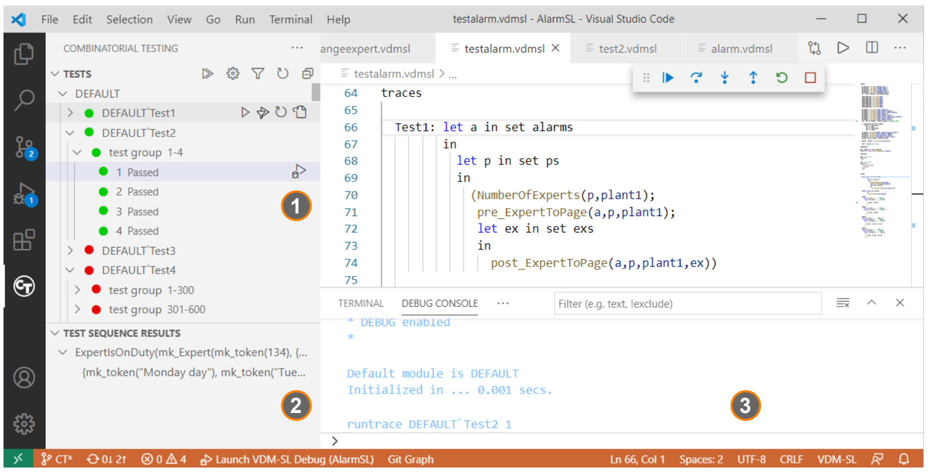
Syntax Highlighting: Syntax highlighting in VS Code is handled on the client side as it is not supported by the protocols. Instead syntax highlighting is performed using TextMate grammars [5]. The TextMate grammars are structured collections of Oniguruma regular expressions[13], implemented using a JSON schema. The grammar file specifies a set of rules, that is used with pattern matching to transform the visible text into a list of tokens and colour these according to a colour scheme.

[12] See https://marketplace.visualstudio.com/vscode
[13] See https://github.com/kkos/oniguruma

Specification Language Server Protocol Client: As custom for language extensions the VDM extensions are activated when a file with a matching file format is opened, e. g., opening a '`.vdmsl`' file actives the VDM-SL extension. On activation the client launches the server and connects to it. When connected, the client forwards all relevant information to the server using the LSP protocol. Furthermore, any responses or notifications from the server triggers feedback to the user, which is handled by the generic LSP client integration in VS Code. In addition to setting up the support for LSP the client also provides customised GUI views to support POG and CT. This is not directly supported by VS Code and LSP since these features are not commonly used for traditional programming languages.

The POG view, illustrated in Figure 4, provides a list of the proof obligations for a specification, where expanding an element displays the actual proof obligation. The view is implemented using the VS Code Webview API[14], which is customised using CSS, HTML and JavaScript.

The CT view, illustrated in Figure 5, shows the tests that are generated based on the traces in the specification, these are structured in a tree view. Traces can be fully or partially executed using either an execution filter or executing a test group at a time. A tests execution sequence and result is displayed in a separate tree view. Tests can also be debugged, which is facilitated using the DAP protocol. The CT view is implemented using the VS Code Tree View API[15]. which reduces the complexity of implementing a view.



ExpertIsOnDuty(mk_Expert(mk_token(134), {...
{mk_token("Monday day"), mk_token("Tue...

```
traces

 Test1: let a in set alarms
        in
          let p in set ps
          in
            (NumberOfExperts(p,plant1);
             pre_ExpertToPage(a,p,plant1);
             let ex in set exs
             in
               post_ExpertToPage(a,p,plant1,ex))
```

```
* DEBUG enabled
*

Default module is DEFAULT
Initialized in ... 0.001 secs.

runtrace DEFAULT`Test2 1
>
```

Fig. 5: VDM-SL VS Code Extension: (1) Combinatorial Testing View; (2) Test Result View; (3) Debug Console

[14] See https://code.visualstudio.com/api/extension-guides/webview.
[15] See https://code.visualstudio.com/api/extension-guides/tree-view.

Debug Adapter Protocol Client: The DAP protocol support is added to the client by specifying a `DebugAdapterDescriptorFactory` which establishes the connection between the generic debugger client and the server upon starting a debug session. This would normally include launching a DAP server and connect to it. However, as the LSP and DAP server is combined into one we simply connect to the port specified for DAP communication.

3.3 Enabling the use of LSP and DAP in VDMJ

Internally, VDMJ provides a set of language services to enable VDM specifications to be processed, and by default the coordination of these services is handled by a command-line processor. So for example, a Parser service is used to analyse a set of VDM source files to produce an Abstract Syntax Tree (AST); if there are no syntax errors, that AST is further processed by a Type Checker; and if there are no type errors, the tree is further processed by an Interpreter (in a read-eval loop) or by a Proof Obligation Generator. If the Interpreter encounters a runtime problem or a debug breakpoint, it stops and interacts with the command-line to allow the user to examine the stack and variables, for example.

Although the command-line is the default way to interact with VDMJ, this is not assumed in the design. An alternative means to interact is provided using the DBGP protocol[16]. This allows VDMJ to interact with a more sophisticated IDE, such as Overture. DBGP messages are exchanged over a local TCP socket, allowing the IDE to coordinate the execution of expressions within the specification, and to debug using a richer user interface. The VDMJ end of the DBGP socket coordinates the same services as the command-line does, but with requests and responses being exchanged via the socket, using DBGP messages.

So adapting VDMJ to use the LSP and DAP protocols is a matter of creating a new handler to accept socket connections from a client, and process the JSON/RPC messages defined in the respective LSP and DAP standards. Since LSP and DAP are expressed in general language terms, there is generally a natural mapping from abstract concepts in these protocols to the specific case of a VDM specification.

Creating the LSP part of the VDMJ handler was relatively simple. The biggest difference to the existing connection handlers is that LSP is an "editing" protocol. That is, rather than being asked to process a fixed set of unchanging specification files, LSP requires the server to accept the creation and editing of files on the fly. So whereas normally, the Parser would only be called once to process a set of fixed specification files, in the LSP handler it can be called repeatedly as edits are passed from the client. Since the VDMJ parser is relatively efficient, it is responsive enough to re-parse the affected file as edits are made. A decision was taken to only type-check the specification when the client deliberately saves the work (i. e., writes changes to disk). This is the same behaviour as Overture.

Execution and debugging via the DAP protocol required the creation of a new "DebugLink" subclass in VDMJ. The existing command-line and DBGP protocols each extend an abstract DebugLink class, which allows the Interpreter service to interact with an arbitrary client to coordinate a debugging session. In this case, the client is

[16] See https://xdebug.org/docs/dbgp

the DAP client. Since the DebugLink was designed to be extended, this was relatively straightforward – in fact the DAPDebugLink is an extension of the command-line class with the protocol aspects changed from textual read/writes to JSON message exchanges. It is about 200 lines of Java. The only complexity in this area is in the case of multi-threaded VDM++ specifications. Special care must be taken to be sure that all of the separate threads and their data are coordinated correctly, with appropriate mutex protection of the (single) communication channel to the DAP client.

In addition to parsing, type-checking and executing a specification, the LSP protocol also enables the server to offer various language services that allow the client to build a more intelligent user interface. For example, the outline of a file (its contents, in terms of the VDM definitions within and their location) can be queried; any name symbol in a file can be used to navigate to that symbol's definition (e. g., moving from a function application to its definition, possibly in a different file); and name-completion services are offered, where the first few characters of a name can be typed, and the server will offer possible completions. These services do not exist in the base VDMJ, and were therefore added as part of the LSP development. They are mostly implemented via visitors (using the VDMJ visitor framework) which process the AST.

The Proof Obligation Generation and Combinatorial Testing features of VDMJ are maked available with the SLSP protocol by adding new "slsp" RPC methods. Internally, the server routes these new requests to the VDMJ components that are able to perform the analyses.

Proof Obligations (POs) can be generated relatively quickly (usually in a fraction of a second, even for hundreds of obligations), so the "slsp/POG/generate" method takes a one-shot approach, generating and returning all the POs in the specification by default, optionally allowing the UI to limit the scope to a sub-folder of specification files or a single file. The returned obligations are represented as VDM source, as a stack of contexts with a primitive obligation at the end. This allows the UI to decide how to represent the indentation of the full obligation without requiring it to understand the VDM AST.

Combinatorial Testing is a more complex problem, requiring multiple interactions between the UI and the server as new "slsp/CT" methods. Unlike Proof Obligations, Combinatorial Tests can expand to millions of test cases and their execution can take many hours. This in turn means that the UI must have some indication of the progress of the execution, and it must also have the ability to cleanly terminate the execution in the event that the test run is taking too long. The base LSP protocol allows for the asynchronous cancellation of an action, and this is implemented in the LSP Server by running CT in a separate thread, allowing the main thread to listen for more interactions from the UI. This complication means that a check has to be added to prevent the user from trying to launch more than one test execution at the same time (which the runtime could not easily support). The server also has to guard against changes to the specification between the expansion of the tests and their subsequent execution. Once a test run is complete, individual tests can be sent to the Interpreter for debugging. This is enabled via the standard DAP protocol, with the addition of a new "runtrace" launch command. So the runtime thinks that the user has started a normal interactive debugging session, but instead of executing something like “print fac(10)”, they are executing “runtrace A’Test 1234”, which will debug test number 1234 of the expansion of the A’Test trace.

In terms of packaging, the SLSP/DAP combination of server functionality is in its own jar, separate from (and dependent on) the standard VDMJ jar. The SLSP/DAP jar is about 174Kb. VDMJ itself is about 2.4Mb.

4 Results and Discussion

In this section we discuss the implementation effort carried out in order to support the LSP, SLSP and DAP protocols for VDMJ and the efforts needed on the client side to provide VDM support. This is compared to the efforts necessary to migrate the Overture language features to a different IDE without using the protocols as performed in [14]. Finally, we discuss to which extent it is currently possible to decouple specification language features from development tools using standardised protocols and what is required to have fully decoupled language support using protocols.

4.1 Assessing the Implementation Effort

The implementation effort is quantified by counting the Lines of Code (LoC) used for implementing support for a given protocol and the features it enables. Providing support the protocols for VDMJ requires 7855 LoC, whereas support for VDM in the VS Code extension only requires 1880 LoC where most are used for the new SLSP features. The few lines of code for the extension can mainly be attributed to the generic LSP and DAP protocol implementations exposed by the VS Code API[17].

A detailed overview of the lines needed to implement the language support is shown in Table 1. In the table the ‘json’ and ‘rpc’ directory contain a basic JSON/RPC system; ‘dap’ and ‘lsp’ are the protocol handlers for each; ‘lsp/lspx’ is the handler for SLSP; ‘vdmj’ is the DebugLink and visitors to implement the various features (i.e. these link to the VDMJ jar); and ‘workspace’ contains the code to maintain the collection of files. For the VS Code extension ‘LSP client’ and ‘DAP client’ is the setup of each client; ‘POG’ and ‘CT’ is the protocol handlers and GUI support for each feature; ‘syntax highlighting’ contains the TextMate grammars for VDM; and ‘SLSP Protocol’ is the definition of the SLSP protocol and expansion of the LSP client.

A comparison between LoC for the VS Code extension, migrating Overture to Emacs (as in [14]) and the Overture IDE (version 3.0.1) not counting the core is seen in Table 2. From the comparison we find that the VS Code extension requires more code than the Emacs migration but Overture consists of far more LoC than both. However, if we only look at the LSP and DAP client support that covers all the normal programming language features they only require 210 LoC. The SLSP parts of the extension are made almost generic, hence these parts can be reused for other specification languages to enable support for the same features in VS Code.

[17] See https://code.visualstudio.com/api/references/vscode-api

[18] The LoC is counted using the “VS Code Counter” extension found at: https://marketplace.visualstudio.com/items?itemName=uctakeoff.vscode-counter

Table 1: LoC measures for the files and directories used to support VDM in VS Code[18]

SLSP/DAP support for VDMJ				
Directory	**Code**	**Comment**	**Blank**	**Total**
dap	699	450	185	1334
json	635	184	139	958
lsp	1653	625	337	2615
lsp/lspx	169	44	30	243
rpc	187	138	56	381
vdmj	2026	451	392	2869
workspace	2655	641	545	3841
Sum	**7855**	**2489**	**1654**	**11998**

VS Code Extension				
Feature	**Code**	**Comment**	**Blank**	**Total**
LSP client	156	16	29	201
DAP client	54	10	15	79
POG	460	51	96	607
CT	732	105	165	1002
Syntax highlight	374	0	0	374
SLSP protocol	104	167	27	298
Sum	**1880**	**349**	**332**	**2561**

For VDMJ to support the protocols it requires considerably more LoC compared to the Emacs approach from [14], which weights against the protocol approach for enabling VDM feature support. However, the protocol solution is a standardisation of the decoupling, hence the server supporting the language features (in this case VDMJ) only has to be implemented once to be used with any development tool supporting the protocols. In comparison, the VS Code extension and migration of the Overture language core to another development tool, as done with Emacs, must be performed for each IDE that is to be supported. Furthermore, the Overture migration depends on Emacs packages, which may not exist in other IDEs or have a different interface. Thus, the analysis of finding suitable packages for another IDE will have to be repeated, possibly making the server solution faster to implement in a new IDE.

Table 2: LoC measure for all IDE related files in the VS Code extension, the Emacs extension and Overture

Project	VS Code (+VDMJ)	Emacs	Overture
Lines of Code	1880 (+7855)	349	>63K

To provide a fair comparison between the three implementation efforts, we compare the features that are available in each. The comparison is done by the same set of features as in [14], this is found in Table 3. As illustrated, the Overture IDE and the Emacs migration both support more features than the VS Code extension. However, some of the features provided by the Emacs migration is only available through a command-line interface and not by a GUI. VS Code does support implementation of integrated command-line interfaces like the Emacs migration, however the implementation efforts related to this are unknown. Providing this kind of command-line interface has intentionally been left out of the VS Code extension, as we wanted to replicate the workflow from Overture.

Table 3: Feature comparison between the VS Code extension, Overture and Emacs. X indicates that a feature is supported. (X) indicates that the feature is only available through a command-line interface, meaning that it is not supported by a graphical user interface.

Features	**VS Code**	**Overture**	**Emacs**
Syntax highlighting	X	X	X
Symbol prettyfication			X
Syntax validation	X	X	X
Evaluation	X	X	(X)
Debugging	X	X	(X)
POG	X	X	(X)
LaTeX report generation		X	(X)
Combinatorial testing	X	X	(X)
Code generation		X	
Auto completion (limited)	X	X	X
Template expansion		X	X
Standard library import		X	

4.2 Protocol Coverage of Language Features

To get an overview of the specification language features that are covered by the protocols, we have examined and grouped the features that are sought after to support specification languages such as VDM, B and Z. This is illustrated in Figure 6, where the language features have been divided into four categories:

1. **Editor**: features that support writing a given specification, e.g. type- and syntax-checking.
2. **Translation**: features that support translating a specification to other formats, e.g. executable code and LaTeX.
3. **Validation**: features that support validation of a specification.
4. **Verification**: features that support verification of a specification.

As a result of the initial implementation and by comparing the features supported by the LSP protocol with the features from Overture, it is found that the protocol is able to support all of the editor related features except syntax highlighting. Thus, specification languages that benefits from the editor features found in Figure 6, can use a direct implementation of the LSP protocol to support this feature set.

The DAP protocol can be used for decoupling the debugging feature as it supports common debug functionality such as different types of breakpoints, variable values and more. This is useful for specification languages that allow execution of specifications.

Additionally, the SLSP extension to LSP facilitates support for both POG and CT. As found in Figure 6 there are still many features related to specification languages that are not supported by protocols. Thus, in order to achieve fully decoupled language server support for VDM one or more protocols must be developed that support the remaining features. From our development of SLSP we believe that it is possible to develop a language-agnostic protocol to support all of these features, which should

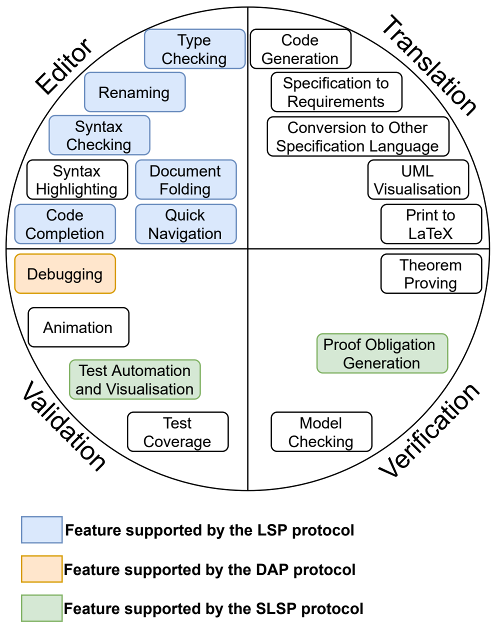


Fig. 6: Specification language features covered by existing protocols

make it possible to apply the protocol to multiple specification languages and potentially increase the industrial uptake of specification languages.

5 Related Work

Support for multiple editors has become common for programming languages, and similar work is now being conducted to achieve the same for modelling and specification languages. Previous work has been focused on providing VDM support for specific platforms, such as Emacs [14], VDMPad [11] and Overture Web IDE [12]. For programming languages it is common to provide language support using standardised protocols to contain the language support in a language-specific server that can be used with multiple language-agnostic clients. This tendency is also becoming apparent for other computer-based languages, such as domain-specific languages (DSL) [3], graphical modelling [13] and formal specification languages such as PVS [10], Dafny [6] and our work for VDM.

Tran and Kulik [14] is able to migrate the Overture core from the Eclipse IDE to Emacs. This is carried out using as many Emacs packages as possible which allows them to provide VDM support with very few additional LoC. This is interesting as it provides knowledge about how few LoC is necessary, which reduces the potential gain from a generalised solution, i. e., if the generalised solution requires a lot of effort you could argue that you might as well have several small dedicated solutions.

Both Oda et al. [11] and Reimer and Saaby [12] presents a web IDE for VDM, based on VDMJ and Overture respectively. Even though both are web based they do not utilise the same language core, thus not providing the same set of language features. Both solutions can probably benefit from unifying the efforts towards shared language support, this could be facilitated using the LSP, DAP and SLSP protocols. Also, VS Code is build using HTML, CSS, and JavaScript which allows the IDE to be launched in a web environment, e. g., Coder[19]. Thus, adding to the number of web IDEs for VDM.

Masci and Muñoz [10] provides support for PVS using the LSP protocol and VS Code. They too use the extensibility of the protocol to provide client-server support for non-LSP features, such as POG and theorem proving. These features are facilitated using a PVS-specific protocol. It would be interesting to investigate if the SLSP protocol could also be used in their case, and what changes may have to be made. This could make way for an addition to the LSP protocol that can be used for a variety of specification languages and possibly be incorporated in the official LSP specification.

6 Concluding Remarks and Future Work

In this paper we have investigated the efforts needed to support VDM in VS Code using the standardised protocols LSP and DAP and our LSP extension SLSP. Also, we evaluated to which extent the VDM language features can be covered by the protocols.

The implementation efforts were estimated using a LoC measure. In total, the VS Code extension consists of 1880 LoC, where only 210 are related to the support of the standardised protocols. This is possible because VS Code provides generic modules for supporting the protocols. Similar generic support is found in other IDEs, we therefore believe that the implementation efforts for these IDEs are comparable to our results. The protocol support for VDMJ totals 7855 LoC which we find appropriate as the server allows the language support to be easily reused in other IDEs as they only have to adhere to the protocols to use the server, which requires little effort compared to creating native support for VDM.

In the language feature coverage evaluation we find that only a subset of the VDM language features are covered by standard protocols. To support features related to specification languages, such as POG and CT we propose the protocol extension SLSP.

Going forward we believe that it would be beneficial to also support other specification language features related to validation, verification and translation, using a similar architecture to decouple the language features from the development tool using a language-agnostic protocol. This would make it easier for multiple development tools to support specification languages, reduce the long term efforts needed to maintain language support, and increase the uptake of specification languages by allowing users to use their preferred development tool.

Acknowledgments We would like to thank Futa Hirakoba for providing information about using VS Code and LSP to support VDM and allowing us to use his implementation of a VS Code extension as a basis for our extensions. We would also like to express our thanks to the anonymous reviewers.

[19] See https://coder.com/.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Bünder, H.: Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages. In: MODELSWARD. pp. 129–140 (2019)
3. Bünder, H., Kuchen, H.: Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol. In: Hammoudi, S., Pires, L.F., Selić, B. (eds.) Model-Driven Engineering and Software Development. pp. 225–245. Springer International Publishing, Cham (2020)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
5. Gray, J.E.: Textmate: Power Editing for Everyone. Pragmatic Bookshelf (2007)
6. Hess, M., Kistler, T.: Dafny Language Server Redesign. Ph.D. thesis, HSR Hochschule für Technik Rapperswil (2019)
7. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
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 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
9. Larsen, P.G., Fitzgerald, J.: The evolution of VDM tools from the 1990s to 2015 and the influence of CAMILA. Journal of Logical and Algebraic Methods in Programming 85(5, Part 2), 985 – 998 (2016), http://www.sciencedirect.com/science/article/pii/S2352220815000954, articles dedicated to Prof. J. N. Oliveira on the occasion of his 60th birthday
10. Masci, P., Muñoz, C.A.: An integrated development environment for the prototype verification system. Electronic Proceedings in Theoretical Computer Science 310, 35–49 (Dec 2019), http://dx.doi.org/10.4204/EPTCS.310.5
11. Oda, T., Araki, K., Larsen, P.G.: VDMPad: A Lightweight IDE for Exploratory VDM-SL Specification. In: Proceedings of the Third FME Workshop on Formal Methods in Software Engineering. p. 33–39. Formalise '15, IEEE Press (2015)
12. Reimer, R.S., Saaby, K.D.: An Open-Source Web IDE for VDM-SL. Master's thesis, Department of Engineering, Aarhus University, Denmark (May 2016)
13. Rodriguez-Echeverria, R., Izquierdo, J.L.C., Wimmer, M., Cabot, J.: Towards a Language Server Protocol Infrastructure for Graphical Modeling. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. p. 370–380. MODELS '18, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3239372.3239383
14. Tran-Jørgensen, P., Kulik, T.: Migrating Overture to a different IDE. In: Gamble, C., Diogo Couto, L. (eds.) Proceedings of the 17th Overture Workshop. pp. 32–47. No. CS-TR- 1530 - 2019 in Technical Report Series, Newcastle University (2019), http://overturetool.org/workshops/17th-overture-workshop.html, null ; Conference date: 07-10-2019 Through 11-10-2019