The Specification Language Server Protocol:
A Proposal for Standardised LSP Extensions

Jonas Kjær Rask   Frederik Palludan Madsen   Nick Battle

DIGIT, Aarhus University, Department of Electrical and Computer Engineering,   Independent
Finlandsgade 22, 8200 Aarhus N, Denmark
                                                                          nick.battle@acm.org
jkr@ece.au.dk   fpm@ece.au.dk

Hugo Daniel Macedo   Peter Gorm Larsen

DIGIT, Aarhus University, Department of Electrical and Computer Engineering,
Finlandsgade 22, 8200 Aarhus N, Denmark

hdm@ece.au.dk   pgl@ece.au.dk

The Language Server Protocol (LSP) changed the field of Integrated Development Environments (IDEs), as it decouples core (programming) language features functionality from editor smarts, thus lowering the effort required to extend an IDE to support a language. The concept is a success and has been adopted by several programming languages and beyond. This is shown by the emergence of several LSP implementations for the many programming and specification languages (languages with a focus on modelling, reasoning, or proofs). However, for such languages LSP has been ad-hocly extended with the additional functionalities that are typically not found for programming languages and thus not supported in LSP. This foils the original LSP decoupling goal, because the move towards a new IDE requires yet another re-implementation of the ad-hoc LSP extension. In this paper we contribute with a conservative extension of LSP providing a first proposal towards a standard protocol decoupling the support of specification languages from the IDE. We hope our research attracts the larger community and motivates the need of a joint task-force leading to a standardised LSP extension serving the particular needs of specification languages.

1   Introduction

Modern Integrated Development Environments (IDEs) go beyond aggregating of the various tools used by developers working with a programming language. IDEs have become platform ecosystems attracting users in search for smart features (e. g., code-completion, show documentation on hovering over a primitive). The ecosystems lead to market segmentation and in some cases users stand by a single ecosystem choice, which in the extreme becomes hate for the alternative IDEs, the popularly known "editor wars". This paves the way to a win-win strategy where programming language maintainers and tool developers jointly gain user adoption when an IDE provides support to their language and vice-versa. However, to put the strategy in practice one hits the $M \times N$ problem [19]: Given M editors and N languages, there are $M \times N$ language support implementations to be developed. Furthermore, each is a re-implementation which opens the way to bug introduction and inconsistent outcomes of invoking the same functionality in different IDEs.

The Language Server Protocol (LSP)[1] changed the field of IDEs, as it decouples core (programming) language features functionality from editor smarts lowering the effort needed to extend a particular IDE with support to a formal language. Thus the problem becomes to produce $M + N$ implementations,

---

[1]https://microsoft.github.io/language-server-protocol/

which additionally may be unified. At the cost of implementing one server for a language, one obtains language smarts (syntax highlighting, type-checking, . . .) support in all the IDEs supporting LSP, and an IDE maintainer can easily support various languages by interacting with the language tools in a standard manner. The additional Debug Adapter Protocol (DAP)[2] provides the same decoupling for executable languages with debugger tool support.

The concept is a success and has been adopted by several programming languages and beyond[3], which is shown by the emergence of several LSP based support for the many programming and specification languages, which have a focus on modelling, reasoning, or proofs (thus containing additional functionalities such as the ones depicted in Figure 3). However, for such languages, the LSP has mainly been extended in an ad-hoc manner with the additional functionalities that are typically not found for programming languages and thus also lacking support in LSP[4][12]. This foils the decoupling goal as the move towards a new IDE requires yet another re-implementation of the LSP extension.

In this paper we contribute with a conservative extension of LSP, the *Specification Language Server Protocol* (SLSP), providing a first proposal towards a standard protocol decoupling the support of specification languages from the IDE, encompassing: Combinatorial Testing [11, 10], Translation/Code-generation [14, 20], Proof Obligation Generation [7, 1] and Theorem Proving [15].

We report on a proof of concept for the protocol that was carried out in the context of a MSc thesis project in [17]. The end result is an implementation of the non-programming language features of the Vienna Development Method (VDM) specification languages that together with a LSP and DAP implement VDM support in Visual Studio Code (VS Code). We hope our research attracts the larger community and congregates the community towards a common effort leading to an LSP extension with effort saving benefits. Moreover, our results confirm that non-LSP based IDEs (Eclipse Rich Client Platforms based on IDE extension plugins) are more difficult to support, given the discrepancy of language specific plugins, thus confirming the value proposed by LSP, DAP, and SLSP.

The remainder of the paper is structured as follows: In Section 2, we provide an overview of the recent protocols used to decouple IDE and language features. Then, Section 3 contains a description of the SLSP protocol. The evaluation of the protocol is described in Section 4 that contains the pilot-study and its results. Section 5 provides an overview the various ad-hoc extensions to LSP that our protocol intends to aggregate. At last, Section 6 concludes the paper, and provides an outline of the future work.

2   Decoupling Using Language-Neutral Protocols

The general purpose of IDEs is to provide support for the development of code and specifications by providing language features such as syntax-checking, hover information and code completion. Supporting such features for a language requires significant effort, which increases for each development tool that should support the language. Decoupling the feature support from the development tool allows extensive reuse as the feature support can be used for multiple development tools. A method to achieve this decoupling is to use a communication protocol in a client-server architecture separating the development tool (client) and the language feature support (server). Thus, a development tool only has to implement the protocol to support a given language, which can be done with little effort compared to native integration [4].

---

[2]`https://microsoft.github.io/debug-adapter-protocol/`

[3]`https://microsoft.github.io/language-server-protocol/implementors/servers/`

[4]e. g., Alloy: `https://github.com/DongyuZhao/vscode-alloy` and
Isabelle: `https://marketplace.visualstudio.com/items?itemName=makarius.isabelle`

This section introduces the existing language-neutral protocols that can be used for decoupling the analysis support from the user interface. These are LSP and the DAP protocols. This is followed by an overview of the features that are not currently supported by those language-neutral protocols.

2.1 The Language Server Protocol

The LSP protocol is a standardised protocol used to decouple a language-agnostic editor (client) and a language-specific server that provides editorial language features such as syntax-checking, hover information and code completion. This is illustrated in Figure 1.
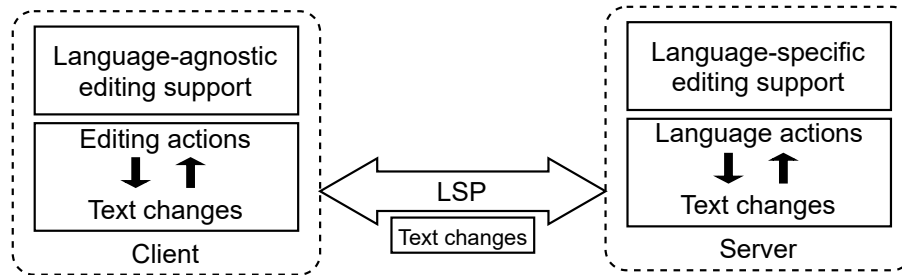

Figure 1: LSP approach to language support. Borrowed from [19].

The client is responsible for managing editing actions without any knowledge of the language by transmitting these to the server as text changes. The server converts the changes into language actions, which is used to perform language-specific support and forward the information to the client. To facilitate this the LSP protocol uses language-neutral data types such as document references and document positions.

The base of the LSP protocol consists of a header and a content part (comparable to HTTP). The content part of a message uses the stateless and lightweight JSON-RPC protocol[5] to facilitate the three base message types: requests, responses and notifications. JSON-RPC can be used within the same process, over sockets and many other message passing environments. This allows the server and client to be on different physical machines. However, most implementations run the server and client as separate process, but on the same physical machine [3]. The request and notification messages provide the entries `method` and `params`, that specify a method (e. g., 'textDocument/didChange') and associated parameters. The LSP protocol specifies a range of different messages to provide the functionality. Furthermore, the request and response messages specify an ID, that provides easy matching of responses to requests.

2.2 The 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 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 language-agnostic debugger component, which is integrated into a development environment as illustrated in Figure 2.
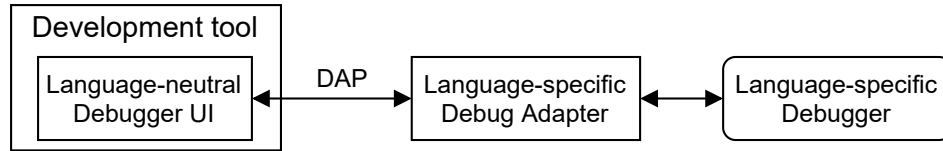
---

[5]`https://www.jsonrpc.org/specification`

Figure 2: The decoupled architecture where the Debug Adapter Protocol (DAP) protocol is used.

The DAP protocol uses a JSON-based wire-format[6] inspired by the V8 Debugging Protocol[7]. This format is similar to but not compatible with the JSON-RPC used in the LSP protocol. Otherwise, it is similar to the LSP protocol with language-neutral data types and a base protocol that has requests, responses and events (similar to notifications from the LSP protocol).

2.3 Support for Specification Languages

The protocols, LSP and DAP, have been developed for programming languages. However, they can also be used with specification languages to support some of the features normally available to programming languages. An overview of common features available for specification languages is illustrated in Figure 3.
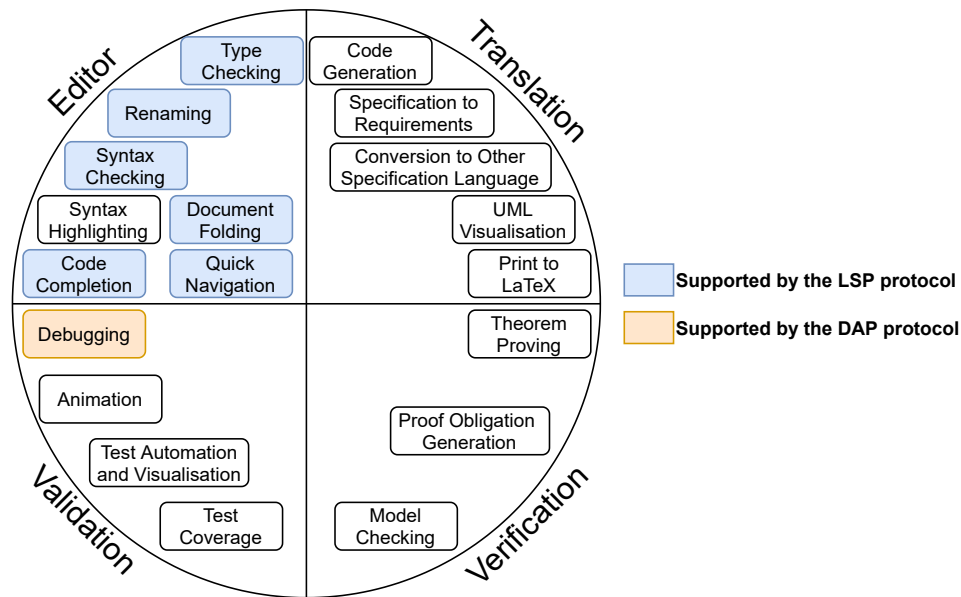

Figure 3: Specification language features covered by existing language-neutral protocols. The individual colours denotes which protocol is needed between the client and language server to support the given feature. Thus, for the LSP protocol a feature is coloured blue, while it is orange for the DAP protocol.

To support editor features not including syntax highlighting a direct implementation of the LSP protocol can be used to decouple the Integrated Development Environment (IDE) from the language core

---

[6]`https://code.visualstudio.com/blogs/2018/08/07/debug-adapter-protocol-website`
[7]`https://github.com/dtretyakov/node-tools/wiki/Debugging-Protocol`

that supports these features. Similarly, the DAP protocol can be used for the debugging feature as it supports common debug functionality such as different types of breakpoints and variable values. However, as clearly illustrated in Figure 3 many of the features that are not commonly found for programming languages cannot be supported using the existing protocols. Thus, there is a need for further protocol development to support all the specification language features.

3 Specification Language Server Protocol

To enable support for specification language features, we propose to use the Specification Language Server Protocol (SLSP). The protocol builds on the design and specification of the LSP protocol by extending the LSP protocol with language independent method calls and data-types. One of the reasons for extending the LSP protocol instead of starting from scratch is that the LSP protocol is the de-facto standard for programming languages [19]. Furthermore, it provides the following benefits:

- LSP defines a base protocol using three message types: *request*, *response* and *notification*.

- LSP includes a number of basic structures than can be used to define messages, such as progress notification and cancel requests.

- LSP defines an infrastructure for communication initialisation between client and server. This enables clients and servers to exchange capabilities, which in turn allows the protocol to be used even if the client or server does not support all the protocol features.

- LSP defines a synchronisation scheme that allows the server to have up-to-date project files even if the user changes them on the client side.

- Finally, the LSP protocol provides support for the editor features found in Figure 3. Thus, these features are directly available in the SLSP protocol.

However, there are a few downsides to extending the LSP protocol. That is, the initialisation and text synchronisation capabilities are non-optional to implement. The initialisation messages are very simple to implement. However, the synchronisation may be slightly over-complicated for some language features, e. g., translation that just needs to know the state of the files when the user activates the features, thus it is not benefiting from the constant synchronisation of the files.

The protocol is developed with a focus on clarifying whether specification language features are able to be supported in a language neutral manner with a client and server communicating using a protocol similar to the LSP protocol. To this effect the initial set of specification language features supported by the protocol is primarily based on the feature support found in the Overture IDE [8, 9] for the dialects of Vienna Development Method (VDM) for which the authors are proficient. For this reason the features and messages may not be expressive enough for some specification languages. However, the protocol is easily extendable with optional entries or additional messages to support missing features. How the protocol should be changed to fully support other languages is left as future work that should be decided on the inputs and requirements from other language specialists.

In the following subsections we describe how the features Proof Obligation Generation (POG), Combinatorial Testing (CT), Translation and Theorem Proving (TP) are supported in the SLSP protocol to support VDM. However, the protocol is based on language-neutral data types, which makes it applicable to any text based language. An overview of the messages in the SLSP protocol is available in Table 1. A detailed description of each of the messages in the SLSP protocol and the design decisions is available in [17]. Furthermore, the support for the features POG, CT and Translation have been tested in a pilot study described in Section 4, however the support for TP has not been tested.

3.1 Proof Obligation Generation

In order for the client to request Proof Obligations (POs) present in the specification, from the server, the message 'generate' is defined in the protocol which enables the server to respond with a list of POs. A PO is defined by the type `ProofObligation` which contains an id, a name, PO type, location in the specification where the PO applies and an optional flag to indicate if the PO is proved. Furthermore, a notification message is also defined which is used for synchronisation of the POs with respect to the specification. This enables the client to request new POG if the specification has changed.

Figure 4 illustrates an example of how a client and a language server communicate during a routine POG session. Similar sequence diagrams for the other features are available in [17].
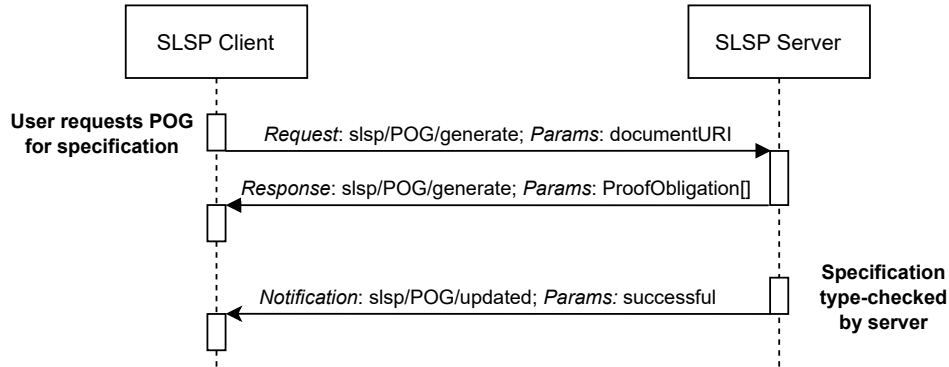

Figure 4: Example of how a client and a language server communicates during POG.

3.2 Combinatorial Testing

For the client to be able to request traces present in the specification the message 'traces' is defined in the protocol. The server is able to respond to this message with a list of the type `CTSymbol` which contains the information necessary for building a tree structure of the traces present in the specification. This includes the name of the trace group, e.g. 'classA' and a collection of traces for the group each specified by a fully qualifying name, their location in the source code and an optional verdict, i.e. if all tests for the trace are passed or failed.

The message 'generate' allows the client to request the server to generate tests for a given trace to get an overview of the number of tests for the trace before execution.

The message 'execute' enables the client to initiate test execution. The message requests execution of a given number of tests in a trace by providing the fully qualifying name of the trace, possible filtering options and the range of tests to execute. The server is allowed to either make a single response containing all test results when test execution has finished or to respond with batches of test results asynchronously by sending partial results back to the client during test execution. The response type is a `CTTestCase` which contains an id, a test verdict and the test case sequence and its result.

3.3 Translation

The protocol supports translation of a specification to any format supported by the server. However, the translation must be able to be carried out purely with information available on the server side as only the single message 'translate' is defined.

The message enables the client to request the server to initiate a translation of the specification to a given format by providing a location on the drive where the server is to store the resulting translation. The server is able to respond with the location of a specific document of the resulting translation that should be opened in the editor.

3.4   Theorem Proving

Theorem Proving (TP) features exists for many specification languages, such as PVS[8] [12], the Isabelle[9] [16] and Lean[10] [13]. However, such functionality is not yet available for VDM. Hence, the support for TP in the SLSP protocol is largely based on the functionality available for PVS in the VSCode-PVS extension [12].

To be able to prove a lemma it is expected that an overview of lemmas in the corresponding specification is available. To this effect the message 'lemmas' has been defined in the protocol, allowing the client to query the server for lemmas in a given specification. Furthermore, to represent a lemma the type `Lemma` has been defined. This type contains its name, a name of the theory that the lemma belongs to, its location, its kind and its status.

To start the proving of a lemma the message 'beginProof' is defined which allows the client to request the server to initiate the theorem prover for the requested lemma. The server response is the initial state of the proof represented by a `ProofState` type which contains an id a status, potential sub-goals and rules.

Usage of Automated Theorem Proving (ATP) is handled in the protocol with the request message 'prove'. The message is send by the client to the server to request the theorem prover to automatically prove a lemma. If a proof has been started the prover should attempt to find a solution for the current lemma at the current proof step. Alternatively, the entry `name` is defined in the message which can be used to specify the name of a lemma that ATP should be applied to. If the ATP process needs to be cancelled before completion the LSP cancel message is used, the same applies for cancelling other commands. Furthermore, the command for initiating ATP often varies between theorem provers. By defining a message for ATP in the protocol specification a single standardised interface, effectively moving the responsibility of calling the specific ATP command(s) to the server. The response to the prove request is of the type `TPProveResponse` which contains a status for the proof, processing time, a list of suggested commands if any and a description of potential counter examples, proof steps, etc.

To facilitate Interactive Theorem Proving (ITP) it should be possible to send commands to a prover at the server side. This is possible with the message 'command' in the protocol. The server should respond with the type `TPCommandResponse` which contains a description of the result of the command as a human-readable string and a proof state of the type `ProofState`. To get the list of available prover commands the message 'getCommands' is defined in the protocol. The response is a collection of type `TPCommand` which contains a name and a description of the command.

To undo a proof step the message 'undo' is defined with the response of type `ProofState` which enables the server to respond with the previous state of the proof. For undoing a specific step that is not the latest one the message includes an `id` entry that specifies the id of the step that the theorem prover should undo. To step through a proof the `command` request should be used. Thus, stepping through a proof is simply a re-transmission of previous prover commands, one for each step.

---

[8]See `https://github.com/nasa/vscode-pvs`.

[9]See `https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2020`.

[10]See `https://github.com/leanprover/vscode-lean`.

Method	Description	Parameters	Response
POG/generate	Generate proof obligations for a file or folder	uri: DocumentUri	ProofObligation[]
POG/updated	Notify that the specification has been updated	successful: boolean	null
CT/traces	Request an outline of the traces in the specification	uri?: DocumentUri	CTSymbol[]
CT/generate	Generate tests for a trace	name: string	numberOfTests: number
CT/execute	Execute tests for a trace, with optional filter and range	name: string filter?: CTFilterOption[] range?: NumberRange	CTTestCase[]
TR/translate	Translates the specification into the specified language	uri?: DocumentUri languageId: string saveUri: DocumentUri	uri: DocumentUri
TP/lemmas	Request list of lemmas in the specification	projectUri?: DocumentUri	Lemma[]
TP/beginProof	Initiate a proof session for a lemma	name: string	ProofState
TP/prove	Automatically prove lemma or request hint	name?: string	TPProveResponse
TP/getCommands	Get list of commands that can be applied to the proof	null	TPCommand[]
TP/command	Apply command to the current step of a proof	command: string	TPCommandResponse
TP/undo	Undo a proof step	id?: number	ProofState

Table 1: List of new messages available in SLSP, for more information see [17]. All methods are prefixed with `slsp/`, e.g., `slsp/POG/generate`. A '?' after the name of an entry indicates that the entry is optional.

As specifications and lemmas can change after they have been proved, it is relevant to be able to re-run a proof. One example of enabling this functionality is to have the client store all commands that have been transmitted to complete a given proof, and keep them stored even if the specification changes. Thus the client is able to re-transmit all the stored commands and check if the proof is still completed. Alternatively, commands could be stored in the source text removing the need for the client to keep a state relating to commands while also alleviating the need to communicate the commands directly to the server as this would be done indirectly through the source text.

4 Pilot Study

To test the SLSP protocol we have carried out a pilot study[11], that implements support for the specification language VDM, more precisely the dialects VDM-SL, VDM++ and VDM-RT. It is implemented in Visual Studio Code (VS Code) as this already supports the protocols LSP and DAP. A detailed description of the support for VDM in VS Code has previously been published [18]. This paper will provide a short description of the extension, VDM VSCode, which is used as a basis for evaluating the use of the language-neutral protocols. The evaluation focuses on the implementation efforts related to using the language-neutral protocols compared to a more language-specific implementation that directly access the language support functionality.

4.1 VDM VSCode

To support VDM in VS Code an extension is developed which implements a client-server architecture that uses the SLSP, LSP and DAP protocols for communication, as illustrated in Figure 5.
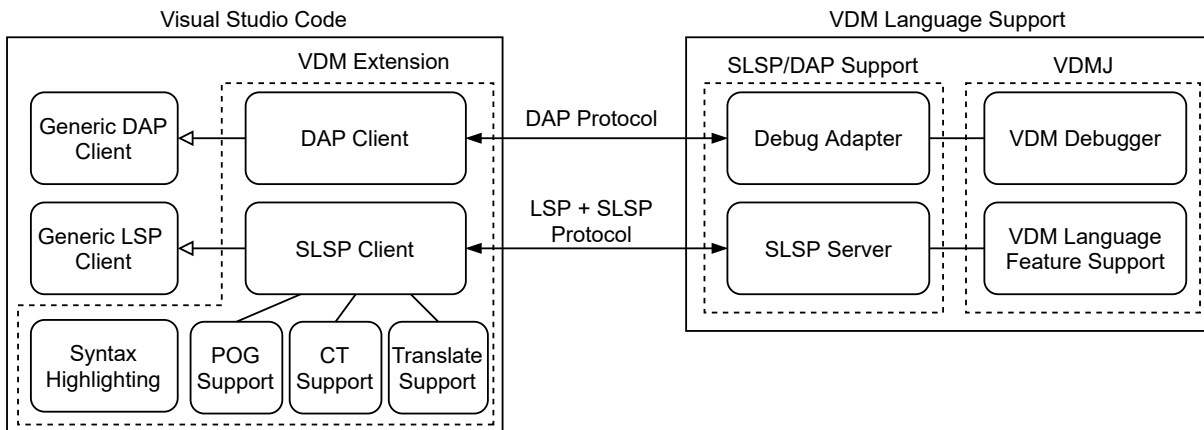

Figure 5: Overview of the architecture of the VDM language extension for VS Code.

The language support is provided by the server that builds on the VDM language core: VDMJ [2]. The functionality of VDMJ is exposed to the protocols through a separate jar that enables the protocol communication and performs function calls to VDMJ. To handle the DAP protocol a debug adapter is implemented that translates DAP messages into actions in the VDM debugger and vice versa. To handle the SLSP and LSP protocol, a SLSP server is created that amongst other things handles the

---

[11]The source code for VDM VSCode is found at: `https://github.com/jonaskrask/vdm-vscode`.

synchronisation of project files such that the client and server always agree on the state of a project. Furthermore, the server handles LSP and SLSP messages and translates these into language actions that are executed using the VDMJ functionality.

The client implementation utilise the VS Code API to provide support for the DAP and LSP protocols. Both of these are supported by generic client modules that provide full support for the protocols with very little extra code. The generic LSP client is easily expandable which allows the SLSP client to reuse the basic message infrastructure. To support the features of SLSP new language-agnostic modules are created that provide message handling and GUI elements for each feature. For the POG feature a table-like view is implemented to show proof obligations and meta-data. For the CT feature a tree view is implemented for navigation between tests and traces. Since the SLSP modules are language-agnostic anyone who adhere to the SLSP protocol will be able to reuse the modules to support any other language. The new views from the VDM VSCode extension are illustrated in Figure 6.
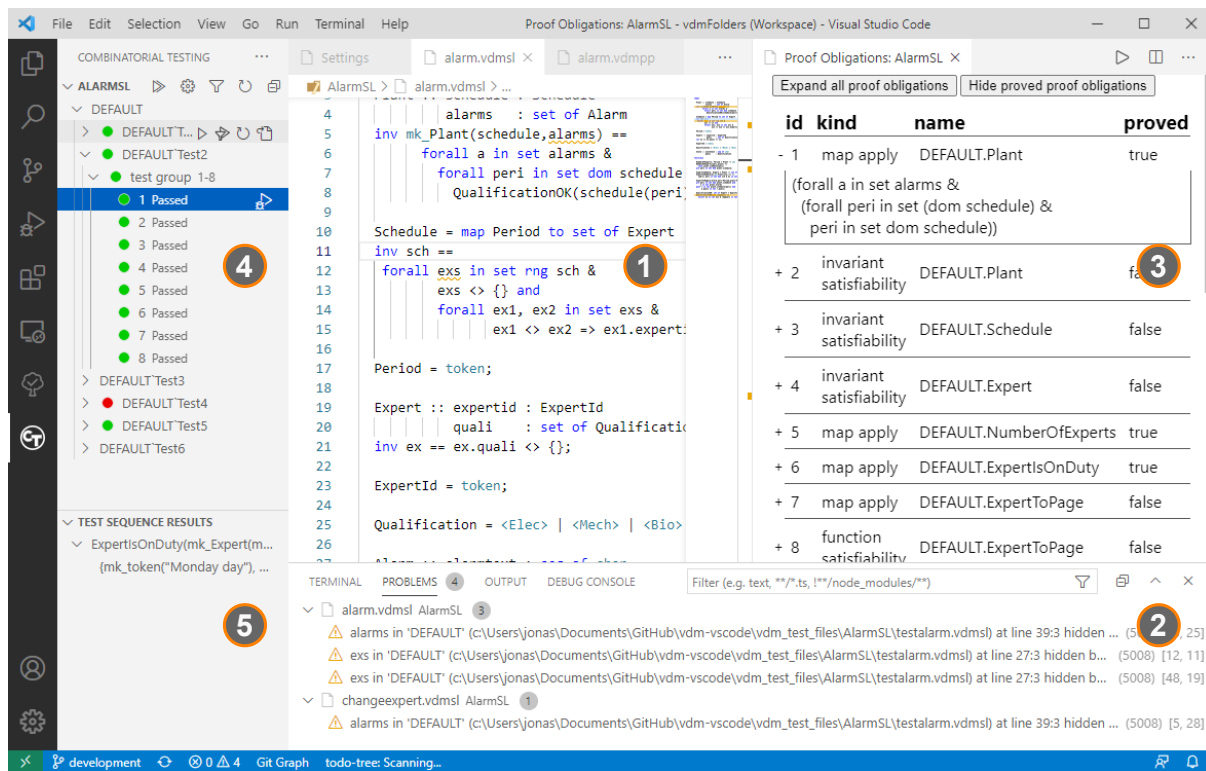

Figure 6: Screenshot of the VDM VSCode Extension: (1) Main Editor; (2) Problems View; (3) Proof Obligation View; (4) Combinatorial Testing View; (5) Test Result View

4.2   Implementation Effort

A concern that might arise, when considering to use the SLSP protocol for the language support, is related to the implementation efforts required. A Lines of Code (LoC) measurement is used to quantify the implementation effort for a given language feature which enables a comparison between VDM VSCode (v1.1.0) not including the server and the Overture IDE (v3.0.1) not including the language core, this is found in Table 2. To be able to properly compare the two IDEs an analysis of the functionality available

for each feature is conducted, which found that VDM VSCode has the same or higher level of support for the implemented features.

Feature	VDM VSCode	Overture IDE
Editor	383	10,163
Debug	60	22,865
POG	494	711
CT	1,014	4,727
Translate to LaTeX	100	1,766
Translate to Word	2	N/A
Sum	**2,068**	**40,232**

Table 2: Comparison of LoC[13](excluding comments and blank spaces) of language feature implementations in VDM VS Code not including the server and the Overture IDE not including the language core. N/A indicates that the feature is not present in the IDE.

From the comparison it is clear, by the sum of LoC, that VDM VSCode requires significantly less LoC on the IDE side than the Overture IDE, with the most significant differences for the editor and debug features. In VDM VSCode these features are supported by the LSP and DAP protocols, respectively. This is made possible due to the generic support for the LSP and DAP protocols available in the VS Code API. Thus, there is a significant reduction in implementation effort when providing support for the features that are generically supported by the IDE, made possible by the language-agnostic nature of the protocols. But what about the new SLSP features that are not supplied with the LSP support?

The features that are supported by the SLSP protocol are the POG, CT, and translation features. For POG the implementation effort for VDM VSCode is slightly less than that of the Overture IDE. In VDM VSCode about half of the LoC are attributed to the protocol handling and the other half to the GUI elements. For CT the LoC are approximately twice of the POG ones, this is expected as the CT feature includes a lot more functionality than the POG feature. Compared to the Overture IDE the effort is significantly reduced. The same is true for the translation features. Thus, it seems that it does not require additional implementation effort to implement the language features in a language-agnostic manner, using the protocols, compared to accessing a more language-specific API, such as that provided by the Overture language core. Furthermore, it should be noted that the POG, CT, and translation modules developed for VDM VSCode are language-agnostic as a result of using the SLSP protocol. That means that any language server that implements the SLSP protocol, can use the modules with very little effort, as evident from the efforts related to the editor and debug features.

Since the language support is divided into a client and a server, implementation effort is also required on the server side to expose the language features through the protocol interface. If you are developing a new language core, the protocol interface can be incorporated from the beginning. However, if a language core is already available this is extra work that must be carried out to use the protocols to provide the language support. Thus, for a complete picture of the effort to implement support for a specification language using the SLSP protocol, the LoC count for the server-side implementation should also be considered. However, it is important to note that this implementation effort only has to be made once for the server to be available across all IDEs that implement support for the protocols.

---

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

4.2.1   Effort Related to the Server

One important aspect of implementing the pilot study was to discover how closely the programming language concepts in the LSP and DAP protocols match the concepts in the VDM specification language. The concepts in the protocols are deliberately abstract, such as a location within a file or a symbol name, rather than being language specific. The problem is to map these abstract concepts into concrete language features in the chosen VDM dialect.

The most significant difference for VDMJ operating via LSP is that the file system does not necessarily represent the latest version of a source file. Rather, the client can send 'didChange' events at any time to indicate that a file is actively being edited. These changes must be syntactically checked on the fly, and the user is under no obligation to save the files to disk before seeing any errors. This clashes with the design of the VDMJ type checker, which needs to see the entire specification. So the LSP server design caches user changes in memory.

Several of LSP's language features map well to VDM. However, one annoyance with outline information is that symbols have to be one of a fixed set of kinds, like 'class', 'method', 'interface' and so on. The set is reasonably rich, but there is no natural mapping for some of VDM's definition categories - for example what is a combinatorial trace, or a permission predicate guarding an operation?

Package	Description	LoC
json	JSON message support	787
rpc	JSON RPC support	284
lsp	LSP/SLSP protocol support	2,033
dap	DAP protocol support	1,265
vdmj	VDMJ interface support	3,374
workspace	Workspace coordination	3,342
Sum		**10,985**

Table 3: Java LoC for the server (excluding comments and blank spaces) not including the VDMJ language core.

Table 3 contains an overview of the implementation efforts of extending VDMJ to handle the SLSP protocol indicated as lines of Java code. The support consists of six packages. The packages 'json' and 'rpc' handle the basic message support, with no knowledge of the SLSP, LSP and DAP protocols, hence you could use those packages to write an arbitrary JSON RPC server The 'lsp' and 'dap' packages contain handlers, for the SLSP, LSP and DAP protocol messages, that delegate to the workspace to get the job done. Lastly, the workspace package understands 'how' to do things, handles all the VDM sources, and uses the 'vdmj' package to talk to VDMJ.

It should be noted that there exists several SDKs that implement the LSP and DAP protocols[14], e. g., LSP4J[15] that makes it easy to implement a language client and server in Java. Using such an SDK would reduce the efforts as they supply the functionality of the 'json', 'rpc' and 'dap'. Also, it will implement most of the 'lsp' package, except the SLSP parts. However, since no such SDK was used for the server it is unknown whether it could be extended for SLSP easily.

---

[14]See `https://microsoft.github.io/language-server-protocol/implementors/sdks/`.
[15]See `https://github.com/eclipse/lsp4j`.

4.2.2 Combining the Efforts

Combining the LoC for the client and server the VDM VSCode extension requires a total of $2068 + 10,985 = 13,053$ LoC. This is still significantly less than the Overture IDE that requires a total of 40,903 LoC. Based on these results, it is expected that the implementation efforts to support future features of the SLSP protocol are less than that of their comparable implementation in the Overture IDE. Similarly, it is expected that feature support for other languages will not require an additional implementation effort by using the protocols compared to more direct support. Thus, the tool and language developers will be able to benefit from the long term advantages of having separate language-agnostic clients and language-specific servers, without an increased initial implementation effort.

5 Related Work

The interest of having a protocol facilitated language support for other formal languages besides VDM is evident in tool implementations for related formal specification languages such as Prototype Verification System (PVS) [12], Dafny [6], Coq[16] [5], Isabelle[17] [16] and Alloy[18]. A common pattern in all the implementations is the leverage of the LSP protocol for editor-related features. Some of them also include language features beyond the scope of the LSP protocol, for which they implement their own language-specific protocols. This is done by extending the LSP protocol with language specific messages resulting in a protocol that is no longer language-agnostic.

This is also present in the paper by Masci and Muñoz [12]. From the architecture it is found that the extension integrates the editing and proof management functionalities of PVS in VS Code, enabling functionalities that developers expect to find in modern verification tools. To enable support for features that are not supported by the LSP protocol, such as interactive analysis of PVS specifications, the protocol is extended resulting in an extended protocol that is not language-agnostic, because the extensions are bound to specific language features of PVS. This allows their language server to be partially reused for IDEs that support the LSP protocol. However, support for the PVS-specific protocol must be separately implemented.

On the other hand, the SLSP protocol presented in this paper is developed with the purpose of supporting features that are common for multiple specification languages. For instance, SLSP supports proof obligation generation. Although its implementation was tailored to support the VDM feature needs, thus its naming; the protocol intends to cover the needs of extensions for languages generating proof goals. One such example are the type-checking conditions generating feature available in the PVS VS Code extension. As such, the development of the SLSP protocol aims to increase the motivation for tool developers to add native support for the protocol, thus reducing the combined effort of implementing language support for specification languages in multiple IDEs. That is why we believe that it is worth exploring the feasibility of using the SLSP protocol in tool implementations for specification languages that already leverages the LSP protocol.

---

[16]See `https://github.com/siegebell/vscoq`.
[17]See `https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2020`.
[18]See `https://github.com/s-arash/org.alloytools.alloy/tree/ls`.

6   Concluding Remarks and Future Work

Our research on how to best extend the LSP and DAP protocols with the VDM suite of specification languages features lead to a new protocol proposal, the SLSP protocol. We implemented the first client and server for it, and it includes the non-programming language features of the VDM specification languages, and together with a LSP and DAP protocol implementation, it extends VS Code to become an IDE for our specification languages and to be our future default implementation supporting them. An overview of our implementation coverage of specification language features is depicted in Figure 7, where we assert which features are covered by each protocol, and which features that have been tested in the VDM VS Code extension.
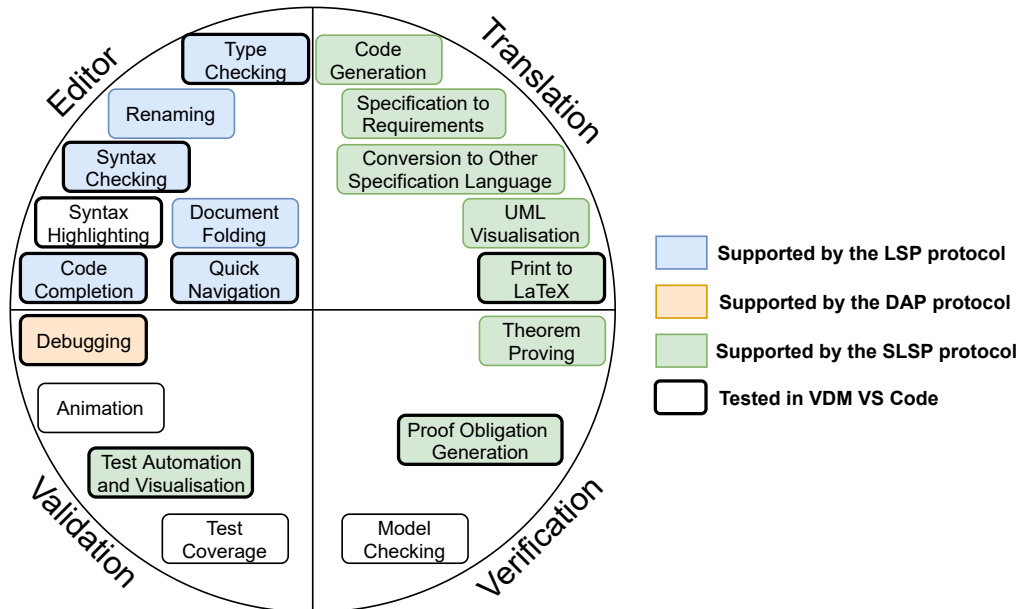

Figure 7: Specification language features supported by the LSP, DAP and SLSP protocols. A boldface edge indicates that a feature has been successfully implemented in the pilot study and is available in the VDM VS Code extension. The individual colours denotes which protocol is needed between the client and language server to support the given feature. Thus, for the LSP protocol a feature is coloured blue, while it is orange for the DAP protocol and green for the SLSP protocol.

Our results show that the language-neutral support for specification languages is feasible, given the ability to extend LSP within our proof of concept. Nevertheless, our work is in its initial stages and more work is needed to fully cover the needs of the VDM language. For example, we did not implement theorem proving in the proof of concept although it is supported by the SLSP protocol as reported in Figure 7. By using LSP, DAP and SLSP, we reduced implementation effort, which will be valuable in the long run, especially when extending other IDEs, as observed in the reduced number of LoC. In addition, the SLSP protocol was designed to be an extension to LSP to support specification language features in a decoupled manner, thus transforming the $M \times N$ problem into $M + N$ as LSP does. We expect our work motivates the other formal specification language community members to join efforts to develop a standardised version of the SLSP protocol that adequately supports the language features found across various formal specification languages, thus extending this result to all of them.

The intention is that when SLSP is completed with the remaining Overture/VDM features, we expect this method to be the one that the VDM community is going forward with. However, we expect that the first step is that SLSP is used for the proof support parts of another specification language. e. g., maturing the protocol to be able to support either Isabelle or Coq; some of the hardest interactive theorem provers to provide an interface.

Moving forward, it is our hope that the SLSP protocol becomes the standard for IDE support of specification language features, such as the LSP protocol is for editor features. To get there, the SLSP protocol has to be tested for other specification languages than VDM through further pilot studies. This will mature the protocol and identify any shortages in it, e. g., for functionality not found in VDM that is available in other languages.

Another interesting path for future work is to utilise the introduction of a client-server architecture for the language support to enable distributed language support. Since the base-protocol is very lightweight a client could be connected to multiple servers, each specialised for a specific feature, e. g., one which is fast at small editorial actions that requires a quick response and another for long-running tasks such as theorem proving. This could also be used for parallel execution of some tasks, e. g., combinatorial testing would be able to benefit from this as each test is independent.

Acknowledgement

References

[1] Bernhard K. Aichernig & Peter Gorm Larsen (1997): *A Proof Obligation Generator for VDM-SL*. In John S. Fitzgerald, Cliff B. Jones & Peter Lucas, editors: *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997), Lecture Notes in Computer Science* 1313, Springer-Verlag, pp. 338–357. ISBN 3-540-63533-5.

[2] Nick Battle (2009): *VDMJ User Guide*. Technical Report, Fujitsu Services Ltd., UK.

[3] Hendrik Bünder (2019): *Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages*. In: *MODELSWARD*, pp. 129–140.

[4] Hendrik Bünder & Herbert Kuchen (2020): *Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol*. In Slimane Hammoudi, Luís Ferreira Pires & Bran Selić, editors: *Model-Driven Engineering and Software Development*, Springer International Publishing, Cham, pp. 225–245.

[5] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring & Benjamin Werner (1993): *The Coq Proof Assistant User's Guide*. Rapport Techniques 154, INRIA, Rocquencourt, France. Version 5.8.

[6] Marcel Hess & Thomas Kistler (2019): *Dafny Language Server Redesign*. Ph.D. thesis, HSR Hochschule für Technik Rapperswil.

[7] C.B. Jones (1985): *The role of proof obligations in software design*. In: *Lecture Notes in Computer Science, Vol. 186*, Springer-Verlag, pp. 27–41.

[8] Peter Gorm Larsen (2001): *Ten Years of Historical Development: "Bootstrapping" VDMTools*. Journal of Universal Computer Science 7(8), pp. 692–709.

[9] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl & Marcel Verhoef (2010): *The Overture Initiative – Integrating Tools for VDM*. SIGSOFT Softw. Eng. Notes 35(1), pp. 1–6, doi:10.1145/1668862.1668864. Available at `http://doi.acm.org/10.1145/1668862.1668864`.

[10] Peter Gorm Larsen, Kenneth Lausdahl & Nick Battle (2010): *Combinatorial Testing for VDM*. In: *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, IEEE Computer Society, Washington, DC, USA, pp. 278–285, doi:10.1109/SEFM.2010.32. Available at `http://dx.doi.org/10.1109/SEFM.2010.32`. ISBN 978-0-7695-4153-2.

[11] Yves Ledru (2002): *The TOBIAS Test Generator and Its Adaptation to Some ASE Challenges*. In: *Workshop on the State of the Art in Automated Software Engineering*, University of California, Irvine.

[12] Paolo Masci & César A. Muñoz (2019): *An Integrated Development Environment for the Prototype Verification System*. Electronic Proceedings in Theoretical Computer Science 310, p. 35–49, doi:10.4204/eptcs.310.5. Available at `http://dx.doi.org/10.4204/EPTCS.310.5`.

[13] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25*, Springer International Publishing, Cham, pp. 378–388.

[14] Paul Mukherjee (1997): *Automatic Translation of VDM-SL Specifications into Gofer*. In John Fitzgerald, Cliff B. Jones & Peter Lucas, editors: *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, Lecture Notes in Computer Science 1313, Springer-Verlag, pp. 258–277. ISBN 3-540-63533-5.

[15] Tobias Nipkow (1989): *Term Rewriting and Beyond — Theorem Proving in Isabelle*. Formal Aspects of Computing 1, pp. 320–338. Available at `ftp://ftp.informatik.tu-muenchen.de/local/lehrstuhl/nipkow/fac89.html`.

[16] Lawrence C. Paulson (1986): *Natural Deduction as Higher-order Resolution*. Journal of Logic Programming 3, pp. 237–258. Available at `http://www.cl.cam.ac.uk/Research/Reports/TR82-lcp-higher-order-resolution.dvi.gz`.

[17] Jonas Kjær Rask & Frederik Palludan Madsen (2021): *Decoupling of Core Analysis Support for Specification Languages from User Interfaces in Integrated Development Environments*. Master's thesis, Aarhus University, Department of Engineering. Available at `https://github.com/overturetool/overturetool.github.io/blob/32b46df36502f1bfcbc4ea8ccad2ab10da04159f/publications/theses/Rask&21.pdf`.

[18] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo & Peter Gorm Larsen (2020): *Visual Studio Code VDM Support*. In: *Proceedings of the 18th International Overture Workshop*, p. 35–49. Available at `https://arxiv.org/abs/2101.07261`.

[19] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer & Jordi Cabot (2018): *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*, MODELS '18, Association for Computing Machinery, New York, NY, USA, p. 370–380, doi:10.1145/3239372.3239383. Available at `https://doi.org/10.1145/3239372.3239383`.

[20] Peter W. V. Tran-Jørgensen, Peter Gorm Larsen & Gary T. Leavens (2017): *Automated translation of VDM to JML-annotated Java*. International Journal on Software Tools for Technology Transfer, pp. 1–25, doi:10.1007/s10009-017-0448-3. Available at `http://dx.doi.org/10.1007/s10009-017-0448-3`.