



AARHUS  
UNIVERSITY

DEPARTMENT OF ENGINEERING

# DECOUPLING OF CORE ANALYSIS SUPPORT FOR SPECIFICATION LANGUAGES FROM USER INTERFACES IN INTEGRATED DEVELOPMENT ENVIRONMENTS

BY

JONAS KJÆR RASK

201507306

AND

FREDERIK PALLUDAN MADSEN

201504477

MASTER'S THESIS

IN

COMPUTER ENGINEERING

SUPERVISOR: PROF. PETER GORM LARSEN

CO-SUPERVISOR: HUGO DANIEL MACEDO

Aarhus University, Department of Engineering

January 5, 2021

---

Jonas Kjær Rask  
201507306

---

Frederik Palludan Madsen  
201504477

---

Peter Gorm Larsen  
Supervisor



# Abstract

Decoupling the core analysis support from the User Interface (UI) in Integrated Development Environments (IDEs) enables extensive reuse when implementing language support in multiple IDEs. This can be achieved by utilising a client-server architecture to decouple the UI from the core analysis support. Communication between the processes in the decoupled architecture is facilitated using a *language-neutral* protocol to enable a *language-agnostic* client and a *language-specific* server to communicate which allows the client to be used with multiple servers and vice versa.

For IDEs that support programming languages there is already a tendency to use this solution but, as this thesis finds, the existing language-neutral protocols Language Server Protocol (LSP) and Debug Adapter Protocol (DAP) lacks support for several language features found in specification languages. Therefore, this thesis proposes a new language-neutral protocol, called the Specification Language Server Protocol (SLSP), that is able to support language features specific to specification languages.

To demonstrate that the language-neutral protocol can facilitate support for specification languages, a pilot study is conducted. In the pilot study a Proof of Concept (PoC) is implemented that provides support for Vienna Development Method (VDM) in Visual Studio Code (VS Code) using the language-neutral protocols.

From the work presented in this thesis it is concluded that it is feasible to support specification language features in the SLSP protocol and to employ the protocol in a client-server architecture that decouples the UI from the core language analysis support. However, the SLSP protocol needs further testing with other specification languages than VDM to validate that it is generally applicable.



# Preface

This Master's thesis is the final part of the Master's degree in Computer Engineering at Aarhus University. It accounts for 30 ECTS points and was conducted from the 31st of August 2020 to the 5th of January 2021.

We would like to thank our academic supervisor, Peter Gorm Larsen, and co-supervisor, Hugo Daniel Macedo, for their attention and valuable advice and feedback during the preparation and execution of the thesis.

We would also like to thank Nick Battle for his development of the language server used in the pilot study and his great feedback in the development of the protocol. Furthermore, we would like to thank Futa Hirakoba for providing information about using VS Code and the LSP protocol to support VDM and allowing us to use his implementation of a VS Code extension as a basis for our extensions.

# Contents

<b>Preface</b>	<b>iii</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Overview . . . . .	1
1.2 Motivation . . . . .	2
1.3 Goals . . . . .	2
1.4 Scope . . . . .	3
1.5 Approach . . . . .	4
1.6 Pilot Study . . . . .	4
1.7 Publication . . . . .	5
1.8 Reading Guide . . . . .	5
1.9 Structure . . . . .	6
<b>2 Background</b>	<b>9</b>
2.1 Computer-Based Languages . . . . .	9
2.2 Toolsets Supporting Specification Languages . . . . .	11
2.3 Toolset Architecture Overview . . . . .	14
2.4 Existing Language-neutral Protocols . . . . .	17
<b>3 Decoupling Using Language-neutral Protocols</b>	<b>22</b>
3.1 Previous IDE Integrations for VDM . . . . .	22
3.2 Features Supported by Standard Protocols . . . . .	23
3.3 Decoupling Architectures . . . . .	24
3.4 Arguments for Extending the LSP Protocol . . . . .	27
<b>4 The Specification Language Server Protocol</b>	<b>29</b>
4.1 Reuse of the LSP Protocol . . . . .	29
4.2 Support for Proof Obligation Generation . . . . .	30
4.3 Support for Combinatorial Testing . . . . .	35
4.4 Support for Translation . . . . .	42
4.5 Support for Theorem Proving . . . . .	45
4.6 Methodology . . . . .	52

## CONTENTS

<b>5</b>	<b>Pilot Study</b>	<b>57</b>
5.1	Supporting VDM in VS Code . . . . .	57
5.2	Implementing the Language Extension . . . . .	59
5.3	Evaluation . . . . .	65
<b>6</b>	<b>Related Work</b>	<b>71</b>
6.1	IDE Support for VDM . . . . .	71
6.2	Protocol Support for Formal Languages . . . . .	73
6.3	Protocol Support for Domain-Specific and Graphical Languages . .	73
<b>7</b>	<b>Concluding Remarks</b>	<b>75</b>
7.1	Introduction . . . . .	75
7.2	Discussion . . . . .	75
7.3	Thesis Project Reflections . . . . .	78
7.4	Evaluation of the Goals . . . . .	78
7.5	Conclusion . . . . .	80
7.6	Future Work . . . . .	81
	<b>Bibliography</b>	<b>85</b>
<b>A</b>	<b>SLSP Protocol Outline</b>	<b>91</b>
A.1	Base Protocol . . . . .	91
A.2	Proof Obligation Generation . . . . .	97
A.3	Combinatorial Testing . . . . .	99
A.4	Translation . . . . .	105
A.5	Theorem Proving . . . . .	107
<b>B</b>	<b>Theorem Proving Using SLSP</b>	<b>113</b>
B.1	Proof Status . . . . .	113
B.2	Using SLSP for the VSCode-PVS Interface . . . . .	114
B.3	Large Version of VSCode-PVS Snippet . . . . .	120
B.4	Large Version of Isabelle Snippet . . . . .	121
<b>C</b>	<b>Extension User Guide</b>	<b>123</b>
C.1	The Interface . . . . .	123
C.2	Example Use . . . . .	125
<b>D</b>	<b>Assessing Performance</b>	<b>127</b>
D.1	Test Traces . . . . .	127
D.2	Performance Profiles . . . . .	129
D.3	Interpreter Execution Time . . . . .	132
D.4	LUHN.vdmsl - A Realistic Specification . . . . .	133

# List of Figures

1.1	An illustration of the approach used in this thesis. It shows the main stages and main transitions between stages. . . . .	4
1.2	Conceptual drawing of the goal of the Proof of Concept. . . . .	5
1.3	Structure of the chapters in this thesis. . . . .	8
2.1	Overview of the Overture architecture. . . . .	14
2.2	Overview of the Rodin Platform architecture. . . . .	16
2.3	Overview of the Community Z Tools architecture. . . . .	17
2.4	Example of language server communication sequence. . . . .	19
2.5	The decoupled architecture where the DAP protocol is used. . . . .	20
2.6	DAP protocol start sequence example. . . . .	21
3.1	Specification language features covered by existing protocols. . . . .	23
3.2	Scenarios of a decoupled design. . . . .	25
4.1	Snippet of the Proof Obligation Explorer in the Overture IDE. . . . .	31
4.2	Example of how a client and a language server communicates during POG. . . . .	32
4.3	Snippet of the CT Overview in the Overture IDE. . . . .	36
4.4	Example for how a client and a language server communicate during CT. . . . .	38
4.5	Screen shot of the VSCode-PVS extension. (1) Editor, (2) Theorem Prover terminal, (3) Theory overview, (4) Proof Explorer and (5) Proof support/suggestions. Further explanation of the figure is available in Section 4.5.3. . . . .	46
4.6	Sequence diagram showing the possible messages, their parameters and their responses. . . . .	48
4.7	Screen shot of the VSCode-PVS extension. (1) Editor, (2) Theorem Prover terminal, (3) Theory overview, (4) Proof Explorer and (5) Proof support/suggestions. . . . .	51
4.8	The processes of implementing language support for a specification language in a given IDE and exposing language features of a language server by leveraging the SLSP protocol. . . . .	53
5.1	Overview of the architecture of the VDM language extension for VS Code. . . . .	59
5.2	Proof obligations in VS Code. . . . .	61
5.3	Combinatorial testing in VS Code. . . . .	63



## LIST OF TABLES

5.4	States of the system during combinatorial testing. . . . .	64
5.5	Translation options available in the VS Code extensions context menu. .	65
5.6	The execution times for the VS Code Extension, the Overture IDE, the VDMJ CLI and the VS Code Extension without partial results. . . . .	69
5.7	The execution times for the VS Code Extension and the VDMJ CLI. The comparison is performed with realistic traces using Luhn’s algorithm as detailed in the VDM model shown in Appendix D.4. . . . .	70
7.1	Specification language features supported by the LSP, DAP and SLSP protocols and their availability in the VS Code extension. . . . .	76
B.1	State machine diagram showing a suggestion for how to transition between the different proof states. . . . .	113
B.2	Screen shot of the VSCode-PVS extension. (1) Editor, (2) Theorem Prover terminal, (3) Theory overview, (4) Proof Explorer and (5) Proof Mate. . . . .	120
B.3	Snippet of theorem proving in the Isabelle VS Code extension. . . . .	121
C.1	View elements of the VS Code VDMSL extension GUI. . . . .	124
D.1	Performance profile of VDMJ executing the trace <code>Test3</code> , executed by the VS Code extension. . . . .	130
D.2	Performance profile of VDMJ executing the trace <code>Test3</code> , executed by the CLI. . . . .	131

## List of Tables

2.1	Features supported in the Overture IDE, the Rodin Platform and Community Z Tools grouped into categories. . . . .	13
5.1	LoC measures for the files and directories used to support the protocols SLSP, LSP and DAP for VDMJ. . . . .	60
5.2	PO functionality available in the VS Code extension and the Overture IDE. . . . .	66
5.3	CT functionality available in the VS Code extension and the Overture IDE. . . . .	66
5.4	Comparison of LoC (exluding comments and blank spaces) of language feature implementations in the VS Code extension and the Overture IDE. N/A indicates that the feature is not present in the IDE. . . . .	67

## LIST OF TABLES

D.1	Comparison of time taken (in seconds) for executing the same specific set of traces in the VDMJ CLI, the Overture IDE and the VS Code extension. The numbers in the parentheses indicate the number of test cases generated from the trace. Each result is the average of three runs. . . . .	128
D.2	Comparison of time taken (in seconds) for executing the same traces in the VS Code extension, the VDMJ CLI and the Overture IDE. 'N/A' means that the trace could not be executed to completion. The comparison was performed with realistic traces using the Luhn algorithm as detailed in the VDM model shown in Appendix D.4. Each result is the average of three runs. . . . .	129
D.3	Execution times (in seconds) as a result of executing the same operation (Listing D.2) three times per interface. . . . .	132

# Glossary

**animation** Creation of graphical interfaces to a specification. Some use the term animation for interpretation or stepping through a specification, that is not how it is used in this thesis.

**back-end** Front-end and back-end refer to the separation of concerns between the user interface (front-end), and the data access layer (back-end) of a piece of software. In the client–server model, the client is considered the front-end and the server is considered the back-end.

**development tool** A development tool is used to create, support and maintain projects implemented using e.g., a specification language or programming language. A development tool can range from a simple editor to a fully fledged GUI.

**framework** Defines the interaction and implements the boilerplate code between tools or components.

**front-end** Front-end and back-end refer to the separation of concerns between the user interface (front-end), and the data access layer (back-end) of a piece of software. In the client–server model, the client is considered the front-end and the server is considered the back-end.

**language feature** Features created for development in a certain language, that serves to maximise the programmer’s productivity, such as syntax highlighting, debugging, type-checking and code completion.

**LaTeX** Is a software system for document preparation, that is often used by academics.

**lemma** Used as placeholder for any statement that can be proved, such as theorems, lemmas and proof obligations.



# Acronyms

**API** Application Programming Interface.

**AST** Abstract Syntax Tree.

**ATP** Automated Theorem Proving.

**CLI** Command-Line Interface.

**CT** Combinatorial Testing.

**CZT** Community Z Tools.

**DAP** Debug Adapter Protocol.

**DSL** Domain-Specific Language.

**GLSP** Graphical Language Server Platform.

**GUI** Graphical User Interface.

**IDE** Integrated Development Environment.

**ITP** Interactive Theorem Proving.

**LoC** Lines of Code.

**LPF** Logic of Partial Functions.

**LSI** Language Server Interface.

**LSIF** Language Server Index Format.

**LSP** Language Server Protocol.

**PO** Proof Obligation.

**PoC** Proof of Concept.

**POG** Proof Obligation Generation.

## *ACRONYMS*

**PVS** Prototype Verification System.

**RCP** Rich Client Platform.

**RPC** Remote Procedure Call.

**SLSP** Specification Language Server Protocol.

**TAP** Test Anything Protocol.

**TP** Theorem Proving.

**UI** User Interface.

**URI** Uniform Resource Identifier.

**VDM** Vienna Development Method.

**VS Code** Visual Studio Code.

# Chapter 1

## Introduction

This chapter introduces the subject and motivation of this thesis project and serves as a general introduction. To set the context of the subsequent chapters, the thesis goals are presented along with the scope of the thesis. This is followed by a description of the approach used to reach the goals and a short introduction to the pilot study developed in the project. In addition, the chapter includes formalities about the thesis, that is, information about published work based on the thesis, a reading guide and a presentation of the structure of the thesis.

### 1.1 Overview

Most modern Integrated Development Environment (IDE) applications are constructed to support *language feature*<sup>7</sup> (syntax-checking, quick navigation, code completion etc.) which serves to maximise the productivity of the programmer when working with a specific computer-based language. Therefore, several common features (besides the basics: compiler, linker, etc.) can be found across IDEs that are useful for most programming languages, such as debugging, syntax highlighting and code completion. The integration of these language features with the programming languages supported by a given IDE is typically left to the developers in the community to implement. Therefore, while there seems to be a common language feature set when looking at the IDE landscape the implementations can vary vastly both in terms of design and implementation language. The Language Server Protocol (LSP)<sup>1</sup> protocol enables programming language services to be decoupled from the Graphical User Interface (GUI) of an IDE such that core analysis support may be contained within a language server. The protocol is utilised in a client-server relationship between the GUI and the associated language support. This allows for easy reuse of language support in IDEs that also complies with the protocol.

The LSP protocol supports some of the language features that are desired for speci-

---

<sup>1</sup>See

<https://microsoft.github.io/language-server-protocol/specification>.

fication languages but not all, as it is tailored towards programming languages. This means that language features specific to specification languages are not supported. Thus, in order to achieve the same decoupled architecture and possibility for reuse, there is a need for a generalised language-neutral protocol that supports the specialised specification language features. This includes features such as proof obligation generation, code generation, translation and even theorem proving.

This thesis argues for and proposes a new language-neutral protocol to be employed in a client-server architecture to provide language support for specification language features. The protocol is used in a pilot study, where language support is provided for Vienna Development Method (VDM) in Visual Studio Code (VS Code), using a client-server architecture that communicates using language-neutral protocols. The pilot study is used in the evaluation of the feasibility of using such a solution for decoupling the language support from the GUI in IDEs for specification languages.

## 1.2 Motivation

The authors of this thesis had previous experience with using the Overture IDE for developing VDM specifications. The Overture IDE was found to be quite cumbersome to work with, have an ageing interface and generally lack some of the features that are a common for modern IDEs for programming languages. From investigating how programming languages are supported in modern extension-based IDEs like VS Code it was found that they use a client-server-based architecture and a standardised communication protocol, which allows the language support to be easily used for multiple IDEs. This kind of generalisation of software components and connections between processes is of interest to the authors. Therefore, it was decided to investigate if the same approach can be used for specification languages, which include lots of language features that are not found in programming languages, such as Proof Obligation Generation (POG).

The work of this thesis is relevant to anyone who provides tool support for or develops specification languages, as it can be used to centralise the language support efforts in one project and thus reduce their effort to support the language in multiple IDEs. In addition, the thesis describes the tasks and processes that are followed to implement support for the VDM specification languages, which interested readers may find useful.

## 1.3 Goals

Multiple modern IDEs that implement support for programming languages, such as C++, Java and Python, use the LSP protocol<sup>2</sup>. Some specification languages and

---

<sup>2</sup>For the full list see <https://microsoft.github.io/language-server-protocol/implementors/servers/>.



## CHAPTER 1. INTRODUCTION

other computer-based languages are also starting to use the LSP protocol to achieve the same decoupling between language support and IDEs [1–3]. However, the LSP protocol is not developed with these languages in mind, with the result that many of the language features are not supported by the LSP protocol. To achieve a complete decoupling for specification languages these features must also be decoupled from the IDE.

As such, the hypothesis of this thesis is:

*It is possible to decouple the language support for specification languages from the user interface using a client-server architecture, with a **language-neutral** protocol between a **language-agnostic** client and a language-specific server.*

In order to adequately test the hypothesis, four goals of the thesis are defined:

- G1:** Explore what language features are relevant for specification languages and to which extend they can be supported by existing standardised protocols.
- G2:** Propose a language-neutral protocol for supporting specification language specific features.
- G3:** Develop a Proof of Concept (PoC) for supporting VDM in VS Code using a client-server-based architecture.
- G4:** Evaluate the feasibility of using the proposed protocol for decoupling a language-agnostic client from a language-specific server.

### 1.4 Scope

The scope of the thesis focuses on proposing a new language-neutral protocol that can be used to facilitate language feature support for specification languages. To test the proposed protocol, a PoC is implemented that provides language support for VDM in VS Code. The PoC is based on the support found in the Overture IDE to provide an alternative to the Overture *front-end*<sup>r</sup>. This does not mean that the protocol can not be used for other specification languages. However, the protocol is not tested with other languages than VDM, which means that some parts of it might benefit from small changes which will become apparent when applying the protocol elsewhere.

Furthermore, the thesis aims to investigate if all the specification language features *can* be supported by the proposed architecture, rather than actually implementing the support. Because of this, only selected features are implemented in the pilot study.

## 1.5 Approach

To reach the goals presented in Section 1.3 an iterative process is followed as illustrated in Figure 1.1.

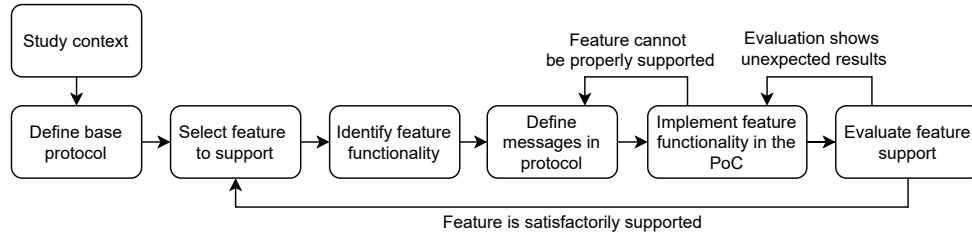


Figure 1.1: An illustration of the approach used in this thesis. It shows the main stages and main transitions between stages.

Initially the context is studied of where the proposed protocol is to be used and the pros and cons of using such an approach for decoupling the language server from the GUI. This provides the knowledge needed to identify the language features relevant for specification languages (**G1**) and enables a well founded selection of protocol design (used for **G2**). The protocol is developed one feature at a time, thus increasing the extent of the protocol in iterations (working towards **G2**). In order to verify that the protocol includes the necessary messages to support a feature, a PoC is implemented that supports the feature using the protocol (**G3**). The results from the PoC is then used to evaluate the feasibility of using the proposed protocol (**G4**).

## 1.6 Pilot Study

To evaluate the feasibility of using the proposed protocol for language support of specification languages a Proof of Concept (PoC) is implemented as part of a pilot study. As illustrated in Figure 1.2, the PoC provides IDE support for VDM in VS Code using a client-server-based architecture enabled by language-neutral protocols as opposed to the platform-based architecture of the Overture IDE. The feature functionalities implemented in the pilot study are based on the ones found in the Overture IDE. This enables a comparison between the solutions that can be used in the feasibility evaluation.

The source code for the PoC is available at:

<https://github.com/jonaskrask/vdm-vscode>.

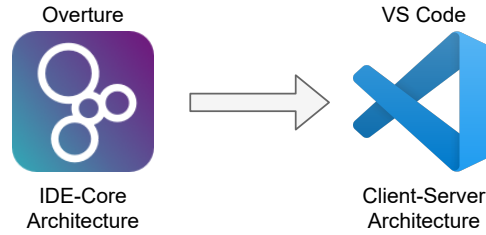


Figure 1.2: Conceptual drawing of the goal of the Proof of Concept<sup>3</sup>.

## 1.7 Publication

In parallel to writing this thesis the authors have published a paper about supporting VDM in VS Code [4]. The language support provided in the paper uses the client-server based approach with language-neutral protocols described in this thesis. This was published in *Proceedings of the 18th Overture Workshop* and presented on the 7th of December 2020.

## 1.8 Reading Guide

The conventions used in the thesis are presented in this section.

**References:** References to peer-reviewed work are referred to using a number surrounded by square brackets, such as [1]. This refers to the first referenced artefact, which can be found in the Bibliography of this thesis. References to websites are referred to using footnotes<sup>4</sup>.

**Emphasis:** Words that are emphasised are written in *italic*, such as *emphasised*.

**Acronyms:** Acronyms appear in parentheses following the full description the first time it is used, such as Vienna Development Method (VDM). A complete list of acronyms is given in the beginning of the thesis.

**Glossary:** The first occurrence of words described in the glossary, is written in *italic* followed by a superscript of  $\tau$ , such as *framework* <sup>$\tau$</sup> . A complete list of glossaries is given in the beginning of the thesis.

**Figures, Listings and Tables:** Figures, listings and tables are referred to using their name, followed by the chapter number and a period, followed by the figure number of that chapter. When a figure contains more than one element each element will be distinguished by a letter. For example the first element of the first figure in chapter 1, would be referenced as Figure 1.1a.

<sup>3</sup>Overture logo borrowed from <https://www.overturetool.org/>. VS Code logo borrowed from <https://github.com/microsoft/vscode/issues/87419>.

<sup>4</sup>Example of a footnote.

**Quotations:** Quotes appear within single quotation marks and are written in *italic*, such as:

*‘This is a quote’*

**Code Elements:** Elements of code are written in plain teletext, such as `CTFilterOption`. Blocks of code are written using listings. An example of a code block written in TypeScript is illustrated in Listing 1.1.

```
interface CTFilterOption {
  key: string,
  value: string | number | boolean
}
```

Listing 1.1: Example of a TypeScript listing

## 1.9 Structure

This section describes the structure and reading guide of this thesis, as illustrated in Figure 1.3. Each chapter is represented by a box. The solid arrows indicate the suggested order of reading. The dashed arrows indicate where additional information for the chapter can be found.

**Chapter 2:** Provides background information in relation to the content of the subsequent chapters in the thesis.

**Chapter 3:** Describes the considerations related to decoupling the language core from the IDE using language-neutral protocols, what other approaches can be used together with their pros and cons and where it is possible to make use of existing solutions.

**Chapter 4:** Introduces the proposed protocol for supporting specification language features. For each feature an investigation of the functionality required to support the feature is carried out, followed by a description of the protocol messages related to the feature. Lastly, the chapter describes a methodology for implementing language support using the protocol.

**Chapter 5:** Contains information about the pilot study. It describes the architecture of the PoC, a description of the language server used and information about each of the main elements that are implemented. Finally, it provides an evaluation of the pilot study, including a comparison to the Overture IDE with regards to functionality, efforts to implement and performance.

**Chapter 6:** Presents related work with regards to IDE support for VDM and using a protocol-based architecture for language support.

## *CHAPTER 1. INTRODUCTION*

**Chapter 7:** Concludes on the work of the thesis followed by suggestions for future work.

**Appendix A:** Contains the full specification of the proposed protocol described in Chapter 4.

**Appendix B:** Describes and exemplifies how the proposed SLSP protocol could be used to facilitate the Theorem Proving (TP) interface found in the VSCode-PVS extension<sup>5</sup> [1].

**Appendix C:** Provides an overview of the GUI for the VS Code extension developed as part of the pilot study in Chapter 5. Furthermore, it also provides examples of using the extension presented as user tasks.

**Appendix D:** Contains and describes the test traces, performance profiles and interpreter execution times used to assess performance of the VS Code extension in Section 5.3.

---

<sup>5</sup>See <https://github.com/nasa/vscode-pvs>.

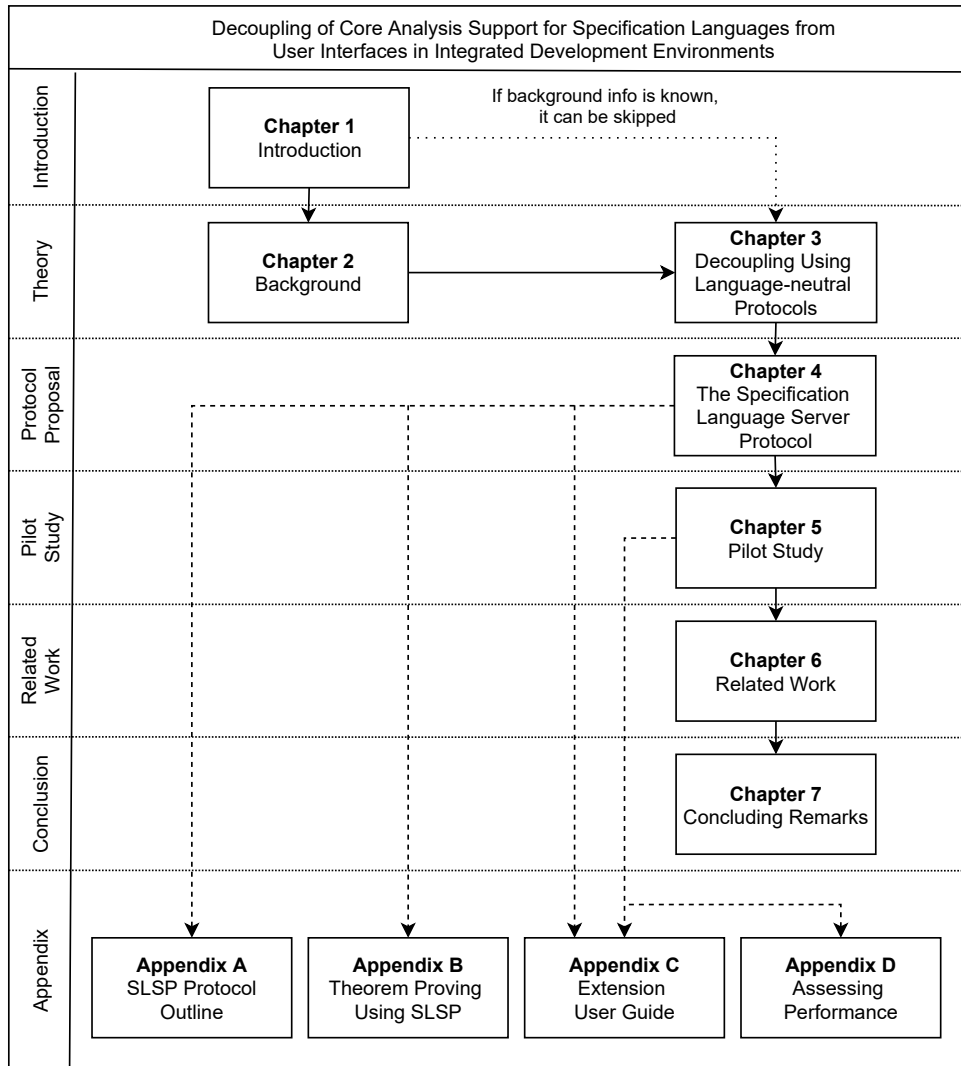


Figure 1.3: Structure of the chapters in this thesis.

## Chapter 2

# Background

This chapter provides background information in relation to the content of the subsequent chapters in the thesis. First a description of computer based languages is presented in Section 2.1 that distinguishes between specification languages and programming languages. Then a description and overview of toolsets that supports specification languages are provided in Section 2.2 together with a classification of the language features they support, while the toolsets architectures are described in Section 2.3. Lastly, a description of existing protocols that aims at decoupling IDEs from parts of common programming language support is provided in Section 2.4.

### 2.1 Computer-Based Languages

This section describes two types of computer-based languages: specification and programming languages, and compares their main similarities and differences. Depending on the purpose of the language it will have different language features which influences the kind of language feature functionality that are needed by the *development tools*<sup>T</sup> to properly support the language. By describing the purpose of the two types of computer-based languages, this section provides insight into what tool features that should be present for the languages.

#### 2.1.1 Specification Languages

A specification language is a formally defined computer-based language which is used in particular in the early phases of development in a more abstract manner compared to traditional programming languages. They are used to describe *what* a system must do, not *how* it should do it. One of the main interests of using specification languages is enabling the creation of proofs of program correctness [5].

Specification languages are used to create formal models of systems that are used to reason about the system properties [6]. Because of the rules that make up the formal languages, formal specifications can be analysed automatically and manipulated by

automated tools [6,7]. This could for example be automated validation of syntax and types, generation of proof obligations based on the specification or to automatically prove theorems derived from the specification.

Three common specification languages are VDM [8], Z [9] and B [10]. They all belong to the same category of specification languages, called model-based specification, and therefore have a lot in common: Properties are specified as invariants constraining the system at any state, with pre- and post-conditions also constraining the system operations at any state. This leads to many similarities in their tool-support.

### 2.1.2 Programming Languages

Programming languages allows instructions to be passed to a computer. There exists a wide array of programming languages with different features and intended purposes. Common for all the programming languages are that the written program is intended to be executed on a computer. For the computer to understand the instructions they must be compiled into binary form. Some programming languages are compiled before starting the execution, whereas other languages interpret the code at runtime.

There exists a lot of different programming languages with more arriving each year and the number of development tools for supporting these languages also increases. This has motivated an effort within the programming language community to support multiple languages in a single development tool and to create language support independent of the tools enabling support for the language in multiple tools.

### 2.1.3 Specification Languages vs. Programming Languages

Specification languages and programming languages are similar on some points. Both rely on a rigorous syntax that consists of rules for determining the grammatical well-formedness of sentences. Both types of languages also define semantics that provides rules for giving meaning to sentences. If the syntax or semantics of the language is not followed, the computer-based tool support will not be able to make sense of what is written. While it is not a requirement that specification languages should be executable, some of them are, and so are programming languages. However, programming languages are not centred around proving properties about the system/program as specification languages are. As a consequence of this the tool-support for specification language features like proofs is not as developed as the support for programming language features like code editing support and debugging.



## 2.2 Toolsets Supporting Specification Languages

Many different formal specification languages exist, some with different dialects and evolutions [8–12]. They are typically supported by a toolset which is a grouping of modules or components that provides the features that are useful when working with the given language. These toolsets are mostly standalone plugins for text editors supporting plugins like jEdit<sup>1</sup>, integrated with command-line tools or integrated into a modular platform such as Eclipse. In this section the major toolsets for VDM, B, Z and their dialects and evolutions will be explored.

### 2.2.1 Tool Support

The Vienna Development Method (VDM) and its dialects VDM-SL, VDM++ and VDM-RT are supported by multiple platforms including VDMTools [13] and Overture [14]. Overture is an open-source platform with an Eclipse based IDE. VDMTools is an open-source platform originally based on the IFAD closed-source VDM-SL and VDM++ Toolboxes which also offers an IDE [13]. Both support a large common feature set relating to validation, verification, translation and development.

The Event-B language is an evolution of B-Method. It is supported by the Rodin Platform [15] which offers an open-source Eclipse based IDE. It is highly extendable with a plugin architecture that allows for installing third party plugins. Examples of these are the ProB interpretation and model checking tool [16] and the Camille Editor [17]. Installing one or more plugins can therefore add support for different features related to modelling, *animation*<sup>7</sup>, visualisation, documentation, theory and proof, code generation and others<sup>2</sup>.

The Z-Notation with the dialect Object-Z is supported by Community Z Tools (CZT) [18]. In its basic configuration it is an open-source Java framework which enables coupling tools to support animation, editing, conversion between formats, type-checking and more [19]. CZT is also available in different toolset configurations either for jEdit in form of plugins or as an Eclipse based IDE<sup>3</sup>.

Alternative open-source and closed-source tools and toolsets to the platforms and frameworks exist. These support either one or more of the same features, but typically with variations in implementation or functionality. Some examples of alternative tools are:

**Z2SAL tool:** Offers model checking capabilities for the Z-Notation language<sup>4</sup>.

**BZ-TT:** A toolset for animation and test generation from both B and Z specifications [20].

<sup>1</sup>See <http://www.jedit.org/>.

<sup>2</sup>See [https://wiki.event-b.org/index.php/Rodin\\_Plug-ins](https://wiki.event-b.org/index.php/Rodin_Plug-ins).

<sup>3</sup>See <http://czt.sourceforge.net/>.

<sup>4</sup>See <http://staffwww.dcs.shef.ac.uk/people/A.Simons/z2sal/>.

**VDM-mode:** ‘*Emacs packages for writing and analysing VDM specifications using VDM-SL, VDM++ and VDM-RT*’<sup>5</sup>.

This varied tool support landscape has driven the development of the Rodin Platform, CZT and Overture, because of the need for a common open and extensible platform that supports the analytic tools needed for its respective language [14, 18, 21].

### 2.2.2 Classification and Listing of Features

To get an overview of the features supported by each toolset for its respective specification language, a classification is made based on the classification of Overture components found in [14]. Thus, features are grouped into the following four categories:

1. **Validation:** Features that supports validation of a specification, i.e., features related to testing for errors in the specification.
2. **Verification:** Features that supports verification of a specification, i.e., features related to proving correctness of the specification.
3. **Translation:** Features that supports translating a specification to other formats, e.g., executable code and *LaTeX*<sup>T</sup>.
4. **Editor:** Features that supports writing a given specification, e.g., type-checking and syntax-checking.

Table 2.1 shows the resulting classification of the features supported by Overture [14], the Rodin Platform (including what is supported by available plugins) [15] and CZT and its modules [18].

### 2.2.3 Common Features

As is evident from the different development stages of each toolset included for the feature categorisation and the inherent differences between the VDM dialects, B and Z, not all features are common across all three toolsets. The following list shows features that are common between the different toolsets within each category extracted from Table 2.1.

**Validation:** Interpreter.

**Verification:** Proof Obligation/Verification Condition Generation.

**Translation:** Print to *LaTeX*.

**Editor:** Type-checking, Syntax Highlighting.

---

<sup>5</sup>See <https://github.com/peterwvj/vdm-mode>.

## CHAPTER 2. BACKGROUND

Category	Overture IDE	Rodin Platform	Community Z Tools
Validation	Interpreter Debugging Test Automation-Support Trace Visualisation Test Coverage Animation	Interpreter Test-case Generation Test Coverage	Interpreter Animation
Verification	Proof Obligation-Generation Automatic Proof-Support	Theorem Proving Proof Obligation-Generation Model Checking	Verification Condition-Generation
Translation	UML Visualisation Code Generation VDM to Alloy Print to LaTeX	Code Generation Conversion to LaTeX UML-B to Event-B Event-B to Requirements	Conversion to LaTeX Conversion to B Conversion to Alloy Conversion to Z/EVES
Editor	Syntax-checking Type-checking Syntax Highlighting Code Completion	Syntax Error Marking Syntax Highlighting Semantic Highlighting Code Completion Quick Navigation Model Elements-Renaming	Type-checking Go-to Declaration Syntax Colouring Folding Document-Elements Content Assist

Table 2.1: Features supported in the Overture IDE, the Rodin Platform and Community Z Tools grouped into categories.

Commonalities and differences in feature support are also found when comparing the features in table 2.1 with IDE features for support of common programming languages such as C++ or Python. The front-end related features of the editor category are all common between the programming IDEs and the toolsets. However, only some features from the categories of validation, verification and translation are found in programming language support. These are from validation, in the form of test automation/generation, test coverage and debugging. Thus, verification and translation is where the toolsets differ from the programming IDEs, as these categories and their features covers functionality, such as Proof Obligation Generation and conversion to LaTeX, that are typically expected for specification languages. The toolsets support these features through specialised GUI views that are specific for the accompanying IDE or the tool itself, as a user-friendly alternative to using the command-line for interaction.

## 2.3 Toolset Architecture Overview

This section provides an overview of the architectures of the development tools: the Overture IDE, the Rodin Platform and the Community Z Tools that supports VDM, B and Z respectively as described in Section 2.2.1.

### 2.3.1 The Overture IDE

The Overture IDE is built on a platform-based architecture which is intended for reuse of common features across extensions [22]. It is comprised of two parts: the *Overture Language Core* and the *Overture Eclipse Extensions*, as shown in Figure 2.1. Where the language core is completely independent of the GUI.

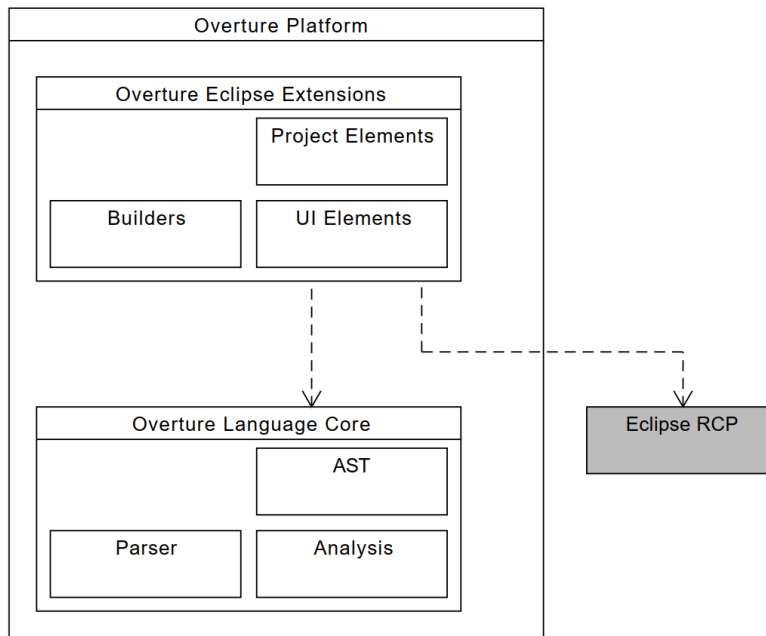


Figure 2.1: Overview of the Overture architecture. Borrowed from [22].

The language core encapsulates and handles any language and notation-related concerns, including parsing, representation and analysis [22]. In other words all the necessary components for manipulating models in the VDM family of languages [23]. All code within the core is self-contained and have no dependencies on Eclipse or other IDEs. This is done to decouple the core language from the GUI implementation. In addition, to provide the general benefits of separation of concerns, the decoupling also opens the possibility of migrating the language core to a different GUI technology or a different IDE. The core consists of an extensible Abstract Syntax Tree (AST) and a parser for constructing the AST from model sources. Any analysis of the AST such as type-checking is implemented using a visitor framework [24]. An important aspect of the language core is its extensibility mechanism

## CHAPTER 2. BACKGROUND

which allows reuse of much of the existing code.

A similar language core exists, i.e., the command-line tool VDMJ developed by Nick Battle [25].

The Overture Eclipse Extensions that are used to build the GUI components of the IDE, are created from the Eclipse Rich Client Platform (RCP). The Eclipse RCP is a powerful and generic framework for building rich client applications, but it requires significant amounts of boilerplate source code and configuration files to build the IDE [22].

According to Coleman et al. [23] it would be useful to create a way of using the core VDM libraries that is not through the Overture IDE or the command-line interface. That would not only act as a check on the development to ensure that it does not depend too heavily on the Eclipse project, but also to give another route for integration with non-Java-based environments. Allowing the core to be accessed by other development tools than the Overture IDE may provide an easier route into the VDM world for people not so familiar with Eclipse.

### 2.3.2 Rodin Platform

The architecture of the Rodin Platform has a clear decoupling between the GUI and core functionalities which is further decoupled by grouping sub-components into four main components as seen in Figure 2.2, these are:

**Event-B UI:** Comprises the Modelling UI (MUI) and Proof UI (PUI).

**Event-B Core:** Comprises the static checker (SC), Proof Obligation Generator (POG) and Proof Obligation Manager (POM).

**Event-B Library Bundles:** Comprises the Abstract Syntax Tree (AST) and Sequent Prover (SEQP) library packages.

**Rodin Platform:** Comprises the Rodin Core and Eclipse Platform.

The Rodin Platform extends on the Eclipse platform with the Rodin Core which is tightly integrated into Eclipse [15]. The Event-B Core uses functionality found in the Rodin Core and Event-B Library Bundles. It also separates static checks of elements well-formedness from proof obligation generation and proof obligation management by separating these concerns into three different sub-components (SC, POG and POM). This in turn also enables a separation of concerns in the Event-B UI between modelling and proving which are then handled by the MUI and PUI sub-component respectively.

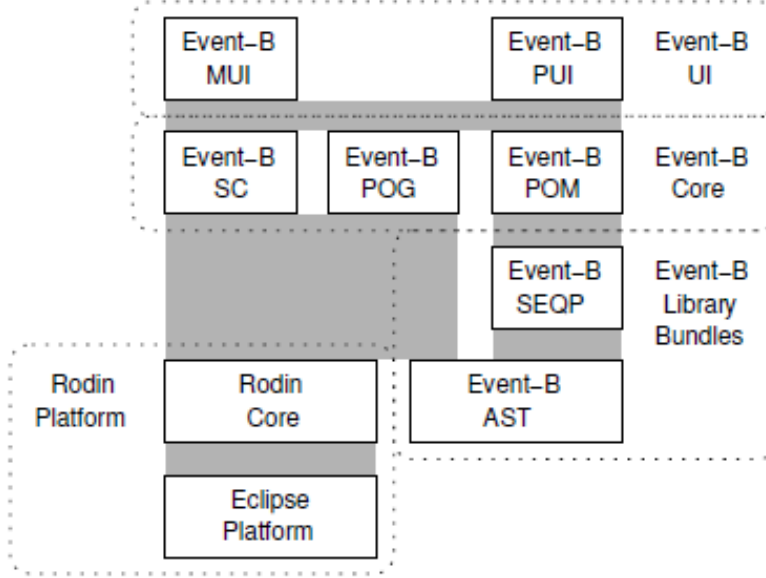


Figure 2.2: Overview of the Rodin Platform architecture. Borrowed from [15].

### 2.3.3 Community Z Tools

The architecture of the CZT<sup>6</sup> framework detailed in Figure 2.3 revolves around the Annotated Syntax Tree (AST)<sup>7</sup> structure which can provide a tree view of Z specifications parsed from various formats. AST classes are directly generated from the interchange format ZML, which is a XML markup designed specifically for Z to support CZT [18]. This allows tools to work directly with the AST representation of a specification and for easy parsing between different tools and session using ZML. To perform operations on the AST the framework makes use of the visitor design pattern [24] which provides a way to separate the structure of a set of objects from the operations performed on these objects. Thus, CZT enables the combination of functionality of tools into a combined core with all language features needed to support Z. For these features to be easier to use, CZT needs to be integrated with an IDE. Effort towards this has been carried out in CZT for Eclipse<sup>8</sup> and CZT plugins for jEdit<sup>9</sup>.

<sup>6</sup>See <http://czt.sourceforge.net>.

<sup>7</sup>Note that AST in a CZT context refers to an Annotated Syntax Tree and not an Abstract Syntax Tree.

<sup>8</sup>See <http://czt.sourceforge.net/eclipse/>.

<sup>9</sup>See <http://czt.sourceforge.net/jedit/>.

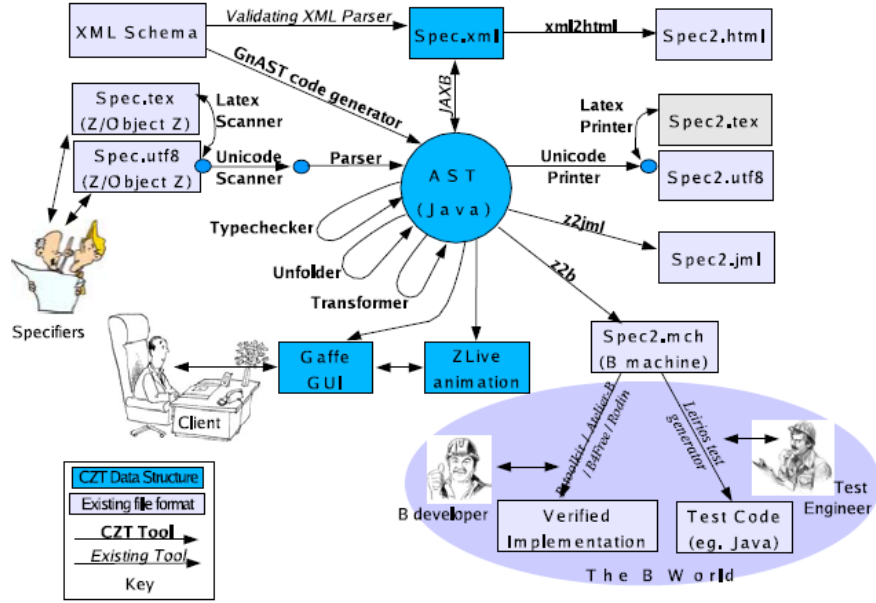


Figure 2.3: Overview of the Community Z Tools architecture. Borrowed from [18].

## 2.4 Existing Language-neutral Protocols

To support the writing of code and specifications many different language features exists, as described in Section 2.2. Supporting such features for a language requires significant effort, as the support has to be implemented for each development tool that should support the language. Decoupling the implementation of the feature support from the development tool allows extensive reuse as a single implementation of language features can be used for multiple development tools. A method to achieve this decoupling is to use a communication protocol in a client-server architecture to decouple the development tool (client) and the language feature support (server). Thus, a development tool will only have to support the protocol to implement feature support for a given language, which can be done with little effort compared to native integration [26]. Using the protocol decoupling approach reduces the  $M \times N$  problem of implementing support for all languages in all development tools to a  $M + N$  problem.

In each of the architectures described in Section 2.3 a decoupling is made of components but on different levels. While each architecture implementation is concerned with decoupling of the core language functionality components and GUI, there is no standardisation of how the core and GUI should interact. However, for programming languages efforts have been made to standardise how features that are common between languages, can be decoupled from a GUI implementation using language-neutral protocols. Two existing examples are the Language Server Protocol (LSP)<sup>10</sup>

<sup>10</sup>See <https://microsoft.github.io/language-server-protocol/>.

and Debug Adapter Protocol (DAP)<sup>11</sup>, which aims to standardise the communication between the GUI and the language service.

This section describes the LSP and DAP protocols, that have potential to be used for decoupling of parts of the specification language features. Other protocols exists that may be useful either by themselves or as inspiration for further protocol development. Examples of these are the Test Anything Protocol (TAP)<sup>12</sup> and the Language Server Index Format (LSIF)<sup>13</sup>.

### 2.4.1 The Language Server Protocol

The Language Server Protocol (LSP)<sup>14</sup> defines a language-neutral protocol to be used between a development tool (client) and a language server that provides editor language features like syntax-checking, hover information and code completion.

A language server runs as a separate process and the client communicates with the server over the stateless, light-weight Remote Procedure Call (RPC) protocol JSON-RPC<sup>15</sup>. JSON-RPC can be used within the same process, over sockets or in many various 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 [27]. Finally the LSP protocol specification assumes that one server serves one tool. Thus, an instance of the server will have to be launched for each client.

An overview of a communication sequence using the protocol can be seen in Figure 2.4, below is an explanation of the different interactions.

**The user opens a file/document in the IDE.** The IDE notifies the language server that a document is open. From now on the contents of the document has to be synchronised between the IDE and the language server.

**The user edits a file/document.** The IDE notifies the sever about the document change and updates the servers document. As this happens the server analyses the document and notifies the IDE with detected errors and warnings.

**The user issues a request on the open document.** This could for example be hovering over a function or executing a ‘Go to Definition’ on a symbol. The IDE will then issue the request with the document Uniform Resource Identifier (URI) and the text position from where the request was made. The server handles the request and responds with the requested information.

**The user closes the document.** A notification is sent from the IDE informing the language server that the document is no longer in memory.

<sup>11</sup>See <https://microsoft.github.io/debug-adapter-protocol/>.

<sup>12</sup>See <http://testanything.org/>.

<sup>13</sup>See <https://code.visualstudio.com/blogs/2019/02/19/lsif>.

<sup>14</sup>See <https://microsoft.github.io/language-server-protocol/>.

<sup>15</sup>See <https://www.jsonrpc.org/specification>.



## CHAPTER 2. BACKGROUND

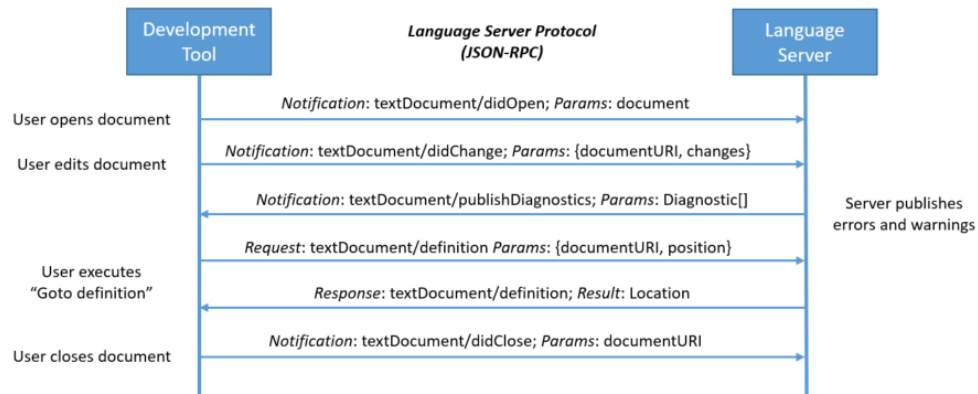


Figure 2.4: Example of language server communication sequence<sup>16</sup>.

As illustrated in the example in Listing 2.1 the protocol communicates using language-neutral data types such as document references and document positions. If the communication should be at language domain level, they would have to make a standardisation of abstract syntax trees and compiler symbols, to be able to use the protocol for multiple languages. Thus, communicating using language-neutral data types not only allows the protocol to be used for all text based languages, but also simplifies the protocol as it is much simpler to standardise a text document reference or position.

```

1 {
2   "jsonrpc": "2.0",
3   "id" : 1,
4   "method": "textDocument/definition",
5   "params": {
6     "textDocument": {
7       "uri": "file:///p%3A/mseng/VSCode/Playgrounds/
8         cpp/use.cpp"
9     },
10    "position": {
11      "line": 3,
12      "character": 12
13    }
14  }

```

Listing 2.1: Example of a 'textDocument/definition' request<sup>16</sup>.

<sup>16</sup>Borrowed from <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>.

## 2.4.2 The Debug Adapter Protocol

The Debug Adapter Protocol (DAP)<sup>17</sup> is used for decoupling IDEs, editors and other development tools from the implementation of a debugger for a specific computer based language. It is developed as a standardisation of an abstract protocol that handles communication between components supporting debugging features. These features includes different types of breakpoints, variable values, multi-process and thread support, navigation through data structures and more.

As current debuggers cannot be expected to implement the protocol, the communication architecture that the protocol is developed for, relies on an intermediary debug adapter component that wraps one or multiple debuggers. This adapter is then part of a two-way communication, utilising the protocol, with a generic debugger component integrated in the development tool where the debugger(s) must be utilised as seen in Figure 2.5. This enables debuggers, with their respective debug adapters, to be reused across development tools that implement the generic debugger component. However, debug adapters and generic debugger components may not support the same feature set and they are expected to run in separate threads or processes. Therefore, messages for handling these complications are defined in the protocol as detailed in the start sequence example in Figure 2.6.

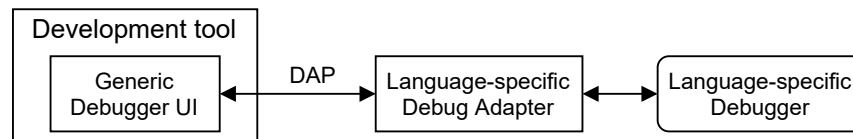


Figure 2.5: The decoupled architecture where the DAP protocol is used<sup>18</sup>.

The protocol specification defines a base message structure with three basic language-neutral message types *requests*, *responses* and *events*. A message example is illustrated in Listing 2.2. The protocol further supports construction of a wide array of generic messages by extending the basic message types. This enables the protocol to handle any messages needed for communication relating to common debug features, while being language-neutral.

<sup>17</sup>See <https://microsoft.github.io/debug-adapter-protocol/>

<sup>18</sup>Inspired by <https://microsoft.github.io/debug-adapter-protocol/overview>

## CHAPTER 2. BACKGROUND

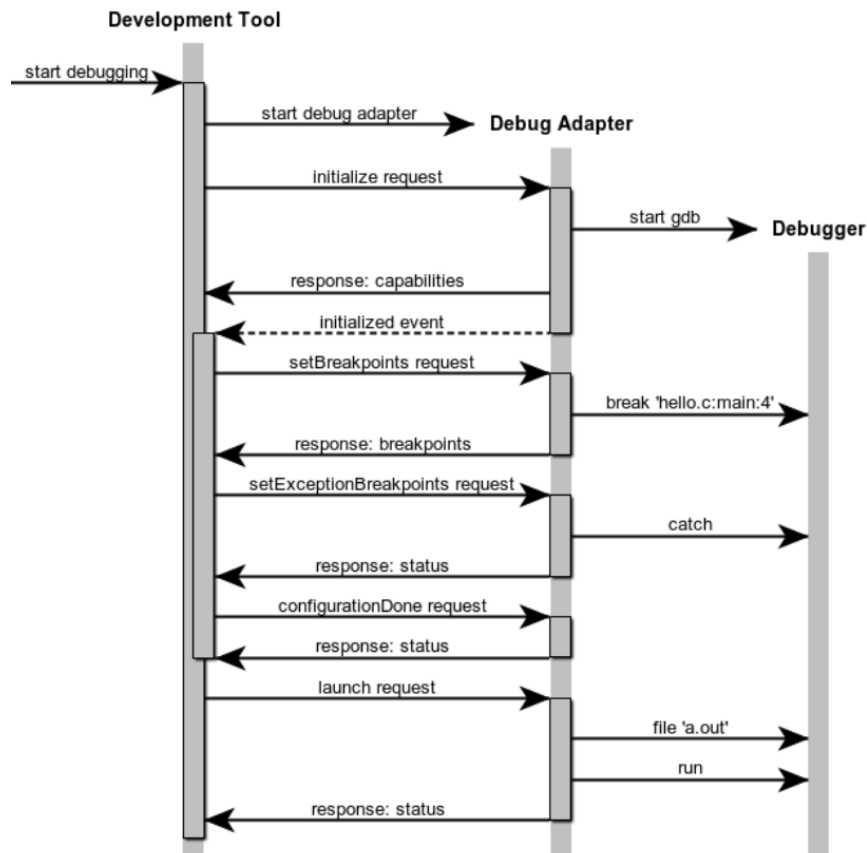


Figure 2.6: DAP protocol start sequence example<sup>19</sup>

```

1 Content-Length: 119\r\n
2 \r\n
3 {
4     "seq": 153,
5     "type": "request",
6     "command": "next",
7     "arguments": {
8         "threadId": 3
9     }
10 }

```

Listing 2.2: JSON formatted 'next' request in the DAP protocol specification<sup>19</sup>

<sup>19</sup>Borrowed from  
<https://microsoft.github.io/debug-adapter-protocol/overview>.

## Chapter 3

# Decoupling Using Language-neutral Protocols

To be able to decouple VDM language features from the Overture IDE, a decoupling approach needs to be determined. To this effect Section 3.1 describes previous IDE integrations for VDM within the Overture community, while Section 3.2 investigates which specification language features are able to be supported by existing protocols. Section 3.3 details different architectures that can be used for decoupling language features from a GUI in an IDE. Lastly, Section 3.4 presents arguments for extending the LSP protocol for the design of a new language-neutral protocol.

### 3.1 Previous IDE Integrations for VDM

Efforts have been made within the Overture community to explore different technologies and settings to integrate VDM support into multiple IDEs to transition away from the Eclipse platform. One such example is the work done by Tran and Kulik [28] where Emacs [29] packages have been developed for Overtures core language components. This has enabled the authors to integrate these components into the Emacs family of text editors. Others again have integrated the core language directly with a web IDE [30, 31]. Thus, the common approach is to leverage the already existing decoupling found in the Overture architecture to integrate the Overture Language Core seen in Figure 2.1 into a new platform such as Emacs or a web IDE. As such, there is no unification of how the integration is carried out between the different platforms. A method to unify the integration would be to adhere to a language-neutral protocol, made for decoupling of a GUI from a language core that enables language features. This is also noted by Tran and Kulik [28], quote:

*‘For Overture development to be sustainable, the next generation of the tool should not favour any particular editor or IDE, but rather support the most popular ones using standardised ways to integrate language*

*tools and IDEs.*

They further highlight the LSP protocol as a prime candidate for such a language-neutral protocol.

### 3.2 Features Supported by Standard Protocols

The protocols LSP and DAP described in Section 2.4 are developed with programming languages in mind. However, they can also be used with specification languages to handle the decoupling of some of the features categorised in Section 2.2.

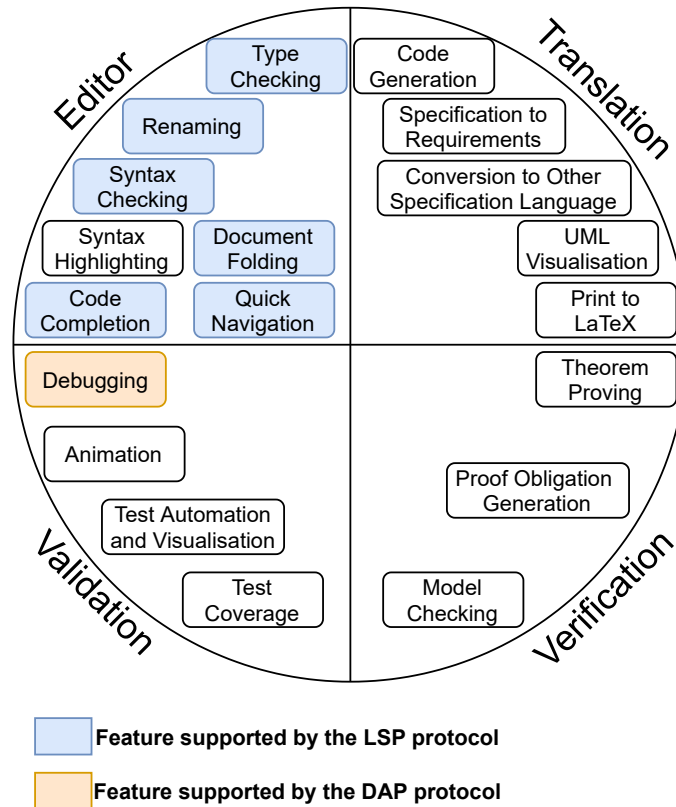


Figure 3.1: Specification language features covered by existing protocols.

For specification languages that benefits from the editor features found in Figure 3.1, a direct implementation of the LSP protocol can be used to decouple the IDE from the language core that supports this feature set. This has been explored in a basic implementation of a VS Code<sup>1</sup> extension connected to a language server for VDM using the LSP protocol. As a result of the initial implementation and by comparing the features supported by the LSP protocol with the specification language feature

<sup>1</sup>See <https://code.visualstudio.com/>.

categories, it is found that the protocol is able to support all of the editor features except syntax highlighting.

Similarly the DAP protocol can also be used for decoupling of the IDE and language core but for features of the validation category seen in Figure 3.1. 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. This was also validated by the basic implementation of the VS Code extension for VDM.

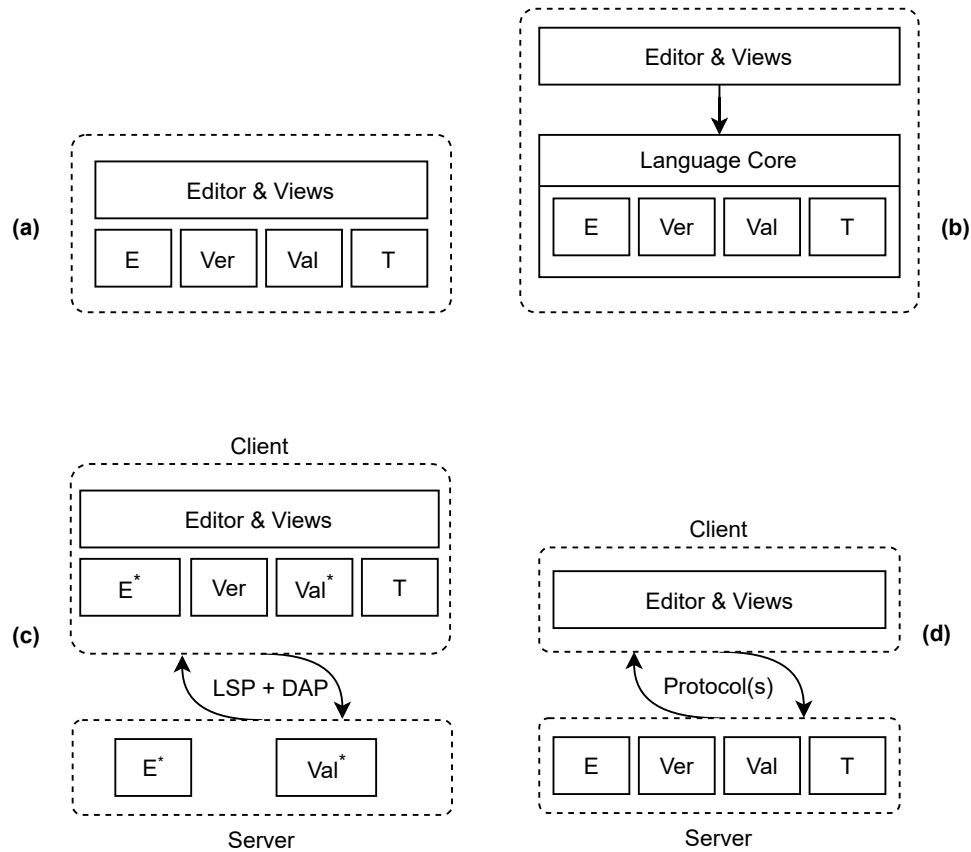
### 3.3 Decoupling Architectures

The considerations relating to using language-neutral protocols for decoupling a language core from an IDE is twofold. Firstly, is the issue of standardisation, not in terms of the protocol providing a uniform way of decoupling, but in terms of uptake and adaption as a standard. Secondly, is the design decisions relating to how the logic should be separated when decoupling using a protocol and how generic a client side can be. Figure 3.2 illustrates four different scenarios according to their level of decoupling: from a monolithic solution (Figure 3.2a) to a fully decoupled solution with a generic client (Figure 3.2d). The figure uses the same division of language features as the one used in Section 3.2.

**Monolithic structure:** This is where the development environment and the language features are tightly interconnected and specially made for a particular language, as seen in Figure 3.2a. The main benefit is that the editor and views can be tailored to the particular language, which is useful for languages with complex semantics or language features [2]. However, the maintenance and development becomes a complex task, even if it is build on an existing platform (e.g., Eclipse RCP<sup>2</sup>). This issue is intensified if the IDE is to support multiple languages and have specialised features for all the small differences in the languages. The same applies if the language is to be supported in multiple IDEs as these may not supply the same building blocks and may even be written in different programming languages.

**Partially decoupled structure:** Here a separate language core supplies all the language features independent of the GUI, as seen in Figure 3.2b. However, the GUI is specifically tailored for a particular language. This is similar to what is used in the Overture IDE, as described in Section 3.1. This solution has the same benefits as the monolithic solution, but it is easier to implement the features in a new platform. A trade-off for this solution is that all of the GUI elements are made for a specific language which makes the support of multiple languages in the same development environment a complex task since the GUI will have to be implemented anew for each language. Additionally if the language core is to be used in another platform the developer will have to find

<sup>2</sup>See [https://wiki.eclipse.org/Rich\\_Client\\_Platform](https://wiki.eclipse.org/Rich_Client_Platform).



**Legend:** E = Editor, Ver = Verification, Val = Validation, T = Translation.  
Blocks marked with '\*' contain a subset of the features.

Figure 3.2: Scenarios of a decoupled design. (a) Monolithic editor (b) editor with a one-way dependency on a specific language core (c) client editor using a server for editor features and debug with extra capabilities to support remaining features (d) generic client editor using a server containing the full language support. Inspired by [2].

ways of supporting the same features in a way suited for the new platform, as was done by Tran and Kulik [28].

**A partially decoupled structure using existing standardised protocols:** The language support is separated between two processes: a client and a server, as seen in Figure 3.2c. They communicate using the existing standardised protocols LSP and DAP. The client considered in this scenario is fully aware of many of the language features, so it can implement more specific capabilities and views at a level close to the monolithic solution. The main benefit of this solution is that the features supported by the server is interacting with the client in a standardised way. This allows a single client implementation to support multiple different languages if they adhere to the protocols. Furthermore, a given language server implementation will be reusable for multiple clients, as described in Section 2.4. This may provide users with a more specialised experience. However, not all language features are supported by existing protocols, which means that some features will have to be implemented purely at the client side. Additionally, this also has the consequence that a client cannot easily manage different languages because it would still need to implement part of the language support code at client side.

**A fully decoupled structure:** Here a language-agnostic client is connected to a language-specific server that supports all of the language features for a particular language, as seen in Figure 3.2d. Communication between the two processes is carried out using one or more protocols designed to support all the necessary language features. The client considered in this solution is generalised to such a degree that it is fully language-agnostic and able to support any language server that complies with the necessary protocols. Additionally, only one server must be developed and maintained per language. Note that providing the same GUI for all languages would promote the same user experience and reduce the learning curve when switching between languages. This may eventually increase the adoption of these kinds of tools, as indicated by Kahani et al. [32]. A disadvantage of this solution is that some language features may only be relevant for a particular language, such as trace visualisation in the Overture IDE that is not used in Rodin or CZT (see Section 2.2). Additionally, although a single client implementation is able to support multiple servers using the same implementation, there may be different preferences for how something should be displayed. This would require a very adaptable client.

An alternative solution is that the server should be able to tell the client exactly which views it needs and where they should be placed. This would result in complex servers, as they would also need a way to understand graphics, as discussed by Rodriguez et al. [2]. Furthermore, the same problem about personalised setups would require multiple server implementations or increased complexity. All of which reduces the time saved from using standardised protocols for supporting the language features.



The goal of this thesis is to provide a language-neutral protocol that would enable an implementation close to the fully decoupled structure. This protocol should support the specification language specific features, such that further decoupling between client and server can be achieved compared to only using existing protocols. However, it is believed that some features and configurations should be the responsibility of the client, e.g., if the feature needs direct access to the project files. It could also be that a given language core supports features or configurations that are only relevant for that specific core, in which case it is left to the language developer to support the features, either with their own protocol or a client-side implementation.

An example of a feature benefiting from being a client responsibility is syntax highlighting where colouring of text can be done faster if there is direct access to it, rather than having to send the text to the server and receive tokens with a slight delay.

### 3.4 Arguments for Extending the LSP Protocol

In order to create a protocol that supports specification language features it is investigated what base format such a protocol should use. The LSP protocol offers a standardised language-neutral decoupling of editor features between a client and a server. As the protocols base messages are extendable, it is possible to extend it with support for the translation, verification and validation features. However, considerations have to be made whether a protocol extension is the optimal approach for supporting such features or if they should have a standalone protocol.

Leveraging the extensibility of the LSP protocol has multiple benefits:

- It is the de facto language protocol standard for programming languages [2].
- Potential message extensions will conform to the protocol specification, promoting reuse of existing tools and frameworks [2].
- The protocol provides a message infrastructure to support synchronisation of text documents (e.g., specifications) between the client and server, as this is an integrated part of the editor features it supports.
- Support for editor features such as syntax- and type-checking, which can be useful in conjunction with non-editor features such as POG [15], are already defined in the protocol.
- Multiple IDEs already supports the LSP protocol<sup>3</sup>. This significantly reduces the complexity of implementing a client as it can take advantage of the features implemented to support the LSP protocol in such environments.

However, multiple drawbacks to this approach are also identified:

---

<sup>3</sup>See <https://microsoft.github.io/language-server-protocol/implementors/tools/>.

- Parts of the LSP protocol, i.e., text-document synchronisation, must be supported on both the client and server side to use the features of the extended protocol. This could potentially introduce additional complexity in the client and the server implementation.
- It is a less modular solution than having one or more protocols which focuses solely on supporting the decoupling of translation, verification and validation features, creating a clear separation of concerns on a protocol level.

The goal of the proposed protocol is to decouple core specification language features from a GUI. As such, the protocol should be implemented in a development tool with graphical interactions, which spans from a simple editor to a fully fledged IDE. Taking this environment into account, development tools that utilise the proposed protocol will also need the editor features found in the LSP protocol. Furthermore, the proposed protocol must support synchronisation of documents between the client and server to properly support specification language features. The synchronisation constitutes the main parts of the LSP protocol that are non-optional. Also, the LSP protocol is modular by design, which means that not all features have to be implemented to use parts of it. This points to the conclusion that the proposed protocol should be designed as an extension of the LSP protocol.

## Chapter 4

# The Specification Language Server Protocol

To support the specification language features in the categories of validation, verification and translation described in Section 2.2, the Specification Language Server Protocol (SLSP) is proposed. It builds on the design and specification of the LSP protocol by means of extension, as this is concluded, in Section 3.4, to be the optimal solution to introduce protocol support for specification language features.

The following sections describes considerations together with noteworthy details of the SLSP protocol and its design. Next, Section 4.1 highlights relevant aspects of reuse of the LSP protocol. Followed by the Sections 4.2, 4.3, 4.4 and 4.5 that describes additions to the SLSP protocol relevant for the features Proof Obligation Generation (POG), Combinatorial Testing (CT), Translation and Theorem Proving (TP) respectively. For reference, an overview of the resulting GUI views from using the SLSP protocol in the pilot study can be found in Appendix C.

### 4.1 Reuse of the LSP Protocol

The SLSP protocol takes advantage of the existing messages and message infrastructure defined in the LSP protocol<sup>1</sup>. This enables multiple levels of reuse:

**Base Protocol:** Three basic message types are defined in the LSP base protocol, namely: *request*, *response* and *notification*. These are reused in the SLSP protocol through the action of extending their existing entries with new types and/or entries, or by defining new variants as illustrated in Listing 4.2. As a result, all SLSP protocol messages conform to the definition of the base messages defined by the LSP protocol.

---

<sup>1</sup>See <https://microsoft.github.io/language-server-protocol/specifications/specification-current/>.

**Basic structures:** The LSP protocol includes a number of basic structures that can be used when defining messages. Some of these are used in the SLSP protocol to create messages that resemble the messages defined in the LSP specification. The structures also include standards for progress notifications used by the server to stream results or publish execution completion percentages. Furthermore, it is defined how to cancel requests and which errors can be used in replies. This is also reused in the SLSP protocol.

**Infrastructure:** The LSP protocol defines communication sequences, messages and replies to ensure errorless client-server communication for the different language features. An example is the ‘initialise’ interaction defined to be the first communication exchange between a client and server to match capabilities (i.e., their support for different features). The SLSP protocol is by design an extension of the LSP protocol and as such is reusing this interaction and extending the involved messages as described in later sections.

**Synchronisation:** The LSP protocol defines messages for text synchronisation between client and server, which are leveraged directly by the SLSP protocol. Therefore, as parts of this synchronisation scheme is non-optional for LSP protocol implementations, the same is the case for an implementation of the SLSP protocol. This allows the message design for features supported in the SLSP protocol to assume that files (i.e., specifications) are available to the server. In addition this synchronisation scheme is also what enables the language-neutral data types used by both protocols.

**Features:** A host of different programming language features are already supported by the LSP protocol. Thus, messages relevant for specification language features, such as syntax- and type-checking, are indirectly reused since these LSP messages are a subset of the SLSP protocol.

These different levels of reuse eliminates the need for designing a lot of the essential but basic message infrastructure together with the corresponding messages that would otherwise be needed in the SLSP protocol. Likewise, it also contributes to the support of a number of editor-related features that is relevant not only for editing specification languages, but also in conjunction with some of the more specialised specification language features.

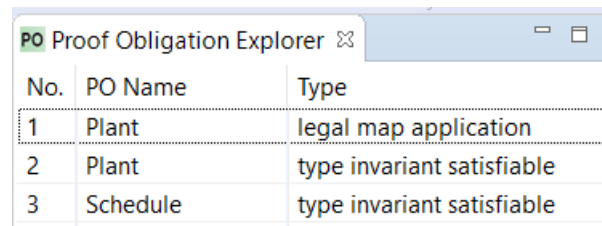
## 4.2 Support for Proof Obligation Generation

In this section it is investigated what functionality is required in order to support Proof Obligation Generation (POG) in an IDE. This is followed by a protocol outline which describes the protocol messages necessary to support POG and the rationale behind these messages.

### 4.2.1 Identifying Feature Functionality

To identify what information needs to be relayed by the SLSP protocol in order to support POG an investigation is carried out of the POG functionality found in the Overture IDE. The result of the investigation is a list of requirements for the IDE which should be satisfied to achieve the same level of POG support as the Overture IDE. The requirements are listed below with a unique identifier for each:

- R1.1** When the user right-clicks on a VDM file an option to perform POG should be displayed in the context menu.
- R1.2** When the user selects POG in the file context menu POG must be started.
- R1.3** The Proof Obligations (POs) should be presented as a list in a separate PO view, as seen in Figure 4.1.
- R1.4** The PO view should contain information about each PO, such as number, name and type.
- R1.5** When selecting a PO in the PO view the VDM that constitute the PO must be displayed.
- R1.6** The user should be able to jump to the location in the source code of the specification where a PO apply.



No.	PO Name	Type
1	Plant	legal map application
2	Plant	type invariant satisfiable
3	Schedule	type invariant satisfiable

Figure 4.1: Snippet of the Proof Obligation Explorer in the Overture IDE.

Additionally, it is found that the ‘Proof Obligation Explorer’ view is only updated when the user activates it. This may result in POs that are outdated and point to an incorrect location in the source code of the specification. A possible addition to the POG functionality could therefore be to implement synchronisation for the view, resulting in the requirement:

- R1.7** The PO view should be updated when the specification is changed.
- R1.8** If POG is not possible after changing the specification, the user should be warned that the POs may be invalid.

The IDE could also be able to support theorem proving. Functionality related to this must be able to show if a PO has been proved or not. This results in the following requirement:

**R1.9** The PO view should show if a PO has been proved.

#### 4.2.2 Protocol Outline

In this section the parts of the protocol relevant for supporting POG is described together with the reasoning for including them. For the full SLSP protocol outline, see Appendix A.

The LSP protocol uses the entry method in its messages to identify the kind of request or notification that is sent, e.g., `textDocument/didOpen`. To avoid conflicting with any current or future methods of the LSP protocol, the messages belonging to the SLSP protocol are prefixed by `slsp/`. As an example a ‘generate proof obligations’ request has the method name `slsp/POG/generate`.

In order to cover the functionality of POG, a new request and a notification is introduced:

**Generate:** Request generation of POs present in the specification.

**Updated:** Notification for synchronisation of the POs with respect to the specification.

The request and notification together with possible triggers for sending them are illustrated in Figure 4.2. Details about the request and notification are found in the sections below.

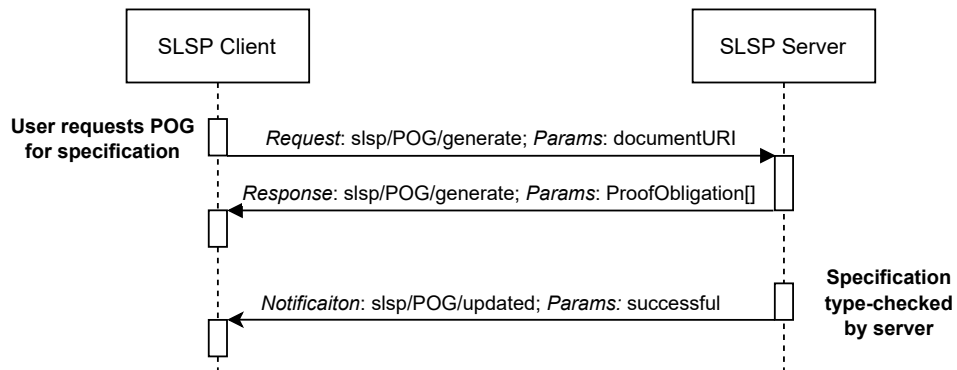


Figure 4.2: Example of how a client and a language server communicates during POG.

##### 4.2.2.1 Initialising the POG Feature

During the initialisation of the communication between the client and the server, both entities report their capabilities to the other as defined by the LSP protocol.

The features that the SLSP protocol provides support for should also be exchanged between the client and server to validate their compatibility. This is made possible by including a new capability under the `experimental` entry in the standard initialise request, as seen in Listing 4.1. A similar entry is defined for the response message of the server as: `proofObligationProvider?: boolean`. If the server supports POG, the client should make related functionality, such as requirement **R1.1** and **R1.3**, available to the user. Likewise it should be hidden if the server does not support POG.

```
Initialize Request (Client Capabilities): {
  method: "initialize"
  params: {
    ...,
    experimental:{
      proofObligationGeneration?: boolean
    }
  }
}
```

Listing 4.1: Extension to the entries of the `initialize` request, the ‘...’ indicates all the entries that are normally included in the message.

#### 4.2.2.2 Generating Proof Obligations

In Section 4.2.1 a collection of functionality that is present in the Overture IDE to support POG is identified. This section describes how this functionality is supported by the SLSP protocol.

- For the client to be able to trigger POG (**R1.2**) the message `POG/generate` is defined as illustrated in Listing 4.2. To communicate to the server which file or folder the POs should be generated for the request has a `uri` entry.
- From **R1.3**, the response to the `generate` message is defined to be an array of POs that has been generated and relevant data about each of them.
- From **R1.4**, each PO includes `id` to be able to uniquely identify them. The `name` and `kind` entries allows the PO to have a short description. The `name` is defined as a string array to allow grouping of the POs, e.g., two POs may belong to the same class, to show this they would have names similar to: `[classA, func1]` and `[classA, func2]`.
- To facilitate **R1.5**, the `source` entry contains the source code of the PO, i.e., the statement that should be proved. It is defined as a string array to allow the server to provide information about the structure of the PO. The client can then concatenate all of the strings to construct the full PO.

```

Generation request: {
  method: "slsp/POG/generate",
  params: {
    uri: string
  }
}

Response: {
  ProofObligation[]
}

interface ProofObligation {
  id: number,
  name: string[],
  kind: string,
  location: Location,
  source: string[],
  proved?: boolean,
}

```

Listing 4.2: Outline of the Generate message and the valid response. The Location type is defined in the LSP specification.

- To allow go-to functionality, as described in **R1.6** the `location` entry is used, which contains the location of where the PO applies in the source code of the specification. The `Location` type is defined in the LSP specification and includes a URI and a range of characters, defined by a line and character number, pointing to the start and end of the range.
- To facilitate **R1.9**, the optional entry `proved` is defined, allowing the server to indicate whether a PO has been proved.

#### 4.2.2.3 Synchronising Proof Obligations

To support synchronisation of the POs as described in **R1.7**, the SLSP protocol includes a notification, `slsp/POG/updated`, that the server can send to the client if changes are made that might have affected the POs. The notification is defined as illustrated in Listing 4.3.

```

Updated Notification:
{
  method: "slsp/POG/updated",
  params: {
    successful: boolean
  }
}

```

Listing 4.3: Outline of the Updated notification.



The intention of the notification is that it should be sent when the server discovers changes to the specification that might have affected the POs, e.g., changes to the names of functions.

To facilitate **R1.8** the notification includes a `successful` entry that indicates if it is possible to run POG. If it is `true` then POG should be possible, e.g., the specification is type-checked with no errors. Based on this message the client can decide to issue a generate request to update the POs. If the entry is `false` then POG is not possible after the update. This indicates that the client should not make a POG request but instead warn the user that the POs may contain information that is no longer correct.

#### 4.2.2.4 Other Protocol Considerations

A valid alternative implementation to the generate request is to split it into multiple messages, effectively distributing the work carried out by the server across the messages. As an example two messages could be specified, one to generate the PO overview containing a minimum of information about each PO and another to receive detailed information about one or more POs. If performance of the server is an issue, in regards to generating the full set of PO information needed by the client, the distribution of server work is advantageous. However, the single message approach introduces the advantage of the client having the full set of PO information available from a single message. This allows the client to respond faster to user interactions, as detailed PO information does not have to be retrieved from the server as the user navigates the GUI. Also, synchronisation is simpler with fewer messages.

### 4.3 Support for Combinatorial Testing

In this section it is investigated what functionality is required in order to provide IDE support for Combinatorial Testing (CT). This is followed by a protocol outline which describes how the functionality requirements are supported by the SLSP protocol.

#### 4.3.1 Identifying Feature Functionality

A number of functionality requirements for CT have been identified from the Overture IDE and forms the basic requirements for supporting CT in an IDE. Some of these requirements also provides the basis for what information needs to be relayed in the SLSP protocol to support CT. Each requirement is listed below with a unique identifier:

**R2.1** The user should be able to display a CT view, similar to the one illustrated in Figure 4.3.

**R2.2** The CT view should be able to show an outline of traces in the specification.

- R2.3** Results from test execution should be saved to a file.
- R2.4** Locally saved test results should be loaded when the CT view is first opened.
- R2.5** It should be possible to filter passed and inconclusive tests from the CT view.
- R2.6** It should be possible to sort executed tests by their verdict.
- R2.7** The user should be able to jump to the location of a trace in the specification from the CT view.
- R2.8** It should be possible to debug a specific test case in the interpreter.
- R2.9** The user should be able to view the results of a test case sequence.
- R2.10** Verdicts of executed tests should be visualised.
- R2.11** Larger numbers of tests should be grouped in the CT view.
- R2.12** The user should be able to execute tests using a filter.
- R2.13** The user should be able to change the settings of a filtered test execution.
- R2.14** It should be possible for the user to execute all tests in a trace.
- R2.15** The user should be able to cancel test execution.
- R2.16** Test execution progress should be visualised in the IDE.

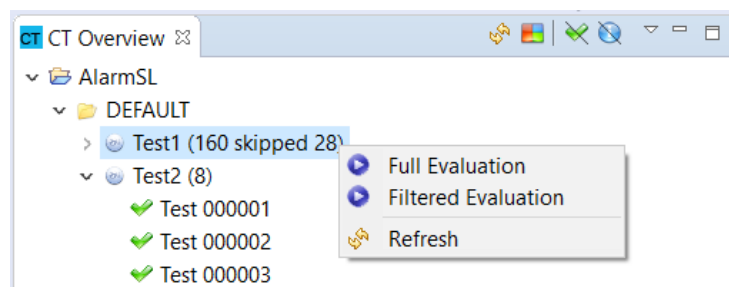


Figure 4.3: Snippet of the CT Overview in the Overture IDE.

Investigating the CT functionality available in the Overture IDE also highlighted that it is not possible to get an overview of the number of tests for a given trace before execution. An additional CT functionality could therefore be to enable test generation for a given trace before execution to provide the overview. This further enables the user to choose only a range of tests within a trace to execute which can be useful for longer running tests. This results in the following additional requirements:

- R2.17** It should be possible to generate tests without executing them.
- R2.18** The user should be able to execute a range of tests in a trace.

It is also found that the Overture IDE requires all the tests of a trace to be executed before displaying the results. The execution can take a long time for some traces and the user may want to know if some tests have already failed so that the execution may be cancelled, hence the requirement:

**R2.19** Test execution results should be available before the execution of all tests is finished.

When working with CT, large test suites can be constructed potentially resulting in millions of test cases being executed or even a combinatorial explosion as considered by Ledru et al. [33]. Thus, next to mitigation strategies such as test filtering (**R2.12**), it is relevant to make requirements about performance when executing large test sets. This leads to the following requirements:

**R2.20** Test execution time should not significantly exceed the time it takes the language core on its own to perform the same tests<sup>2</sup>.

**R2.21** The user should be able to navigate the GUI during test execution.

### 4.3.2 Protocol Outline

This section describes the parts of the protocol relevant to support the CT feature and the reasoning for including them. For the full protocol, see Appendix A.

As with supporting POG, CT support is leveraging that the SLSP protocol is an extension of the LSP protocol. Therefore, all of the new methods described in this section use the prefix `slsp/CT/` to indicate that they are used for the CT feature.

In order to cover the functionality of CT three new requests are defined:

**Traces:** Request an outline of the traces that are present in the specification.

**Generate:** Request a generation of test cases from a trace.

**Execute:** Request execution of a number of tests.

Besides these new requests, the LSP notification `$/progress` is utilised for reporting partial results from test executions and to report the progress of the executions as a percentage. The requests, notification and possible triggers for sending them are illustrated in Figure 4.4 and they are further detailed in the sections below.

#### 4.3.2.1 Initialising the Combinatorial Testing Feature

Feature capabilities should be reported by both the server and client to validate compatibility before initiating feature related communication. Therefore, the additional entry `combinatorialTest` is defined to enable the client to report its support for CT under the `experimental` capability entry of the standard `initialise` request. This is included similarly to that of the `proofObligationGeneration` capability seen in Listing 4.1. Furthermore, the entry `combinatorialTestProvider` is defined for the server to report if

<sup>2</sup>For the pilot study this means that the VS Code extension must be able to execute a trace in approximately the same time as VDMJ is able to without the protocol communication.

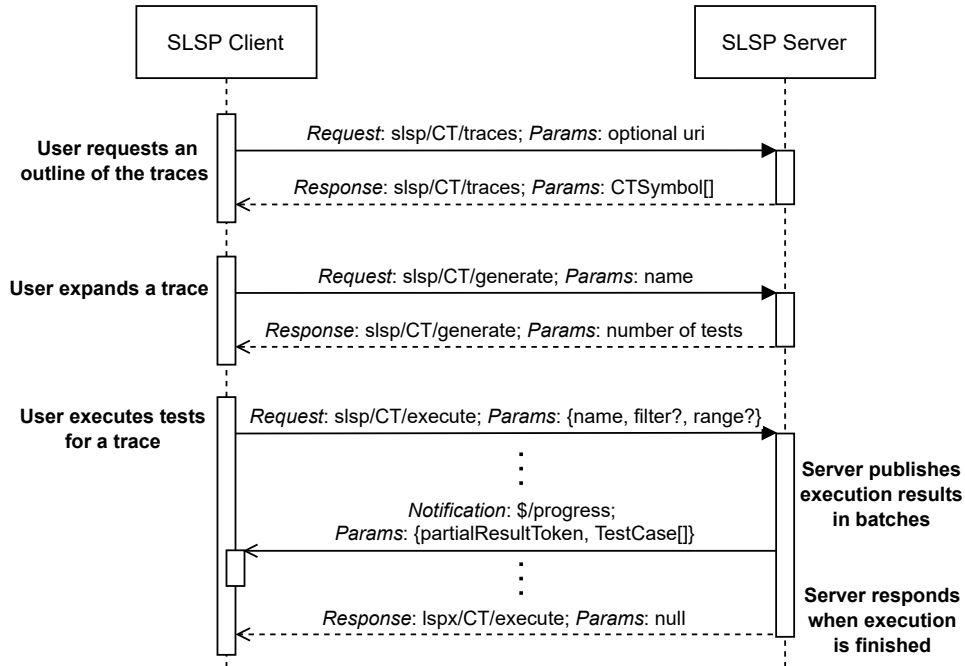


Figure 4.4: Example for how a client and a language server communicate during CT.

it is capable of supporting CT. When responding with the CT capability the server should also indicate if it supports progress reporting while working on an execute request. This results in the initialisation response detailed in Listing 4.4.

```

Response (Server Capabilities): {
  capabilities: {
    ...,
    experimental: {
      combinatorialTestProvider?: boolean | { workDoneProgress?:
        boolean }
    }
  }
}

```

Listing 4.4: Outline of the extension of the capabilities response message with the `combinatorialTestProvider` entry. The ‘...’ indicates other entries included in the response.

Using this entry, the server can communicate if it supports reporting of test execution progress which is necessary for requirement **R2.16** to be satisfied.

#### 4.3.2.2 Trace Overview

In order to be able to show a trace overview in the GUI, as stated by requirement **R2.2**, a new request message is defined with the method name `slsp/CT/traces` together with the specialised response type `CTSymbol` found in Listing 4.5. This allows the client to request traces which are returned by the server as an array of `CTSymbols`. These contains the information needed by the client to show traces in a given project grouped under a symbol name, e.g., a VDM module or class.

The `Trace` type in Listing 4.5 which defines the `traces` entry provides further information about a given trace:

**Name:** A fully qualifying name that uniquely identifies the trace. This is needed to support multiple requirements relating to test generation and test execution such as **R2.17** and **R2.14**.

**Location:** The location of the trace in the specification. This information is needed for the go-to functionality stated by requirement **R2.7**.

**Verdict:** The test execution verdict of the trace, i.e., a combined verdict for all tests of a trace. It is used for verdict visualisation as stated in requirement **R2.10**.

```
interface CTSymbol {
    name: string,
    traces: Trace[]
}

interface Trace {
    name: string,
    location: Location,
    verdict?: VerdictKind,
}
```

Listing 4.5: The `CTSymbol` type and the `Trace` type

#### 4.3.2.3 Generating Tests

When generating a test as specified by requirement **R2.17**, information about the trace that uniquely identifies it is needed. This is provided by the `Trace` type illustrated in Listing 4.5 as it has a name that is fully qualifying. In order to be able to generate tests using a trace name, a new request message is defined with the method name `slsp/CT/generate`. It uses the entry `workDoneToken` as specified by the LSP protocol, enabling the server to publish the progress of a request to generate tests. This is realised using the token value of the `workDoneToken` entry in a `$/progress` notification, with the progress being reported as a percentage and an optional message<sup>3</sup>. The response type for the generate message is simply a number

<sup>3</sup>For more information about the Work Done Progress notifications see <https://microsoft.github.io/language-server-protocol/specification#workDoneProgress>.

representing the number of tests that the server has generated. This allows the user to execute a range of tests as specified by requirement **R2.18**.

#### 4.3.2.4 Executing Tests

In Section 4.3.1 a large subset of requirements relates specifically to test execution. This section describes how this functionality is supported by the SLSP protocol.

To enable execution of tests as specified in **R2.14**, **R2.12** and other requirements a new request message is defined with the method name `slsp/CT/execute`. The response to this message is an array of `TestCase` elements, which contain information about the result of a test case. The message and its response are illustrated in Listing 4.6.

```
Execute request: {
  method: "slsp/CT/execute",
  params: {
    name: string,
    filter?: CTFilterOption[],
    range?: NumberRange,
    workDoneToken?: ProgressToken,
    partialResultToken?: ProgressToken
  }
}

Response: {
  TestCase[] | null
}
```

Listing 4.6: Outline of the `Execute` message and the response from the server

Most of the entries in the `execute` request is optional, as indicated by the question mark. The only non-optional entry is `name`, this is used to identify the trace that generates the tests that should be executed. If nothing other than `name` is specified all of the tests for the given trace will be executed and returned upon completion as required by **R2.14**.

The `filter` entry is used to provide support for filtered trace execution as specified by **R2.12**. As the execution filter is server specific, the type `CTFilterOption` is used for the entry, the content of this is found in Listing 4.7. The `key` is used to identify the filter, the `value` is the value that should be applied for the filter. If a filter option is valid or not is decided by the server, e.g., in the Overture IDE an option

```
interface CTFilterOption {
  key: string,
  value: string | number | boolean
}
```

Listing 4.7: Outline of the `Execute` message and the response from the server

is the ‘Subset Limitation (%)’ that must be an integer between 1-100. This can be specified using `CTFilterOption` as: `{key: "subset limit", value: 100}`. As multiple filters may be applied to the same execution the entry is an array of filter options.

To support execution of a range of tests as specified by **R2.18** the `range` entry is available. This is used to specify a range of tests to be executed using a start test id and an end test id.

To visualise the progress of the execution as required by **R2.16** the `execute` message uses the entry `workDoneToken` described in section Section 4.3.2.3.

The entry `partialResultToken` is a LSP specific entry. If included, it is used by the server to publish partial results instead of waiting for the execution to finish and respond with all the results at the same time, as required by **R2.19**. When using the token the results can be published in batches of `TestCases` using the `$/progress` notification.

An outline of the response type `TestCase` can be seen in Listing 4.8. The `id` entry is used to identify which test case the result belongs to. The `verdict` entry shows if a test passed, failed, was inconclusive or filtered, using the enum type `VerdictKind`, as required by **R2.10**. The `sequence` entry contains the test case execution sequence, as required by **R2.9**. This is sent as an array of the type `CTResultPair` which contains one string to describe a step in the executed sequence and another to describe the result of the step, in case the step was not executed the second string is `null`.

```
interface TestCase {
  id: number,
  verdict: VerdictKind,
  sequence: CTResultPair[]
}
```

Listing 4.8: Outline of the `TestCase` interface, used in the response to an `execute` request.

Cancellation of a test execution request (**R2.15**) is supported by utilising the LSP notification `$/cancelRequest`<sup>4</sup>. When received by the server, it should cancel the execution and respond with an error.

#### 4.3.2.5 Debugging a Test

The DAP protocol supports all of the debugging features needed to debug a specification and its implementation. Thus, it is a prime candidate to use for supporting trace debugging as stated in requirement **R2.8**. Two solutions to couple the DAP protocol with the SLSP protocol are evaluated in the following.

<sup>4</sup>See <https://microsoft.github.io/language-server-protocol/specification#cancelRequest>.

The first possible solution takes advantage of the fact that the DAP protocol has support for attaching a client to an already running debugger. Leveraging this capability it is possible to define a CT specific message in the SLSP protocol that starts the execution of a test in the interpreter and have the client attach to the process using the DAP protocol. However, this would require the server to implement the attach functionality and enable it to be activated by the client. This creates a tight coupling where a feature made available through the SLSP protocol relies on the DAP protocol and its implementation.

The other possible solution is to not directly define messages to support this functionality in the SLSP protocol, and instead have it handled purely using the DAP protocol. It requires that the interpreter has the ability to execute a test in the specification based on the SLSP trace information. As such, the client must relay the necessary trace information acquired from the SLSP protocol through the DAP protocol. Thus, the responsibility of supporting the trace debugging functionality is moved to the implementations of the client and server and how they leverage the SLSP protocol and the DAP protocol. This in turn breaks the tight coupling identified in the first solution and is why no CT debugging messages are defined in the SLSP protocol.

#### **4.3.2.6 Other Protocol Considerations**

During a CT session both the client and the server has their own local representation of the actual traces present in a specification. Thus, as the user is able to change the specification on the fly, desynchronisation can happen between the actual traces and the local representation of the traces in the server and the client. A possible solution could be to define a notification message in the protocol, enabling the server to notify the client of a change to the specification which has affected a given trace. However, to obtain the knowledge of whether a change in the specification has affected a given trace, both test generation and execution would have to be performed for the trace. Both these actions are computationally demanding, especially for test execution, which makes this an infeasible strategy. As such, the only option is to publish such a notification on any kind of specification change, which defeats its purpose and it is therefore not defined in the SLSP protocol specification. This pushes the responsibility of synchronisation to the client implementation as described in Section 5.2.3, effectively making it an active choice of the user through the action of re-generating or re-executing tests.

## **4.4 Support for Translation**

In this section it is investigated what functionality is required in order to support translation in an IDE. This is followed by a protocol outline which describes the protocol messages necessary to support translation and the rationale behind these messages.



#### 4.4.1 Identifying Feature Functionality

To identify what information is needed to be relayed by the SLSP protocol in order to support translation, an investigation is carried out of the translation functionalities found in the Overture IDE. The result of the investigation is a list of requirements for a given IDE to support the same level of translation as the Overture IDE. The requirements are listed below with a unique identifier for each:

**R3.1** The user should be able to interact with the editor to translate a given specification to LaTeX.

**R3.2** The user should be able to interact with the editor to translate a given specification to Java.

**R3.3** A translated specification should be saved to the drive.

#### 4.4.2 Protocol Outline

In this section the protocol parts to support translation is described together with the reasoning for including each of them. For the full SLSP protocol outline, see Appendix A.

The translate feature covers multiple sub-features as detailed in Figure 3.1. However, common for the group of features are the one-way conversion of a specification to another language, e.g., a programming, markup or natural language. Therefore, the SLSP protocol should not only support the translation kinds found in the Overture IDE and as a result the translation feature is agnostic of the translation format. To support translation only one new request is defined:

**Translate:** Request translation of the given specification to another language format.

The new request described in this section uses the prefix `slsp/TR/` to indicate that it belongs to the translation feature.

##### 4.4.2.1 Initialising the Translation Feature

As with other features facilitated by the SLSP protocol, support for the translation feature should be validated by the client and the server. Thus, the entry `translate` is defined to enable the client to report its support for translation in the standard initialise request. A corresponding `translateProvider` entry is defined for the server to report its support for translation in the response message. As detailed in Listing 4.9, the `translateProvider` entry defines a `languageId` as a single string or a string array if multiple languages are supported (this notation is based on the LSP specification where it is used when one or more values are possible). The id specifies one or more language formats that the server can provide translation

to. The range of valid language ids extends on the list found in the LSP specification<sup>5</sup>. Additionally, the `workDoneProgress` entry enables the server to report its support for notifying the client about translation progress.

```
Response (Server Capabilities):{
  capabilities: {
    ...,
    experimental:{
      translateProvider?: boolean | {
        languageId: string | string[],
        workDoneProgress?: boolean
      }
    }
  }
}
```

Listing 4.9: Outline of the entry `translateProvider`, used in the response to an initialise request.

#### 4.4.2.2 Translating a Specification

Translation to multiple languages should be facilitated by the protocol as is evident by requirement **R3.1** and requirement **R3.2**. Therefore, a request message that is agnostic to the language format of the translation is needed in the protocol. This request message is defined with the method name `slsp/TR/translate` as illustrated in Listing 4.10. The entries of the message are described as follows:

**uri:** A URI to the project or file that is to be translated.

**languageId:** An ID defining the language format of the translation, some of which are defined in the LSP specification<sup>6</sup>.

**saveUri:** A URI defining the location where the resulting translation is to be saved. This enables saving as defined by requirement **R3.3**.

**workDoneToken:** A token to identify the request for which the server reports its work performed progress.

Additionally, a response message is defined for the server. It specifies a URI that points to either the save location of the translated specification or a single translated file. This enables the client to show one or more translated files after a successful translation.

<sup>5</sup>See <https://microsoft.github.io/language-server-protocol/specifications/specification-current/>.

<sup>6</sup>See <https://microsoft.github.io/language-server-protocol/specification#textDocumentItem>.

```

Translate Request (client):{
  method: "slsp/TR/translate",
  params:{
    uri?: DocumentUri,
    languageId: string,
    saveUri: DocumentUri
    workDoneToken?: ProgressToken
  }
}

```

Listing 4.10: Outline of the translate request message

## 4.5 Support for Theorem Proving

Multiple different VS Code extensions exists that enables Theorem Proving (TP). Such extensions includes VSCode-PVS<sup>7</sup> [1], the Isabelle [34] VS Code extension<sup>8</sup> and the Lean [35] VS Code extension<sup>9</sup>. However, common for these extensions are a non-standardised language-specific decoupling between the GUI and the theorem prover in the server. Thus, although they might use a language-neutral protocol like the LSP protocol to offer language features such as type-checking and syntax-checking, this is not the case for the TP feature.

In this section it is investigated what functionality is required in order to support TP, i.e., Interactive Theorem Proving (ITP) and Automated Theorem Proving (ATP), in an IDE. This is followed by a protocol outline that describes the protocol parts necessary to support TP and the rationale behind them.

### 4.5.1 Identifying Feature Functionality

To identify what information is needed to be relayed by the SLSP protocol in order to support TP an investigation is carried out of the TP functionality found in VSCode-PVS and the Isabelle VS Code Extension. VSCode-PVS is illustrated in Figure 4.5 (see Appendix B.3 for a large version of the figure), and a snippet from the Isabelle extension is found in Appendix B.4. As the Lean VS Code extension resembles that of the Isabelle extension both functionally and visually, a figure of the Lean extension is not included. The result of the investigation is a list of requirements for a given IDE, which should be satisfied to provide similar support as found in the extensions. The requirements are listed below, with a unique identifier for each. Furthermore, it is noted that multiple statements can be proved including theorems, lemmas and POs, these will all be denoted as *lemmas*<sup>T</sup> for the sake of simplicity. Also, the term *theory* is used for a collection of lemmas that are related.

<sup>7</sup>See <https://github.com/nasa/vscode-pvs>.

<sup>8</sup>See <https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2020>.

<sup>9</sup>See <https://github.com/leanprover/vscode-lean>.

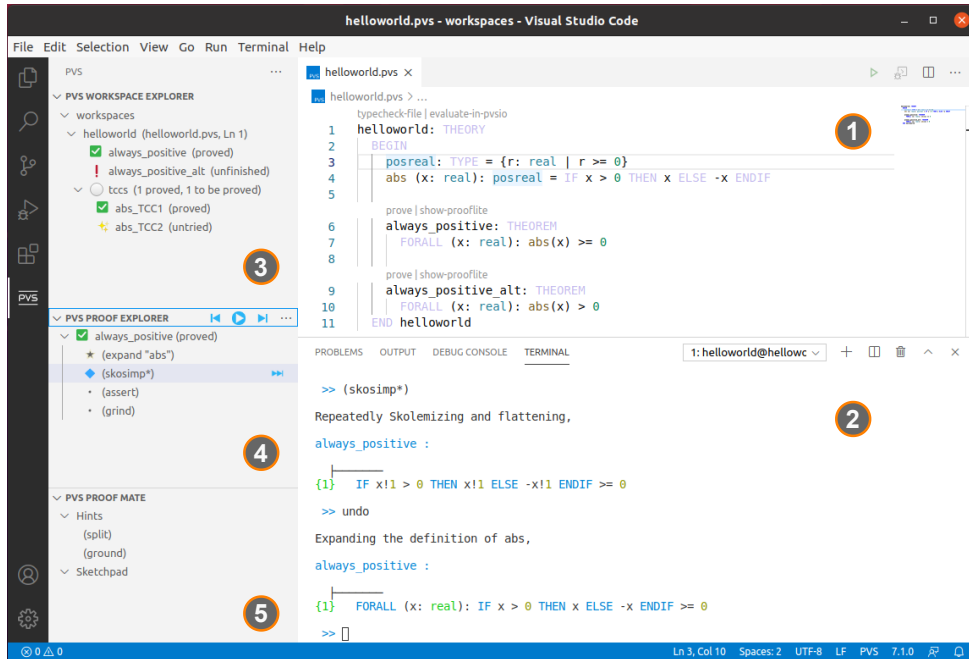


Figure 4.5: Screen shot of the VSCode-PVS extension. (1) Editor, (2) Theorem Prover terminal, (3) Theory overview, (4) Proof Explorer and (5) Proof support/suggestions. Further explanation of the figure is available in Section 4.5.3.

- R4.1** The user must be able to view a list of theories and lemmas for a given specification in a theory explorer view.
- R4.2** Theories and lemmas should be uniquely identifiable and have a proof status.
- R4.3** A list of valid TP commands should be available to the user.
- R4.4** The user should be able to view subgoals for a lemma.
- R4.5** The user should be able to view the proof steps for a lemma.
- R4.6** Proofs should be stored locally.
- R4.7** Locally stored proofs should be loaded on startup.
- R4.8** Clicking on a lemma in the theory explorer should go to the lemma definition.
- R4.9** The user should be able to re-run a proof.
- R4.10** The user should be able to cancel a TP command.
- R4.11** The user should be able to undo a TP command.
- R4.12** The user should be able to step through a proof.

**R4.13** The user should be able to postpone completion of a proof.

**R4.14** The progress of a postponed proof should be stored locally.

**R4.15** It should be possible to attempt to automatically prove a lemmas using one or more automated theorem provers.

## 4.5.2 Protocol Outline

In this section the protocol parts to support TP is described together with the reasoning for including each of them. For the full SLSP protocol, see Appendix A.

The protocol assumes a console-like communication with the prover, where the user can prove one lemma at a time by initialising a proof session for a specific lemma in the theorem prover. This interaction is similar to the one found in VSCode-PVS, which is further explained in Section 4.5.3. However, a more dynamic interaction like the one found in the Isabelle extension should also be possible as it is a matter of the client implementation.

To support TP six new requests are defined:

**Lemmas:** Request a list of all lemmas found in the specification.

**Begin Proof:** Request initialisation of a theorem prover to start a proof session.

**Prove:** Request proving of a given lemma in the specification using ATP.

**Get Commands:** Request a list of valid commands for the theorem prover.

**Command:** Request execution of a command in the theorem prover.

**Undo:** Request undoing of a command executed in the theorem prover.

A communication scenario including all six requests is shown in Figure 4.6. As detailed in the figure, the request messages use the prefix `slsp/TP/` to indicate that they belong to the TP feature.

### 4.5.2.1 Initialising the Theorem Proving Feature

The entry `theoremProving` is defined in the protocol which allows the client to report that it supports TP in the standard initialise request. For the server to acknowledge its support for TP the entry `theoremProvingProvider` is defined as detailed in Appendix A. The entry enables the server to respond with a boolean, stating whether it supports the feature.

### 4.5.2.2 Theory Overview

To be able to prove a lemma it is expected that an overview of lemmas in the corresponding specification is available as stated in requirement **R4.1**. To this effect the

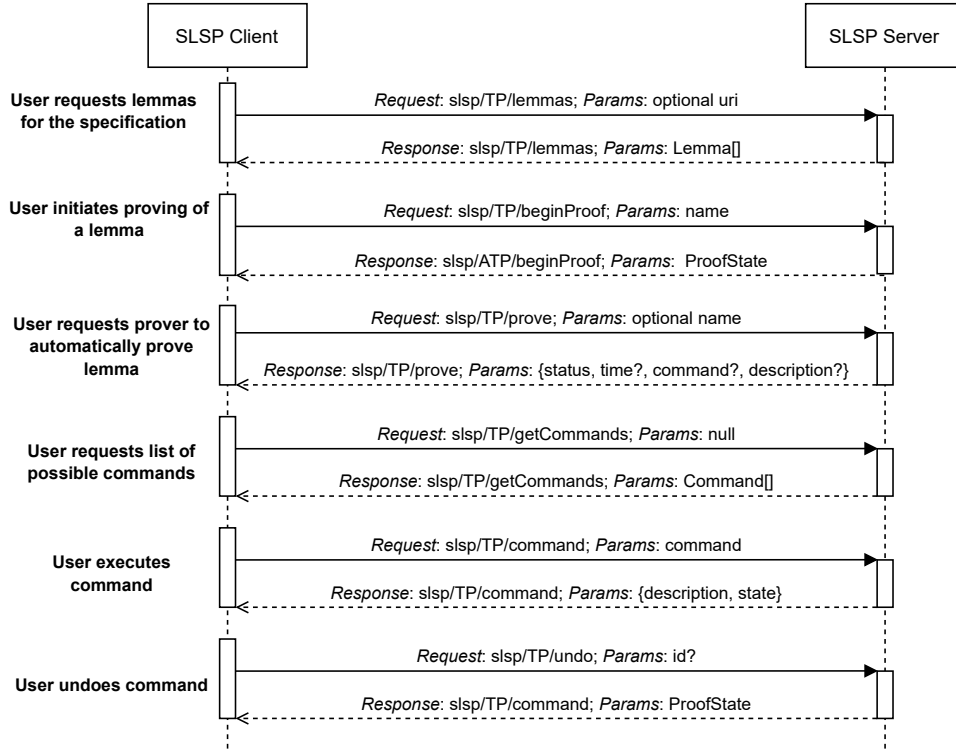


Figure 4.6: Sequence diagram showing the possible messages, their parameters and their responses.

request message `slsp/TP/lemmas` has been defined in the protocol specification, allowing the client to query the server for lemmas in a given specification. To represent a lemma in the specification, the type `Lemma` has been defined as illustrated in Listing 4.11, with entries described as follows:

**name:** A unique name for the lemma used for requirement **R4.2**.

**theory:** Name of the theory that the lemma belongs to.

**location:** Location of the lemma in the specification enabling requirement **R4.8**.

**kind:** The kind of lemma, e.g., theorem, lemma or corollary.

**status:** A proof status as stated in requirement **R4.2**. A suggestion for how to transition between the statuses is found in Appendix B.1.

#### 4.5.2.3 Proving Lemmas

To start the proving of a lemma the request `slsp/TP/beginProof` is defined. This will initiate the theorem prover for the requested lemma. The response contains

```
interface Lemma{
    name: string,
    theory: string
    location: Location,
    kind: string,
    status: ProofStatus,
}

type ProofStatus = "proved" | "disproved" | "untried" | "unfinished"
                  | "timeout" | "unchecked";
```

Listing 4.11: Outline of the type Lemma.

information about the initial state of the proof using the type `ProofState`, as illustrated in Listing 4.12. The `ProofState` type is also used for ITP to facilitate requirement **R4.5** and its entries are defined as follows:

- id:** Proof step ID, the initial response will have id 1. Subsequent steps have incremental values. The ID is used to facilitate requirements **R4.5**, **R4.11**, **R4.12**, and **R4.13**.
- status:** Status of the proof using the type `ProofStatus`, with the possibility to include a theorem prover specific status as a string instead.
- subgoals:** Lemma sub-goals left to prove after the step as stated in **R4.4**. At the beginning of the proof this will be the lemma itself.
- rules (optional):** Contains the rules used for a given step in a proof.

```
interface ProofState{
    id: number,
    status: ProofStatus | string,
    subgoals: string[],
    rules?: string[]
}
```

Listing 4.12: Outline of the type `ProofState`.

#### 4.5.2.4 Automated Theorem Proving

To facilitate the use of ATP as described in requirement **R4.15** the protocol defines the request message `slsp/TP/prove`. The message is used to ask 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 request has the entry `name` that contains the name of a lemma that ATP should be applied to.

Theorem provers will often have a specific command for ATP. By defining a message for ATP in the SLSP protocol specification a single standardised interface is defined,

pushing the responsibility of calling the specific ATP command(s) to the server. For PVS the command `grind` would be executed, while Isabelle would execute `sledgehammer`.

If the ATP process needs to be cancelled before completion the LSP cancel message is used, the same applies for cancelling other commands (**R4.10**).

The response for the `prove` request contain the following entries:

**status:** Status of the proof after ATP using the `ProofStatus` type as in Listing 4.11.

**time (optional):** Execution time of the ATP process in milliseconds.

**command (optional):** Suggestions to commands that can be applied at the current step to complete the proof.

**description (optional):** Human-readable information from the prover, e.g., a counter example or proof steps to reach proof.

#### 4.5.2.5 Interactive Theorem Proving

To facilitate ITP the user should be able to send commands to a prover at the server side. To get the list of available prover commands (**R4.3**) the request `slsp/TP/getCommands` is defined. The response contains an array of commands which are defined by a name and a description.

Prover commands are sent to the server using the request `slsp/TP/command` that has the following entry:

**command:** Command identified by a string.

The response contains a `description` of the result of the command as a human-readable string and a `state` entry of the type `ProofState` that contains information about the state of the proof after applying the command, such as proof status and subgoals. The type is described in Section 4.5.2.3.

To undo a step (**R4.11**) the request `slsp/TP/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 request includes an `id` entry that specifies the ID of the step that the theorem prover should undo.

As specifications and lemmas can change after proving them, it is relevant to be able to re-run the proof (**R4.9**). A solution to achieve this is for the client to store all commands that have been transmitted to complete the proofs, and keep them stored even if the specification changes. This allows the client to re-transmit all the stored commands and check if the proof is still completed.

To handle the requirements **R4.9**, **R4.12** and **R4.13** concerned with stepping through the 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.



### 4.5.2.6 Other Protocol Considerations

Some commands have sub-commands, e.g., in Isabelle: `Apply(simp add: something)`, where `Apply` is the command and `simp` is a sub-command. Providing information about such sub-commands is not considered in the `getCommands` request, as it resembles the auto-completion feature from the LSP protocol. Thus, it may be better supported by utilising a language server specific to the prover. Otherwise, a message could be included in the SLSP protocol that is sent while typing commands to get suggestions based on the currently typed arguments. However, this is left as future work if such functionality is desired.

### 4.5.3 Applying Theorem Proving Support

This section provides an understanding of the messages defined in the protocol for TP and how they can be applied to satisfy the requirements described in Section 4.5.1. The communication is largely based on the TP setup used in VSCode-PVS illustrated in Figure 4.7, for a larger version of the figure see Appendix B.3. Following is an explanation of the numbered views in the figure and how they can be supported by the protocol:

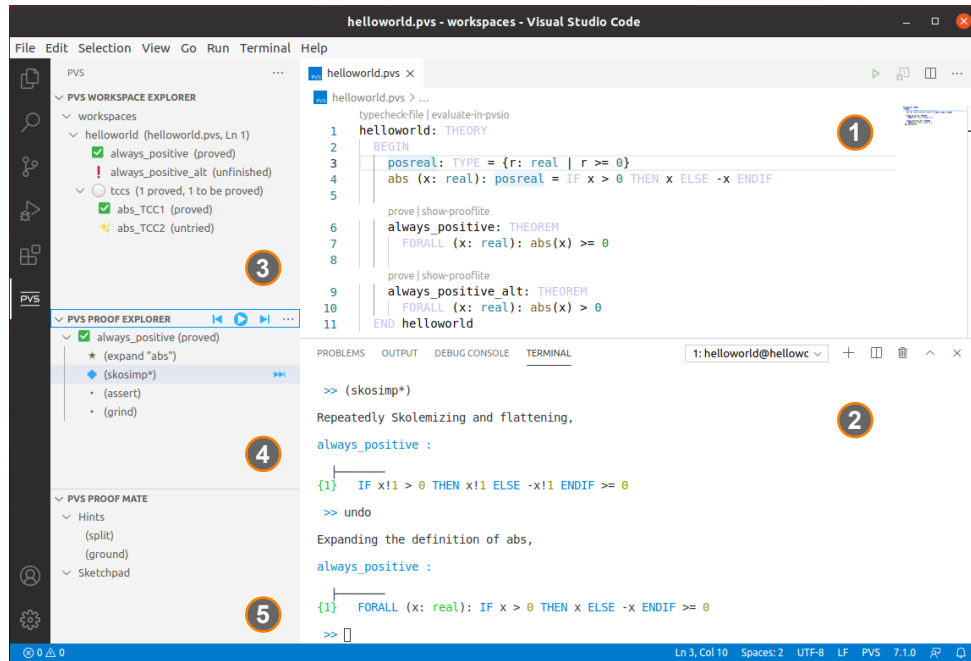


Figure 4.7: Screen shot of the VSCode-PVS extension. (1) Editor, (2) Theorem Prover terminal, (3) Theory overview, (4) Proof Explorer and (5) Proof support/suggestions.

1. Shows the specification/theory editor that contain the lemmas which must be proved and functions that produce POs. This should be supported by the functionality provided by the LSP protocol.
2. Shows the theorem prover terminal in a proof session, where the user is able to carry out the proof of a lemma using commands. Launching the terminal can be supported using the `beginProof` message. Writing proof commands can be supported using the `command` message. When performing a command the new goal is displayed in the bottom, as illustrated in the bottom of the terminal. This information is available through the response to `beginProof` and `command` which includes an entry of the type `ProofState`. The terminal can also show a list of the available commands by performing a double-tab, this behaviour can be facilitated with the `getCommands` message.
3. Shows an overview of the lemmas in the theory (TCCs are the same as POs). This view can be facilitated using the `lemmas` message. As illustrated it also displays a proof status, such as `prove`, `unfinished` and `untried`. This information is part of the `Lemma` type.
4. Shows the ‘Proof Explorer’ that contain the steps of the proof. The step marked with a star has been executed, the blue diamond is the current step and the dot steps are stored steps that were used in a previous proof session to complete the proof. This behaviour can be facilitated by storing all the commands that have been used to complete a proof on the client side.  
  
As displayed in the top of the ‘Proof Explorer’, the user is also able to step back, play a proof and step forward. The step back can be carried out using the `undo` message which causes the server to omit a step and return the previous goal, as displayed in (2). The client would however keep the step stored, but mark it as not executed. The step forward corresponds to executing the step marked as the current step (the blue diamond) using the `command` message. Play proof is possible by transmitting all the stored steps in sequence.
5. Shows the ‘Proof Mate’ which is able to show hints/suggestions for possible steps to apply for the proof. This can be facilitated using the `prove` message which returns a list of possible steps. The `status` of the `prove` response can be used to indicate if the steps can complete the proof or only progress it.

A detailed description of supporting the VSCode-PVS GUI using the TP messages from the SLSP protocol is found in Appendix B.

## 4.6 Methodology

The process of supporting a specification language in a given IDE by leveraging the SLSP protocol can be divided into four overall steps as illustrated in Figure 4.8.

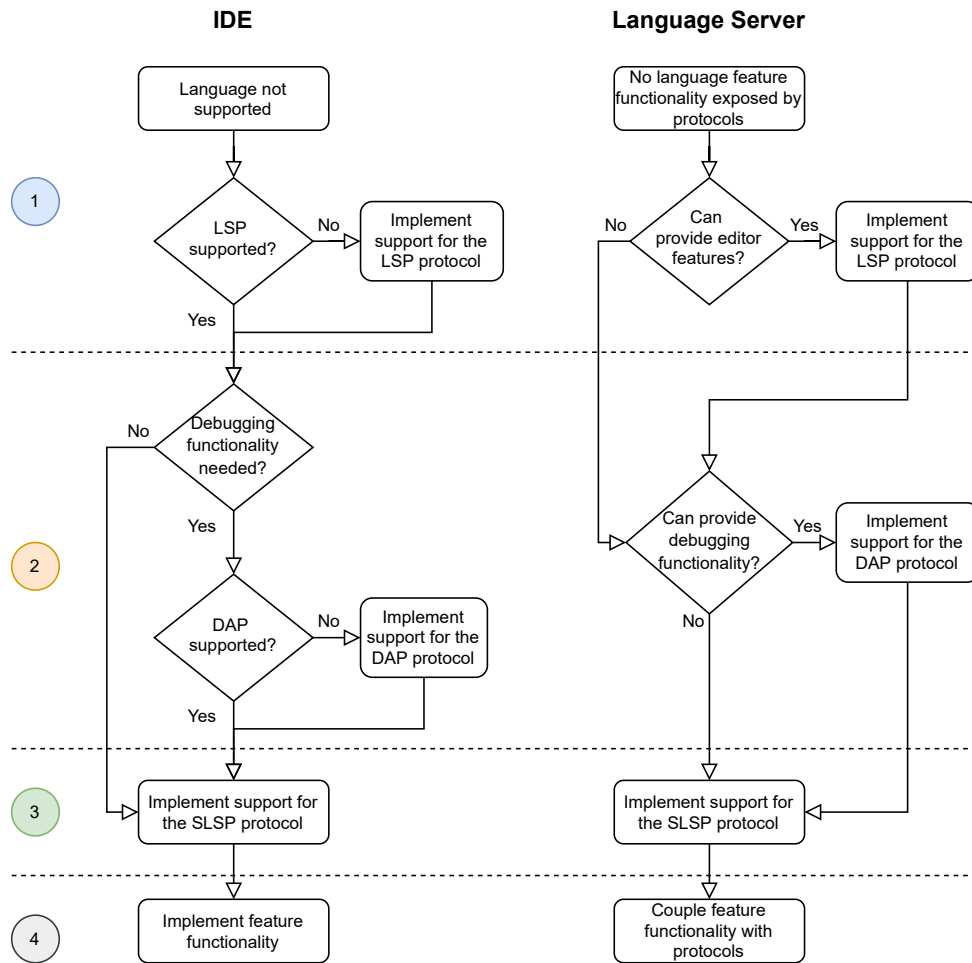


Figure 4.8: The processes of implementing language support for a specification language in a given IDE and exposing language features of a language server by leveraging the SLSP protocol.

## CHAPTER 4. THE SPECIFICATION LANGUAGE SERVER PROTOCOL

The processes depicted in the figure can be completed in parallel if no existing IDE and server support are available. Alternatively, only the process for the missing implementation must be completed. Following is a description of each of the steps involved in the process for the IDE:

1. As the SLSP protocol extends the existing LSP protocol, the IDE needs to support the core elements of the LSP protocol. That is, the message infrastructure, basic messages, types and document synchronisation mechanisms. Furthermore, if language features that are supported by the LSP protocol is needed, e.g., type-checking and syntax-checking, parts of the protocol specific to these features also needs to be implemented. Thus, it is beneficial to consider an IDE that has native support for the LSP protocol, as this can reduce the implementation effort.
2. Debugging functionality is not supported by the SLSP protocol but by the separate DAP protocol. As such, if debugging of the specification language is possible the IDE should support the DAP protocol to be able to handle debugging functionality. In addition, the CT functionality ‘debug trace in interpreter’ can be implemented using the DAP protocol. If the mentioned functionality is not relevant for the language this step can be skipped entirely.
3. Support for the SLSP protocol needs to be implemented and integrated with the existing support for the LSP protocol.
4. To leverage the feature functionality available through the implementations for the LSP, DAP and SLSP protocols, front-end logic and views specific to the IDE needs to be implemented. If support for the LSP and DAP protocols are native to the IDE the existing GUI implementation for these protocols can be leveraged.

The support for a specification language furthermore relies on a language server to provide the necessary language features. Thus, the server must support the SLSP protocol, and by dependency also the core of the LSP protocol, and possibly the DAP protocol, to be able to expose the language features it can provide to the IDE. This process is also illustrated in Figure 4.8, however the approach differs from that of the IDE process by focusing on the available language features in the server. The description of the steps involved is as follows:

1. If the server can provide editor features, support for the LSP protocol should be implemented. This includes not only the core parts of the protocol but also those specific to the editor features that the server can provide.
2. If the server can provide debugging functionality, support for the DAP protocol should be implemented. This functionality can alternatively be provided by a different server or not at all if the specification language can not be executed.

#### *CHAPTER 4. THE SPECIFICATION LANGUAGE SERVER PROTOCOL*

3. Support for the SLSP protocol must be implemented to handle communication related to the specification language features that the server can provide. If support for the LSP protocol has not been implemented in step one, the core LSP protocol, i.e., the message infrastructure, basic messages, types and document synchronisation mechanisms, must be implemented in this step.
4. To expose the specification language features that the server can provide, i.e., editor features, debugging functionality and/or specific specification language features, logic that couples the protocol messages with related functionality must be implemented.

As is evident from the process of implementing protocol support in a server, it is entirely possible to have a language server that only exposes features specific to specification languages, e.g., POG and CT, using the SLSP protocol. However, the adequacy of such a server implementation is questionable, as editor related features are highly relevant in a proper IDE.



## Chapter 5

# Pilot Study

The feasibility of supporting VDM in a different IDE than the Overture IDE using the SLSP protocol described in Chapter 3 must be evaluated. To this effect a pilot study is conducted as it can be used for a small scale implementation of a Proof of Concept (PoC), which can then be evaluated using relevant criteria.

The IDE chosen for integrating VDM support as part of the PoC is VS Code. It is chosen as it is well suited for extensibility and offers build-in support for the DAP and LSP protocols. The integration of VDM support in VS Code is detailed in Section 5.1 and an overview of the implementation is provided in Section 5.2. The evaluation of the feasibility of supporting VDM in VS Code based on the pilot study is detailed in Section 5.3.

The source code for the VS Code extension is available at: <https://github.com/jonaskrask/vdm-vscode>.

### 5.1 Supporting VDM in VS Code

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. The client implementation is tightly coupled to VS Code, however 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.

#### 5.1.1 Visual Studio Code

VS Code in its basic configuration is a free lightweight source code editor bundled with support for a few of programming languages. However, it is designed to be able to support a multitude of functionality and features through a rich ecosystem of extensions, with extended support for extension development through various Ap-

plication Programming Interfaces (APIs) exposed in the VS Code framework. VS Code employs a folder and workspace system for handling projects structures, i.e., their sub-folders and source code files. Language-specific features and functionality are delegated to extensions which keeps the source code editor and its related GUI parts language-agnostic. VS Code also offers generic support for the LSP and DAP protocols enabling developers of programming language servers and debuggers to easily integrate with VS Code.

### 5.1.2 Leveraging the Generic LSP Support in VS Code

As detailed in Section 3.4 the SLSP protocol is an extension of the LSP protocol. As a result of this, when implementing support for a specification language in an IDE, using the SLSP protocol, existing client implementations for the LSP protocol can be reused. Thus, no implementation effort is needed for editor features such as type-checking or for the document synchronisation necessary for the specification language features supported in the SLSP protocol. This is leveraged in the pilot study by extending the native LSP client in the VS Code extension to handle the SLSP protocol. Likewise, the client is extended with view logic to handle the specification language features supported by the SLSP protocol.

It should be noted that if generic support for the LSP protocol is not present in the IDE, additional implementation effort is necessary. Namely to implement the features supported by the LSP protocol, which are necessary for the SLSP protocol.

### 5.1.3 The VDM Language Extension

The extensibility model of VS Code reduces the complexity of developing support for VDM in VS Code on the client side of the SLSP protocol. Particularly for the editor features (see Figure 3.1) as VS Code has native support for the LSP protocol. The complexity of implementing specialised functionality for specification language features such as POG and CT is also reduced. The reduction arises from the VS Code APIs that can also ease the implementation of specialised functionality and graphical elements. Below is an overview of the components in the extension architecture, presented in Figure 5.1, that are specific to the SLSP protocol and the features it supports:

**Generic LSP Client:** Handles the integration of the features enabled by the LSP protocol and provides the message infrastructure to handle the protocol on the client side.

**SLSP Client:** Extends the Generic LSP Client with the message infrastructure to support the SLSP protocol.

**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.

**SLSP Server:** Implements the message infrastructure to handle the SLSP protocol on the server side, thus also supporting the LSP protocol.

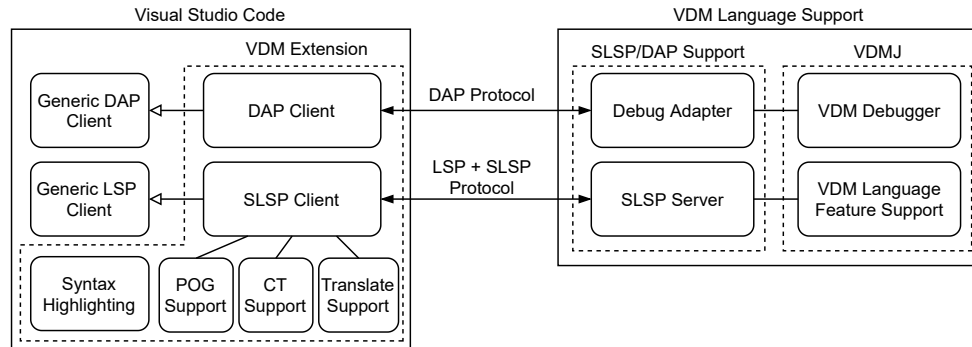


Figure 5.1: Overview of the architecture of the VDM language extension for VS Code.

The extension further adds support for debugging of a specification using the generic DAP client in VS Code and also syntax highlighting of VDM using TextMate grammars<sup>1</sup>.

The modules for supporting the SLSP features, i.e., POG, CT and translate, are implemented in an almost generic fashion. POG and translate are implemented completely generic. By providing generic support in the GUI any other language, with a language server that supports the SLSP protocol, will require no additional code than the connection to the server in order to support these features in VS Code.

It should be noted that support for the TP feature is not implemented in the VS Code extension since the language server for VDM does not support TP.

An illustration of the GUI of the extension can be found in Appendix C.1 together with a description of each of the view elements.

## 5.2 Implementing the Language Extension

This section describes the implementation efforts put into the language extension. The language server in the extension is implemented by exposing the existing language core, VDMJ, using the SLSP protocol. For the client implementation the

<sup>1</sup>See <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide>.

native support for the LSP and DAP protocols in VS Code is leveraged. However, support for the SLSP specific features are implemented from scratch. Section 5.2.1 shortly describes the implementation of the language server. Section 5.2.2, Section 5.2.3 and Section 5.2.4 describes the implementation of the features supported by the SLSP protocol.

### 5.2.1 Exposing the VDMJ Language Core via SLSP

The language server that provides the VDM language support is included in the VDMJ<sup>2</sup> project and implemented by Nick Battle. This section provides a short description of the VDMJ project and what is required to expose the language core via the protocols. This is followed by an overview of the Lines of Code (LoC) that constitutes the support.

Internally, VDMJ provides a set of language services to enable VDM specifications to be processed. By default the coordination of these services is handled by a command-line processor. Although the Command-Line Interface (CLI) is the default way to interact with VDMJ, this is not assumed in the design. An alternative means of interaction is provided using the DBGP protocol<sup>3</sup>. This allows VDMJ to interact with a more sophisticated IDE. Therefore, adapting VDMJ to use the SLSP and DAP protocols is a matter of creating a new handler process for the JSON/RPC messages defined in the respective protocol specifications.

The implementation effort to expose the VDMJ language core via the SLSP protocol only needs to be carried out once to provide support for VDM in any IDE that supports the protocol. However, this effort must not be neglected since similar server implementation efforts must be carried out for other specification languages if they are to support the SLSP protocol.

Directory	Code	Comment	Blank	Total
dap	699	450	185	1334
json	635	184	139	958
lsp	1728	653	350	2731
lsp/slsp	246	66	43	355
rpc	187	138	56	381
vdmj	2026	451	392	2869
workspace	2737	647	556	3940
<b>Sum</b>	<b>8012</b>	<b>2523</b>	<b>1678</b>	<b>12213</b>

Table 5.1: LoC measures for the files and directories used to support the protocols SLSP, LSP and DAP for VDMJ.

To get an overview of the efforts on the server side the LoC for extending VDMJ with support for the SLSP and DAP protocols is presented in Table 5.1. In the table the

<sup>2</sup>See <https://github.com/nickbattle/vdmj>.

<sup>3</sup>See <https://xdebug.org/docs/dbgp>.

‘json’ and ‘rpc’ directory contain a basic JSON/RPC system; ‘dap’ and ‘lsp’ are the protocol handlers for each; ‘lsp/slsp’ are the handlers for SLSP specific messages; ‘vdmj’ links to the VDMJ jar; and ‘workspace’ contains the code to maintain the collection of files.

## 5.2.2 Proof Obligation Generation Functionality

Some implementation effort is needed to offer at least the same level of functionality for POG in VS Code as is available in the Overture IDE, identified in Section 4.2.1. To this effect the message infrastructure to handle POG messages as specified in the SLSP protocol detailed in Section 4.2 are implemented, enabling the VDM language extension to generate POs.

To show POs generated by the server a specialised proof obligation view is implemented using the VS Code Webview API<sup>4</sup>. The webview can be fully customised using CSS, HTML and JavaScript, but it has the drawback of requiring an increased implementation effort. Alternatively the VS Code Tree View API<sup>5</sup> can be used, but the formatting of the Tree View is too restrictive which is an issue for fully displaying a PO.

The implemented ‘PO View’ combines the ‘Proof Obligation Explorer’ and the ‘Proof Obligation View’, found in the Overture IDE, into one which enables the user to expand PO meta data to show the corresponding PO as seen in Figure 5.2.

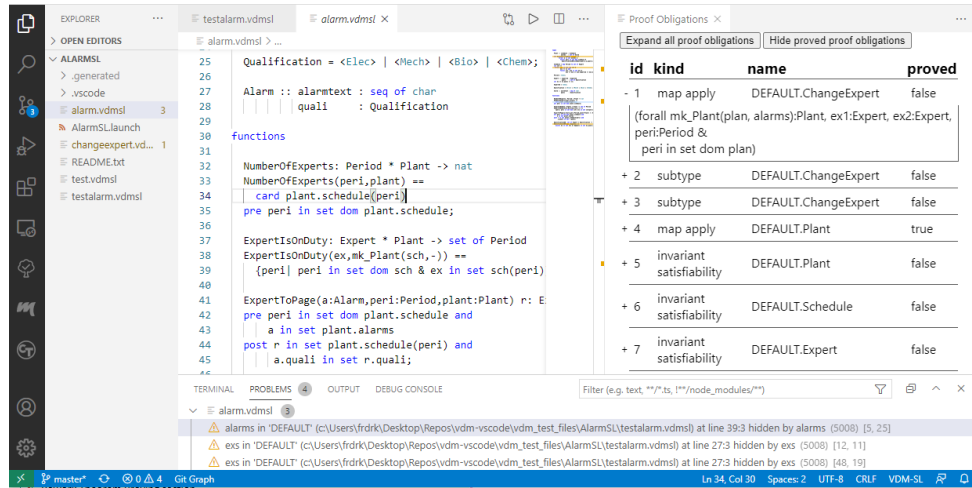


Figure 5.2: Proof obligations in VS Code.

The VDM language extension also implements additional PO functionality compared to the Overture IDE including:

<sup>4</sup>See <https://code.visualstudio.com/api/extension-guides/webview>.

<sup>5</sup>See <https://code.visualstudio.com/api/extension-guides/tree-view>.

**PO View Sorting:** Enables the user to sort POs by their meta data, i.e., ‘id’, ‘name’ or ‘kind’.

**Expand All POs:** Allows the user to either expand or collapse all POs in the view.

**Hide Proved POs:** This allows the user to hide or show proved POs.

**PO Validity:** This generates and displays new POs in the view as the specification passes a type-check or displays a warning if the specification fails the type-check.

A full overview of the POG functionality available in both the Overture IDE and the VS Code extension is found in Table 5.2. Furthermore, an example of use for the POG functionality can be found in Appendix C.2.

### 5.2.3 Combinatorial Testing Functionality

To implement the functionality for CT identified by the requirements in Section 4.3.1, substantially more effort is needed compared to implementing the POG functionality. This is evident not only from the larger number of requirements but also from the LoC presented in Table 5.4. The messages and types in the SLSP protocol, detailed in Section 4.3.2, are implemented to enable the client to query for a trace outline, to generate tests, to execute tests and to get test results. However, sending a test to the interpreter for debugging is implemented by leveraging the DAP protocol and the native DAP client. To keep the client language- and server-agnostic, the functionality for changing test execution filter options has been separated into its own class. This class implements a filter options interface. Thus, language or server specific filtering functionality can be injected into the client as needed for the specific implementation.

To implement the display of traces and tests, including handling of corresponding user interactions, the VS Code Tree View API is used. The API reduce the complexity of implementing the view compared to using the Webview API. This comes at the cost of being restricted to a simple tree view structure, as illustrated in Figure 5.3, and not having the more advanced capabilities offered by the VS Code Webview API. However, these restrictions does not hinder viewing and interacting with tests as described by the requirements in Section 4.3.1.

As discussed in Section 4.3.2.6 tests and their results have to be stored by the client side of the SLSP protocol. Whenever a test execution is stopped, either by completing execution of all tests or by cancelling the request, a snapshot of the combinatorial test’s state is saved to a file. This includes every trace, its tests and their results. However, unlike the Overture IDE this is only used for loading the state of the combinatorial tests whenever the extension is reloaded. Thus, tests and their results are stored directly in memory for a given CT session. This enables a fast and responsive CT view in the GUI, as the objects representing view elements, such as tests and their results, does not have to be created from a file before they are displayed in the view. Consequently, the extension could starve the system of memory as it needs

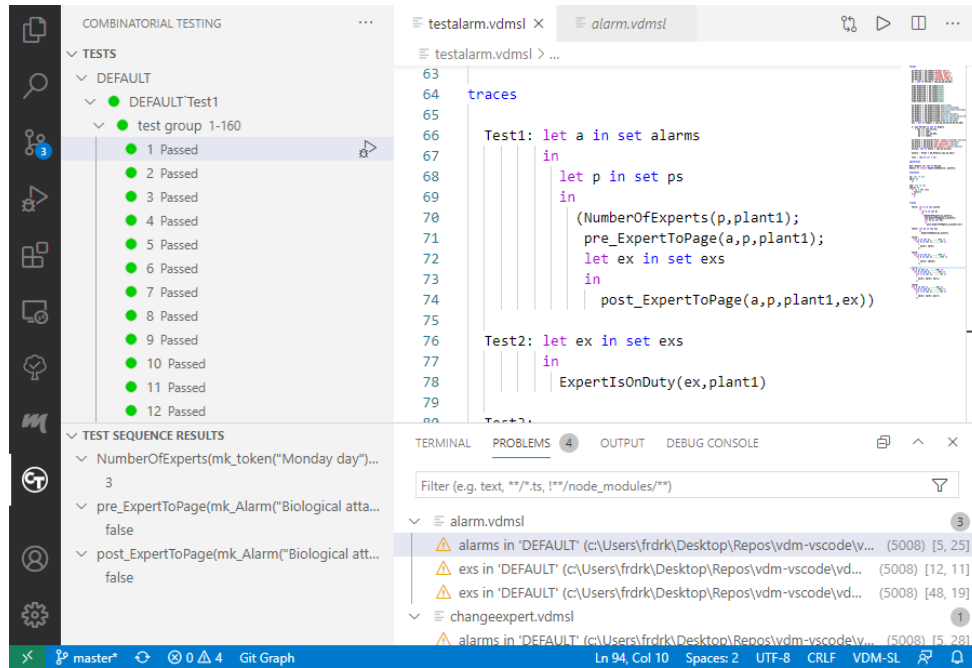


Figure 5.3: Combinatorial testing in VS Code.

to keep potentially millions of objects in memory. However, the implementation of the Tree View in VS Code is well optimised resulting in less than one gigabyte of reserved system memory after execution of a million tests.

The CT feature is utilised in a dynamic system where the user can change the specification on the fly. This action can cause the client representation of CT traces to be either knowingly or unknowingly desynchronised from the specification. Thus, possible transitions between a synchronised, a known desynchronised and unknown desynchronised client state are identified to ensure correct transitions between the states. This is needed to guarantee that the client is able to recover from a desynchronised state, as no trace synchronisation messages are specified in the SLSP protocol. The resulting states and their transitions are illustrated in Figure 5.4.

A complete overview of functionality implemented to support the CT feature can be found in Table 5.3. Furthermore, an example of use for the CT functionality can be found in Appendix C.2.

## 5.2.4 Translation Functionality

The translation functionality requires little implementation effort, as evident from the small number of requirements for the translation functionality, identified in Section 4.4. The only GUI element required for a translation is the addition of a new

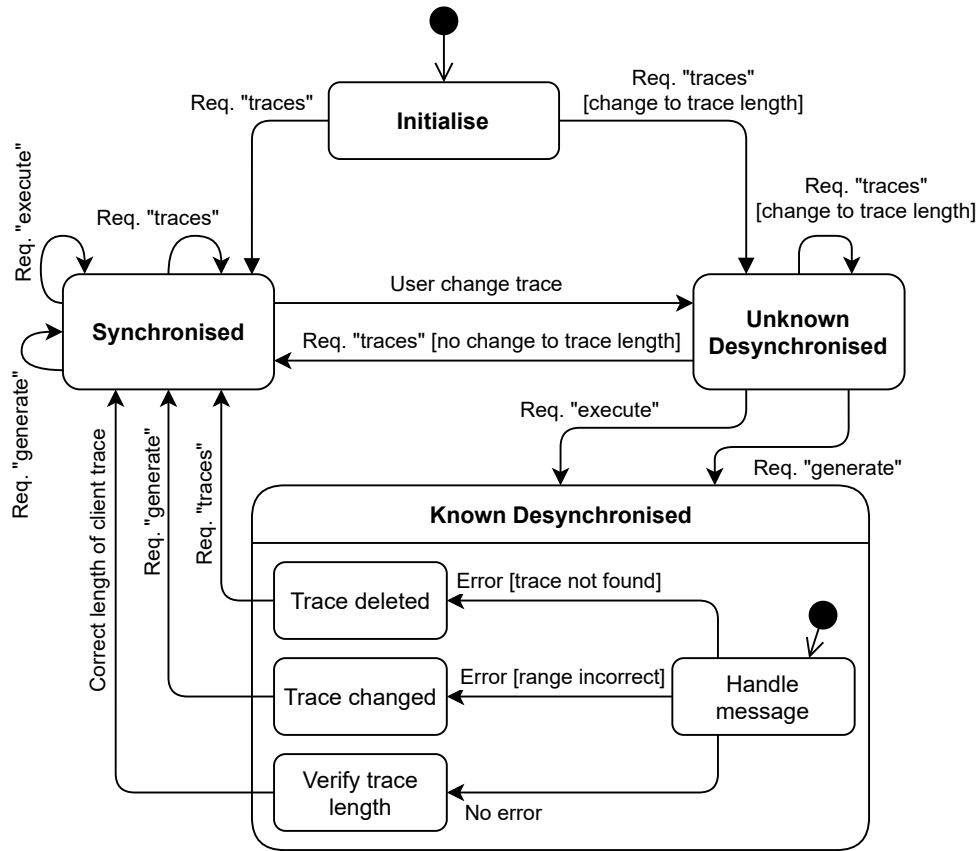


Figure 5.4: States of the system during combinatorial testing.

context menu item. As illustrated in Figure 5.5 two new items have been added, namely ‘Translate to LaTeX’ and ‘Translate to Word’. Hence, support for translation to the formats ‘.tex’ and ‘.doc’ is available in the VS Code extension. The implementation for translation is designed to be language-agnostic, also with regards to the language type/format it translates to. As such, it can be reused for any language with only a couple of additional LoC for each additional translation.

An example of use for the translation functionality can be found in Appendix C.2.

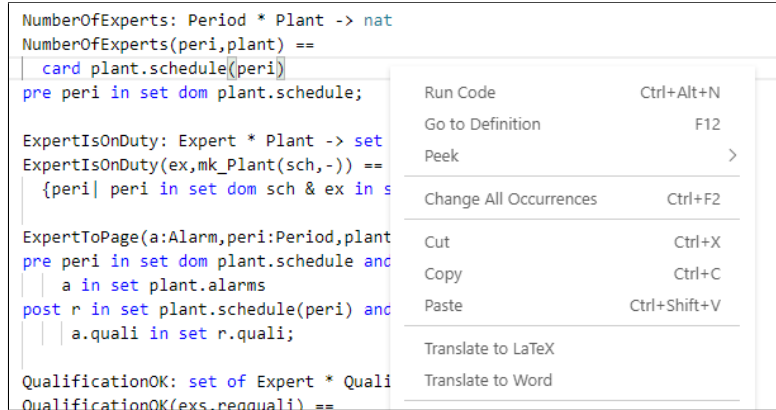


Figure 5.5: Translation options available in the VS Code extensions context menu.

## 5.3 Evaluation

The feasibility of supporting VDM in VS Code using the SLSP protocol is evaluated in terms of available functionality together with Lines of Code (LoC) of the implementation and its performance. The LoC measurement is used to quantify the implementation effort for a given VDM language feature which enables a comparison between the VS Code extension and the Overture IDE. Hence, Section 5.3.1 contains a comparison of the functionality available in the extension and the Overture IDE for POG and CT. This is followed by a comparison of the LoC for the feature implementations in Section 5.3.2. Lastly, the protocol performance impact is discussed in Section 5.3.3.

### 5.3.1 Comparing Feature Functionality

To get an overview of the functionality available for the features CT and POG in both the VS Code extension and the Overture IDE, the functionality is listed in Table 5.2 for POG and in Table 5.3 for CT. This serves as insight into the completeness of the features which is necessary to properly compare LoC.

It should be noted that the VS Code extension also supports translation to LaTeX and Word formats. However, these features only consists of simple right-click actions, which is the same as the Overture IDE.

Table 5.2 shows that the VS Code extension offers the necessary functionality extrapolated from the requirements in Section 4.2.1, such as PO Generation and a PO View. This is also true for the Overture IDE, as the requirements are based on the functionality available in this tool. However, the VS Code extension is able to provide further functionalities that are not available in the Overture IDE such as view sorting and PO validity (described in Section 5.2.2).

As presented in Table 5.3, the VS Code extension is able to offer almost the same CT

Functionality	VS Code extension	Overture IDE
PO Generation	X	X
PO Explorer view	X	X
PO View	X	X
Go-to Symbol	X	X
PO View Sorting	X	
Expand All POs	X	
Hide Proved POs	X	
PO Validity	X	

Table 5.2: PO functionality available in the VS Code extension and the Overture IDE.

functionalities as the Overture IDE, except for the ability to sort by trace verdicts. However, the extension provides the additional functionality of generating tests before execution. This enables the user to not only get an overview of the number of tests that are to be executed, but also to choose a specific range of tests to execute. Furthermore, the VS Code extension also enables the test results to be viewed as they become available.

Functionality	VS Code extension	Overture IDE
Trace Outline	X	X
Filtering Tests	X	X
Save and Load Traces	X	X
Sort by Verdict		X
Go-to Trace	X	X
Send To Interpreter	X	X
View Test Results	X	X
Visualise Verdicts	X	X
Test Grouping	X	X
Filtered Test Execution	X	X
Full Test Execution	X	X
Change Execution Filter	X	X
Cancel Test Execution	X	X
Visualisation of Progress	X	X
Generate Tests	X	
Execute Test Range	X	
Stream Execute Results	X	

Table 5.3: CT functionality available in the VS Code extension and the Overture IDE.

Comparing the functionality offered for the CT and POG feature in the two tools, it is clear that the VS Code extension is able to not only match the level of functionality for both features as found in the Overture IDE but also to surpass it.



### 5.3.2 Comparing LoC of Feature Implementations

Each specification language feature that must be implemented in a given IDE, requires at least a minimum of implementation effort. This is evident in Table 5.4, where the LoC varies between the different features in the given IDE, but also for the same feature in the different IDEs.

Feature	VS Code extension	Overture IDE
Editor	167	10,163
Syntax Highlight	815	671
Debug	54	22,865
POG	484	711
CT	915	4727
Translate to LaTeX	74	1766
Translate to Word	2	N/A
<b>Sum</b>	<b>2511</b>	<b>40,903</b>

Table 5.4: Comparison of LoC (exluding comments and blank spaces) of language feature implementations in the VS Code extension and the Overture IDE. N/A indicates that the feature is not present in the IDE.

For the VS Code extension, the 167 LoC for the ‘editor’ feature is entirely negligible considering that it effectively implements support for an entire set of features. This includes not only the ‘editor’ features specified in Table 2.1, but also a growing number of additional LSP protocol related features as highlighted by Bänder [27]. The same is true for the ‘debug’ feature, as the 54 LoC provides support for fully fledged debugging capabilities. These insignificant code amounts are only made possible due to the native support for the LSP protocol and DAP protocol available in the VS Code API.

To support syntax highlighting for VDM 815 LoC is needed but the count can vary depending on the granularity of the highlighting.

To implement support for the POG feature enabled by the SLSP protocol, only 484 LoC were necessary where about half are used for the GUI view elements. For the CT support the code count nearly doubles to 915 LoC. However, this is expected as a total of 21 CT functionality requirements have been identified in Section 4.3.1. This is in comparison to the POG functionality where only 9 requirements are identified in Section 4.2.1.

The translate to LaTeX feature functionality is implemented in the VS Code extension using only 74 LoC. The implementation for LaTeX is reused for the translation to the Word format resulting in only 2 LoC for the Word translation.

Comparing the LoC for the debug and editor feature implementations in the VS Code extension with that of the Overture IDE, the implementation effort needed to support these features is found to be substantially lower for the VS Code extension as evident in Table 5.4. However, the implementation effort of features dependent on

the SLSP protocol such as POG and CT has less of a difference in LoC. For the POG feature the difference between the VS Code extension and the Overture IDE is small whereas for the CT feature, the LoC for the Overture IDE is about 5 times higher. The translation to LaTeX is implemented using a considerably higher LoC count in the Overture IDE compared to the VS Code extension. In addition the translation feature in the VS Code extension can be reused for other translations, whereas the Overture IDE implementation is specialised for translation to LaTeX.

The only feature where the Overture IDE uses less LoC is syntax highlighting, as it only performs keyword highlighting with all keywords coloured the same. However, the VS Code extension implements a scheme where different keywords has different underlying meaning, which results in a different and more advanced colourisation.

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.

It should be noted that the LoC count for the VS Code extension found in Table 5.4 only covers the client-side implementation of the SLSP protocol, just as the count for the Overture IDE only covers the front-end implementation. 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. As detailed in Section 5.2.1 this is considerably higher than for the client at 8012 LoC for supporting the SLSP and DAP protocols in the server. However, this implementation effort only has to be made once for the server to be available across all IDEs that implement support for the protocols.

### 5.3.3 Protocol Performance Impact

The VS Code extension implements a client-server architecture with communication between the two processes using the SLSP protocol. Many of the user interactions related to VDM features have the effect that multiple messages is passed between the client and the server. Thus, a performance overhead is potentially introduced by the protocol for every action that needs to be processed by the server. To best expose any potential overhead, the action of executing CT tests is used, as it is the most communication demanding task available using the SLSP protocol in its current iteration. During CT execution, the server will send partial tests results back to the client with a given batch size.

The language server of the VS Code extension leverages VDMJ as its language core. Thus, the CLI for VDMJ is used for a performance comparison in terms of execution time of a set of test traces. To serve as a point of reference, execution times for executing the traces using the Overture IDE have also been measured. To produce comparable results, the trace executions are performed on the same machine, where the processes are allowed to use as much memory as necessary to avoid any impact from memory starvation. The time recorded is for the entire trace execution, i.e., from the user starts trace execution and until all results are available.

### 5.3.3.1 Stress Test Using Simple Tests

For stress testing, seven simple traces have been created as detailed in Appendix D.1. Each trace generates the same type of test, deliberately being as simple as possible to minimise the servers processing time of each test. This effectively increases the amount of the total execution time that is used for protocol communication, i.e., sending partial results to the client, relative to validating each test. The only difference between the traces are the number of tests that they generate as illustrated in Figure 5.6, with the raw measurement data detailed in Table D.1.

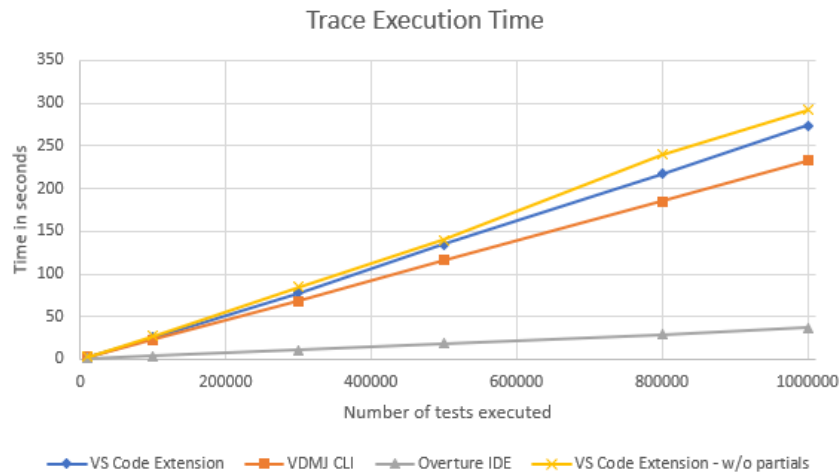


Figure 5.6: The execution times for the VS Code Extension, the Overture IDE, the VDMJ CLI and the VS Code Extension without partial results.

As illustrated in Figure 5.6 the execution time is substantially lower for the Overture IDE across all traces. This can be attributed to the traces being unrealistically simple which causes the differences between the CT implementation in the Overture IDE and VDMJ to result in a significant performance difference. As found in Table D.2 this difference is less pronounced with traces that generates realistic tests.

Figure 5.6 further shows that the VS Code extension has the highest execution time across all traces, and that the time difference to the VDMJ CLI tool increases slightly as traces generates larger numbers of tests. However, looking at the execution of traces using the VS Code extension without partial results indicates that the difference is also present without the server sending partial results to the client. In fact, not using partial results *increases* the total execution time slightly. This is likely caused by the computational efforts related to transferring and displaying all the tests at the same time instead of distributing the effort across the entire execution time.

Some of the computational overhead introduced by the VS Code extension is likely attributed to the interpreter being slightly slower when it is executed in the VS Code server and not the CLI. This is evident from the performance profiles found in appendix D. These show that the function calls to VDMJ performed by the VS Code

server uses slightly more CPU time compared to the same function calls to VDMJ when using the CLI. The same conclusion is reached by testing the interpreter outside the CT functionality, as described in Appendix D.3.

Some of the overhead for the VS Code extension is however also caused by the protocol communication. This is evident from the performance profiles where it is found that sending the test results using the SLSP protocol uses 9% of the total CPU time when executing 300k tests.

### 5.3.3.2 Test on Realistic Traces

Performance in terms of execution time is also measured using realistic traces. As opposed to the simple tests that have no specification testing value, the realistic traces are performing tests on a specification. Therefore, significantly more work must be performed by the interpreter for each test. Concretely, the traces generates tests for Luhn's algorithm specified in VDM-SL, the specification is found in Appendix D.4. The execution times for the tests is illustrated in Figure 5.7.

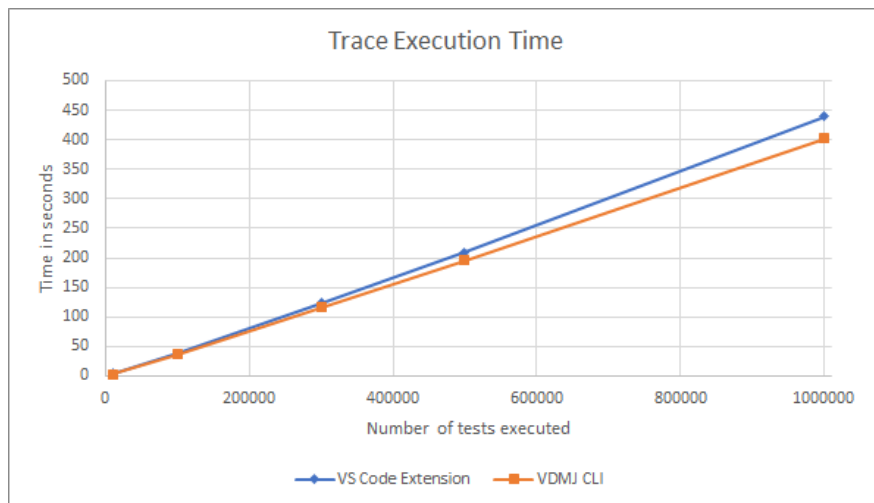


Figure 5.7: The execution times for the VS Code Extension and the VDMJ CLI. The comparison is performed with realistic traces using Luhn's algorithm as detailed in the VDM model shown in Appendix D.4.

From the trace execution times it is found that the difference between the VS Code extension and the VDMJ CLI increases almost the same for the realistic tests as for the stress tests. This shows that the difference is present even in cases where the execution of the tests requires more computations. However, it does not provide additional information on whether the difference is mostly caused by the interpreter or the protocol communication.

In order to gain further insight into the impact of the protocol communication the difference in interpreter execution time must be reduced and optimally removed.

## Chapter 6

# Related Work

This chapter presents related work with regards to IDE support for VDM and using a protocol-based architecture for language support. In Section 6.1 other work carried out to support VDM in multiple IDEs is presented. In Section 6.2 other efforts to support formal languages using a protocol-based decoupling is presented, with focus on the VS Code extension for Prototype Verification System (PVS). Section 6.3 presents efforts to provide protocol-based support for other categories of computer-based languages, which show efforts to use the LSP protocol outside its intended use and other protocols using the same architecture.

### 6.1 IDE Support for VDM

Previous work on IDEs for VDM has been focused on using the VDM language cores, i.e., the Overture core or VDMJ, to create support in different IDEs such as Emacs [28], VDMPad [30] and Overture Web IDE [31].

#### 6.1.1 Migrating Overture to a Different IDE

Tran and Kulik [28] is able to migrate the Overture language core from the Eclipse based 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 to integrate the Overture language core in another IDE. The few LoC reduces the potential advantage of a generalised solution that consists of generic IDE components which can be used for any language with little configuration (such as the LSP client in VS Code). That is, if the development of the generalised solution requires a lot of effort you might as well have several small dedicated solutions, one being the Emacs implementation.

The Emacs migration uses a total of 349 LoC, which enables almost all the features found in the Overture IDE except code generation and standard library import.

Compared to the PoC implemented in the pilot study described in Chapter 5 that uses 2511 LoC their solution uses several times less LoC. However, the components of the PoC that use the most LoC, i.e., POG and CT, are only available through a CLI in the Emacs migration.

Furthermore, the language features implemented in the PoC only use approximately 200 LoC when leveraging the native modules available through the VS Code API (i.e., LSP and DAP). The support for the SLSP features should require far less LoC to implement in VS Code for any other specification language, since they can reuse the modules created for the PoC.

### 6.1.2 Web Support for VDM

Oda et al. [30] and Reimer and Saaby [31] each presents a web IDE for VDM, based on VDMJ and the Overture language core respectively. Even though both are web based they do not utilise the same language core, hence not providing the same set of language features. Both solutions can probably benefit from unifying the efforts towards shared language support, which could be facilitated using the LSP, DAP and SLSP protocols.

VDMPad is a lightweight tool specialised for the earlier stages of the development that involves incremental and exploratory production of formal specifications. The web server is build on top of the Squeak Smalltalk system [36]. The server uses VDMJ as the *back-end*<sup>1</sup> to provide VDM language support accessed through the VDMJ CLI. The CLI is text-based which means that VDMPad has to communicate with the CLI using text-based commands and interpret the replies that are made to be human-readable. Thus, if the CLI commands change slightly the server cannot function. This communication path may benefit from using a standardised interface to VDMJ, such as the SLSP protocol, to leverage an object oriented communication that is less likely to be affected by changes to VDMJ. Using the protocol interface will also enable VDMPad to utilise all the protocol supported features with little effort. Furthermore, it will allow VDMPad to become a generalised lightweight solution for any formalism whose language support implements the protocol.

The Overture Web IDE aims to implement a fully fledged IDE for VDM using the Overture language core as the back-end. It is implemented with the goal of reusing as much software as possible. This could probably benefit from the standardised language support interface provided by the LSP, DAP and SLSP protocols to achieve an even more comprehensive software reuse, e.g., the editor could be implemented using the Monaco editor<sup>1</sup>, also used by VS Code. Thus, achieving full editor support through the LSP protocol by simply spawning the editor and connecting it to the language server.

---

<sup>1</sup>See <https://github.com/microsoft/monaco-editor>.

## 6.2 Protocol Support for Formal Languages

The interest to have protocol facilitated language support for other formal languages besides VDM is evident in tool implementations such as PVS [1], Dafny [3], Coq<sup>2</sup> [37], and Isabelle<sup>3</sup> [34]. The solutions all use 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.

One of the tools is the VS Code extension VSCode-PVS, by Masci and Muñoz [1], that provides language support for PVS. The aim of VSCode-PVS was to reduce the steep learning curve of formal technologies. This is accomplished by providing functionalities that developers expect to find in IDEs but that are not present in the standard Emacs front-end of PVS such as auto-completion, point-and-click navigation and live diagnostics.

The architecture of VSCode-PVS builds on the LSP protocol, with a decoupling between editor front-end and language server back-end. To supply features that are not part of the LSP protocol, e.g., discharging POs and theorem proving, they use the extensibility of the protocol. However their protocol extension is created specifically for PVS, hence not language-neutral. This allows their language server to be partially reused for IDEs that support the LSP protocol. But, support for the PVS-specific protocol must be specifically implemented.

Compared to the work of this thesis, the goal of the SLSP protocol is to be language-neutral, enabling uniform support for features that are common for some specification languages. This aims to increase the motivation for tool developers to support the SLSP protocol generically, thus reducing the combined effort of implementing language support for specification languages in multiple IDEs. Therefore, it could be interesting to use the SLSP protocol to enable PVS support.

Looking at the efforts of creating VSCode-PVS, they have used approximately 7k LoC (3K for the editor front-end, 4K for the language server). This is similar to the efforts for the PoC produced as part of this thesis.

## 6.3 Protocol Support for Domain-Specific and Graphical Languages

As previously described the LSP protocol is commonly used for the language support of programming-languages<sup>4</sup>, such as C++, Python, Java. The same tendency is becoming apparent for other computer-based languages, such as Domain-Specific Languages (DSLs) [26, 38] and graphical modelling [2, 38].

<sup>2</sup>See <https://github.com/siegebell/vscoq>.

<sup>3</sup>See <https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2020>.

<sup>4</sup>For an extensive list see <https://microsoft.github.io/language-server-protocol/implementors/servers/>.

Bünder and Kuchen [26] create a language server using the LSP protocol for a simple DSL and implements a client for it in three different IDEs. This shows that client support can be created in a short amount of time. Furthermore, they provide a SWOT analysis of using the LSP protocol, where they highlight the advantage of supporting multiple IDEs and find that it would be an advantage to use the LSP protocol to support other types of languages. These conclusions also applies to the use of the SLSP protocol as it builds on the same architecture and message structure as the LSP protocol.

Rodriguez-Echeverria et al. [2] propose an architecture for supporting graphical modelling languages using the LSP protocol by creating a translation layer from nodes and edges to text and vice versa. This illustrates that the protocol can be used for other computer-based languages than programming languages, and shows that language features which does not directly match the use case of the LSP protocol can still benefit from it. The same is evident from the SLSP protocol that is an extension of the LSP protocol rather than a translation layer. However, some features for specification languages may benefit from the architecture they propose, where the elements are translated to another format before they are sent to the server. This could be creating GUI elements specific to a specification language or supporting theorem proving for VDM by automatically translating to another language, e.g., Isabelle to interact with an Isabelle language server, and translate the responses back to VDM.

Walsh et al. [38], propose a generalisable approach to designing hybrid model editors as language servers using Xtext<sup>5</sup> [39], the LSP protocol and the Graphical Language Server Platform (GLSP)<sup>6</sup>. Xtext is a language workbench that can be used to create DSLs and have language support for the DSL generated by Xtext. Xtext furthermore implements the LSP protocol to support some of the generated features. GLSP is a platform that provides extensible components for the development of diagram editors including edit functionality in web-applications via a client-server protocol. It follows the architectural pattern of the LSP protocol, but applies it to graphical modelling and diagram editors for browser/cloud-based deployments. This is facilitated using the Graphical Language Server Protocol which builds on the client-server protocol defined in Sprotty<sup>7</sup>. Also considering the proposed SLSP protocol, it shows that the LSP-based language support architecture is being applied for multiple types of computer-based languages. However, their solution is based on a stand-alone protocol to implement the non-LSP features, whereas the SLSP protocol extends the LSP protocol.

<sup>5</sup>See <https://www.eclipse.org/Xtext/>.

<sup>6</sup>See <https://www.eclipse.org/glsp/>.

<sup>7</sup>See <https://github.com/eclipse/sprotty>.



## Chapter 7

# Concluding Remarks

This chapter concludes this thesis by presenting the main findings and evaluates on the achievement of the goals and hypothesis stated in Chapter 1. It also outlines possible future work and reflects on the thesis process as a whole.

### 7.1 Introduction

This thesis has explored the feasibility of using language-neutral protocols in client-server-based language support for specification languages. An investigation of the language features that are supported by existing protocols has been carried out. The investigation revealed a significant lack of support for features specific to specification languages. To bridge the gap the thesis proposes a new language-neutral protocol, the Specification Language Server Protocol (SLSP), that extends the LSP protocol to include support for POG, CT, translation and TP. Most of the protocol is realised in a PoC as part of a pilot study where support for VDM is implemented in VS Code.

The remaining part of this chapter is organised as follows. First, the current iteration of the proposed SLSP protocol is discussed in Section 7.2. Then a reflection of the thesis process is presented in Section 7.3. Next, achievement of the thesis project goals are discussed in Section 7.4. Afterwards, the thesis project as a whole and the hypothesis is concluded upon in Section 7.5. Lastly, future work for the thesis project is proposed in Section 7.6.

### 7.2 Discussion

In this section the state of the work carried out in the thesis is discussed. Followed by a discussion on the consequences of using the proposed solution which should be taken into account when choosing how to provide language support.

### 7.2.1 Features Supported by Language-Neutral Protocols

In Section 3.2 it is presented which of the specification language features can be supported by the currently standardised language-neutral protocols, LSP and DAP. Figure 7.1 illustrates the extent that specification language features can now be supported using language-neutral protocols, when including the proposed SLSP protocol. Furthermore, the figure also provides an overview of the features available in the VS Code extension. Evaluating each category of features it is found that:

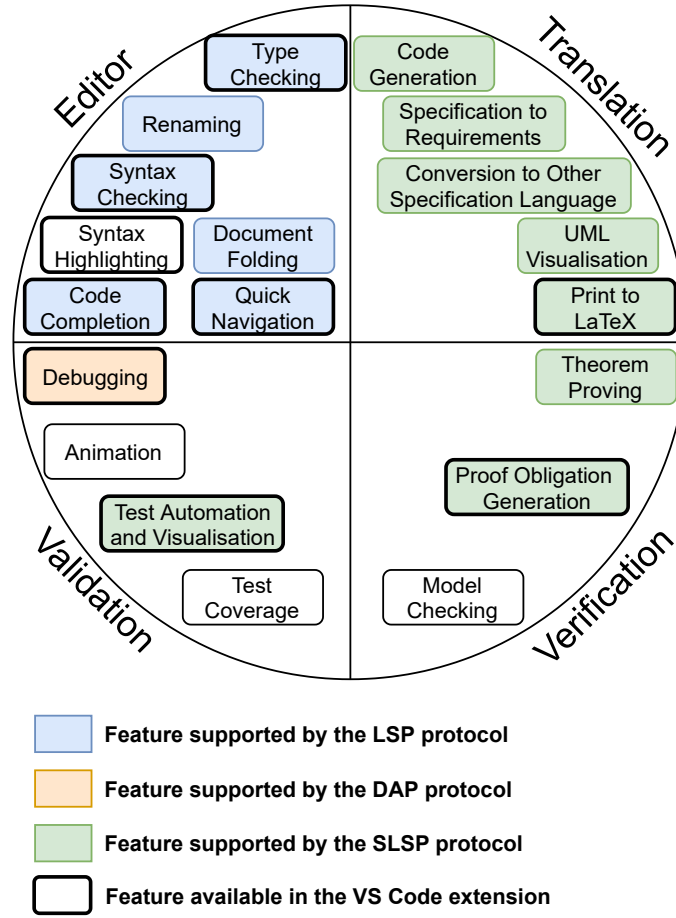


Figure 7.1: Specification language features supported by the LSP, DAP and SLSP protocols and their availability in the VS Code extension.

**Editor:** All editor features are supported by the LSP protocol, except syntax highlighting. This is demonstrated in the PoC by implementing support for syntax-checking, type-checking, quick navigation and code completion. Based on the PoC it is concluded that the other features supported by the LSP protocol can be used for support of specification languages. A suggestion to a partial protocol support of syntax highlighting is described in Section 7.6.6.

## CHAPTER 7. CONCLUDING REMARKS

**Translation:** The SLSP protocol enables basic support for translation that is applicable to any language. The support is demonstrated by implementing support for translation to formats compatible with LaTeX and Word. This provides confidence that the protocol should be able to handle any other translation. However, some features may benefit from more specialised protocol support, such as for code generation to be able to support more advanced code generation and enable code generation options.

**Verification:** The SLSP protocol supports POG, which is demonstrated in the PoC. However, the data that can be relayed using the protocol are based on VDM and the functionality found in the Overture IDE. Other languages may be able to supply additional information about POs. Because of the scope of the thesis and the time available it has not been possible to test the protocol for other languages than VDM, which would increase the confidence that the protocol is able to support all aspects of POG.

The theorem proving support in SLSP has not been implemented in the PoC as there is currently no dedicated theorem prover for VDM. This means that the feature support has not been validated, so it may not be able to fully support theorem proving.

**Validation:** The debugging feature is supported by the existing DAP protocol, which is demonstrated in the PoC. Test automation functionality is supported by the SLSP protocol which facilitates CT. This is also demonstrated in the PoC. As for POG the CT feature is only validated for VDM which means that there might be functionalities, relevant for other languages, that are not yet covered.

### 7.2.2 Consequences from the Proposed Solution for Decoupling

There are some consequences to using the proposed solution that should be considered before applying the same solution to other projects. As described in Section 5.2.1, a significant amount of effort must be carried out to implement support for the protocols in the server. That is, if a language core is available (as VDMJ was for the pilot study) the features must be exposed via the protocols. Some frameworks exist to reduce this effort, such as LSP4J<sup>1</sup>, but it is still a significant part of providing the language support. If language support must be created from the beginning, the protocol can be incorporated from the beginning which may reduce the overall effort.

For the IDEs it is also worth noting that a substantial reduction in implementation effort is only possible when native support for the protocols is available, as is the case for the LSP and DAP protocols in VS Code. Native support for the LSP protocol is

---

<sup>1</sup>See <https://github.com/eclipse/lsp4j>.

available in more than 20 IDEs<sup>2</sup>, while native support for the DAP protocol is only available in a few<sup>3</sup> where most are variations of VS Code. As such, a language that implements support in an IDE that does not have native protocol support will not be able to benefit from the reduced effort.

It is furthermore evident that the proposed solution for decoupling introduces a performance overhead compared to a tightly integrated solution. However, it has not been possible to precisely determine the significance of the performance overhead. This may affect longer running tasks such as executing traces that generates many tests.

In summary, the combined efforts to provide support for specification languages in IDEs are significantly reduced, if the proposed solution is adopted by all IDE developers and all language developers. However, some efforts must still be carried out on both sides of the protocol.

### 7.3 Thesis Project Reflections

The process of working with the master thesis has fostered different skills and insights for the authors. General research skills, such as how to approach the literature, determine its relevancy and extract only relevant information, has improved throughout the thesis process. This is as a valuable and highly relevant ability that can be used in the future when working with research articles and other scientific artefacts. Knowledge about how to structure and write content for a scientific document like a thesis has also been gained through valuable feedback from the supervisor and co-supervisor. This also includes knowledge about how to structure and write a research article as a research paper has been produced as part of the master thesis work. Writing the research paper has further allowed us to engage with the scientific community around VDM and gain knowledge about how a research paper is presented. Another valuable learning outcome has come from the frequent communication and collaboration with Nick Battle in relation to the development of the proposed protocol and insight into the development of the language server.

### 7.4 Evaluation of the Goals

Besides investigating and testing the hypothesis, this thesis project also had four goals that should be accomplished. This section revisits these goals and evaluates whether they have been accomplished or not.

---

<sup>2</sup>For the full list see: <https://microsoft.github.io/language-server-protocol/implementors/tools/>.

<sup>3</sup>See <https://microsoft.github.io/debug-adapter-protocol/implementors/tools/>.

### 7.4.1 Revisiting the Goals

The goals are presented in Chapter 1 and are shown again below:

- G1:** Explore what language features are relevant for specification languages and to which extend they can be supported by existing standardised protocols.
- G2:** Propose a language-neutral protocol for supporting specification language specific features.
- G3:** Develop a PoC for supporting VDM in VS Code using a client-server-based architecture.
- G4:** Evaluate the feasibility of using the proposed protocol for decoupling a language-agnostic client from a language-specific server.

### 7.4.2 Evaluation

In this section the goals are evaluated by explaining to what extent each goal is considered to be achieved, and which parts of the thesis that relate to the goal.

- G1:** This goal is *achieved*. The relevant features for specification languages are described in Chapter 2, where the features are divided into four categories. Through an investigation of IDEs that support the specification languages VDM, Z and B, common features are determined. Furthermore, the existing protocols are presented. This knowledge is combined in Section 3.2 where an overview of the features supported by the existing protocols is presented.
- G2:** This goal is *partially achieved*. In Chapter 4 the SLSP protocol is presented. The protocol is able to support the specification language specific features: Proof Obligation Generation, Combinatorial Testing, Translation and Theorem Proving. However, some features are not supported by the protocol in its current iteration, such as model checking and animation. For each of the supported features an investigation of IDE functionality has been carried out resulting in a list of feature functionality requirements. The requirements are used to aid the development of the protocol and argue that the protocol is able to provide support for a given feature.  
  
Although the protocol is language-neutral, the development of it has been focused on providing support for VDM and it has only been used with VDM. Testing the protocol for other specification languages may show that the protocol needs changes to be usable by all specification languages.
- G3:** This goal is *partially achieved*. A PoC has been developed that provides language support for VDM-SL, VDM++ and VDM-RT. The front-end is implemented in VS Code and acts as a client with regards to the protocols. This is connected to a language server that provides the language support using the

VDMJ language core. The PoC includes support for several editor features, debugging, Proof Obligation Generation, Combinatorial Testing and translation to LaTeX and Word. However, it is only considered partially achieved as the theorem proving support is not tested since the language server for VDM does not yet support carrying out proofs. Thus, the theorem proving part of the SLSP protocol has not been properly tested.

**G4:** This goal is *partially achieved*. The feasibility of using language-neutral protocols for decoupling the IDE from the language core is investigated on a conceptual basis in Chapter 3. In Chapter 4 it is argued how the protocol supports the language features. This is tested in the pilot study described in Chapter 5. This also includes an evaluation in Section 5.3 that shows that the effort needed to support VDM using the SLSP protocol is significantly below that of the Overture IDE. However, it also shows that the proposed solution introduces some computational overhead, which causes a reduction in performance for some long running tasks.

The reason for considering the goal partially achieved is that ideally such an evaluation should be carried out by new and independent users. We did inform students of the modelling of mission critical systems course about an early prototype, but unfortunately we have not been able to receive systematic feedback from the few new users of the PoC.

## 7.5 Conclusion

In this thesis it has been investigated if language features for specification languages can be supported using a decoupling between a GUI and a language service provider in an IDE. This is based on the hypothesis of the thesis, found in Section 1.3, that states:

*It is possible to decouple the language support for specification languages from the user interface using a client-server architecture, with a **language-neutral** protocol between a **language-agnostic** client and a language-specific server.*

In the thesis it is found that the decoupling of language support can be carried out at different levels, going from an architecture where the IDE and the language support is tightly coupled to a fully decoupled architecture where the client is completely language-agnostic. The thesis works towards the fully decoupled architecture, by proposing a language-neutral protocol, the Specification Language Server Protocol (SLSP). The protocol enables support for the specification language features Proof Obligation Generation (POG), Combinatorial Testing (CT), Translation and Theorem Proving (TP).

To evaluate the usability of the SLSP protocol, a pilot study has been carried out. In the pilot study a PoC is implemented that provides language support for VDM in

## CHAPTER 7. CONCLUDING REMARKS

VS Code using the newly defined protocol and the standardised protocols LSP and DAP. The PoC shows that IDE support for language features relevant to VDM can be implemented with less effort, than the native support found in the Overture IDE, by leveraging a client-server architecture, the proposed SLSP protocol and the LSP and DAP protocols. Furthermore, the PoC also shows that limited effort is needed to support language features supported by the language-neutral protocols if the IDE has native support for these. This overall points to a great advantage in efforts needed to implement specification language support in a new IDE when using language-neutral protocols in a client-server architecture. Comparing the performance of the PoC with that of the native CLI for VDMJ showed that there is a small performance reduction that is evident for longer running processes such as executing large traces.

Based on the thesis, it is concluded that *the hypothesis is true*. However, further investigation must be carried out for the proposed protocol to determine that all common specification language features can be supported and that it is feasible to use the protocol for other specification languages than VDM. This will furthermore make a strong case for the standardisation of the SLSP protocol.

## 7.6 Future Work

This section describes relevant future work derived from the knowledge obtained throughout the thesis project. The items described in this section are considered less essential to the goals of the thesis but highly relevant for future work of both the SLSP protocol as well as the VS Code extension. Thus, the future work can be distributed across two areas, namely the SLSP protocol described in Chapter 4 and the VS Code extension developed as part of the pilot study described in Chapter 5.

### 7.6.1 Standardising the SLSP Protocol

As described in Section 6.2 VSCode-PVS leverages a language-specific protocol to support specification language features. A natural first step to move towards a standardisation of the SLSP protocol proposed in this thesis would be a collaboration to test the feasibility of supporting PVS using the SLSP protocol. In addition, it is also worthwhile to investigate the feasibility of supporting other specification languages such as B and Z to further strengthen the case for standardising the SLSP protocol.

When the SLSP protocol has been tested for other specification languages a request can be made to Microsoft to adopt the SLSP protocol and include it in the LSP specification. This would allow a larger number of developers to review and develop the protocol. Assuming that an adoption happens the language developers will be able to rely on the tool vendors, that supports the LSP protocol, to also implement support for the specification language features of the SLSP protocol in their IDE. This would enable a broader exposure of the specification language features, which in addition could cause programming languages to adopt some of the features, such

as POG for static analysis.

Alternatively, the SLSP protocol can be published as a stand-alone protocol for specification languages but such an approach will probably have less impact on the general community.

### 7.6.2 Enabling a Fully Decoupled Architecture

The architecture of the current PoC is close to the fully decoupled structure described in Section 3.3 where the client is effectively generic, i.e., language features are entirely provided by the server. To get to the fully decoupled structure, the responsibility to provide syntax highlighting rules must be moved from the client to the server. However, the client should still be responsible for applying the rules. This has also been noted on the forum for the LSP protocol, where one of the main developers state that:

*‘In general it is our [experience] that syntax highlighting must be fast and it best done inside the editor itself. So requesting this from the server might not be optimal.’<sup>4</sup>*

To this effect we propose a solution, where support for supplying syntax highlighting patterns is incorporated into either the SLSP protocol or the official LSP protocol. This should use a standardised format, e.g., JSON formatted regular expressions, such as TextMate<sup>5</sup> as used in VS Code. This will enable the server to supply the language-specific syntax highlighting patterns on initialisation that can be used by the client to apply the syntax highlighting. Hereby, effectively achieving a fully decoupled architecture. However, this requires that all IDEs are able to use a common format for syntax highlighting.

### 7.6.3 Support for Theorem Proving

Prior efforts have been made to offer proof support for TP in VDM. Agerholm and Sunesen [40] and Vermolen et. al [41] leverages a translation of VDM to HOL. In [41] they further build a proof system for the VDM++ proof obligations based on the HOL theorem prover. Agerholm and Frost [42] have made effort to create a VDM-LPF proof engine which is an extension of Isabelle with the Logic of Partial Functions (LPF) and VDM-SL data types. Although these efforts are not directly concerned with a language-neutral decoupling protocol, they and others manifest the relevancy of proof support for VDM and offers useful knowledge for continuing work towards proof support for VDM in the VS Code extension.

<sup>4</sup>See <https://github.com/microsoft/language-server-protocol/issues/682>.

<sup>5</sup>See [https://macromates.com/manual/en/language\\_grammars](https://macromates.com/manual/en/language_grammars).



### 7.6.4 Support for Model Checking

Model checking is currently not supported in the SLSP protocol. However, it is highly relevant for multiple specification languages such as Alloy [43] and TLA+ [44]. To investigate the functionality required for model checking two prime candidates are the VS Code extensions for Alloy<sup>6</sup> and TLA+<sup>7</sup>. The Alloy extension uses the LSP protocol for editor support while the model checking functionality is handled by Alloy specific messages, that are transmitted using the LSP base protocol. The TLA+ extension does not use any language-neutral protocols, instead the VS Code extension communicates with the official TLA+ tools<sup>8</sup> using the CLI. Providing support for model checking in the SLSP protocol could also motivate further investigation into model checking for VDM<sup>9</sup>.

### 7.6.5 From PoC to Official Release

The VS Code extension developed to support VDM as part of the pilot study is in its current iteration only at the level of a PoC. However, it is of relevance to take the implementation to a level that can be released as an official ‘version 1.0’ on the VS Code extension marketplace. To this effect, multiple areas of the extension needs to be improved. The current implementation only supports handling single-root projects, i.e., folders. As such, support for the VS Code concept of a workspace, which enables handling of multi-root projects, needs to be implemented. Furthermore, the implementation of the extension should be improved with regards to its code design to ease future development such as adding new features or more detailed syntax highlighting. Lastly, the difference in trace execution time for the native CLI for VDMJ and the VS Code extension should be investigated further to determine if it is possible to reduce it and thereby improve performance of longer running tasks such as trace execution.

In addition to the extension, aspects of the protocol can also be improved namely with regards to the functionality of filter options for the CT feature. The protocol should allow for the server to supply a set of valid options for filtered CT execution, as defining valid options is currently the responsibility of the client although they are specific to the server implementation.

### 7.6.6 Expanding Feature Support of the PoC

Moving forward with an official release version of the VS Code extension, support for additional features should be added. As evident from Figure 7.1 not all features supported by the LSP and SLSP protocols are available in the VS Code extension.

---

<sup>6</sup>See <https://github.com/s-arash/VSCoAlloyExtension>.

<sup>7</sup>See <https://github.com/alygin/vscode-tlaplus>.

<sup>8</sup>See <https://github.com/tlaplus/tlaplus>.

<sup>9</sup>Interest for model checking for VDM has previously been expressed in [14, 45] (not limited to).

## *CHAPTER 7. CONCLUDING REMARKS*

An obvious feature candidate is the coverage reporting that is available in VDMJ. Providing the LaTeX encoded coverage report can be done using the existing translate feature. However, providing visualisation in the specification as the Overture IDE does using ‘.cov’ files will need more effort and possibly an extension to the SLSP protocol. Other possibilities are the LSP protocol related features such as document folding and renaming, which require further development on the server.

## Bibliography

- [1] P. Masci and C. A. Muñoz, “An Integrated Development Environment for the Prototype Verification System,” *Electronic Proceedings in Theoretical Computer Science*, vol. 310, p. 35–49, Dec 2019.
- [2] R. Rodriguez-Echeverria, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, “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, (New York, NY, USA), p. 370–380, Association for Computing Machinery, 2018.
- [3] M. Hess and T. Kistler, *Dafny Language Server Redesign*. PhD thesis, HSR Hochschule für Technik Rapperswil, 2019.
- [4] J. K. Rask, F. P. Madsen, N. Battle, H. D. Macedo, and P. G. Larsen, “Visual Studio Code VDM Support,” in *Proceedings of the 18th Overture Workshop*, 2020.
- [5] M. . Gaudel, “Formal specification techniques,” in *Proceedings of 16th International Conference on Software Engineering*, pp. 223–227, 1994.
- [6] A. van Lamsweerde, “Formal Specification: a Roadmap,” in *ICSE ’00: Proceedings of the Conference on The Future of Software Engineering*, (New York, NY, USA), pp. 147–159, ACM Press, 2000.
- [7] M. Iglewski and T. Müldner, “Comparison of formal specification methods and object-oriented paradigms,” *Journal of Network and Computer Applications*, vol. 20, no. 4, pp. 355 – 377, 1997.
- [8] C. B. Jones, *Systematic software development using VDM*, vol. 2. Prentice Hall Englewood Cliffs, 1990.
- [9] M. Spivey, *The Z Notation – A Reference Manual (Second Edition)*. Prentice-Hall International, 1992.
- [10] J.-R. Abrial and J.-R. Abrial, *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

- [11] S. Garland, J. Guttag, K. Jones, J. Horning, A. Modet, and J. Wing, *Larch: Languages and Tools for Formal Specification*. Monographs in Computer Science, Springer New York, 2012.
- [12] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer, “Principles of OBJ2,” in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’85, (New York, NY, USA), p. 52–66, Association for Computing Machinery, 1985.
- [13] P. G. Larsen, “Ten Years of Historical Development: “Bootstrapping” VDM-Tools,” *Journal of Universal Computer Science*, vol. 7, no. 8, pp. 692–709, 2001.
- [14] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, “The Overture Initiative – Integrating Tools for VDM,” *SIGSOFT Softw. Eng. Notes*, vol. 35, pp. 1–6, January 2010.
- [15] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, “Rodin: an open toolset for modelling and reasoning in Event-B,” *STTT*, vol. 12, no. 6, pp. 447–466, 2010.
- [16] M. Leuschel and M. Butler, “ProB: A Model Checker for B,” in *FME 2003: Formal Methods* (K. Araki, S. Gnesi, and D. Mandrioli, eds.), (Berlin, Heidelberg), pp. 855–874, Springer Berlin Heidelberg, 2003.
- [17] J. Bendisposto, F. Fritz, M. Jastram, M. Leuschel, and I. Weigelt, “Developing Camille, a text editor for Rodin,” *Software: Practice and Experience*, vol. 41, no. 2, pp. 189–198, 2011.
- [18] P. Malik and M. Utting, “CZT: A Framework for Z Tools,” in *ZB 2005: Formal Specification and Development in Z and B* (H. Treharne, S. King, M. Henson, and S. Schneider, eds.), (Berlin, Heidelberg), pp. 65–84, Springer Berlin Heidelberg, 2005.
- [19] T. Miller, L. Freitas, P. Malik, and M. Utting, “CZT Support for Z Extensions,” in *Integrated Formal Methods* (J. Romijn, G. Smith, and J. van de Pol, eds.), (Berlin, Heidelberg), pp. 227–245, Springer Berlin Heidelberg, 2005.
- [20] B. Legeard, F. Peureux, and M. Utting, “Automated Boundary Testing from Z and B,” in *FME 2002: Formal Methods—Getting IT Right* (L.-H. Eriksson and P. A. Lindsay, eds.), (Berlin, Heidelberg), pp. 21–40, Springer Berlin Heidelberg, 2002.
- [21] L. Voisin and J.-R. Abrial, “The rodin platform has turned ten,” in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pp. 1–8, Springer, 2014.

## BIBLIOGRAPHY

- [22] L. D. Couto, P. G. Larsen, M. Hasanagic, G. Kanakis, K. Lausdahl, and P. W. V. Tran-Jørgensen, “Towards Enabling Overture as a Platform for Formal Notation IDEs,” in *2nd Workshop on Formal-IDE (F-IDE)*, (Oslo, Norway), June 2015.
- [23] J. W. Coleman, A. K. Malmos, C. B. Nielsen, and P. G. Larsen, “Evolution of the Overture Tool Platform,” in *Proceedings of the 10th Overture Workshop 2012*, School of Computing Science, Newcastle University, 2012.
- [24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [25] N. Battle, “VDMJ User Guide,” tech. rep., Fujitsu Services Ltd., UK, 2009.
- [26] H. Bünder and H. Kuchen, “Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol,” in *Model-Driven Engineering and Software Development* (S. Hammoudi, L. F. Pires, and B. Selić, eds.), (Cham), pp. 225–245, Springer International Publishing, 2020.
- [27] H. Bünder, “Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages.,” in *MODELSWARD*, pp. 129–140, 2019.
- [28] P. Tran-Jørgensen and T. Kulik, “Migrating Overture to a different IDE,” in *Proceedings of the 17th Overture Workshop* (C. Gamble and L. Diogo Couto, eds.), no. CS-TR- 1530 - 2019 in Technical Report Series, pp. 32–47, Newcastle University, 2019. null ; Conference date: 07-10-2019 Through 11-10-2019.
- [29] R. M. Stallman, “EMACS the Extensible, Customizable Self-Documenting Display Editor,” *SIGPLAN Not.*, vol. 16, p. 147–156, Apr. 1981.
- [30] T. Oda, K. Araki, and P. G. Larsen, “VDMPad: A Lightweight IDE for Exploratory VDM-SL Specification,” in *Proceedings of the Third FME Workshop on Formal Methods in Software Engineering*, Formalise ’15, p. 33–39, IEEE Press, 2015.
- [31] R. S. Reimer and K. D. Saaby, “An Open-Source Web IDE for VDM-SL,” Master’s thesis, Department of Engineering, Aarhus University, Denmark, May 2016.
- [32] N. Kahani, M. Bagherzadeh, J. Dingel, and J. R. Cordy, “The Problems with Eclipse Modeling Tools: A Topic Analysis of Eclipse Forums,” in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS ’16, (New York, NY, USA), p. 227–237, Association for Computing Machinery, 2016.

- [33] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron, “Filtering TOBIAS Combinatorial Test Suites,” in *FASE 2004* (M. Wermelinger and T. Margaria-Steffen, eds.), (Springer-Verlag Berlin Heidelberg), pp. 281–294, LNCS 2984, 2004.
- [34] L. C. Paulson, “Natural deduction as higher-order resolution,” *Journal of Logic Programming*, vol. 3, pp. 237–258, 1986.
- [35] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, “The Lean Theorem Prover (System Description),” in *Automated Deduction - CADE-25* (A. P. Felty and A. Middeldorp, eds.), (Cham), pp. 378–388, Springer International Publishing, 2015.
- [36] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, “Back to the future - the story of squeak, a practical smalltalk written in itself,” *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 318–326, 1997.
- [37] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner, “The Coq proof assistant user’s guide,” Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
- [38] L. Walsh, J. Dingel, and K. Jahed, “Toward Client-Agnostic Hybrid Model Editor Tools as a Service,” in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS ’20*, (New York, NY, USA), Association for Computing Machinery, 2020.
- [39] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
- [40] S. Agerholm and K. Sunesen, “Reasoning about VDM-SL Proof Obligations in HOL,” tech. rep., IFAD, 1999.
- [41] S. Vermolen, J. Hooman, and P. G. Larsen, “Automating Consistency Proofs of VDM++ Models using HOL,” in *Proceedings of the 25th Symposium on Applied Computing (SAC 2010)*, (Sierre, Switzerland), ACM, March 2010.
- [42] S. Agerholm and J. Frost, “An Isabelle-based theorem prover for VDM-SL,” in *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs’97)*, LNCS, Springer-Verlag, August 1997. Also available as technical report IT-TR: 1997-009 from the Department of Information Technology at the Technical University of Denmark.
- [43] D. Jackson, “Alloy: a lightweight object modelling notation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.
- [44] L. Lamport, “Specifying Concurrent Systems with TLA+,” in *Computational System Design*, (Amsterdam), IOS Press, 1999.

## *BIBLIOGRAPHY*

- [45] D. Jackson, “Abstract model checking of infinite specifications,” in *FME’94: Industrial Benefit of Formal Methods* (M. B. M. Naftalin, T. Denvir, ed.), pp. 519–531, Springer-Verlag, October 1994.

All links were last followed on January 3rd, 2021.





## Appendix A

# SLSP Protocol Outline

This chapter details the SLSP protocol specification in a similar way used in the official LSP protocol specification<sup>1</sup>.

Section A.1 is copied verbatim from the official LSP specification<sup>2</sup>.

### A.1 Base Protocol

The base protocol consists of a header and a content part (comparable to HTTP). The header and content part are separated by a ‘\r\n’.

#### A.1.1 Header Part

The header part consists of header fields. Each header field is comprised of a name and a value, separated by ‘: ’ (a colon and a space). The structure of header fields conform to the HTTP semantic. Each header field is terminated by ‘\r\n’. Considering the last header field and the overall header itself are each terminated with ‘\r\n’, and that at least one header is mandatory, this means that two ‘\r\n’ sequences always immediately precede the content part of a message.

Currently the following header fields are supported:

---

<sup>1</sup>See <https://microsoft.github.io/language-server-protocol/specifications/specification-current/>.

<sup>2</sup>See <https://microsoft.github.io/language-server-protocol/specification#baseProtocol>.

## APPENDIX A. SLSP PROTOCOL OUTLINE

Header Field Name	Value Type	Description
Content-Length	number	The length of the content part in bytes. This header is required.
Content-Type	string	The mime type of the content part. Defaults to application/vscode-jsonrpc; charset=utf-8

The header part is encoded using the ‘ascii’ encoding. This includes the ‘\r\n’ separating the header and content part.

### A.1.2 Content Part

Contains the actual content of the message. The content part of a message uses JSON-RPC to describe requests, responses and notifications. The content part is encoded using the charset provided in the Content-Type field. It defaults to utf-8, which is the only encoding supported right now. If a server or client receives a header with a different encoding than utf-8 it should respond with an error.

(Prior versions of the protocol used the string constant utf-8 which is not a correct encoding constant according to specification.) For backwards compatibility it is highly recommended that a client and a server treats the string utf-8 as utf-8.

### Example

```
Content-Length: ... \r\n
\r\n
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "textDocument/didOpen",
  "params": {
    ...
  }
}
```

### Base Protocol JSON structures

The following TypeScript definitions describe the base JSON-RPC protocol:

#### Abstract Message

A general message as defined by JSON-RPC. The language server protocol always uses “2.0” as the jsonrpc version.

## APPENDIX A. SLSP PROTOCOL OUTLINE

```
interface Message {  
  jsonrpc: string;  
}
```

### A.1.3 Request Message

A request message to describe a request between the client and the server. Every processed request must send a response back to the sender of the request.

```
interface RequestMessage extends Message {  
  /**  
   * The request id.  
   */  
  id: number | string;  
  
  /**  
   * The method to be invoked.  
   */  
  method: string;  
  
  /**  
   * The method's params.  
   */  
  params?: array | object;  
}
```

### A.1.4 Response Message

A Response Message sent as a result of a request. If a request doesn't provide a result value the receiver of a request still needs to return a response message to conform to the JSON RPC specification. The result property of the ResponseMessage should be set to `null` in this case to signal a successful request.

## APPENDIX A. SLSP PROTOCOL OUTLINE

```
interface ResponseMessage extends Message {
  /**
   * The request id.
   */
  id: number | string | null;

  /**
   * The result of a request. This member is REQUIRED on success.
   * This member MUST NOT exist if there was an error invoking the
   * method.
   */
  result?: string | number | boolean | object | null;

  /**
   * The error object in case a request fails.
   */
  error?: ResponseError;
}

interface ResponseError {
  /**
   * A number indicating the error type that occurred.
   */
  code: number;

  /**
   * A string providing a short description of the error.
   */
  message: string;

  /**
   * A primitive or structured value that contains additional
   * information about the error. Can be omitted.
   */
  data?: string | number | boolean | array | object | null;
}

export namespace ErrorCodes {
  // Defined by JSON RPC
  export const ParseError: number = -32700;
  export const InvalidRequest: number = -32600;
  export const MethodNotFound: number = -32601;
  export const InvalidParams: number = -32602;
  export const InternalError: number = -32603;
  export const serverErrorStart: number = -32099;
  export const serverErrorEnd: number = -32000;
  export const ServerNotInitialized: number = -32002;
  export const UnknownErrorCode: number = -32001;

  // Defined by the protocol.
  export const RequestCancelled: number = -32800;
  export const ContentModified: number = -32801;
}
```

## APPENDIX A. SLSP PROTOCOL OUTLINE

### A.1.5 Notification Message

A notification message. A processed notification message must not send a response back. They work like events.

```
interface NotificationMessage extends Message {  
    /**  
     * The method to be invoked.  
     */  
    method: string;  
  
    /**  
     * The notification's params.  
     */  
    params?: array | object;  
}
```

### \$ Notifications and Requests

Notification and requests whose methods start with '\$/' are messages which are protocol implementation dependent and might not be implementable in all clients or servers. For example if the server implementation uses a single threaded synchronous programming language then there is little a server can do to react to a '\$/cancelRequest' notification. If a server or client receives notifications starting with '\$/' it is free to ignore the notification. If a server or client receives a requests starting with '\$/' it must error the request with error code `MethodNotFound` (e.g. -32601).

### A.1.6 Cancellation Support

The base protocol offers support for request cancellation. To cancel a request, a notification message with the following properties is sent:

*Notification:*

- method: '\$/cancelRequest'
- params: `CancelParams` defined as follows:

```
interface CancelParams {  
    /**  
     * The request id to cancel.  
     */  
    id: number | string;  
}
```

A request that got canceled still needs to return from the server and send a response back. It can not be left open / hanging. This is in line with the JSON RPC protocol that requires that every request sends a response back. In addition it allows for returning partial results on cancel. If the request returns an error response on cancellation it is advised to set the error code to `ErrorCodes.RequestCancelled`.

### A.1.7 Progress Support

The base protocol offers also support to report progress in a generic fashion. This mechanism can be used to report any kind of progress including work done progress (usually used to report progress in the user interface using a progress bar) and partial result progress to support streaming of results.

A progress notification has the following properties:

*Notification:*

- method: `$/progress`
- params: `ProgressParams` defined as follows:

```
type ProgressToken = number | string;
interface ProgressParams<T> {
  /**
   * The progress token provided by the client or server.
   */
  token: ProgressToken;

  /**
   * The progress data.
   */
  value: T;
}
```

Progress is reported against a token. The token is different than the request ID which allows to report progress out of band and also for notification.

## A.2 Proof Obligation Generation

Besides specialised initialisation entries, the following messages are related to the POG feature in the protocol:

- **Generate request**
- **Specification Updated notification**

### A.2.1 Initialisation

*Client Capability:*

- property name (optional):  
`experimental.proofObligationGeneration`
- property type: `boolean`

*Server Capability:*

- property name (optional):  
`experimental.proofObligationProvider`
- property type: `boolean`

### A.2.2 Generate Request

The generate proof obligations request is sent from the client to the server to request generation of proof obligations for a document or folder.

*Request:*

- method: `'slsp/POG/generate'`
- params: `GeneratePOParams` defined as follows:

```
export interface GeneratePOParams {  
  /**  
   * Uri to the file/folder for which proof obligations  
   * should be generated.  
   */  
  uri: DocumentUri;  
}
```

*Response:*

- result: `ProofObligation[] | null`

## APPENDIX A. SLSP PROTOCOL OUTLINE

- error: code and message set in case an exception happens during the generate request.

```
/**
 * Parameters describing a Proof Obligation (PO) and meta data.
 */
export interface ProofObligation {
  /**
   * Unique identifier of the PO.
   */
  id: number;
  /**
   * Name of the PO.
   * Array describe the hieracy of the name,
   * e.g. ["classA", "function1"].
   */
  name: string[];
  /**
   * Type of the PO.
   */
  type: string;
  /**
   * Location where the PO applies.
   */
  location: Location;
  /**
   * Source code of the PO.
   * String array can be used to provide visual formatting
   * information, e.g. the PO view can put a "\n\t" between
   * each string in the array.
   */
  source: string | string[];
  /**
   * An optional flag indicating if the PO has been proved.
   */
  proved?: boolean;
}
```

### A.2.3 Specification Updated Notification

The updated notification is sent from the server to the client to notify that the specification has been updated. Meaning that the POs that has been sent to the client before this notification are now out-of-sync with the specification.

*Notification:*

- method: 'slsp/POG/updated'
- params: POGUpdatedParams defined as follows:



## APPENDIX A. SLSP PROTOCOL OUTLINE

```
export interface POGUpdatedParams {  
  /**  
   * Describes the state of the specification.  
   * True if POG is possible.  
   * False otherwise, e.g. the specification is not type-correct.  
   */  
  successful: boolean;  
}
```

### A.3 Combinatorial Testing

Besides specialised initialisation entries, the following messages are related to the CT feature in the protocol:

- **Traces request**
- **Generate request**
- **Execute request**

#### A.3.1 Initialisation

*Client Capability:*

- property name (optional): `experimental.combinatorialTesting`
- property type: `boolean`

*Server Capability:*

- property name (optional):  
`experimental.combinatorialTestProvider`
- property type: `boolean | CombinatorialTestOptions` where  
`CombinatorialTestOptions` is defined as follows:

```
export interface CombinatorialTestOptions extends  
  WorkDoneProgressOptions {  
}
```

#### A.3.2 Trace Request

The trace request is sent from the client to the server to request an outline of the traces available in a given specification.

*Request:*

## APPENDIX A. SLSP PROTOCOL OUTLINE

- method: ‘slsp/CT/traces’
- params: CTracesParameters defined as follows:

```
export interface CTracesParameters {  
  /**  
   * An optional uri to the file/folder for which Traces should be  
   * found.  
   */  
  uri?: DocumentUri;  
}
```

### *Response:*

- result: CTSymbol[] | null
- error: code and message set in case an exception happens during the traces request.

## APPENDIX A. SLSP PROTOCOL OUTLINE

```
/**
 * Describes a grouping of traces, e.g. a class, classA, may
 * have multiple traces which are all combined in a CTSymbol.
 */
export interface CTSymbol {
  /**
   * Name of Trace group, e.g. "classA".
   */
  name: string;
  /**
   * Traces in the group.
   */
  traces: CTTrace[];
}

/**
 * Overview information about a trace
 */
export interface CTTrace {
  /**
   * Fully qualified name of the trace.
   */
  name: string;
  /**
   * Location in the source code of the trace.
   */
  location: Location;
  /**
   * An optional combined verdict of all the tests from the trace.
   */
  verdict?: VerdictKind;
}

/**
 * Kinds of test case verdicts.
 */
export enum VerdictKind {
  Passed = 1,
  Failed = 2,
  Inconclusive = 3,
  Filtered = 4,
}
```

### A.3.3 Generate Request

The generate request is sent from the client to the server to request generation of tests from a trace.

*Request:*

- method: 'slsp/CT/generate'
- params: CTGenerateParameters defined as follows:

```
export interface CTGenerateParameters
  extends WorkDoneProgressParams {
  /**
   * Fully qualified name of the trace, which test cases should be
   * generated based on.
   */
  name: string;
}
```

*Response:*

- result: CTGenerateResponse | null
- error: code and message set in case an exception happens during the generate request.

```
export interface CTGenerateResponse {
  /**
   * The number of tests that is generated from the trace.
   */
  numberOfTests: number;
}
```

### A.3.4 Execute Request

The execute request is sent from the client to the server to request execution of tests from a trace using optional filtering options.

*Request:*

- method: 'slsp/CT/execute'
- params: CTExecuteParameters defined as follows:

## APPENDIX A. SLSP PROTOCOL OUTLINE

```
export interface CTEExecuteParameters
  extends WorkDoneProgressParams, PartialResultParams {
  /**
   * Fully qualified name of the trace, which test cases should be
   * executed from.
   */
  name: string;
  /**
   * Optional filters that should be applied to the execution.
   * If omitted the server should use default settings.
   */
  filter?: CTFilterOption[];
  /**
   * An optional range of tests that should be executed.
   * If omitted all tests for the trace are executed.
   */
  range?: NumberRange;
}

/**
 * Mapping type for filter options for the execution of CTs.
 */
export interface CTFilterOption {
  /**
   * Name of the option. E.g. "reduction", "seed" or "limit".
   */
  key: string;
  /**
   * Value of the option. E.g. "random", 999, 100.
   */
  value: string | number | boolean;
}

/**
 * Describes a range of numbers, e.g. 1-10.
 */
export interface NumberRange {
  /**
   * Start number, if omitted 'end' should be considered as
   * the absolute number of tests that must be returned
   */
  start?: number;
  /**
   * End number, if omitted tests from 'start' to last
   * should be returned.
   */
  end?: number;
}
```

## APPENDIX A. SLSP PROTOCOL OUTLINE

*Response:*

- **result:** CTestCase[] | null
- **partial result:** CTestCase[]
- **error:** code and message set in case an exception happens during the execute request.

```
/**
 * Test case information.
 */
export interface CTestCase {
  /**
   * ID of the test case.
   */
  id: number;
  /**
   * Test case verdict
   */
  verdict: VerdictKind;
  /**
   * Test case execution sequence and result.
   */
  sequence: CResultPair[];
}

/**
 * Kinds of test case verdicts.
 */
export enum VerdictKind {
  Passed = 1,
  Failed = 2,
  Inconclusive = 3,
  Filtered = 4,
}

/**
 * Test sequence result pair.
 */
export interface CResultPair {
  /**
   * The operation/function that was executed.
   */
  case: string;
  /**
   * The result of the operation/function. Null if no result.
   */
  result: string | null;
}
```

## A.4 Translation

Besides specialised initialisation entries, the following message is related to the translate feature in the protocol:

- **Translate Request**

### A.4.1 Initialisation

*Client Capability:*

- property name (optional):  
`experimental.translate`
- property type: `boolean`

*Server Capability:*

- property name (optional):  
`experimental.translateProvider`
- property type: `boolean | TranslateOptions` where `TranslateOptions` is defined as follows:

```
export interface TRTranslateOptions extends WorkDoneProgressOptions {
  /**
   * language id as a string. See the LSP specification for valid
   * ids.
   */
  languageId: string | string[];
}
```

### A.4.2 Translate Request

The translate request is sent from the client to the server to request translation of a specification.

*Request:*

- method: `'slsp/TR/translate'`
- params: `TranslateParams` defined as follows:

## APPENDIX A. SLSP PROTOCOL OUTLINE

```
export interface TRTranslateParams {  
  /**  
   * DocumentUri specifying the root of the project to translate.  
   */  
  uri?: DocumentUri;  
  /**  
   * Language id as a string.  
   * See the LSP specification for valid ids.  
   */  
  languageId: string;  
  /**  
   * DocumentUri specifying the location of the resulting  
   * translation. This should be an existing empty folder.  
   */  
  saveUri: DocumentUri;  
}
```

### Response:

- **result:** TranslateResponse defined as follows:

```
export interface TRTranslateResponse {  
  /**  
   * URI specifying the "main" file of the resulting translation  
   * If multiple files are generated, this is the uri to where  
   * "main" is.  
   */  
  uri: DocumentUri;  
}
```



## A.5 Theorem Proving

Besides specialised initialisation entries, the following messages in the protocol is related to the theorem proving feature:

- **Lemmas Request**
- **Begin Proof Request**
- **Prove Request**
- **Get Commands Request**
- **Command Request**
- **Undo Request**

### A.5.1 Initialisation

*Client Capability:*

- property name (optional):  
`experimental.theoremProving`
- property type: `boolean`

*Server Capability:*

- property name (optional):  
`experimental.theoremProvingProvider`
- property type: `boolean`

### A.5.2 Lemmas Request

The lemmas request is sent from the client to the server to get an exhaustive list of lemmas in the specification.

*Request:*

- method: `'slsp/TP/lemmas'`
- params: `LemmasParams` defined as follows:

## APPENDIX A. SLSP PROTOCOL OUTLINE

```
interface TPLemmasParams {  
    /**  
     * The scope of the project files.  
     */  
    projectUri?: DocumentUri  
}
```

*Response:*

- result: Lemma [ ] defined as follows:

```
/**  
 * Parameters describing a Lemma and meta data.  
 */  
interface Lemma {  
    /**  
     * Unique name of the lemma.  
     */  
    name: string,  
    /**  
     * Name of the theory that the lemma belongs to.  
     */  
    theory: string,  
    /**  
     * Identifies the location of the lemma.  
     */  
    location: Location,  
    /**  
     * Theorem, Lemma, corollary etc.  
     */  
    kind: string,  
    /**  
     * Status of the proof of the lemma  
     */  
    status: ProofStatus  
}
```

Where ProofStatus is defined as follows:

```
/**  
 * Type describing the status of a proof  
 */  
type ProofStatus = "proved" | "disproved" | "untried" | "unfinished"  
    " | "timeout" | "unchecked";
```

## APPENDIX A. SLSP PROTOCOL OUTLINE

### A.5.3 Begin Proof Request

The begin proof request is sent from the client to the server to initialise a proof session for a lemma.

*Request:*

- method: ‘slsp/TP/beginProof’
- params: TPBeginProofParams defined as follows:

```
interface TPBeginProofParams {  
    /**  
     * Name of the lemma that is to be proved.  
     */  
    name: string  
}
```

*Response:*

- result: ProofState defined as follows:

```
/**  
 * Parameters describing the state of a proof and meta data.  
 */  
interface ProofState {  
    /**  
     * Proof step id.  
     */  
    id: number,  
    /**  
     * Status of the proof.  
     */  
    status: ProofStatus | string,  
    /**  
     * Subgoals, empty if proved.  
     */  
    subgoals: string[],  
    /**  
     * Rules used for this step.  
     */  
    rules?: string[]  
}
```

### A.5.4 Prove Request

The prove request is sent from the client to the server to request the theorem prover to automatically prove a lemma.

## APPENDIX A. SLSP PROTOCOL OUTLINE

*Request:*

- method: 'slsp/TP/prove'
- params: TPProveParams defined as follows:

```
interface TPProveParams {  
    /**  
     * Name of the lemma that is to be proved.  
     * If proof in progress that lemma is assumed.  
     */  
    name?: string  
}
```

*Response:*

- result: TPProveResponse defined as follows:

```
interface TPProveResponse {  
    /**  
     * Status of the proof.  
     */  
    status: ProofStatus,  
    /**  
     * Processing time in milliseconds  
     */  
    time?: number,  
    /**  
     * Suggested commands to apply  
     */  
    command?: string[],  
    /**  
     * Humans-readable description of:  
     * Counter example, proof steps, etc.  
     */  
    description?: string  
}
```

### A.5.5 Get Commands Request

The get commands request is sent from the client to the server to request a list of commands that can be applied to a proof.

*Request:*

- method: 'slsp/TP/getCommands'
- params: null.

## APPENDIX A. SLSP PROTOCOL OUTLINE

*Response:*

- **result:** TPCCommand defined as follows:

```
/**
 * Parameters describing a theorem proving command.
 */
interface TPCCommand {
    /**
     * Command name.
     */
    name: string,
    /**
     * Description of the command.
     */
    description: string
}
```

### A.5.6 Command Request

The command request is sent from the client to the server to apply a command to the current step of a proof.

*Request:*

- **method:** 'slsp/TP/command'
- **params:** TPCCommandParams defined as follows:

```
interface TPCCommandParams {
    /**
     * The command and arguments identified by a string.
     */
    command: string
}
```

*Response:*

- **result:** TPCCommandResponse defined as follows:

## APPENDIX A. SLSP PROTOCOL OUTLINE

```
interface TPCCommandResponse {  
    /**  
     * Description of the result of the command,  
     * e.g. accepted, error, no change.  
     */  
    description: string,  
    /**  
     * State of the proof after the command.  
     */  
    state: ProofState  
}
```

### A.5.7 Undo Request

The undo request is sent from the client to the server to undo a proof step.

*Request:*

- method: 'slsp/TP/undo'
- params: TPUndoParams defined as follows:

```
interface TPUndoParams {  
    /**  
     * Id of the step that must be undone.  
     * If empty, undo last step.  
     */  
    id?: number  
}
```

*Response:*

- result: ProofState.

## Appendix B

# Theorem Proving Using SLSP

This chapter provides a suggestion to how to support TP using the SLSP protocol. First, a suggestion to which proof status should be assigned to a lemma based on the interactions with the proof is illustrated. Next, the chapter includes a step by step description of how the VSCode-PVS<sup>1</sup> [1] interface for TP can be supported using the SLSP protocol. This was shortly described in Section 4.5.3. Lastly, the chapter includes large versions of the VSCode-PVS snippet illustrated in Section 4.5.3 and the Isabelle VS Code extension.

### B.1 Proof Status

Figure B.1 illustrates the proof statuses that a lemma can have and a suggestion to the transitions between the states.

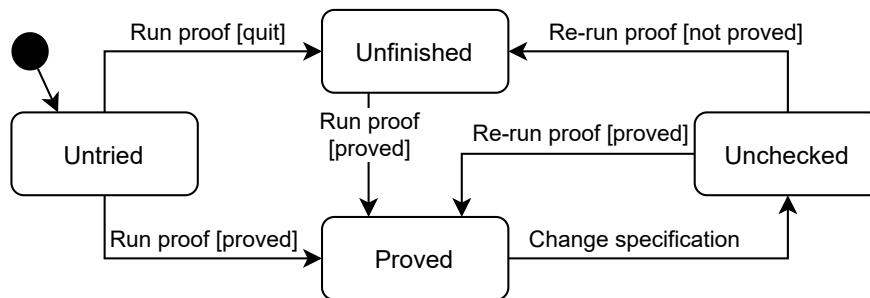


Figure B.1: State machine diagram showing a suggestion for how to transition between the different proof states.

<sup>1</sup>See <https://github.com/nasa/vscode-pvs>.

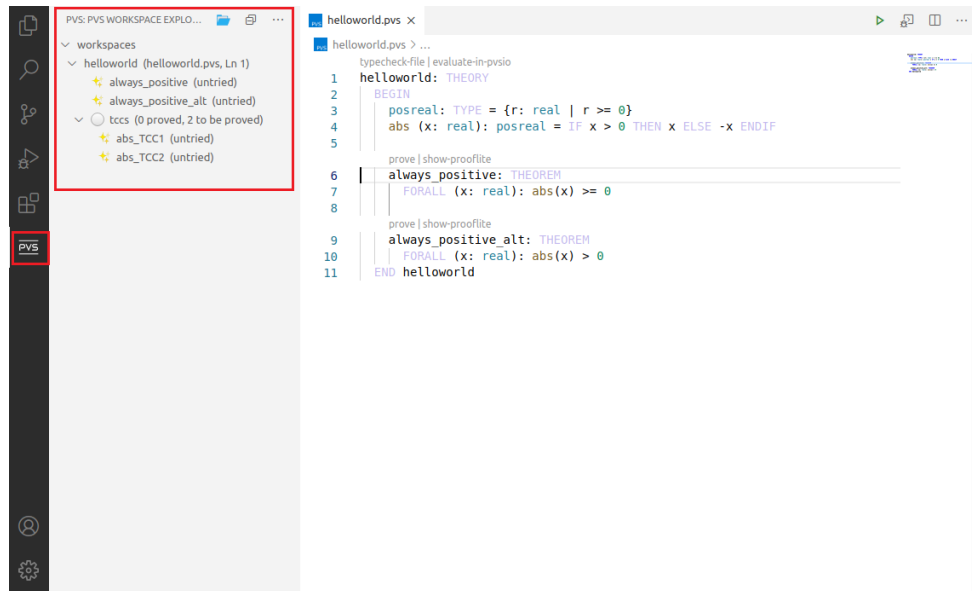
## B.2 Using SLSP for the VSCode-PVS Interface

This section describes how messages relating to the TP feature support found in the SLSP protocol can be used to facilitate actions and GUI elements found in VSCode-PVS. *Note* that it has not been implemented using the SLSP protocol, this simply illustrates that it is possible to implement the GUI using the SLSP protocol.

### Step 1: Lemma Overview

**Actions:** Launch VS Code and open the ‘PVS Workspace Explorer’.

**Protocol:** `lemmas` provide a list of lemmas that can be displayed in the view



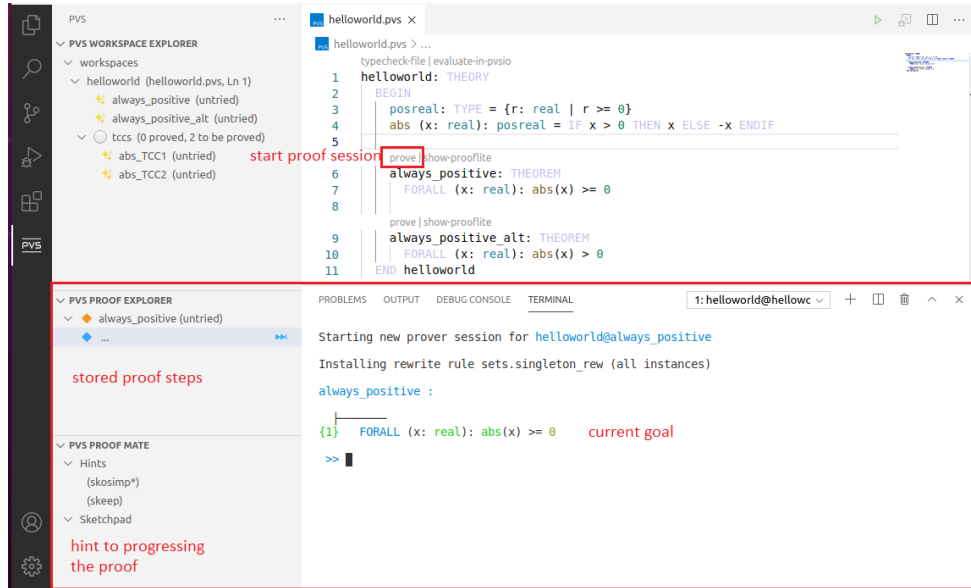
### Step 2: Starting a Proof Session

**Actions:** Start proof session for the lemma: `always_positive`. This launches a terminal to the theorem prover that displays the current goal. In the ‘Proof Explorer’ proof steps that have been performed are shown. In the ‘Proof Mate’ suggestions to proof steps that can progress the proof are listed.

**Protocol:** `beginProof` starts a proof session which launches the terminal. `prove` provides a list of commands that can progress the proof which can be displayed in the ‘Proof Mate’.



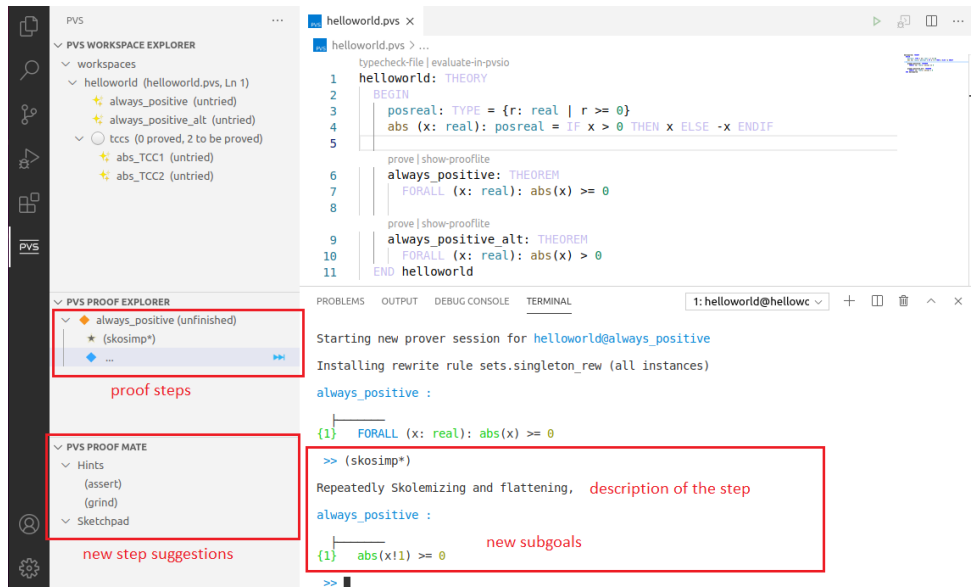
## APPENDIX B. THEOREM PROVING USING SLSP



### Step 3: Performing a Proof Step

**Actions:** Perform the suggested proof step (`skosimp*`), which causes a new goal to be displayed. In the 'Proof Explorer' the step is displayed and the step tracker has moved down a step. In the 'Proof Mate' new suggestions to proof steps that can further progress the proof are shown.

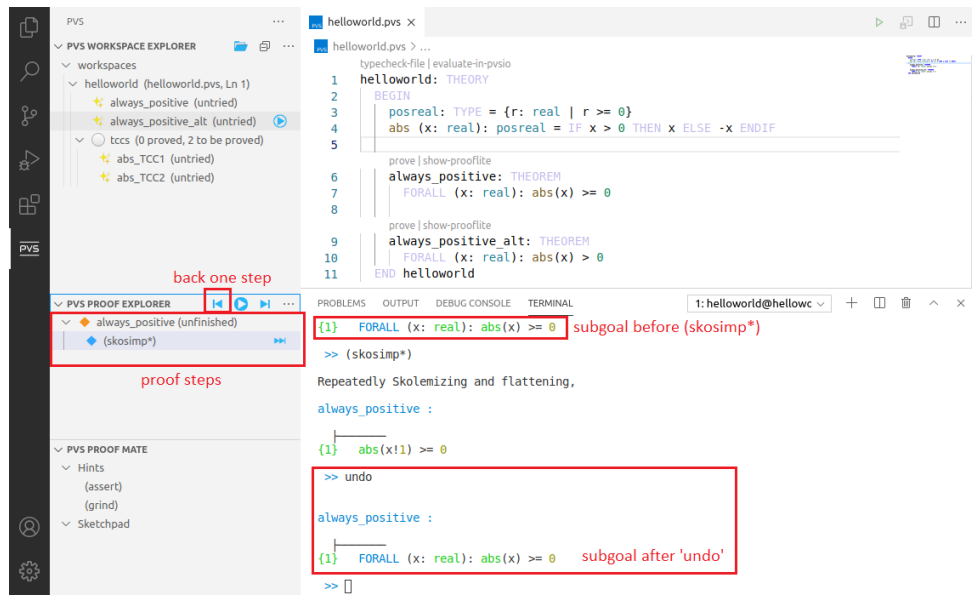
**Protocol:** `command` sends the proof step to the server, which handles the step and responds with the new subgoals. `prove` provides a new list of commands that can be used to progress the proof.



### Step 4: Undoing a Proof Step

**Actions:** Undoes a proof step by pressing the ‘back one step’ button. This performs an *undo* step in the terminal, and the previous subgoals are displayed again. Furthermore, the step tracker in the ‘Proof Explorer’ moves back one step but the command is still stored.

**Protocol:** `undo` causes the server to undo the latest command and responds with the sub-goals from the previous step.

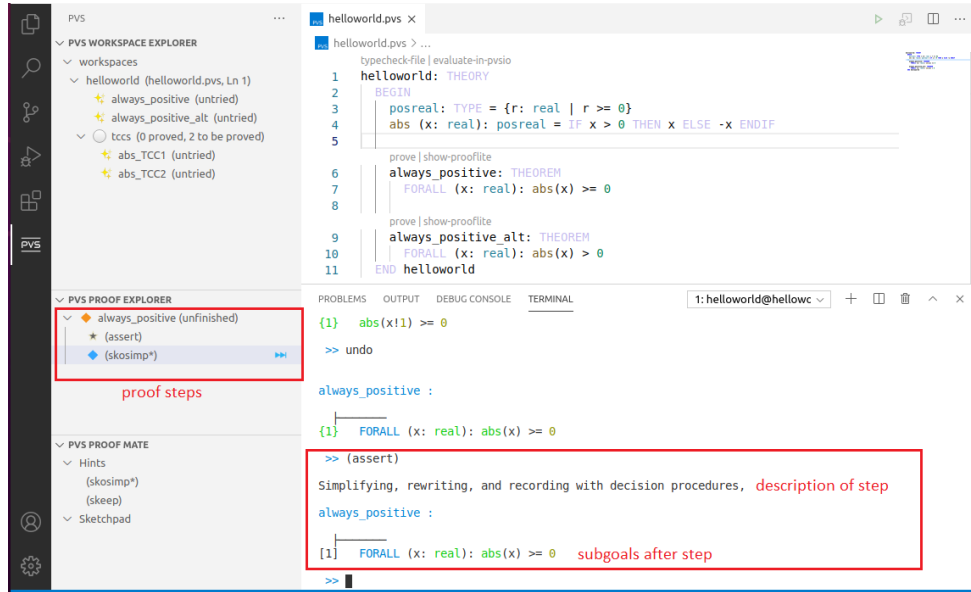


### Step 5: Performing a Different Proof Step

**Actions:** Perform the proof step (`assert`), which has no effect on the proof, hence the same sub-goals are displayed. In the ‘Proof Explorer’ the step is seen and the step tracker has moved down a step.

**Protocol:** `command` sends the proof step to the server which handles the step and responds with the new sub-goals after the command.

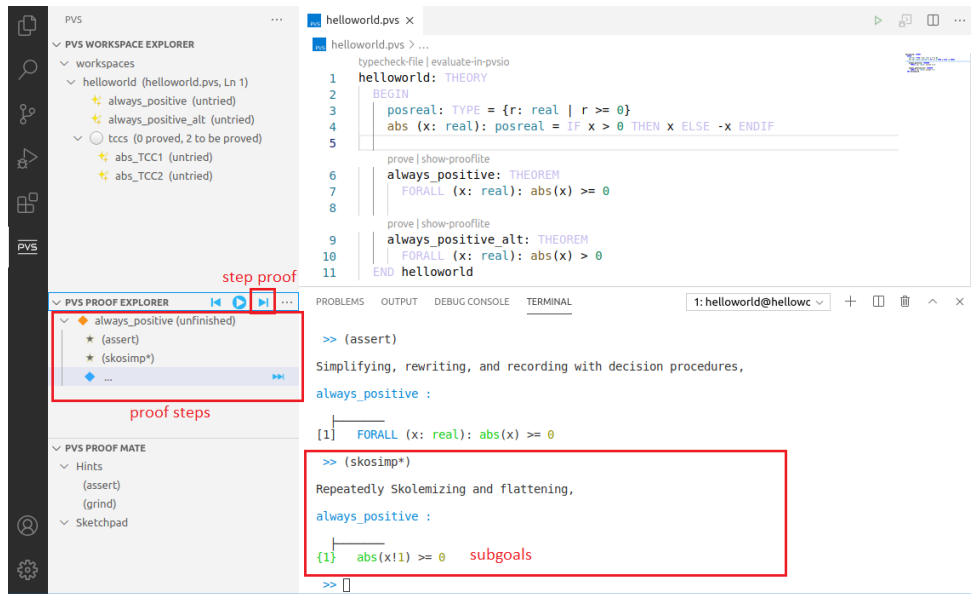
## APPENDIX B. THEOREM PROVING USING SLSP



### Step 6: Replay Stored Proof Step

**Actions:** Perform the stored proof step by pressing the ‘Step proof’ button, which causes a new sub-goal to be displayed. In the ‘Proof Explorer’ the step tracker has moved down a step.

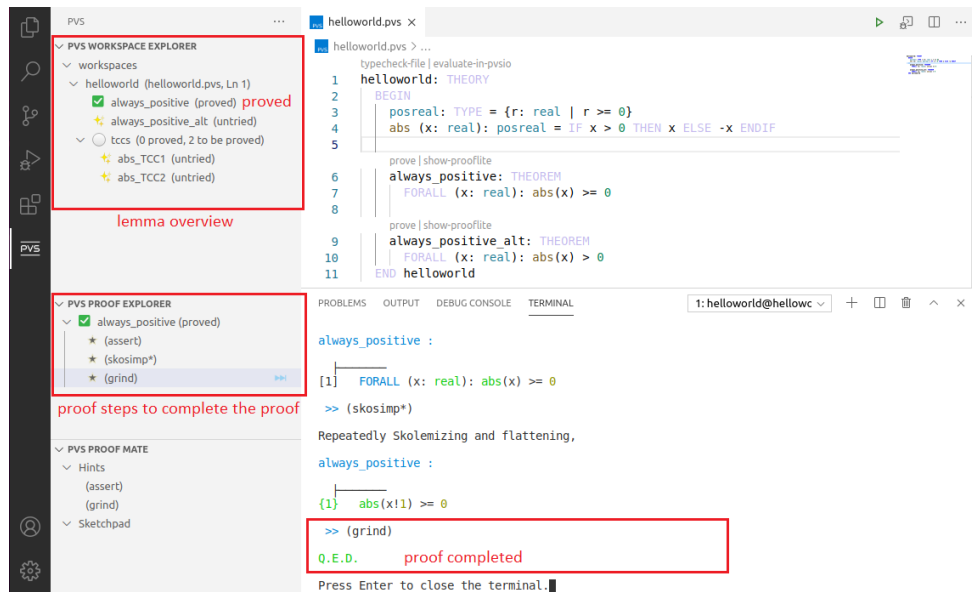
**Protocol:** command sends the stored proof step to the server which handles the step and responds with the sub-goals after the command.



## Step 7: Complete the Proof

**Actions:** Perform the suggested proof step (`grind`), which causes Q.E.D. to be displayed, indicating that the proof has been completed. In the ‘Proof Explorer’ the steps that have been performed to complete the proof are displayed, these are stored for later use. In the ‘Workspace Explorer’ the status of the lemma is changed to ‘proved’.

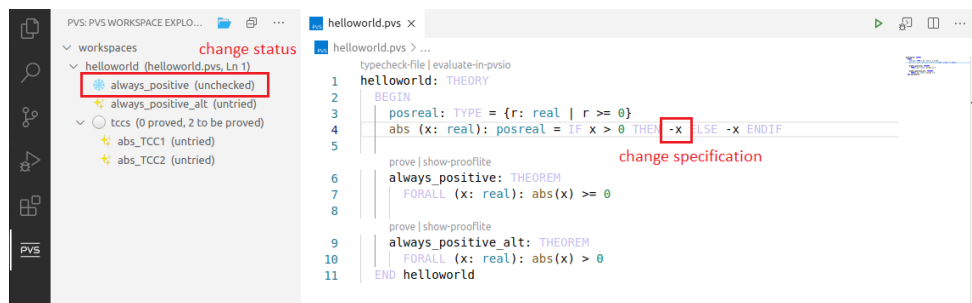
**Protocol:** `command` sends the proof step to the server which handles the step and responds with the proved proof status.



## Step 8: Change the Specification

**Actions:** Make a small change in the specification. This changes the proof status for `always_positive` as it has not been proved after the specification changes.

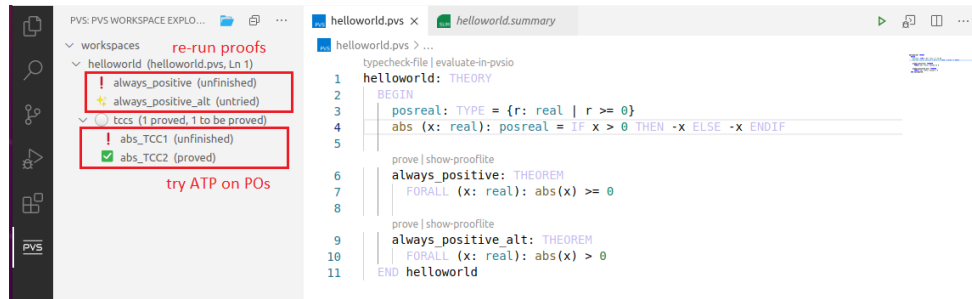
**Protocol:** `lemmas` provides a list of lemmas that are now available for the specification and updates the status.



## Step 9: Re-run All Proofs

**Actions:** Run the command ‘Re-run All Proofs’ which re-runs the proofs that have previously been performed and tries to automatically prove the others. As the proof steps performed for the lemma `always_positive` is no longer able to complete the proof, hence the status is changed to ‘unfinished’. One other proof is proved and another is not able to be automatically proved.

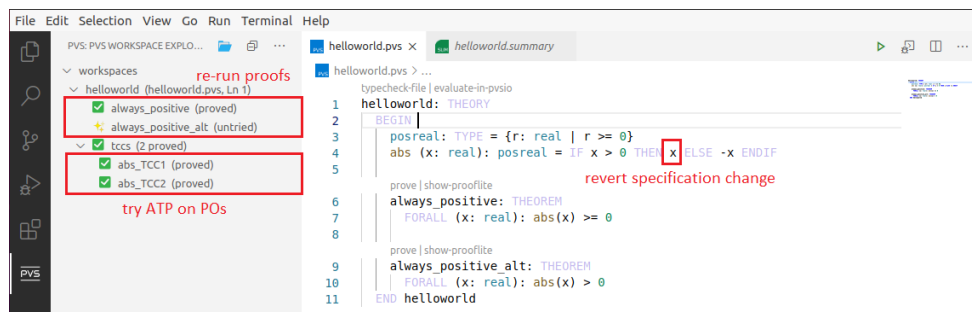
**Protocol:** `beginProof` and `command` is used to launch a proof session and re-transmit the stored proof steps, without displaying the terminal to the user. The status after transmitting all the commands is displayed in the ‘Workspace Explorer’. `prove` is used to try ATP on the two TCCs (alternative wording for POs), the response from the server includes the result of the ATP process.



## Step 10: Revert the Specification Change and Re-run All Proofs

**Actions:** The change to the specification is reverted and ‘Re-run All Proofs’ is performed again. This changes the status of the `always_positive` lemma to ‘proved’. The POs are proved using ATP.

**Protocol:** `beginProof` and `command` is used to launch a proof session and re-transmit the stored proof steps, without displaying the terminal to the user. The status after transmitting all the commands is displayed in the ‘Workspace Explorer’. `prove` is used to try ATP on the two POs, the response from the server includes the result of the ATP process.



### B.3 Large Version of VSCode-PVS Snippet

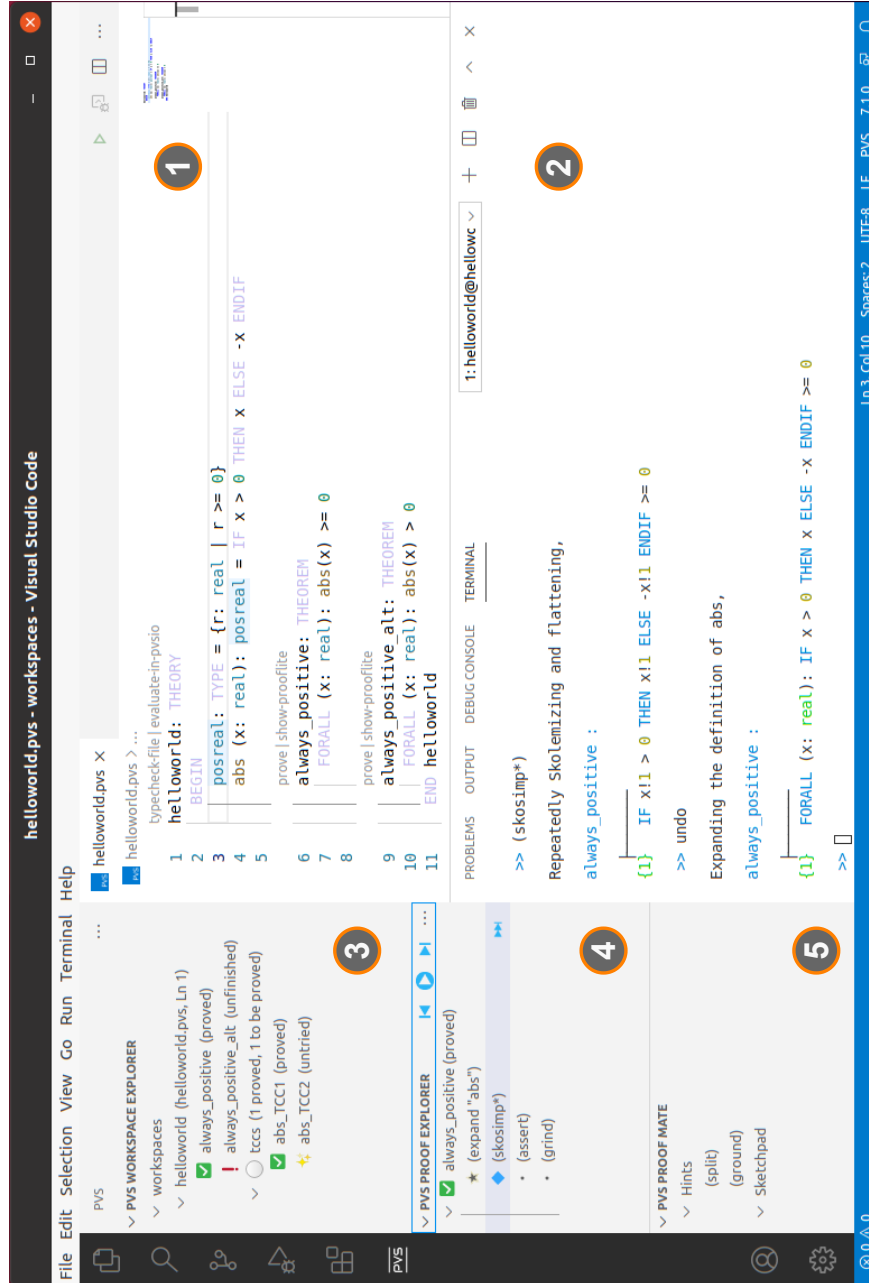


Figure B.2: Screen shot of the VSCode-PVS extension. (1) Editor, (2) Theorem Prover terminal, (3) Theory overview, (4) Proof Explorer and (5) Proof Mate.

## B.4 Large Version of Isabelle Snippet

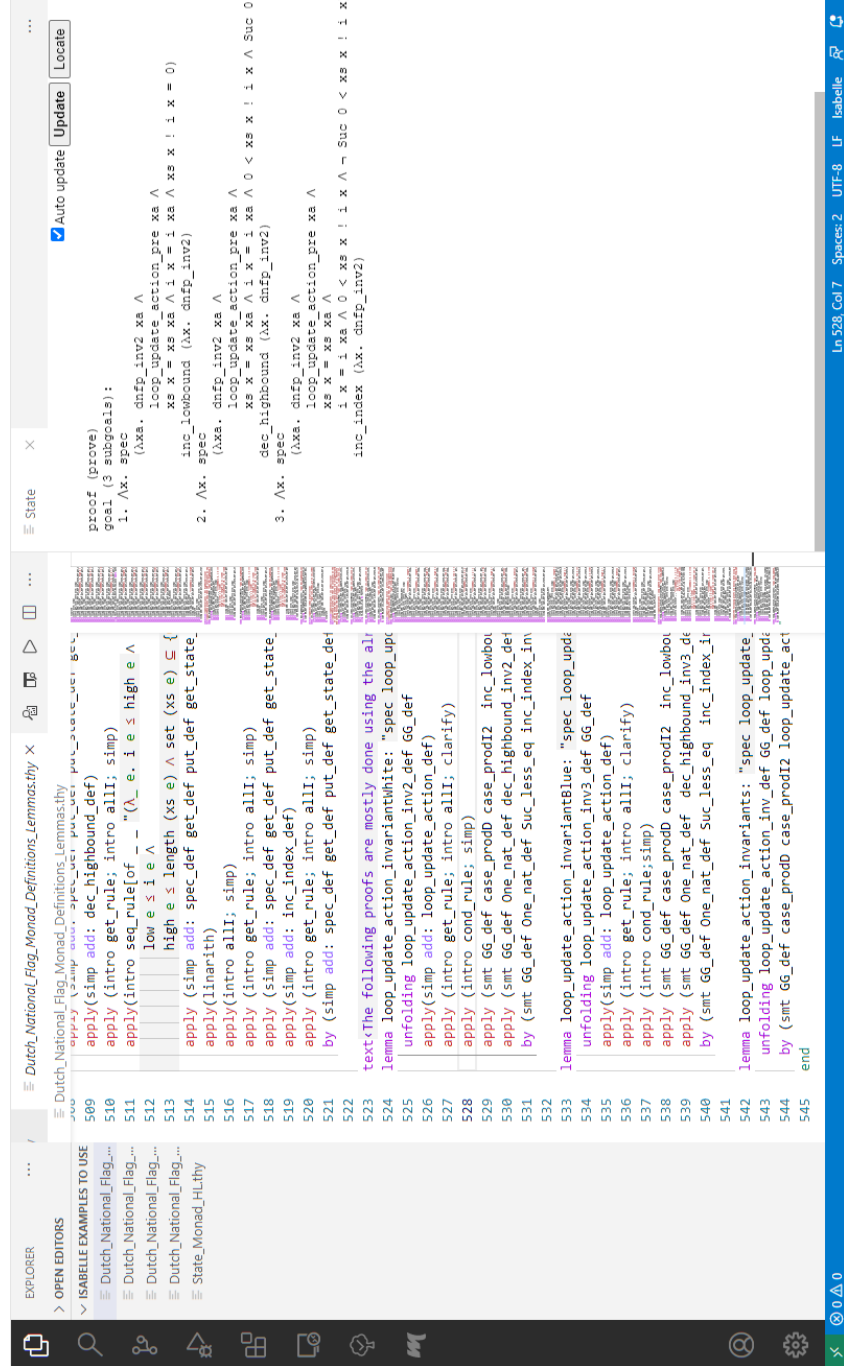


Figure B.3: Snippet of theorem proving in the Isabelle VS Code extension.





## Appendix C

# Extension User Guide

This chapter presents an overview of the GUI for the VS Code extensions for VDM in Appendix C.1 and provides examples of use by presenting representative user tasks in Appendix C.2.

### C.1 The Interface

Following is a description of the view elements composing the GUI of the VS Code VDM-SL extension (the GUI is identical for all the VDM dialects) as illustrated in Figure C.1:

- 1: The editor where the user can view the specification with syntax highlighting, errors and warnings are underlined. Furthermore, the user is able to access the context menu to translate the specification to LaTeX or Word formats and start POG.
- 2: The debug console which provides a CLI to the VDMJ debugger enabling the user to issue commands during a debugging session.
- 3: The POG view which enables the user to view and interact with POs. This interaction includes filtering, expanding meta-data to show the actual PO and go-to functionality.
- 4: The CT view where the user can view and execute traces present in the specification. This includes viewing trace and test status, generating tests from traces, filtering visible tests, filtered trace execution and send to interpreter functionality.
- 5: The test sequence results view enables the user to view the results of a given test executed in a trace.

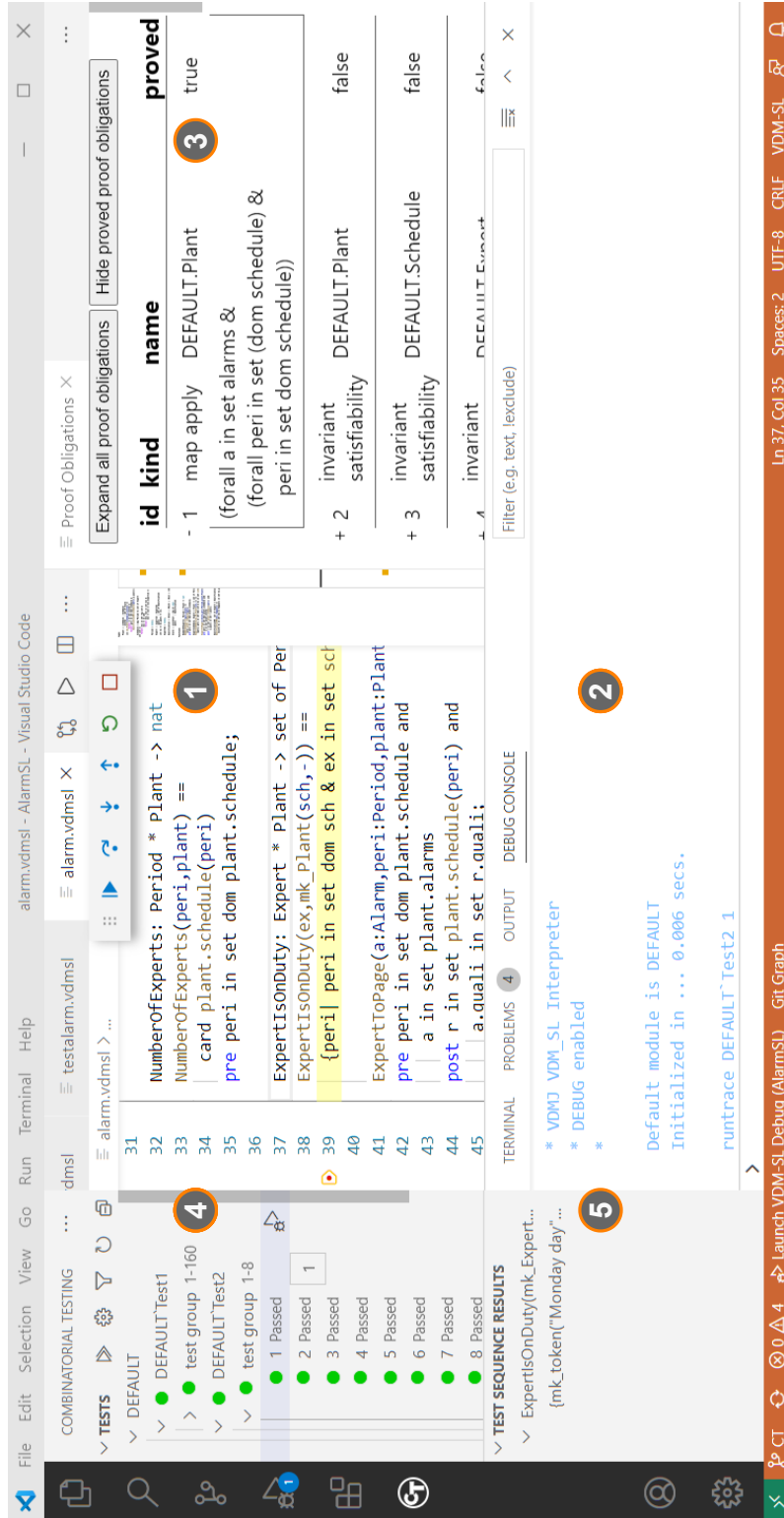


Figure C.1: View elements of the VS Code VDM-SL extension GUI.

## C.2 Example Use

This section showcases interactions related to the feature functionality available in the VS Code extension. This is carried out through a set of representative tasks for a given user. The tasks are represented by a *goal* and a *workflow*.

**Goal:** View and interact with proof obligations of the specification.

**Workflow:** To view proof obligations for a specification two actions exists. The user can right click in the view of the specification to access the context menu and click the ‘Run Proof Obligation Generation’ menu item. Alternatively the user can right click in the editor of a specification file in the explorer window and click the ‘Run Proof Obligation Generation’ menu item. Any of these actions displays a new window in the GUI that enables the user to view all POs for the specification listed by their meta-data in a table. The user can then interact with any of the PO meta-data by clicking it to expand and display the actual PO. By double-clicking a PO, the user is able to go to the location of the PO in the specification. Furthermore, the user can filter proved POs and expand all POs by pressing the ‘Hide proved proof obligations’ button and the ‘Expand all proof obligations’ button, respectively. The latter button also enables the user to collapse all expanded POs.

**Goal:** View traces for a specification, execute a trace and view its test results.

**Workflow:** To view traces for the specification the user has to navigate to the CT menu icon in the activity bar. This allows the user to generate a trace outline in the combinatorial testing view. Clicking the button ‘Generate trace outline’ displays a tree view of the traces present in the specification. From here the user is able to hover on a given trace and choose to either do a full evaluation, filtered evaluation, generate tests or go to the location of the trace in the specification. Choosing a full evaluation generates the tests from the trace and executes them. While executing, the user is able to view the progress of the test execution in a notification-bar view which is visible until either all tests have been executed or the user cancels the execution. Tests and their results are visible to the user in batches as they are finished executing. The user is furthermore able to interact with a finished test by clicking it to view its test results in the separate ‘Test sequence results’ view.

**Goal:** View errors and warnings for the specification.

**Workflow:** The user can use the problem view to get an overview of any syntax and type errors and warnings that are present in the specification. From here the user is able to click a warning or error and go to the location in the specification. Alternatively warnings and errors are highlighted for the user directly in the specification by a red and yellow line respectively.

**Goal:** Translate specification to LaTeX or Word format.

**Workflow:** The user is able to right click in the editor window to view the context menu. This enables the user to chose between the ‘Translate to LaTeX’ and ‘Translate to Word’ menu items. Clicking either of these translates the specification to the given format and allows the user to view the resulting files in an associated folder generated in the root folder of the project.

**Goal:** Initiate debugging of a specification and traverse a data structure.

**Workflow:** The user navigates to the debugger icon in the activity bar. From here the user can initiate a debug session by clicking ‘Start debugging’. This displays a debug terminal window from which the user can interact directly with the VDMJ debugger by issuing commands. To traverse a data structure the user must pause the execution of the specification. The user can then start the debugging of the specification by issuing the ‘print’ command in the debugger terminal for the relevant operation or function. If breakpoints have been applied the debugger stops when reaching the first break-point and the user is able to step-into, step-over and step-out of elements of the specification using the debug interface available in the editor window.

## Appendix D

# Assessing Performance

### D.1 Test Traces

Listing D.1 shows a simple VDM-SL specification file which defines seven traces:

- **Test1**: Generates 1,000 test cases.
- **Test2**: Generates 10,000 test cases.
- **Test3**: Generates 100,000 test cases.
- **Test4**: Generates 300,000 test cases.
- **Test5**: Generates 500,000 test cases.
- **Test6**: Generates 800,000 test cases.
- **Test7**: Generates 1,000,000 test cases.

The tests allows for a comparison of performance between the VS Code extension, the VDMJ CLI and the Overture IDE in terms of test execution time for varying test case sizes. As can be seen in Listing D.1, the tests are extremely simple. However, this is intentional as the comparison of interest is between the VS Code extension and the VDMJ CLI to test the impact on performance caused by the SLSP protocol communication. If any difference is observed in execution time between the VS Code extension and the VDMJ CLI it potentially stems from a messaging overhead introduced by the SLSP protocol. The overhead will be exposed by a large number of simple tests as this requires a proportionally large amount of protocol communication.

Table D.1 shows the measured execution times for the simple test traces when executed by the VDMJ CLI, the Overture IDE and the VS Code extension.

In addition to the simple test traces, realistic traces are also used to measure any performance difference between the the VS Code extension and the VDMJ CLI in terms of trace execution time. These are the traces `First1000`, `First10k`, `First100k`, `First300k`, `First500k` and `First1000k` detailed in Appendix D.4. Table D.2 shows the measured execution times for the traces when

## APPENDIX D. ASSESSING PERFORMANCE

```

functions
  testFunc: nat -> nat
  testFunc(x) == x;

traces
Test1:
  let a in set {1, ..., 1000} in
    testFunc(a);

Test2:
  let a in set {1, ..., 10000} in
    testFunc(a);

Test3:
  let a in set {1, ..., 100000} in
    testFunc(a);

Test4:
  let a in set {1, ..., 300000} in
    testFunc(a);

Test5:
  let a in set {1, ..., 500000} in
    testFunc(a);

Test6:
  let a in set {1, ..., 800000} in
    testFunc(a);

Test7:
  let a in set {1, ..., 1000000} in
    testFunc(a);

```

Listing D.1: Traces used for measuring trace execution time.

Platform	Test1 (1k)	Test2 (10k)	Test3 (100k)	Test4 (300k)	Test5 (500k)	Test6 (800k)	Test7 (1000k)
VDMJ CLI	0.5	3	23	68	116	185	233
Overture IDE	0.5	1	4	11	19	29	37
VS Code extension	0.5	3	26	78	134	217	274

Table D.1: Comparison of time taken (in seconds) for executing the same specific set of traces in the VDMJ CLI, the Overture IDE and the VS Code extension. The numbers in the parentheses indicate the number of test cases generated from the trace. Each result is the average of three runs.

## APPENDIX D. ASSESSING PERFORMANCE

executed by the VS Code extension, the VDMJ CLI and the Overture IDE respectively.

Trace	VS Code	VDMJ CLI	Overture IDE
First1000	<1	<1	<1
First10k	4	3	3
First100k	38	36	27
First300k	124	116	N/A
First500k	210	196	N/A
First1000k	440	402	N/A

Table D.2: Comparison of time taken (in seconds) for executing the same traces in the VS Code extension, the VDMJ CLI and the Overture IDE. ‘N/A’ means that the trace could not be executed to completion. The comparison was performed with realistic traces using the Luhn algorithm as detailed in the VDM model shown in Appendix D.4. Each result is the average of three runs.

### D.2 Performance Profiles

Following are the performance profiles obtained from executing traces using the VS Code extension and the VDMJ CLI respectively. Figure D.1 shows the CPU times of functions for the VDMJ process when the trace `Test3` is executed using the VS Code extension. Figure D.2 shows the CPU times of functions for the VDMJ process when the trace `Test3` is executed using the the VDMJ CLI. Comparing the profiles of identical functions it is clear that the functions use slightly more CPU time in the VS Code server. Furthermore, it is shown that the `send()` function that sends the results to the client uses 9% of the total CPU time.

## APPENDIX D. ASSESSING PERFORMANCE

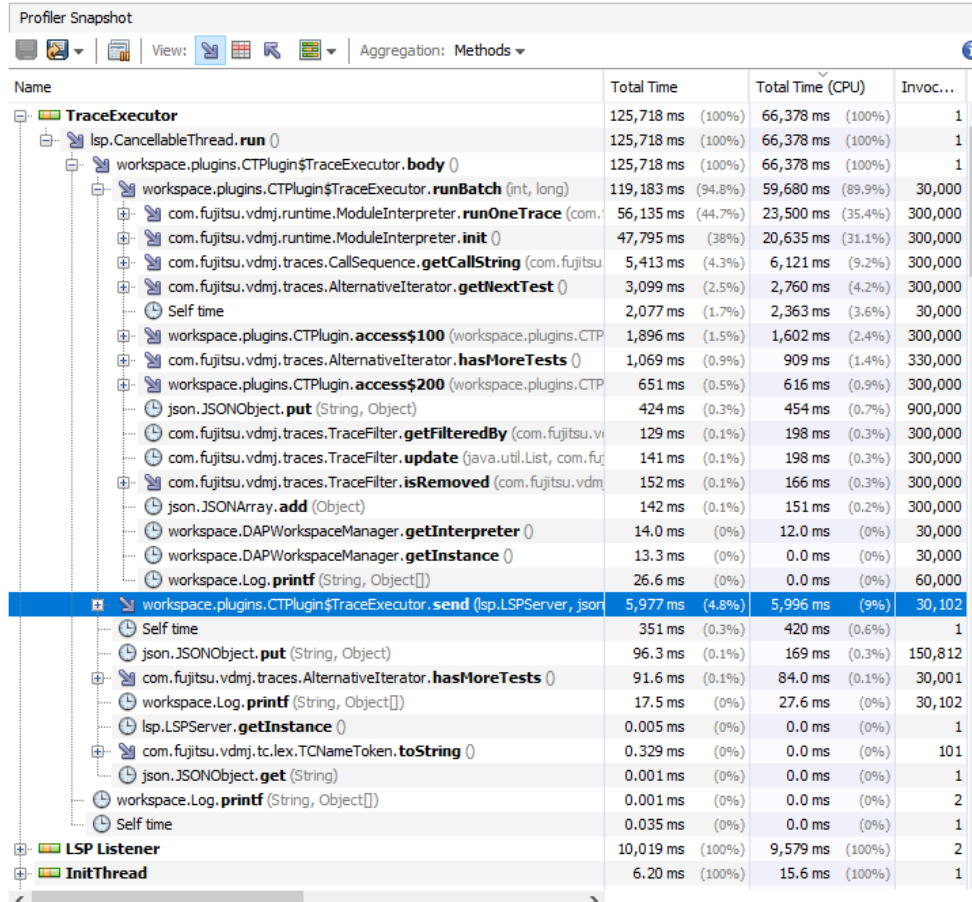


Figure D.1: Performance profile of VDMJ executing the trace Test 3, executed by the VS Code extension.



## APPENDIX D. ASSESSING PERFORMANCE

Profiler Snapshot

View: Aggregation: Methods

Name	Total Time	Total Time (CPU)	Invoc...
com.fujitsu.vdmj.runtime.Interpreter.runtrace (String, int, int, boolean, fl...	88,944 ms (100%)	31,415 ms (100%)	1
<b>com.fujitsu.vdmj.runtime.ModuleInterpreter.runOneTrace (com.fujitsu...</b>	<b>37,177 ms (41.8%)</b>	<b>12,563 ms (40%)</b>	<b>300,000</b>
com.fujitsu.vdmj.runtime.ModuleInterpreter.init ()	33,026 ms (37.1%)	11,963 ms (38.1%)	300,001
com.fujitsu.vdmj.messages.ConsolePrintWriter.println (String)	14,643 ms (16.5%)	3,021 ms (9.6%)	600,002
com.fujitsu.vdmj.in.definitions.INNamedTraceDefinition.getIterator (c...	1,323 ms (1.5%)	1,161 ms (3.7%)	1
Self time	904 ms (1%)	894 ms (2.8%)	1
com.fujitsu.vdmj.traces.CallSequence.getCallString (com.fujitsu.vdmj...	771 ms (0.9%)	778 ms (2.5%)	300,000
com.fujitsu.vdmj.runtime.ModuleInterpreter.getTraceContext (com.fi...	432 ms (0.5%)	709 ms (2.3%)	300,001
com.fujitsu.vdmj.traces.AlternativeIterator.getNextTest ()	391 ms (0.4%)	338 ms (1.1%)	300,000
com.fujitsu.vdmj.traces.AlternativeIterator.count ()	41.3 ms (0%)	37.1 ms (0.1%)	1
com.fujitsu.vdmj.lex.LexStreamReader.read (char[])	0.054 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.lex.LexTokenReader.init ()	0.020 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.lex.LexTokenReader.nextToken ()	0.145 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.runtime.Interpreter.\$1.<clinit> ()	0.025 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.runtime.ModuleInterpreter.getDefaultName ()	0.021 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.runtime.ModuleInterpreter.findTraceDefinition (com...	0.066 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.traces.TraceFilter.getFilteredCount ()	0.002 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.messages.ConsolePrintWriter.print (String)	0.119 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.traces.AlternativeIterator.hasMoreTests ()	134 ms (0.2%)	0.0 ms (0%)	300,000
com.fujitsu.vdmj.traces.TraceFilter.isRemoved (com.fujitsu.vdmj.trac...	21.5 ms (0%)	0.0 ms (0%)	300,000
com.fujitsu.vdmj.traces.TraceFilter.getFilteredBy (com.fujitsu.vdmj.tr...	23.3 ms (0%)	0.0 ms (0%)	300,000
com.fujitsu.vdmj.traces.TraceFilter.update (java.util.List, com.fujitsu.v...	25.7 ms (0%)	0.0 ms (0%)	300,000
com.fujitsu.vdmj.values.IntegerValue.toString ()	27.2 ms (0%)	0.0 ms (0%)	300,000
com.fujitsu.vdmj.commands.CommandReader.println (String)	0.031 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.messages.RTLogger.getLogSize ()	0.008 ms (0%)	0.0 ms (0%)	1
Self time	0.713 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.commands.CommandReader.prompt ()	0.033 ms (0%)	0.0 ms (0%)	1
com.fujitsu.vdmj.commands.CommandReader.readLine ()	0.0 ms (0%)	0.0 ms (0%)	1
CTMainThread	0.097 ms (100%)	15.6 ms (100%)	1
CTMainThread	0.071 ms (100%)	15.6 ms (100%)	1
InitThread	0.232 ms (100%)	0.0 ms (-%)	1
CTMainThread	16.3 ms (100%)	0.0 ms (-%)	1
InitThread	0.034 ms (100%)	0.0 ms (-%)	1
CTMainThread	0.145 ms (100%)	0.0 ms (-%)	1

Figure D.2: Performance profile of VDMJ executing the trace Test 3, executed by the CLI.

### D.3 Interpreter Execution Time

This section validates that the VDMJ interpreter is slower when executed by the language server compared to executing it through the CLI. This is validated by executing the operation in Listing D.2 and comparing execution times between the CLI and the Language Server Interface (LSI). The execution times are found in Table D.3.

	CLI	LSI
Run 1	5.968	6.285
Run 2	5.953	6.246
Run 3	5.998	6.299
<b>Average</b>	<b>5.973</b>	<b>6.277</b>

Table D.3: Execution times (in seconds) as a result of executing the same operation (Listing D.2) three times per interface.

As evident from the execution times in Table D.3 the VDMJ interpreter is slightly slower when executed through the LSI compared to execution through the CLI.

```
work: nat ==> ()
work(n) ==
  for a = 1 to n do
    for b = 1 to n do
      let - = luhns("276328483246826343264873264832686482364826")
    in skip;
```

Listing D.2: Test operation that uses the LUHN algorithm.

## D.4 LUHN.vdmsl - A Realistic Specification

```

/**
 * A specification of the Luhn check digit algorithm.
 */
types
  Digit = nat          -- A decimal digit, 0-9
  inv d == d < 10;

functions
  luhn: seq1 of Digit -> Digit -- Non empty list input
  luhn(data) ==
    total(data) * 9 mod 10;

  -- Convenience function for "12345"
  luhns: seq1 of char -> Digit
  luhns(number) ==
    luhn(strToSeq(number));

  -- Convenience function for numbers
  luhnn: nat -> Digit
  luhnn(number) ==
    luhn(natToSeq(number));

  total: seq of Digit -> nat
  total(data) ==
    if data = []
    then
      0
    else
      let multiplier = (len data) mod 2 + 1,
          product = hd data * multiplier
      in
        total(tl data) + -- recurse
          if product < 10
          then product
          else (product mod 10) + 1
  measure len data;

  strToSeq: seq1 of char -> seq1 of Digit
  strToSeq(s) ==
    [ cases i :
      '0' -> 0, '1' -> 1, '2' -> 2, '3' -> 3, '4' -> 4,
      '5' -> 5, '6' -> 6, '7' -> 7, '8' -> 8, '9' -> 9
    end | i in seq s];

```

Listing D.3: The LUHN algorithm specified in VDM-SL.

## APPENDIX D. ASSESSING PERFORMANCE

```
natToSeq: nat -> seq of Digit
natToSeq(n) ==
  if n < 10
  then [n]
  else natToSeq(n div 10) ^ [n rem 10]
measure n;

operations
work: nat ==> ()
work(n) ==
  for a = 1 to n do
  for b = 1 to n do
    let - = luhns("276328483246826343264873264832686482364826")
    in skip;

traces
/**
 * Generate all the possible 1, 2 and 3 digit sequences and check
 * that the luhn
 * calculation completes without breaking any constraints.
 */
First1000:
  let a,b,c in set {0,...,9} in
  (
    luhn([a]);
    luhn([a,b]);
    luhn([a,b,c])
  );

First10k:
  let a,b,c,d in set {0,...,9} in
  (
    luhn([a]);
    luhn([a,b]);
    luhn([a,b,c]);
    luhn([a,b,c,d])
  );

First100k:
  let a,b,c,d,e in set {0,...,9} in
  (
    luhn([a]);
    luhn([a,b]);
    luhn([a,b,c]);
    luhn([a,b,c,d]);
    luhn([a,b,c,d,e])
  );
```

## APPENDIX D. ASSESSING PERFORMANCE

```
First300k:
  let a,b,c,d,e in set {0,...,9} in
  let f in set {0,...,2} in
  (
    luhn([a]);
    luhn([a,b]);
    luhn([a,b,c]);
    luhn([a,b,c,d]);
    luhn([a,b,c,d,e]);
    luhn([f,a,b,c,d,e])
  );

First500k:
  let a,b,c,d,e in set {0,...,9} in
  let f in set {0,...,4} in
  (
    luhn([a]);
    luhn([a,b]);
    luhn([a,b,c]);
    luhn([a,b,c,d]);
    luhn([a,b,c,d,e]);
    luhn([f,a,b,c,d,e])
  );

First1000k:
  let a,b,c,d,e,f in set {0,...,9} in
  (
    luhn([a]);
    luhn([a,b]);
    luhn([a,b,c]);
    luhn([a,b,c,d]);
    luhn([a,b,c,d,e]);
    luhn([a,b,c,d,e,f])
  );

/**
 * The Luhn algorithm will detect any single-digit error, as well
 * as almost all
 * transpositions of adjacent digits. It will not, however,
 * detect transposition
 * of the two-digit sequence 09 to 90 (or vice versa).
 *
 * See http://en.wikipedia.org/wiki/Luhn\_algorithm
 */
AllOneDigitErrors:
  let input = [7,9,9,2,7,3,9,8,7,1] in
  let pos in set inds input in
  let replacement in set {0,...,9} \ {input(pos)} in
  let corrupt = input(1,...,pos-1) ^ [replacement] ^ input(pos
    +1,...,len input) in
  checkFail(corrupt, 3);
```

## APPENDIX D. ASSESSING PERFORMANCE

```

AllAdjacentTranspositions:
  let input = [7,9,9,2,7,3,9,8,7,1] in
  let pos in set inds tl input be st -- ie. one less than the
    length
    input(pos+1) <> input(pos)
    and {input(pos+1), input(pos)} <> {0,9} in
  let replacement = [input(pos+1), input(pos)] in
  let corrupt = input(1,...,pos-1) ^ replacement ^ input(pos
    +2,...,len input) in
  checkFail(corrupt, 3);
/**
 * It will detect 7 of the 10 possible twin errors (it will not
 * detect
 * 22 <> 55, 33 <> 66 or 44 <> 77).
 *
 * See http://en.wikipedia.org/wiki/Luhn\_algorithm
 */
AllTwinErrors:
  let input = [1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,0,0] in
  let pos in set inds tl input be st input(pos) = input(pos+1) in
  let rep in set {0,...,9} \ {input(pos)} in
  let corrupt = input(1,...,pos-1) ^ [rep, rep] ^ input(pos
    +2,...,len input) in
  checkFail(corrupt, 0);

/**
 * Because the algorithm operates on the digits in a right-to-
 * left manner and zero
 * digits affect the result only if they cause shift in position,
 * zero-padding the
 * beginning of a string of numbers does not affect the
 * calculation.
 *
 * See http://en.wikipedia.org/wiki/Luhn\_algorithm
 */
ZeroPadding:
  let input = [7,9,9,2,7,3,9,8,7,1] in
  let number in set {1, ..., 10} in
  let padding = [p-p | p in set {1, ..., number}] in
  checkOK(padding ^ input, 3);

operations
/**
 * These operations support the traces above
 */
checkFail: seq1 of Digit * Digit ==> bool
checkFail(data, expected) ==
  return luhn(data) <> expected -- Expect failure!
post RESULT = true;

checkOK: seq1 of Digit * Digit ==> bool
checkOK(data, expected) ==
  return luhn(data) = expected -- Expect success!
post RESULT = true;

```