AARHUS
UNIVERSITY
SCIENCE AND TECHNOLOGY

AN OPEN-SOURCE WEB IDE FOR VDM-SL

BY

RASMUS S. REIMER
201181008

AND

KASPER D. SAABY
20094357

MASTER'S THESIS
IN
COMPUTER ENGINEERING

SUPERVISOR: PROF. PETER GORM LARSEN

Aarhus University School of Engineering

9th of May 2016

Rasmus S. Reimer	Kasper D. Saaby	Peter Gorm Larsen
201181008	20094357	Supervisor

Abstract

Integrated Development Environments (IDEs), are used in different types of software development. A variety of IDEs exists which supports traditional programming languages, such as Java and C#. In addition, during the past years, several IDEs have emerged as web services. These web IDEs presents opportunities, which includes reducing requirements of users local machines. However, the number of desktop IDEs and web IDEs which support modelling languages are limited.

This thesis project investigates how a web IDE can be developed, which supports the modelling language called VDM-SL. The investigations focus on reusing software provided by the growing open-source community. To this end, an open-source software evaluation method is applied, in order to select the most suitable software for reuse. In addition, a pilot study is conducted as part of the evaluations, which provide further insight into the feasibility of the software reuse approach.

The pilot study shows, that the reuse-based approach is indeed feasible, but different software granularities should be considered. Additionally, the pilot study shows, that certain features of the software are particularly important when considering the software for reuse.
The experiences obtained during the pilot study have been used to derive guidelines for porting an existing desktop IDE to a web IDE.

Acknowledgements

This thesis constitutes the final project of our Master's degree in Computer Engineering at Aarhus University. We would like to thank our academic supervisor Peter Gorm Larsen, for providing valuable advice and feedback during this process.
In addition, we would like to thank the researchers in the Overture core group, who have provided valuable feedback on the web IDE, which has been developed as part of this thesis project.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the selected thesis subject, along with the concepts of software reuse, open-source software, web services, and integrated development environments. These concepts provide the basis for this thesis, and are used to motivate the selected subject and the selected approach. Chapter 2 extends on the concepts presented in this chapter, by providing additional theory.

1.1. Overview

A software development project often follows a well-defined process, which defines a set of activities. These activities are defined to help guide the project, and ensure that certain artefacts are produced. Different processes exist, but many include four fundamental activities [1]. These activities include creating a software specification, which describes the functionality and constraints for the software. Then, the software is designed and implemented according to the specification. Next, the software is validated to ensure that the customer's needs have been met by the resulting software. Finally, it is defined how the software shall be maintained and updated while in operation.

In addition to these activities, researchers have been advocating the use of formal methods for many years [1]. With formal methods, a model of the software is defined, which can be formally analysed. The user requirements can be translated into such a model, which then serves as a formal specification [1, 3]. The model is therefore an unambiguous description of what the system should do.

1.1.1 Software Reuse

Software may be developed from scratch during the development processes. However, a different approach is to use a reuse-based strategy, where the development process is geared toward maximizing reuse of existing software [1]. The possible benefits of this approach are lower software development and maintenance costs, faster delivery, and increased software quality [1]. On the other hand, selecting unsuitable software for reuse can have the reverse effect.

Reusable software units can be of radically different sizes [1]. Reusing a small software unit may only provide a single feature. However, it may be possible to reuse the software without any mod-

ifications, and only require a limited amount of effort to integrate the feature [1]. Reusing a large software unit instead, could provide many features. Large software units can sometimes be reused by re-configuring or extending its behaviour. Extensible software provides interfaces for injecting these extensions. However, the provided features can have other costs associated with them. The consequences of reusing large software units can include lack of control over features, problems with interoperability, lack of control over software evolution, and lack of developer support [1]. However, even if a specific software unit cannot be reused, it is still possible that the software can serve as inspiration regarding architectural (or design) decisions [1].

1.1.2 Web services

Web services have evolved from being a simple means of sharing information, to delivering specific functionality accessible over the web [4]. Web services have emerged which delivers Software as a Service (SaaS), where the software runs on a server, instead of a local machine. This means that software can be highly distributed. Additionally, this change in software organization has led to software reuse being a dominant approach to developing web services [1].

1.1.3 Open-source Software

A possible source of reuse is provided by the open-source movement, which has generated a large amount of reusable software available under different licensing terms. An added benefit of using open-source software is that often multiple developers will be maintaining the software and contributing with new features [1, 5]. However, selecting software to reuse can be challenging, which is why methods are emerging to help guide this process [6]. Open-source software differs in a number of parameters, including granularity, provided features, and software quality [5, 6]. This further complicates the selection process.

1.1.4 Software Granularity

Regarding code as the smallest unit of software reuse, the next level of (coarser) granularity can be defined as functions. Functions provide a specific functionality, which can be executed from other functions. The functionality provided can e.g. be an implementation of an *algorithm*. An algorithm is a well-defined description, of how to solve a specific problem, such as sorting an array of integer values.

The Object-Orientated Programming (OOP) paradigm provides the concept of classes and objects. A class is a collection of coherent methods for manipulating the object's state. Methods and functions are closely related concepts, but in this thesis, when a function is defined inside a class, the function is referred to as a method. Classes can inherit from other concrete or abstract classes, and can thus extend or override functionality defined in the inherited class. Some programming languages include the concept of interfaces. Interfaces define methods which shall be provided by any class implementing the interface.

Libraries are defined as a set of integrated objects, which are used as utilities for accomplishing intermediate programming tasks. A *component* is defined as a set of integrated objects, libraries, and possibly other components, which provide a specific feature. With these definitions in place, an *application* is defined as a set of integrated objects, libraries, and components, which provides a set of coordinated features for accomplishing a variety of tasks.

1.1.5 Development Environment

Software can be developed using only a text editor for writing code, and a command-line tool for compiling or interpreting the code. However, developing software typically involve many different tasks, using different tools [7]. Among these tasks are writing, executing, and debugging the code. Writing tests is also an important part of software development [8]. The tests are often executed using third-party tools. Additionally, more high-level tasks can be necessary, such as version control and deployment of the software.

1.1.5.1 Integrated Development Environment

To avoid having to learn a variety of incoherent tools, an *Integrated Development Environment (IDE)* can be used [7]. IDEs are designed to facilitate either programming, modelling, or both through the integration of assisting tools in a coherent *Graphical User Interface (GUI)* [9]. However, a particular IDE usually only supports a small number of languages, restricting the choice of which IDE can be used in the development. The features provided by an IDE can include an editor, a file explorer, debugging integration, code execution or model evaluation, and support for testing [9]. In this thesis, an *editor* is defined as a component for text editing. Additionally, the editor maintains an edit history and provides interfaces for integrating features that can manipulate and format the text.

A large number of commercial and open-source IDEs exists, among which are Visual Studio[1], IntelliJ[2], Eclipse[3], and XCode[4]. Common for these IDEs is that they are all *desktop IDEs*. Hence, software have to be installed on the user's local machine [9]. Additionally, desktop IDEs can have a large resource demand regarding memory and CPU, which places certain demands on the local machine[5].

1.1.5.2 IDE for Modelling

As mentioned above, a model can be created as part of the development process, that serves as a formal specification. The model is expressed using formally defined semantics, making it closely related to traditional programming languages. The process of developing a model can therefore be supported by having similar features to those used for traditional programming languages. Desktop IDEs already exists for modelling languages that provide many of such features, as described in section 2.5. The Eclipse-based Overture Tool is one of these desktop IDEs. This desktop IDE supports the VDM languages by integrating the Overture core. The Overture core includes a parser, type checker, and an interpreter for evaluating VDM models.

Certain features exist for both modelling and programming languages. Sometimes these features use a naming convention, which relates mostly to the programming languages, such as code completion. However in this thesis, when the feature is related to modelling languages, the name used for programming is used to avoid altering the usual understanding of the feature.

[1] See `https://www.visualstudio.com/`, accessed 26th of April 2016.

[2] See `https://www.jetbrains.com/idea/`, accessed 26th of April 2016.

[3] See `https://eclipse.org/ide/`, accessed 26th of April 2016.

[4] See `https://developer.apple.com/xcode/`, accessed 26th of April 2016.

[5] See `https://www.visualstudio.com/en-us/visual-studio-2015-system-requirements-vs#3`, accessed 7th of April 2016.

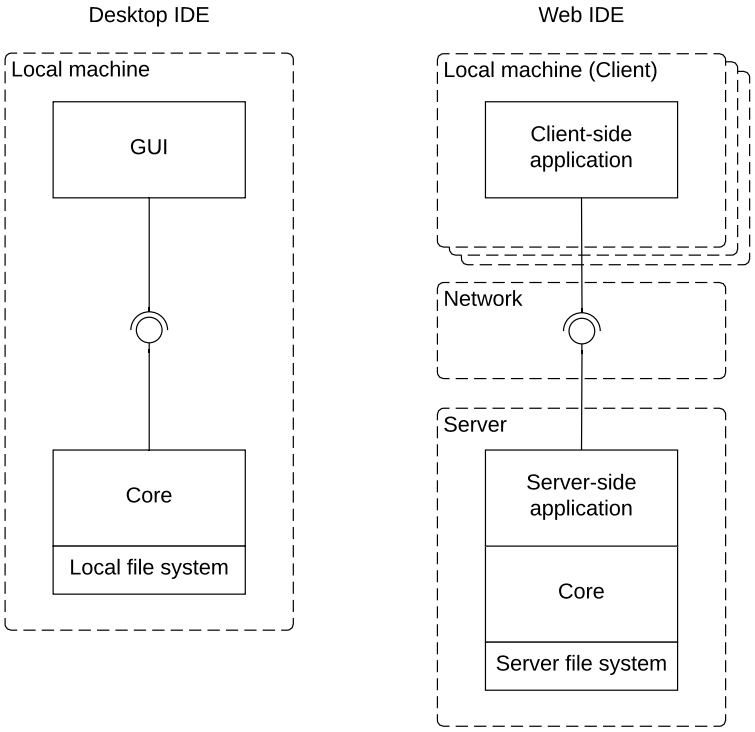


Figure 1.1: Illustration of the difference between a desktop IDE and a web IDE.

1.1.5.3 IDE as a Web Application

During the past years, several *web IDEs* have emerged [7]. These web IDEs provide developers with a development environment on any machine, provided that the machine has internet access. This makes the amount of hardware resources available on the local machine less relevant since most processing is handled on a server [9]. Additionally, no software need to be installed on the local machine, except for a web browser.

Figure 1.1 illustrates an important difference between web IDEs and desktop IDEs. A web IDE is subject to the client-server model inherent in web applications. The *client-side application* runs in a browser on a client and provides the web IDE's GUI. The *server-side application* runs on a server and provides the web IDE's core features. Hence, the client-server model introduces the network layer [9]. A desktop IDE does not include the network layer since the GUI and core are co-located. Hence, a web IDE is subject to additional security and Quality of Service (QoS) challenges compared to a desktop IDE.

1.1.6 Chapter structure

The remaining part of this chapter is organized as follows. Section 1.2 presents the motivation for this thesis. In section 1.3, the hypothesis and goals are presented. Then, section 1.4 presents the scope of this thesis project and expectations for the reader. Section 1.5 presents the applied approach for testing the defined hypothesis and achieving the defined goals. In section 1.6, a reading guide is presented. Lastly, section 1.7 presents the structure of this thesis.

1.2. Motivation

The reuse-based approach to software development, combined with the large amount of open-source software, presents a variety of opportunities. These opportunities include developing software rapidly and of high quality [1, 5]. Additionally, supplying the resulting software as open-source enables maintenance and support to be provided by a diverse community [1].

An IDE is a source of productivity within software development [9]. Providing an IDE as a web application can be beneficial for both students and professionals.
As mentioned above, web IDEs can be less demanding regarding hardware resources, than some desktop IDEs. In addition, the development environment can be delivered preconfigured and with all dependencies installed.
A web IDE can also be used for educational purposes, such as a programming course [9, 7]. Web-based programming assignments can be developed, that combines the web IDE with the course's teaching material. Teachers can formalize programming assignments and include test cases. The students can use the test cases to validate their solutions of the assignments. Hence, teachers can formalize, grade, and provide feedback on the assignments using the web IDE [9].
A web IDE enables additional features to be added, that would be less accessible through traditional desktop IDEs. These features include real-time collaboration and code sharing [9]. Additionally, making changes and upgrading the web IDE can be cheaper since changes are automatically rolled out to the users when the web IDE is re-deployed [1].
Open-source web IDEs already exists that can be extended to support additional programming languages. Furthermore, open-source software exists, which provide one or more relevant features for web IDEs.

This thesis project explores the possibility of developing a web IDE using a reuse-based software development approach. Most existing web IDEs only support programming languages and not modelling languages [9, 7]. Therefore, the focus of this thesis project is on supporting the modelling language VDM-SL, which is introduced in section 2.5. During this process, a method for selecting relevant open-source software is assessed. The end result is a set of guidelines for developing a web IDE.

1.3. Hypothesis and Goals

Hypothesis: It is possible to develop a web IDE supporting VDM-SL using a reuse-based development approach and open-source software.

Testing this hypothesis involves integration with the Overture core which is a large and complex library. It is unknown how well the Overture core performs in a multi-user environment. In addition to testing the defined hypothesis, a set of subsequent goals have been defined which are addressed.

Goal 1: Explore the possibilities and limitation of integrating the Overture core in a web IDE.

Goal 2: Investigate whether a method can be used to assess the feasibility of reusing a particular open-source software unit.

Goal 3: Provide guidelines for the development of web IDEs using a reuse-based approach.

1.4. Scope

The scope of this thesis project is limited to gathering experience in the development of a web IDE supporting VDM-SL. However, the resulting guidelines and experiences are kept sufficiently general, to be applicable for the development of web IDEs supporting other programming and modelling languages.

This scope requires an understanding of the challenges involved in selecting and reusing open-source software. Additionally, a variety of web technologies are explored, to address these challenges, be productive, and ensure long-term maintainability. Since the integration of VDM-SL is realized using the Overture core, it is necessary to study the Overture core as well. Therefore, an evaluation of the integration with the Overture core is also provided.

It is expected that the reader of this thesis has a basic understanding of traditional programming languages, such as Java. Additionally, the reader should possess basic knowledge on HyperText Transfer Protocol (HTTP) communication including Uniform Resource Locators (URLs). Lastly, a basic understanding of concurrency and related issues, is also expected.

1.5. Approach

The applied approach for testing the defined hypothesis is described below.

Literature study: In order to select features that a web IDE supporting VDM-SL should include, a literature study is conducted, along with investigating a set of existing desktop and web IDEs. Additionally, to select appropriate open-source software for reuse, a method for aiding the selection process is studied.

Discovering and rating open-source software: An internet search is conducted to discover suitable open-source software units of different granularities. The features defined through the literature study are used as criteria on which to rate the open-source software.

Pilot study: The best rated open-source software is used in a pilot study, in order to gain deeper insight into the feasibility of the selected approach, regarding software granularity. Additionally, the pilot study also helps to assess how the Overture core performs in a multi-user environment.

Evaluation: The result of the pilot study is evaluated, focusing on the advantages and disadvantages of using open-source software. Additionally, any issues concerning the integration of the Overture core is discussed.

Generalization: The approach for evaluating open-source software and the result of the pilot study is used to derive a set of guidelines. These guidelines covers both the selection of open-source software and development of web IDEs.

1.6. Reading Guide

This section presents the formats and styles that are used throughout this thesis.

Diagrams
Diagrams provided in this thesis do not follow a predefined standard. The diagrams are kept simple and illustrative to support the readers understanding of the text, without assuming any prior knowledge of modelling standards.

Emphasis
Emphasized sentences and words are written in *italic*.

Footnotes
A footnote is attached to the sentence or paragraph to which it refers, as a superscript integer. The associated resource is found at the bottom of the page, on which the superscript integer is provided.

Numbering
Figures and tables are numbered using the chapter number and the number of the figure or table in that chapter, e.g. Figure 2.1 refers to the 1st figure in chapter 2.

Radar charts
The evaluation results are presented in either a table as seen in Table 1.1 or using a radar chart as seen in Figure 1.2. The radar chart contains a dimension for each criterion. A coloured line is drawn between adjacent dimensions, representing a subject under evaluation. The further a coloured line reaches out on a dimension, the better the rating is on that criterion. Each line has a unique thickness to be able to see lines on top of each other.

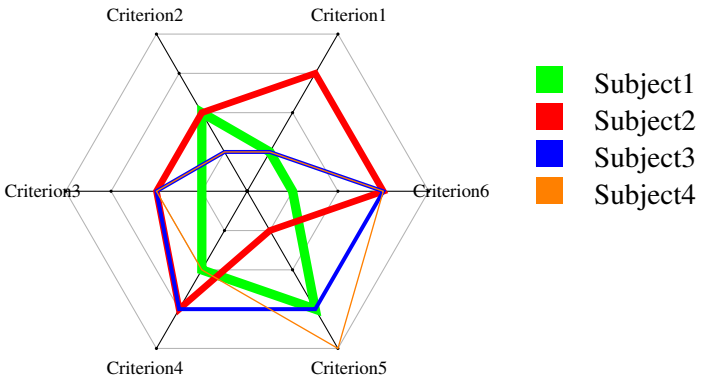


Figure 1.2: Radar chart example.

References
Reference to cited literature is referred to using an integer inside square brackets e.g. [1], which correspond to an entry in the bibliography.

Tables
Tables are used to a large extend to present IDE criteria, and all these tables follow the format seen in Table 1.1 below. The leftmost column contains the criteria, where each criterion represents an IDE feature. For each criterion, two to four ratings are defined. A rating is defined using an explanatory text, which describes what is required to achieve the particular rating, on the specific

criterion. Higher ratings always includes the lower ratings of a criterion.

Table 1.1: Criteria description example.

Criterion	**Rating**			
	-	**+**	**++**	**+++**
Criterion1	Description1a	Description1b	Description1c	Description1d
Criterion2	Description2a	Description2b		
...	...	...	...	...
CriterionN	DescriptionNa	DescriptionNb	DescriptionNc	

1.7. Structure

The thesis structure is illustrated in Figure 1.3. Chapters are illustrated as solid boxes. The left-hand side of each swim lane describes the theme of the chapters contained in the swim lane. Arrows between the chapters indicate dependencies. The arrows start at a chapter which is depended on by the chapter pointed at.

Chapter 2: This chapter presents theory and concepts that are used throughout the thesis.

Chapter 3: This chapter presents the results of a literature study and a study of existing desktop and web IDEs, which is a set of criteria that is used in the subsequent evaluations.

Chapter 4: This chapter presents the evaluations of open-source software of different granularities, using the criteria defined in chapter 3. Additionally, the evaluation results are compared and discussed.

Chapter 5: This chapter presents the pilot study, in which the best rated open-source software from chapter 4 are applied.

Chapter 6: This chapter presents a set of guidelines for developing web IDEs using open-source software. The guidelines are derived from the knowledge gained throughout the thesis project.

Chapter 7: This chapter concludes the thesis, and discusses whether the hypothesis is true or false. Additionally, an evaluation of the defined goals is provided. Lastly, future work to improve the result of the pilot study is presented.

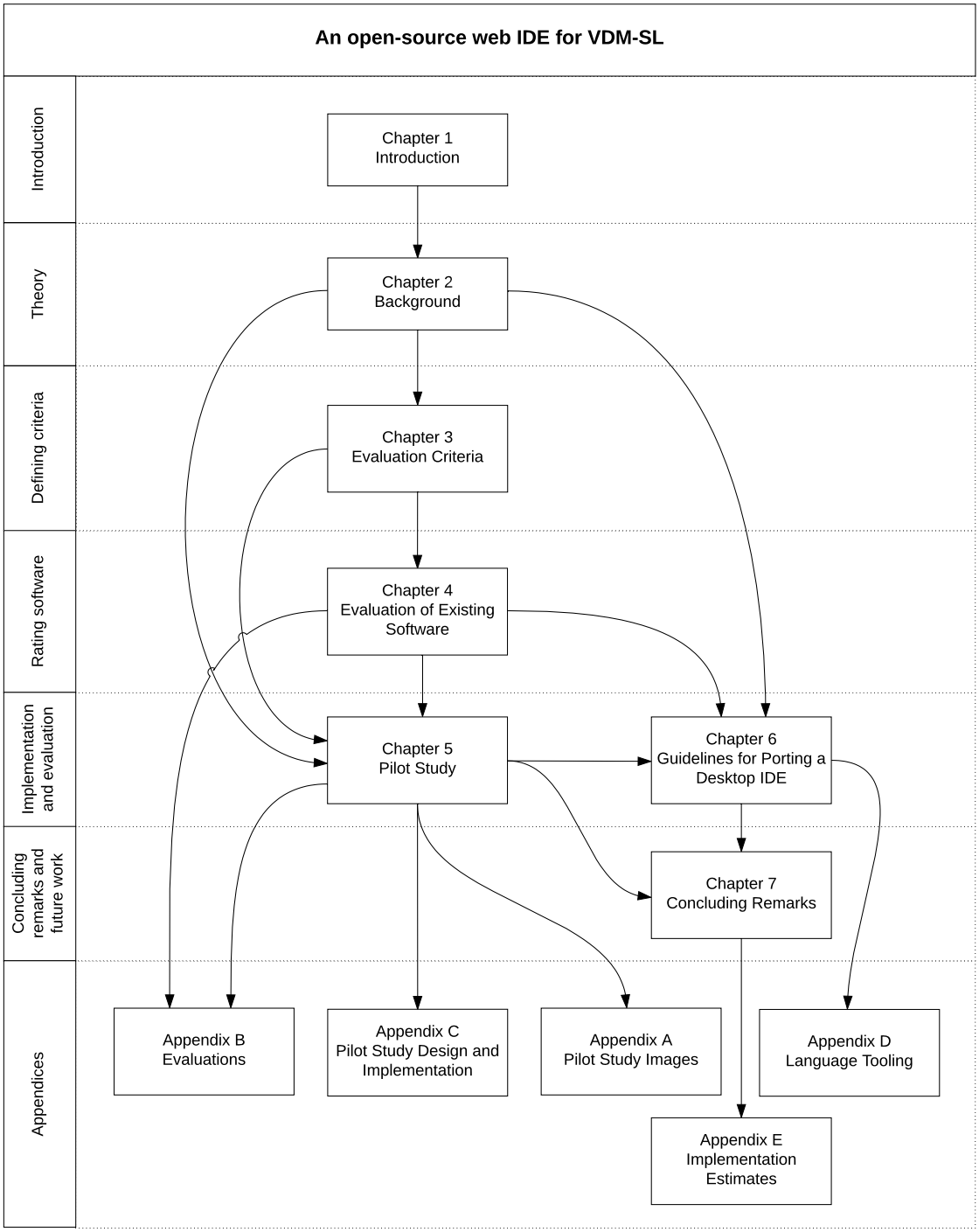


Figure 1.3: Thesis structure.

Chapter 2

Background

This chapter extends on the concepts described in chapter 1, by introducing additional concepts and technologies that form a basis for succeeding chapters. Reusable constructs are introduced together with a method for evaluation of open-source software. Certain state-of-the-art technologies are presented, along with technologies that forms a basis for the research in this thesis project. The technologies and approaches introduced in this chapter are applied in chapters 3-5.

2.1. Introduction

Web applications have evolved from being a means of sharing information to delivering web services, such as Software as a Service (SaaS) [4]. Additionally, the features supported in modern browsers and the use of JavaScript enables development of highly dynamic web-based User Interfaces (UIs) [10]. To cope with these advances, a multitude of web technologies have emerged for the development and deployment of web applications [11]. These includes frameworks, tools, and architectural patterns. Additionally, many of these technologies are open-source and supported by large software organizations and communities.

A large amount of free software is provided by the open-source community [12]. This presents opportunities for software development. However, deciding to reuse open-source software requires careful evaluation, to ensure the quality and functionality of the software is satisfying. There are a number of properties worth considering when evaluating software for reuse. These includes legal rights of reuse, openness from developers, and the community surrounding the software. Since the amount and quality of available information can vary, it can be necessary to use a method to help guide the decision.
In this thesis project, open-source software is evaluated for reuse in a web IDE supporting VDM-SL. In addition, the Qualification and Selection of Open-source Software (QSOS) method is used for the evaluations [12].

The remaining part of this chapter is structured as follows. In section 2.2 additional reusable concepts are described. Afterwards, section 2.3 presents the method used to evaluate and select open-source software. Then, section 2.4 describes a set of IDE features and relates these to an example IDE. Section 2.5 gives a description of the Vienna Development Method (VDM), and tools for working with VDM. Lastly, section 2.6 introduces technologies related to web development.

2.2. Software Reuse

In addition to the software granularities defined in section 1.1, this section describes architectural patterns and frameworks as reusable concepts.

2.2.1 Architectural Patterns

A *pattern* is an abstract description or template that convey knowledge about how to structure applications or parts of applications [1]. The patterns encapsulate knowledge about how specific problems can be solved, in a way that supports software quality factors and best practices [13]. Different patterns have different advantages and disadvantages and are intended for different scenarios. An important engineering skill is to be able to select the appropriate pattern for a given situation.

The typical *architectural pattern* used in web applications is the Model-View-Controller (MVC) composite pattern [1]. The MVC pattern separates the application into models, views, and controllers. The precise communication flow between the three entities can vary, however an example is shown in Figure 2.1.

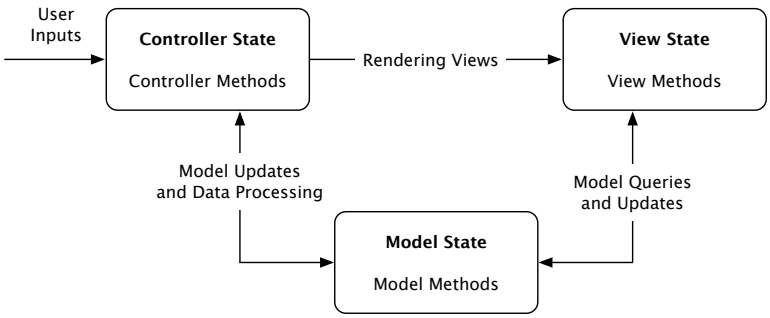


Figure 2.1: Basic example of the MVC pattern. Inspired by [1].

User input is routed to a method in a controller where the input is processed by calling domain logic implemented in the models. After the input has been processed, a response is rendered in the view.

2.2.2 Frameworks

A *framework* is a set of integrated software artefacts which facilitate software development in specific application domains [13]. A framework handles tasks common for the specific type of software, such as handling HTTP communication in web applications. Frameworks support software reuse in an object-oriented manner by providing generic features that are likely to be needed in applications of a similar type [1, 14]. Frameworks achieve this through the adaptation and extension of abstract and concrete classes. The components handling the business logic can then be integrated on top of the framework, creating a desired separation of concerns [13].

In the web application domain, a large number of client-side and server-side frameworks exists. Client-side frameworks are typically JavaScript frameworks, which provide an architecture to help structure an application. Additionally, the client-side frameworks can include mechanisms for event handling, and communication with the server-side application [10].

Server-side frameworks typically handle communication with client-side applications and external services both synchronously and asynchronously. Server-side frameworks usually assumes that data is persisted in a database, and therefore provide mechanisms for integrating with different types of databases [1].

Due to the way reuse is provided by frameworks, e.g. through inheritance, implementation of interfaces and abstract classes, frameworks are generally programming language specific. However, some multi-language frameworks do exists [15, 16].

2.3. Reusable Software Evaluation

As discussed in section 1.1, developing software using a reuse-based approach can have a variety of benefits. However, the approach can also have a number of drawbacks if unsuitable software is reused. Approaches have been proposed to address these drawbacks [6]. These approaches define a method or guidelines to evaluate and select the most appropriate open-source software for reuse. The approach described below is a method that includes four steps and activities for each step. Additionally, the method defines a set of criteria to be used in the evaluations.

2.3.1 The Qualification and Selection of Open Source Method

This section describes the four steps of the method called Qualification and Selection of Open Source software (QSOS) [12]. This method is an iterative approach, where each iteration consists of four steps. Additionally, an increasing level of detail can be used in each iteration, meaning that the software is studied in greater detail. The four steps are described below.

1. Define: In this step, criteria are defined which are used during the next three steps. These criteria are subdivided into three categories: type of software, type of license, and type of community. Additionally, the type of software is composed of a maturity analysis of the software, and a functional coverage analysis of the software. In the maturity analysis, a set of criteria is imposed by the method.

2. Evaluate: In this step, open-source software is evaluated using the criteria defined in step 1. Information on the open-source software is retrieved in order to provide a rating for the software. Each criterion is assigned a discrete value from 0 to 2, and a description is defined for each discrete value explaining what is required to receive the rating.

3. Qualify: In this step, the evaluation results are qualified. Filters are defined based on the context and criteria of the software. Filters can be defined such that only software of a certain type, or distributed under a certain license is considered. Maturity filters can be used to designate the relevance of each maturity criteria, and functional coverage filters for describing if certain criteria are required, optional, or not required. The filters are used in the next step to assign weights to the criteria.

4. Select: In this step, software matching the criteria is compared and possibly selected. Scores are rendered for the criteria, based on the filters defined in the previous step, for comparison and selection of the software units.

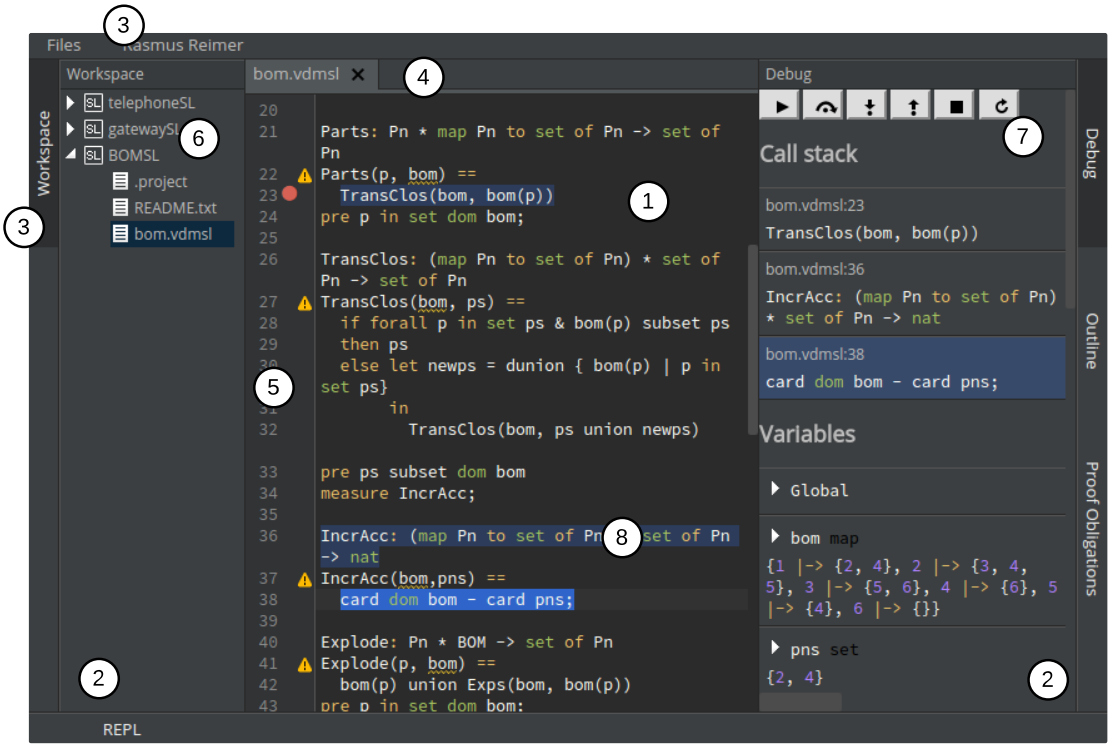


Figure 2.2: An example IDE window.

2.3.2 Application of QSOS

In this thesis project, different granularities of software reuse are considered. This amounts to a large quantity of software, that needs to be evaluated. Therefore, the QSOS method is applied in a pragmatic manner which means that certain steps are relaxed. Only relevant maturity criteria and non-optional criteria are defined in step 1. Additionally, some of the criteria imposed by the method are omitted, and more attention is given to defining criteria relating to the features of the software. All of the criteria are considered equally important, therefore no weights are defined in step 3. Hence, the time required to evaluate the software is reduced.

2.4. Integrated Development Environment

This section explains the terminology used in this thesis for features related to IDEs. Explanations are associated with a reference, to where on Figure 2.2 the particular feature can be seen. A feature name followed by a number in parenthesis refers to the circle with the same number on Figure 2.2.

An IDE consists of an *editor* (1), multiple *panels* (2) and *menus* (3). The editor allows editing the content of an open file. It has *tabs* (4) for each open file, which allows switching between the files. The editor also has a *gutter* (5) which can display line numbers, breakpoints, *linting* (code errors and warnings), etc. Panels contain UI for the features of an IDE. There are menus for toggling panels on the left, right, and bottom edges of the IDE window. In the top is a menu for some IDE features, such as creating new projects.
The *file explorer* (6), shown in the left panel, is used for managing the files and directories of

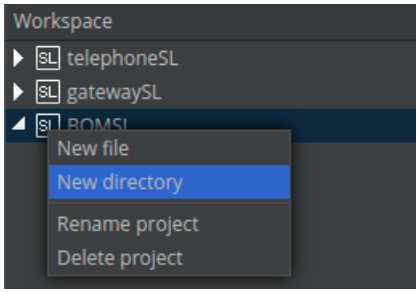


Figure 2.3: An example of a context menu.

projects in the workspace. It also allows opening files, which loads the file content in the editor.
The *debugging* (7) feature, shown in the right panel, displays controls and information for debugging. It also interacts with the editor by highlighting *stack frames*.
The *outline* feature displays a list of types, values, and functions defined in the open file.
The *context menu* is a menu which is customized, based on the context it is in. Figure 2.3 shows an example of a context menu, activated by right-clicking on a project in the file explorer.

2.5. Vienna Development Method

The Vienna Development Method (VDM) is a formal method that focuses on the development and analysis of system models [3]. VDM defines a formal modelling language called the VDM Specification Language (VDM-SL). VDM-SL is one of three VDM dialects that can be used to describe data and functionality [17]. Data is defined using types, on which constraints can be added using data type invariants. Functionality is defined as operations over the types.

Overture is a community-based initiative that develops the *Overture Tool*, which is an open-source tool for VDM [3]. The Overture Tool is developed in Java and consists of the Overture core, and a desktop IDE based on the Eclipse platform [18]. The Overture core contains the features for supporting VDM-SL. Figure 2.4 shows which features are currently supported, and which features are only available as prototypes, or not yet available.

The central features available in the Overture core are syntax checking, type checking, and the interpreter [3]. An Abstract Syntax Tree (AST) is generated from a VDM model, by parsing, and type checking the model. The AST is used by many features of the Overture core, such as the proof obligations generator and code completion. Expressions can be evaluated, through the interpreter, optionally against an AST. The Overture core also includes a debugger that conforms to an extension of the debugging protocol, called common Debugger Protocol (DBGP) [19]. DBGP describes how to communicate between a debugger engine and an IDE.

VDMPad is a lightweight web IDE that supports a limited number of the features available in the Overture Tool [20]. It provides a simple UI for experimenting with VDM-SL. VDMPad can be used to evaluate VDM-SL expressions based on a model, and a state of the model. In addition, VDMPad supports unit testing.

VDMTools is another desktop IDE for VDM [21]. This is a comprehensive desktop IDE for the VDM dialects, supporting all the features of the Overture Tool. These features include static

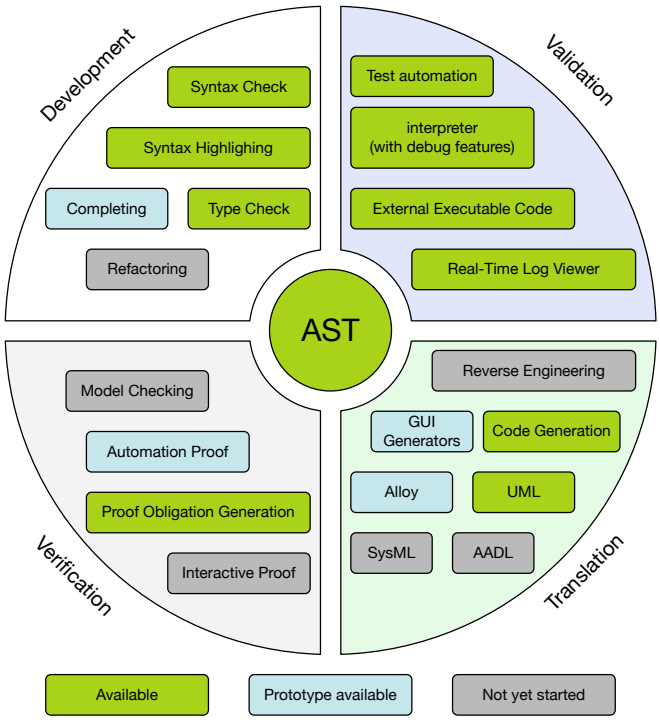


Figure 2.4: Overview of Overture Tool components. Inspired by [2].

semantics checking, code generation, test coverage analysis, and debugging support. However, VDMTools is closed-source and is therefore not an appropriate candidate for integrating into a web IDE. Hence, the open-source Overture core is used instead.

2.6. Web Applications

The World Wide Web is based on a client-server model, where servers stores resources and clients (web browsers) accesses them [4]. Resources are identified by a URL on the server, which clients can request using HTTP.

2.6.1 Client-side Application

Client-side applications provides the Graphical User Interface (GUI) in web applications. HyperText Markup Language (HTML) is used for defining how elements of the GUI are structured, including position and nesting. Cascading Style Sheets (CSS) are used for defining how elements of the GUI are styled, including colours, sizes, and fonts.

Client-side applications are programmed in JavaScript, or languages which compile to JavaScript, since it is the only language that most browsers can interpret [22]. With JavaScript, it is possible to send asynchronous HTTP requests to the server-side application. This technique is called Asynchronous JavaScript And XML (AJAX) [23]. The server-side application can respond to these requests with data in any text format, such as XML or JavaScript Object Notation (JSON). This

allows the client-side application to fetch data from the server-side application. The client-side application can also use AJAX to send data to the server-side application.

Multiple frameworks exist for developing client-side applications running in the browser [24, 25]. These frameworks mostly use the MVC architecture and communicates with the server-side application through an API using AJAX [26]. Recently, new client-side frameworks have been developed which aligns more with the future of the web standards as defined by the World Wide Web Consortium (W3C)[1]. These frameworks include Angular2, React, and Polymer [15, 27, 28].

2.6.2 Server-side Application

Web APIs have become increasingly popular [4]. Web APIs are server-side applications that structures how resources can be accessed. Different protocols have been defined for the communication between client-side and server-side applications. Two of these protocols are the Simple Object Access Protocol (SOAP) and REpresentational State Transfer (REST) protocols [29].

SOAP is developed by Microsoft, and relies exclusively on XML messaging [29]. The XML messages can be communicated over HTTP using SOAP bindings, where the XML message is added to the body of the HTTP requests.
To use SOAP with JavaScript can be difficult, but is mitigated to a certain degree by using third-party JavaScript libraries, which handles the communication.

In contrast, REST integrates better with the web standards [29]. A REST API can serve data in either XML or JSON. Additionally, REST makes extensive use of the HTTP protocol, e.g. by using URLs to identify resources, and HTTP verbs to specify the action to perform on the resources [30]. JavaScript has better support for HTTP verbs and JSON than SOAP bindings and XML, hence a REST API is easier to integrate into a client-side application.

A REST API is stateless, meaning that no state is memorized by the server-side application between requests from client-side applications, such as session information. The client-side application is therefore required to include relevant contextual information in each request. Statelessness supports scalability by reducing the memory load of the server-side application, enabling it to service a larger number of client-side applications concurrently [30]. Additionally, the stateless principle also enables load-balancing, since intermediate nodes (servers and proxies) do not have to take state affinity into account.

2.6.3 Full-duplex Communication

HTTP is a request-response model, which only allows the client-side application to initiate communication. Hence, a client-side application has to poll the server-side application for data changes.

Another method for communication was recently introduced, allowing full-duplex communication [31]. This technology, which is called WebSockets, is built on top of TCP sockets. Besides allowing full-duplex communication, WebSockets also have lower network latency compared to AJAX [32].

[1] See `https://www.w3.org/`, accessed 7th of April 2016.

2.6.4 Concurrency

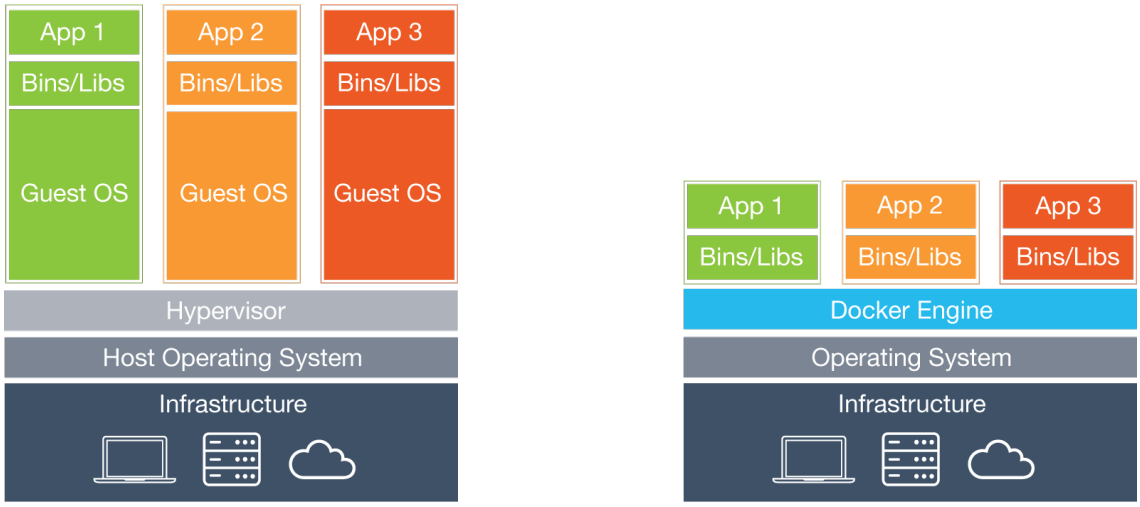
Web applications are concurrent applications. There are potentially multiple client-side applications, sending concurrent requests, to the server-side application. How the server-side application handles the concurrent requests, depends on the language and framework used.

JavaScript runs in a single thread and uses an *event loop* and an *event queue* as its concurrency model [33]. Programming in JavaScript is based on listening for and handling events. All events, such as user interactions or a completing HTTP request, can be handled by adding event listener functions to the events. When an event occurs, all registered event listener functions are added to the event queue.
The event loop handles each listener in the event queue synchronously, which means that functions cannot be preempted [33]. However, this also means that if an event listener function takes a long time to complete, the UI will become unresponsive since the UI event listeners are queued but not handled.

WebWorkers presents a solution to this problem [10]. A WebWorker is a JavaScript object which runs in its own thread. A WebWorker is isolated from the rest of the application, meaning no shared state and no access to the UI. They define a simple event-based interface for communicating with the rest of the application.

2.6.5 Deployment

Before deploying a web application on a server, the server has to be configured correctly, and the necessary software has to be installed. A Virtual Machine (VM) can be used, to easily deploy the web application to any server or local machine. However, using VMs has overhead as every VM has its own copy of an Operating System (OS), as seen in Figure 2.5a.



(a) Diagram showing layers involved when running a virtual machine.

(b) Diagram showing layers involved when running a Docker container.

Figure 2.5: Diagram showing the different between virtual machines and Docker containers[2].

[2]Taken from `https://www.docker.com/what-docker`, accessed 10th of April 2016.

Docker containers, addresses this overhead [34]. Docker containers reuse the hosts OS instead of running a guest OS, as seen in Figure 2.5b [35]. However, the containers still provide an isolated environment for the web application, including the file system, processes, network addresses, and ports. The environment can be customized, with the necessary configuration and software dependencies for the web application. The process of customizing the environment and executing the web application, can be automated with scripts called Dockerfiles.

Chapter 3

Evaluation Criteria

In chapter 2, theory and concepts are presented, including a method to evaluate and select open-source software. In this chapter, a set of criteria is defined which are used in the evaluations. Hence, this chapter is related to step 1, of the QSOS method. These criteria cover features which a web IDE supporting VDM-SL should ideally include. Afterwards, chapter 4 presents an evaluation of existing software based on the criteria defined in this chapter.

3.1. Introduction

Before any evaluation can be performed, a set of criteria needs to be defined. The criteria defined in this chapter are used in chapter 4, to evaluate software for reuse in a web IDE. The criteria describe the desired features of an ideal solution, and will assist in the selection of the most appropriate building blocks. Ratings have been defined for each criterion, in order to represent how well the criterion is supported. The ratings are defined with an explanatory text, which have been defined as part of this thesis, since no definition have been found in literature.

VDM-SL is a modelling language and therefore has different IDE feature requirements compared to a regular programming language, such as Java. The Overture core supports many of the desirable features for VDM-SL, as seen in Figure 2.4 [17]. A web IDE supporting VDM-SL should ideally include all of these features, and the features are therefore included, in the set of criteria. Additional criteria are based on features, selected through a literature study, of web IDEs and desktop IDEs [9, 36]. The criteria are separated into five categories, which are described below.

Editor: This category describes criteria for the editor in a web IDE.

Development environment: This category describes criteria for language agnostic features of a web IDE, e.g. a file explorer and theming.

Coding assistance: This category describes criteria for language specific features that work without evaluating a model, e.g. reference resolving and refactoring.

Runtime evaluation: This category describes criteria for language specific features that relate to evaluating a model, e.g. debugging and automated testing.

Quality: This category describes criteria for quality features of the software, e.g. documentation and extensibility.

The categorization of the criteria is an attempt to group criteria with certain semantic similarities. Some of the criteria fit in more than one category and in those cases the criteria are placed in the most fitting category. However, it has not been possible to find literature giving a well-defined categorization of criteria for IDE features. The categorization selected in this thesis should therefore be regarded as a way of grouping the criteria to make them more manageable.

The remaining part of this chapter is organized as follows: Section 3.2 describes the editor criteria. Then, the development environment criteria are described in section 3.3, which covers the language agnostic criteria. Afterwards, section 3.4 describes the coding assistance criteria, which are language specific criteria. Then, section 3.5 describes another set of language specific criteria, which are the runtime evaluation criteria. Finally, section 3.6 describes a set of quality criteria.

3.2. Editor

This section describes the criteria selected for the editor category. This category includes all criteria for the editor in a web IDE. The criteria are binary in the sense that either they are supported fully or not at all.

Annotate scrollbar: Annotate the scrollbar with locations of search results, warnings, and errors.

Auto indention: Keep the indention level when inserting a new line.

Close brackets: Automatically add a closing bracket after the cursor when inserting an opening bracket.

Code completion: Show a list of code completion suggestions.

Code folding: Allow toggling visibility of parts of a model, such as a function body.

Code templates: Insert common or user defined model constructs, e.g. methods, loops, or simple design patterns.

Context menu: Show a context menu at a term to allow actions such as refactoring or reference resolving.

Edit history: Maintain a history of changes to a model allowing undoing changes.

Highlight matching brackets: Highlight the matching bracket when placing the cursor at a bracket.

Indent guides: Show a vertical line indicating an indention level.

Line wrapping: Wrap lines that are longer than the editor's width.

Linting: Highlight warnings and errors in the gutter, and in the model.

Multiple open files: Manage multiple open files allowing switching between them.

Multiplex modes: Allow mixing languages within a file e.g. literate programming.

Programmable gutters: Allow symbols in the gutter to show breakpoints, warnings, and errors.

Search: Navigate to a text string in the editor by searching.

Search and replace: Search for and replace a text string with another text string.

Syntax highlighting: Colour terms based on language-specific syntax.

3.3. Development Environment

This section describes the criteria selected for the development environment category. This category includes all criteria for language agnostic features, meaning that they are not concerned with the language used in the IDE. Table 3.1 describes the ratings defined for the development environment criteria.

Code sharing: Sharing of files and projects between multiple users, with restricted privileges, such as only evaluating the model.

Collaboration: Collaboration allows multiple users to work simultaneously in the same file, displaying all users cursors and changes to the model in real-time.

File explorer: Allows navigating the file structure of projects and finding files. Also includes functionality to create, move, rename and delete files, folders, and projects.

Terminal: Terminal for a command-line shell for accessing the operating system services, e.g. a package manager.

Theming: Theming allows for visual customization of the IDE, e.g. colour themes, font style, and layout.

Version control: Version controlling the model through integration with a Version Control System (VCS), such as Git[1] or SubVersion (SVN)[2].

[1] See `https://git-scm.com/`, accessed 28th of April 2016.
[2] See `http://subversion.apache.org/`, accessed 28th of April 2016.

Table 3.1: Description of the ratings of the criteria in the development environment category.

Criterion	Rating			
	-	+	++	+++
Code sharing	No support.	Can share a link to a project, allowing the recipient to view and evaluate the model.		
Collaboration	No support.	Allows one user to change the model and other users to view the changes in real-time.	Allows multiple users to change the model simultaneously.	
File explorer	Has file tree view.	Can create, delete, move and rename files and directories.	Has file search facility.	
Terminal	No support.	Has access to an isolated file system, allowing installation of software.		
Theming	No support.	Can change the editor styling.	Can change the IDE styling.	
Version control	No support.	Can display changes.	Can import a model from a VCS, and deposit changes, in the model to the VCS.	Has full integration with a VCS, such as Git, SVN, etc.

3.4. Coding Assistance

This section describes the criteria selected for the coding assistance category. This category includes all criteria for features concerned with assisting the user in developing models. Table 3.2, describes the ratings, defined for the coding assistance criteria.

Code formatting: Reformatting a model to follow recommended standards for improving readability.

Code generation: Generation of code, such as Java or C++, from a model.

Dependency management: Detecting missing dependencies, such as libraries, and the ability to retrieve dependencies using a supplied dependency specification.

Outline: Displaying definitions overview e.g. methods, variables, and imported definitions per file.

Pretty print to LaTeX: Print models as LaTeX documents.

Proof Obligation Generator (POG): Verify dynamic type correctness and semantic consistency of a model.

Refactoring: Support modelling by providing behaviour preserving transformations to improve model quality and avoid "code smells" [37].

Reference resolving: Navigation to variable and method declarations, and provide a list of usages of variables and methods.

Table 3.2: Description of the ratings of the criteria in the coding assistance category.

Criterion	Rating			
	-	+	++	+++
Code formatting	No support.	Can format a model to follow a language format.		
Code generation	No support.	Supported.		
Dependency management	No support.	Can notify about missing dependencies.	Can install missing dependencies.	Can list and install additional dependencies.
Outline	No support.	Supported.		
Pretty print to LaTeX	No support.	Supported.		
Proof obligation generator	No support.	Supported.		
Refactoring	No support.	Can manipulate content in the editor.		
Reference resolving	No support.	Can navigate to implementation.	Can list usages of a function or variable.	

3.5. Runtime Evaluation

This section describes the criteria selected for the runtime evaluation category. This category consists of criteria covering features concerned with evaluating a model. Table 3.3, describes the ratings, defined for the runtime evaluation criteria.

Code coverage: Displaying evaluated paths as a result of evaluating a model, starting from a specified entry-point.

Debugger: Defining breakpoints, evaluating a model, and halting evaluation at breakpoints. When the evaluation is halted, state information is displayed, and the model can be stepped through, which updates state information.

External executable code: Calling external Java ARchive (JAR) files from a model.

Interpreter: Evaluating models with support for creating run configurations defining entry-points, supplying arguments, and optionally setting environment variables. Additionally, UI can be provided for text I/O.

Read-Evaluate-Print-Loop (REPL): Evaluation of expressions in the context of a model through a Command-Line Interface (CLI).

Testing: Definition and execution of test cases and subsequently displaying test results.

Table 3.3: Description of the ratings of the criteria in the runtime evaluation category.

Criterion	Rating			
	-	+	++	+++
Code coverage	No support.	Supported.		
Debugger	No support.	Support for evaluating a model, defining breakpoints in the editor, and views for displaying model state information.		
External executable code	No support.	Can interface with external JAR files.		
Interpreter	No support.	Supported.		
REPL	No support.	Supported.		
Testing	No support.	Can configure test-case runners.	Can run tests and display test results.	

3.6. Quality

This section describes the criteria selected for the quality category. This category does not relate to specific features of the software, but instead the quality of the software. Table 3.4, describes the ratings, defined for the quality criteria.

Documentation: Documentation of the software, aiding in the development of additional features through extensions.

Extensibility: Possibilities to add additional features to the software, e.g. defined interfaces and whether extensions can be developed for both client-side and server-side applications.

Maturity: The release stage of the software, e.g. alpha, beta, stable, or Long-Term Support (LTS).

The extensibility feature is influenced by many quality features. These include how well the software is documented, supported by core developers, and the number of contributors in the surrounding community. These quality features are important for enabling developers to extend the software to support additional languages [8]. In addition, it is desirable that the software is mature and in a stable release, indicating that the software has been tested through usage. This can lead to discovering and elimination of many issues and inexpediencies, improving the overall software quality [38]. An LTS release means that the software receives bug fixes and updates with backwards compatibility over a long term.

The community behind the software will also be taken into consideration. This can indicate whether the software will continue to be maintained and improved. The community involved with the software have not been defined as a feature alongside documentation, extensibility, and maturity since the community is difficult to quantify.

Table 3.4: Description of the ratings of the criteria in the quality category.

Criterion	Rating			
	-	+	++	+++
Documentation	None.	Exists but is severely limited.	Limited documentation for developing extensions.	Comprehensive documentation for developing extensions.
Extensibility	No support.	Extensions can be added.		
Maturity	Alpha.	Beta.	Stable.	LTS.

Chapter 4

Evaluation of Existing Software

In chapter 3, a set of criteria is defined. These criteria cover features a web IDE supporting VDM-SL should ideally include. In this chapter, the criteria are used to evaluate existing software. Additionally, a visual comparison of the evaluation results will be provided. Hence, this chapter is related to step 2 and 4, of the QSOS method. The software ranges from entire applications to small software units. The evaluations will provide insights into the appropriateness of reusing a given software unit. The outcome of the evaluations is applied in a pilot study, in chapter 5. The pilot study is intended to provide deeper insights into the extensibility and integration possibilities of the software.

4.1. Introduction

Reusing software can be performed at many different levels of granularity. The different granularities can have different benefits and shortcomings as described in section 1.1. In this chapter, software units of different granularities are evaluated. Initially, a coarse-grained approach with complete applications is evaluated. Subsequently, software of increased granularities is evaluated, including frameworks and editors.

The evaluations are solely based on interacting with the software, and documentation provided by community websites. This restriction is applied due to the limited project time frame. An in-depth investigation of all the software can be too comprehensive. However, a cursory evaluation will provide an indication of whether the software is worth investigating in depth. The intention behind the evaluation is to assess which software units requires the least amount of effort to be reused. Therefore, certain assumptions have been made in the evaluation. It is assumed that all features can be added if the documentation claims that the software is extensible. If UI components exist for supporting a feature, it is assumed that interfaces exist to populate the UI components. Therefore, a feature is rated as supported if the feature is supported "out of the box", perhaps through an existing extension. Additionally, a feature is rated as not supported, even if it is possible to add the feature by developing an extension.

Figure 4.1 shows how software of different granularities is related according to the criteria defined in chapter 3. At the coarsest level of granularity, are existing web IDEs. Hence, a set of open-source web IDEs is evaluated according to the criteria defined in section 3.2-3.6. The results of these evaluations are used for deriving web IDEs with potential in forming the basis for a web IDE

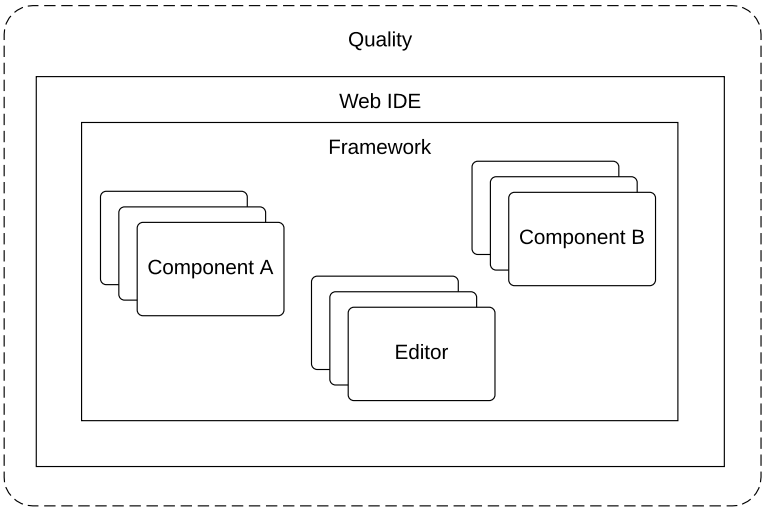


Figure 4.1: Software reuse composition.

supporting VDM-SL. However, extending or reconfiguring a complete web IDE can be difficult [1]. A fine-grained approach is therefore also considered.

A fine-grained approach means that components are evaluated. The components are building blocks which need to be integrated into an application, whereas the web IDEs constitute an (almost) complete solution to developing a web IDE. Each component may only provide a small subset of all the desired features. In addition, each component may be maintained by different organizations, which can have consequences regarding interoperability and compatibility [5]. When using a component-based approach, a platform is required on which to integrate the components. In many types of software development, including web development, a framework forms the basis for the software [13]. A set of frameworks is therefore evaluated.

Frameworks are described in section 2.2.2. Both client-side and server-side frameworks are evaluated. Multiple frameworks exist for different programming languages. Additionally, some of these frameworks are supported by well-established software organizations. Hence, promising frameworks exist which are well documented. This can aid the evaluation process since a large amount of information is available. However, the large number of frameworks can also make it cumbersome to find the most appropriate framework for the given situation.

Editors are the only components considered in the fine-grained approach since no other suitable components are found. Editors constitute major components of a web IDE. The editor is therefore a desirable component to avoid having to develop and maintain. As with existing web IDEs, there exist a set of open-source editors. Additionally, some of these editors are used by the evaluated web IDEs.

The evaluations are presented in the following sections, which are organized as follows. In section 4.2 a set of existing web IDEs is evaluated. At the next level of granularity, a set of client-side and server-side frameworks are evaluated in section 4.3. Then, in section 4.4 editors are evaluated. Lastly, section 4.5 presents a summary of the evaluations.

4.2. Web IDEs

A number of open-source web IDEs exist, which differs in complexity and the number of features they support "out of the box". Additionally, some web IDEs are intended for specific software development domains, such as web development, other are more general purpose [7].
To find the most appropriate web IDE for reuse, an assessment has been performed on the discovered web IDEs. Afterwards, the most promising candidates have been selected for further evaluation. The web IDEs selected through this assessment are described below.

Che: This is a web IDE supporting multiple static and dynamic languages through the use of docker containers. These languages include Java, C++, Python, Ruby, and JavaScript. Additionally, Che includes a variety of features for managing projects, such as version control and dependency management.

Cloud9: This is a commercial web IDE supporting multiple static and dynamic languages, such as C++, JavaScript, and PHP. Additionally, Cloud9 is contributed to by a large community.

Codebox: This is a simple web IDE supporting static and dynamic languages, such as Java and JavaScript. Additionally, Codebox can run both as a desktop IDE as well as a web IDE.

Orion: This is a simple web IDE made by the Eclipse Foundation, which is primarily intended for working with JavaScript, HTML, and CSS.

These web IDEs are open-source, allowing them to be reused either by cloning and modifying the code base, or by developing extensions.

In the remaining part of this section, the evaluation results are presented followed by an evaluation summary. The evaluation of the editor criteria is presented in Table 4.1. The evaluation of the development environment, coding assistance, runtime evaluation, and quality criteria are visualized in Figure 4.2-4.5. Radar charts are used to provide a visual intuition of how the web IDEs compare. The radar charts are accompanied by additional comments when needed. The evaluation of the development environment, coding assistance, runtime evaluation, and quality criteria can also be seen in table-form in Appendix B.

4.2.1 Editor

Table 4.1: Evaluation results of editor criteria.

Criterion	Web IDE			
	Che	Cloud9	Codebox	Orion
Annotate scrollbar	+	-	-	+
Auto indention	+	+	+	+
Close brackets	+	+	+	+
Code completion	+	+	+	+
Code folding	+	+	+	-
Code templates	+	+	+	-
Context menu	-	+	-	+
Edit history	+	+	+	+
Highlight matching brackets	+	+	+	+
Indent guides	-	+	+	-
Line wrapping	+	+	+	+
Linting	+	+	+	+
Multiple open files	+	+	+	-
Multiplex modes	+	+	+	-
Programmable gutters	+	+	+	+
Search	+	+	+	+
Search and replace	+	+	+	+
Syntax highlighting	+	+	+	+

Both Cloud9[1] and Codebox[2] uses the Ace editor, whereas Che uses the CodeMirror editor. These editors are evaluated in section 4.4. Neither Ace nor CodeMirror includes a context menu, however this feature has been added in Cloud9. Orion does not use a third party editor. In addition, Orion's editor is missing five of the criteria in this category, and are therefore rated worse among the web IDEs.

4.2.2 Development Environment

Figure 4.2 shows a comparison of the four web IDEs evaluated on the development environment criteria. The figure shows that none of the web IDEs meet all the criteria since each of the web IDEs lack support for at least one criterion. Cloud9 is rated best overall, but it has no support for the version control criterion. Codebox performs second best by achieving a better rating than Che in the terminal, theming, and file explorer criteria.

[1] See `https://docs.c9.io/docs/the-editor`, accessed 24th of March 2016.
[2] See `http://help.codebox.io/ide/shortcuts.html`, accessed 24th of March 2016.

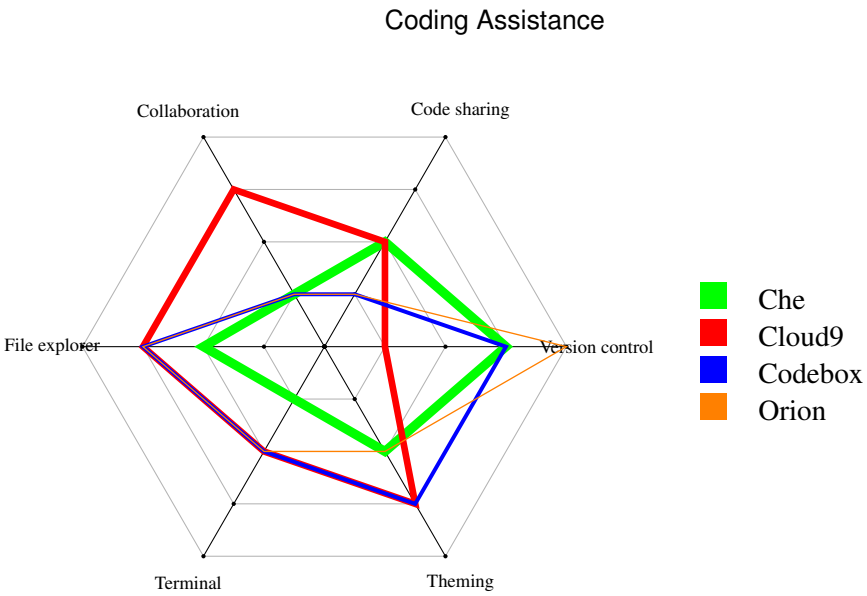


Figure 4.2: Evaluation results for development environment criteria.

Collaboration
Cloud9 supports real-time collaboration, where multiple users can edit files simultaneously, similar to Google Docs[3].

Terminal
In Cloud9 workspaces are running in Docker containers, which is described in section 2.6.5. This provides an isolated environment for each user. Codebox includes a terminal interface, and deploying the web IDE in a Docker container will provide a development environment which can be customized by the user. Orion includes a shell which can be used to access the file system[4].

Version control
Orion has full support for Git, including merging, branching, and viewing file changes[5]. Che and Codebox support basic Git tasks, such as commit, push, and pull. Additionally, Che also supports merging and showing commit history.

4.2.3 Coding Assistance

Figure 4.3 shows a comparison of the coding assistance criteria. Che is rated best overall by performing better or equal to the other three web IDEs on all criteria. Cloud9 is rated second best, by performing better, or equal to, Codebox on all criteria.

[3]See `https://www.google.dk/intl/da/docs/about/`, accessed 30th of March 2016.
[4]See `https://wiki.eclipse.org/Orion/Documentation/User_Guide/Reference/Shell_page`, accessed 16th of March 2016.
[5]See `https://wiki.eclipse.org/Orion/Documentation/User_Guide/Tasks/Working_with_Git`, accessed 16th of March 2016.

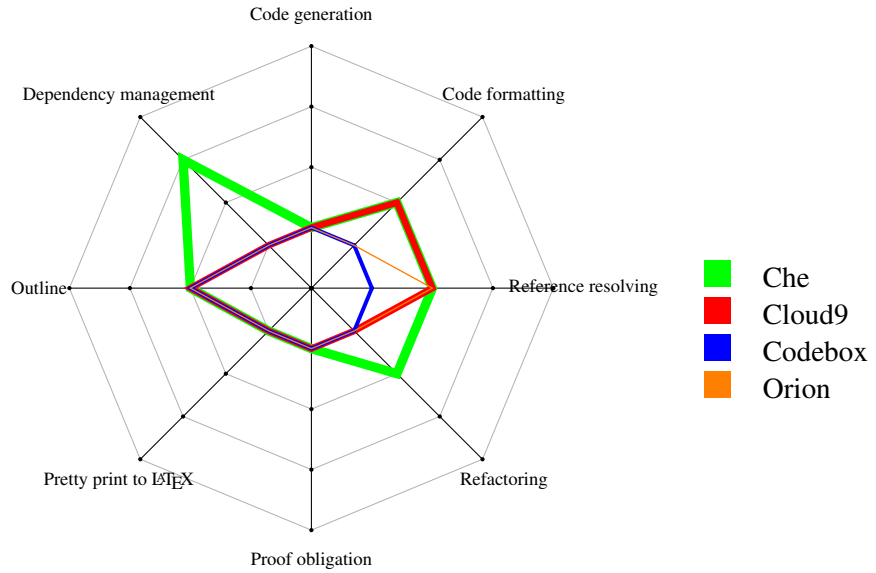


Figure 4.3: Evaluation results for coding assistance criteria.

4.2.4 Runtime Evaluation

Figure 4.4 shows a comparison of the runtime evaluation criteria. Cloud9 achieves the best overall rating, even though Cloud9 and Che are very evenly matched. Cloud9 is only rated better than Che on the REPL criteria. Codebox fares poorly on these criteria, only supporting the debugging criteria.

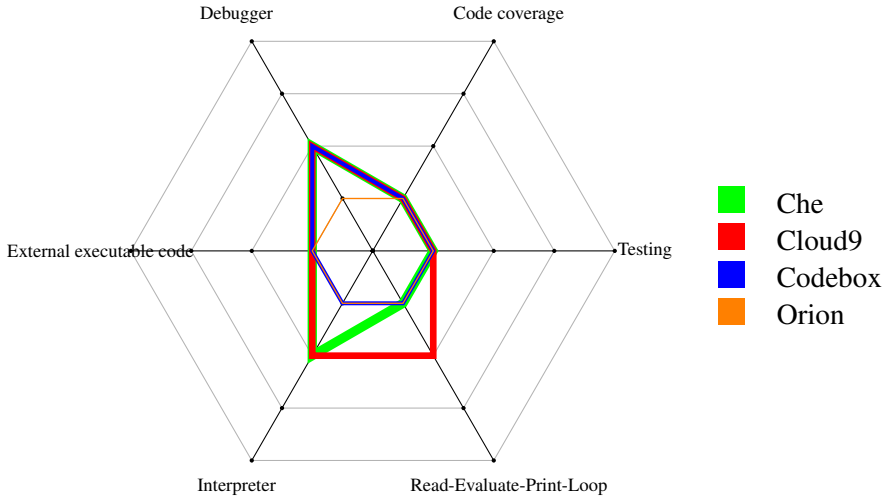


Figure 4.4: Evaluation results for runtime evaluation criteria.

Debugger
In Che and Cloud9, run configurations for debugging are created as described below, in the evaluation of the interpreter criteria. Codebox can debug files, but does not provide UI for supplying arguments or environment variables. Additionally, Che, Cloud9, and Codebox support defining breakpoints in the editor and provides UI for displaying state information.

External executable code
This criterion is not supported by any of the web IDEs since it is specific for VDM-SL.

Interpreter
In Cloud9, run configurations can be created with arguments and environment variables. Text-based I/O is supported through a terminal. In Che, run configurations are created using Dockerfiles, in which arguments and environment variables can be defined. Text-based output is displayed during execution of the Dockerfile. However, text-based input cannot be supplied during execution, using Dockerfiles. For supplying text-based input at runtime using Che, the Dockerfile must only be used for the compilation. In addition, the last command in the Dockerfile should be a sleep command. This will allow manual execution through a terminal, connected to the Docker container.

4.2.5 Quality

Figure 4.5 shows a comparison in the quality category. As seen in the figure, Orion is rated best in this category. Che is rated better than Cloud9 since Che is the most mature of the two web IDEs. Codebox performs poorly in this category, only supporting the documentation and extensibility criteria. Additionally, Codebox is no longer maintained.

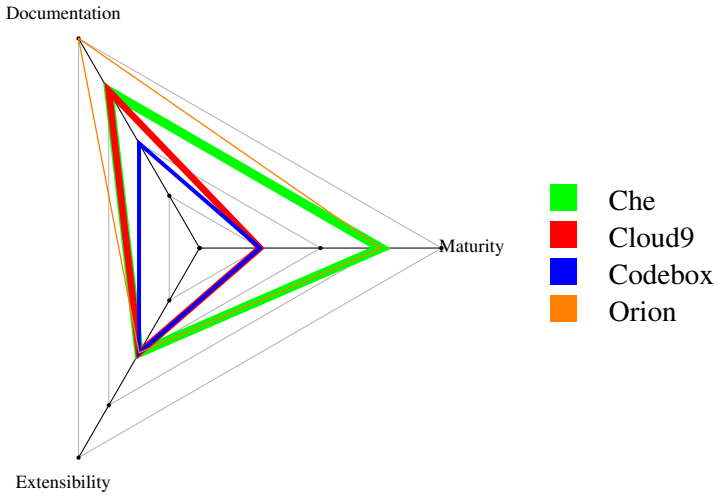


Figure 4.5: Evaluation results for quality criteria.

Documentation
Che[6], Cloud9[7], and Codebox[8] have documentation, but this documentation is very limited with respect to the development of extensions. For Orion[9], both user guides and developer guides are available on the community website. The developer guides provide documentation and examples on how to develop client-side extensions.

[6]See `https://eclipse-che.readme.io/docs/`, accessed 6th of May 2016.
[7]See `https://cloud9-sdk.readme.io/docs`, accessed 6th of May 2016.
[8]See `http://help.codebox.io`, accessed 6th of May 2016.
[9]See `https://wiki.eclipse.org/Orion/Documentation/Developer_Guide`, accessed 16th of March 2016.

Extensibility
In Che, Cloud9, and Codebox both client-side and server-side extension is possible.
Extending Orion is only possible on the client-side[10]. The example server shipped with Orion does not define an API for extending its behaviour.

Maturity
Since Cloud9 is in alpha release, later versions of the extension architecture might not be backwards compatible with the current version. Che is in stable release 4.1.0[11] and Orion is in stable release 11.0[12].
As mentioned, Codebox is no longer being maintained. This makes Codebox less desirable to reuse since no community exists for maintaining the code base.

4.2.6 Evaluation Summary

The results of the evaluations are summarized, in Table 4.2. The table displays the evaluation result in each category in percentages. The percentages are calculated for each category, as the web IDEs achieved rating in a category, divided by the maximum achievable rating in that category. The average is calculated, by summing the web IDEs rating percentages across the categories, and dividing the sum by the number of categories.

As seen in Table 4.2, Cloud9 and Che are rated better than Codebox and Orion. Additionally, Che achieves a marginally better rating than Cloud9.

Table 4.2: Summation of web IDE evaluation results.

Category	Rating percentage			
	Che	Cloud9	Codebox	Orion
Editor	89	94	89	72
Development environment	45	73	64	64
Coding assistance	60	30	10	20
Runtime evaluation	29	43	14	0
Quality	71	43	29	86
Average percentage	**59**	**57**	**41**	**48**

Che and Orion have relations to the Eclipse Foundation, which is a well-established community for open-source software, supported by corporate sponsors such as Google, IBM, and Oracle[13]. This provides the web IDEs with a solid community to take part in support and maintenance.
Cloud9 is a privately held company founded in 2010 and is backed by Atlassian[14], among other. Cloud9 has built a large community for support and maintenance, judging from activity on com-

[10] See `https://wiki.eclipse.org/Orion/Documentation/Developer_Guide/Architecture#Server_architecture`, accessed 17th of March 2016.
[11] See `https://projects.eclipse.org/projects/ecd.che/releases/4.1.0`, accessed 2nd of May 2016.
[12] See `https://projects.eclipse.org/projects/ecd.orion/releases/11.0`, accessed 2nd of May 2016.
[13] See `https://www.eclipse.org/corporate_sponsors/`, accessed 18th of March 2016.
[14] See `https://c9.io/site/about`, accessed 27th of March 2016.

munity sites[15], blogs[16], and Github repository[17].
The fact that Codebox is no longer being maintained is a serious disadvantage for the web IDE since a community for taking part in support and maintenance no longer exists.

4.3. Frameworks

Instead of reusing an entire web IDE, another option is to use a well-established framework. The framework forms a platform on which to integrate components, such as an editor, a file explorer, etc. A framework might not include as many features as existing web IDEs. However, frameworks are worth considering from a quality perspective. A well-established framework might have better documentation and a larger community than a web IDE. Developers are also more likely to have experience with a framework than with a web IDE since frameworks are used in many applications.

In the remaining part of this section, the evaluation results are presented and discussed. The client-side and server-side frameworks can only be evaluated on the quality criteria defined in chapter 3. The results are visualized using radar charts, to give a better intuition of how the frameworks compare. The results can also be seen in table-form in Appendix B.

4.3.1 Client-side Frameworks

This section presents evaluation results of three of the most used open-source client-side frameworks for developing web applications.

Angular2: Angular2 is a framework developed and used by Google [15]. In addition to JavaScript, Angular2 can be used with either TypeScript or Dart which are statically typed languages [39, 40].

Ember: Ember is a framework developed by individual contributors and backed by sponsors, which include LinkedIn and Yahoo![18]. An Ember application uses the traditional Model-View-Controller (MVC) approach for structuring the application [24].

React + Redux: React is a framework for building UI components developed and used by Facebook [27]. Redux is a library typically used together with React for the application architecture[19].

Figure 4.6 shows the results of evaluating these frameworks against the quality criteria.

[15] See `https://community.c9.io/`, accessed 27th of March 2016.
[16] See `https://c9.io/blog/`, accessed 27th of March 2016.
[17] See `https://github.com/c9/core`, accessed 27th of March 2016.
[18] See `http://emberjs.com`, accessed 31st of March 2016.
[19] See `http://redux.js.org/docs/basics/UsageWithReact.html`, accessed 31st of March 2016.

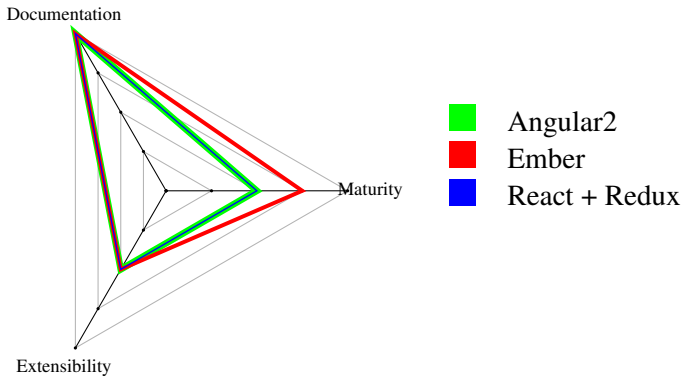


Figure 4.6: Evaluation results for quality criteria.

Documentation
All three frameworks are documented extensively. However, the fact that Angular2 can use a statically typed language, makes its documentation easier to use, since the framework API is documented with types.

Extensibility
The frameworks are all extensible in the scope of developing web applications.

Maturity
Angular2 and React are in beta, which means that APIs can still change and parts of the framework are not yet stable. Ember is in stable release, which means that updates will be backward compatible.

4.3.2 Server-side Frameworks

Integration with the Overture core needs to be performed in the server-side application, since client-side applications cannot run Java. To ease the integration, and to enable code reuse from the existing Overture tools, only frameworks supporting Java have been selected for evaluation. The selected frameworks are described below.

Play: This is a lightweight MVC framework for developing web applications in both Java and Scala.

Spring Framework: This is a comprehensive framework for developing applications, ranging from embedded applications running resource-constrained devices, to enterprise-scale web applications[20].

Both Spring and Play are well-established frameworks, which are supported by software companies[21,22]. Evaluation results of the frameworks in the quality criteria are seen in Figure 4.7. The results show that the frameworks are equally matched in these criteria. Additionally, only the maturity criterion is lacking in order for the frameworks to achieve the best rating.

[20] See `http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/overview.html`, accessed 28th of March 2016.
[21] See `https://pivotal.io/spring-app-framework`, accessed 26th of April 2016.
[22] See `https://www.lightbend.com`, accessed 26th of April 2016.

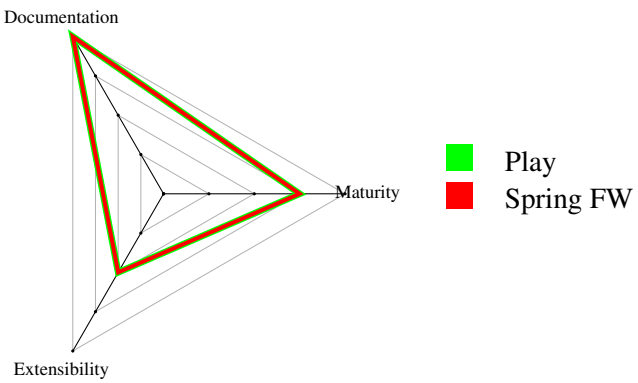


Figure 4.7: Evaluation results for quality critera.

Documentation
The community website for Play includes comprehensive documentation and guides on how to use the framework. In addition, tutorials are provided, which demonstrates different usages of the framework. Springs community website also provides extensive documentation, guides, and tutorials for using the framework. Hence, both frameworks achieve the best ratinr on this criterion.

Extensibility
Both frameworks are extensible in the scope of developing web applications.

Maturity
Play is in release 2.5.2, but there is no information on the type of release. However, it is assumed that this is a stable release. Spring is in stable release 4.2.5. There is no mention of LTS releases for either of the frameworks.

4.4. Editors

The two most prominent editor components are Ace[23] and CodeMirror[24]. Both of these editors are developed in JavaScript and are intended for usage in web applications. As the evaluation of the editors shows, Ace and CodeMirror supports a large number of the editor criteria defined in section 3.2. In addition, the usage of these editors in web applications is extensive. Ace is used by Cloud9[25], Wikipedia[26], and GitHub[27] among others. CodeMirror is used by Che[28], Adobe Brackets[29], Google Chrome DevTools[30], Firefox Developer Tools[31], and BitBucket[32] among others.

[23] See `https://ace.c9.io`, accessed 31st of March 2016.
[24] See `http://codemirror.net/`, accessed 31st of March 2016.
[25] See `https://ace.c9.io/#nav=about`, accessed 16th of March 2016.
[26] See `https://www.mediawiki.org/wiki/Extension:CodeEditor`, accessed 16th of March 2016.
[27] See `https://github.com/blog/905-edit-like-an-ace`, accessed 16th of March 2016.
[28] See `https://eclipse-che.readme.io/v1.0/docs/plug-ins`, accessed 6th of May 2016.
[29] See `http://brackets.io/`, accessed 16th of March 2016.
[30] See `https://developer.chrome.com/devtools`, accessed 16th of March 2016.
[31] See `https://hacks.mozilla.org/2013/11/firefox-developer-tools-episode-27-edit-as-html-codemirror-more/`, accessed 16th of March 2016.
[32] See `http://blog.bitbucket.org/2013/05/14/edit-your-code-in-the-cloud-with-bitbucket/`, accessed 16th of March 2016.

In the remaining part of this section, the evaluation results are presented and discussed. The quality evaluation results are visualized using radar charts, to give a better intuition of how the editors compare. The quality evaluation results can also be seen in table-form in Appendix B.

4.4.1 Editor

Table 4.3 shows the results of evaluating Ace and CodeMirror on the editor criteria.

Table 4.3: Evaluation results of editor criteria.

Criterion	Editor	
	Ace	CodeMirror
Annotate scrollbar	-	+
Auto indention	+	+
Close brackets	+	+
Code completion	+	+
Code folding	+	+
Code templates	+	+
Context menu	-	-
Edit history	+	+
Highlight matching brackets	+	+
Indent guides	+	-
Line wrapping	+	+
Linting	+	+
Multiple open files	+	+
Multiplex modes	+	+
Programmable gutters	+	+
Search	+	+
Search and replace	+	+
Syntax highlighting	+	+

4.4.2 Quality

Figure 4.8 shows the quality evaluation results of Ace and CodeMirror.

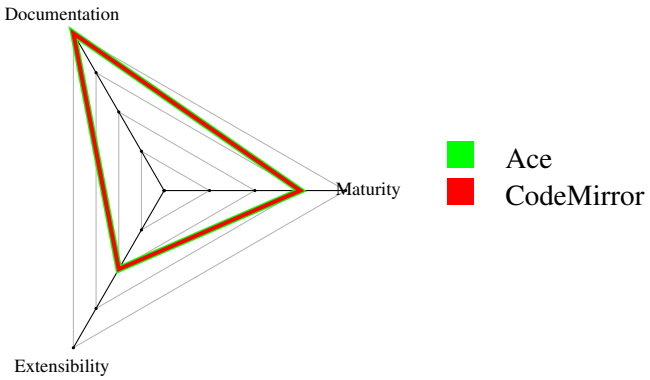


Figure 4.8: Evaluation results for quality criteria.

The editors are rated equally in the quality criteria. In addition, the editors are rated almost equally in the editor criteria. The differences are that only Ace supports the indent guides criterion and only CodeMirror supports the annotate scrollbar criterion. None of them supports the context menu criterion. However, VDM syntax highlighting rules have been developed for CodeMirror, as part of VDMPad [20]. Hence, selecting CodeMirror for the web IDE enables reuse of the VDM syntax highlighting rules.

4.5. Evaluation Summary

The evaluations show that the web IDEs support some of the criteria defined in chapter 3. However, the best rating achieved, among the web IDEs, is still below 60%. Additionally, the shortcomings in the quality category means that extending or reconfiguring the software can be difficult. Hence, it must be considered carefully whether the gain in the supported criteria is worth the effort.

As opposed to the web IDEs, the frameworks and editors can only be expected to support a subset of the criteria. The finer granularity of the frameworks and editors means that their expected contributions are more specific with respect to the criteria. Additionally, the evaluation results show that both client-side and server-side frameworks, and the editors, achieves better ratings in specific categories.

As mentioned above, these evaluations are solely based on interacting with the software and the documentation provided. As such, the evaluations are intended to provide an overall assessment as to whether it is appropriate to reuse the software. From these results, it cannot be concluded which approach, with respect to granularity, is the most appropriate. However, the results provide a basis on which to select software for each approach. If a coarse-grained approach is taken, then both Cloud9 and Che are worth investigating further. If a fine-grained approach is taken, then CodeMirror, React with Redux, and the Play framework should be selected. However, additional consideration can still influence the selection of the software. An in-depth investigation, such as a pilot study, can provide additional insights into the feasibility of reusing the software.

Chapter 5

Pilot Study

In chapter 4, existing software of different granularities is evaluated based on the defined criteria. In this chapter, a subset of the evaluated software, which comprises web IDEs, frameworks, and editors is used in a pilot study. This pilot study is intended to serve as a platform on which to experiment with different development choices, and provide further insights into the feasibility of developing a web IDE using a reuse-based approach. Hence, this chapter is related to step 2 and 4 of the QSOS method, with the addition of increasing the level of detail of the evaluation. Afterwards, the lessons learned from the pilot study is applied in chapter 6, to provide guidelines for the development of web IDEs.

5.1. Introduction

A pilot study is a scientific tool, which can be used to conduct a preliminary analysis, before committing to extensive development. The evaluations in chapter 4, build on a set of simplifying assumptions, which are presented in section 4.1. In order to get a more accurate picture of how appropriate the software is to reuse, the software needs to be studied in greater detail. By conducting a pilot study, insights can be gained into how difficult the software will be to extend or integrate into another solution.

This chapter describes the pilot study which is carried out as part of this thesis project to gain experience with the software. The pilot study is based on the results of the evaluations in chapter 4 of existing software, to assess the feasibility of a reuse-based approach to developing a web IDE. The initial approach in the pilot study is to select the two best rated web IDEs (Cloud9 and Che). Further experimentation is performed on these web IDEs to assess their extensibility and how integrating the Overture core can be achieved.

A more fine-grained software reuse approach is also investigated. The purpose of this investigation is to address possible integration difficulties by avoiding the complete web IDEs, in which the interfaces are dictated, and instead, gain a more fine-grained control over how the web IDE can be composed. In this investigation, the evaluations of frameworks and editors in chapter 4 are exploited, starting with the selection of frameworks to form the basis for a web IDE.

Angular2 is selected as the client-side framework for its support of a strongly typed language.

Angular2 is still in beta release, but it is nearing its first release candidate[1]. The missing features in the Angular2 beta are mainly internationalization and UI animation. Neither of these are crucial in developing a web IDE for VDM-SL.

The Play framework is selected as the server-side framework since it supports Java and includes features to improve productivity. A more detailed discussion of why this framework is selected and a description of the framework is provided in appendix C.2.

Using a well-established framework is beneficial in the development process and for adding extensions in the future [13]. These frameworks are often well documented and developers might be familiar with the framework from previous projects.

The remaining part of this chapter is structured as follows. The investigation into extending existing web IDEs is described in section 5.2. Then, the investigation into using frameworks and editors is described in section 5.3. Finally, section 5.4 presents an evaluation of the pilot study compared to the best rated existing web IDE.

5.2. Extending Existing Web IDEs

Cloud9 and Che, are rated best in section 4.2 given their supported features which cover development environment, coding assistance, and runtime evaluation. Since Cloud9 and Che are so evenly matched, both have been selected for further experimentation. These experimentations primarily concern the extensibility of the web IDEs, which is a concern due to the web IDEs' poor ratings in the quality category.

5.2.1 Extending Cloud9

The attempt at integrating with Cloud9 is aborted due to certain difficulties. The main problem area is the undocumented APIs. Both the client-side and server-side application of Cloud9 are developed in JavaScript. This adds to the difficulties since JavaScript is dynamically typed and gives no guarantees on types, the number of function parameters, and return values [41]. This means that APIs cannot be inferred from inspecting the source code of Cloud9, making familiarization with the API more difficult, and slow.

5.2.2 Extending Che

The attempt at integrating with Che is also aborted due to limited documentation which is rapidly becoming outdated. Examples on how to develop extensions are trivial and also outdated. In addition, the process of building extension as JARs[2], is cumbersome and failing at the time of writing. Hence, further development with Che would be cumbersome and slow.

[1] See `https://github.com/angular/angular/milestones`, accessed 22nd of March 2016.

[2] See `https://eclipse-che.readme.io/docs/developing-extensions`, accessed 24th of March 2016.

5.2.3 Evaluation

These existing web IDEs are in the early stages of opening up for extensibility for developers outside the core teams. Since documentation, community, maturity and support are lacking, extending the web IDEs would be difficult and slow.

5.3. Using Frameworks and Components

This section describes a component-based approach to developing a web IDE using open-source software. The best rated components and frameworks in chapter 4 are applied. The result of the pilot study is evaluated according to the criteria defined in chapter 3, and compared to the best rated web IDEs in chapter 4.

5.3.1 Client-side Application

The client-side application is intended to investigate how a modern framework can be used to develop the UI for a web IDE. This section describes the results of the investigation. A more detailed description of the selected framework, editor, and the architecture of the client-side application is provided in appendix C.1. In addition, images and descriptions of the web IDE's features are presented in appendix A.

Angular2 and TypeScript are used for developing UI components, communicating with the server-side, and structuring the business logic. Angular2 is selected over Ember and React, mainly due to its support for strongly typed languages. The main issue with Cloud9 is undocumented and weakly typed APIs, which makes extending the web IDE difficult. Using Angular2 which is well documented and strongly typed, is expected to address this issue.

The CodeMirror editor is also used in the pilot study. CodeMirror and Ace are nearly identical in features. However, for the pilot study, CodeMirror is desired over Ace, since it allows reusing the syntax highlighting rules from VDMPad.

For communicating with the server-side application, both AJAX and WebSockets are used. Communicating through a WebSocket is preferred over AJAX when sending messages frequently, and fast response times are required. This is due to the fact that a WebSocket keeps a TCP connection open, whereas AJAX opens and closes a TCP connection for each HTTP request, which adds additional latency [31].
Debugging requires frequent messages to be sent for requesting stack frames and performing stepping actions. Hence, the debugging feature uses a WebSocket.
Saving files also require frequent messages. However, this is implemented using AJAX to gain experience with the two different communication models.

The Google Sign-In library[3] is used to add a sign-in feature to the web IDE. This allows users to use their Google account to sign-in and get a personal workspace for VDM-SL projects.

[3] See `https://developers.google.com/identity/sign-in/web/`, accessed 11th of April 2016.

5.3.1.1 Evaluation of the Client-side Application

As part of the pilot study, a subset of the criteria defined in chapter 3 is implemented. These include debugging, an outline view, a proof obligations view, a file explorer, and a REPL. Angular2 proves useful for structuring the application and separating the UI from the business logic.

However, since Angular2 is a new framework and is in beta release, no mature UI library exists for Angular2. This means that no standard integration of components, such as CodeMirror, exists and therefore have to be developed. CodeMirror and the Google Sign-In library does not follow the Angular2 principles, such as data binding, dependency injection, and components [15]. However, Angular2 proves sufficiently flexible to enable integration.

The client-side application contains many components which have to react to changes from other components. An example is the editor which has to open files, show stack frames, and breakpoints based on changes in the debug component. This is handled by emitting events which are being listened to using the observable concurrency pattern [42]. Angular2 uses the observable library called RxJS[4], which is developed by Microsoft. The observable concurrency pattern makes this event-based communication between components manageable.

Saving a file is performed automatically every time the file is changed, by sending the entire content of the file to the server-side application. This approach puts unnecessary load on the network compared to only sending the changes. However, by using a debounce mechanism[5] the network load is reduced. After a change happens to the file, the debounce mechanism waits with sending the changed content, until 300ms has passed with no new changes. Hence, instead of saving the files on each key-press, the file is saved after a 300ms pause with no key-press.

Using TypeScript for the client-side application increases development productivity since the IDE used for developing the web IDE can offer code completion, based on types. In addition, TypeScript helps to understand the Angular2 framework API, since the documentation specifies types for all APIs. Finally, bugs are caught relating to types during the compilation step.

5.3.2 Server-side Application

The server-side application of the pilot study is intended to investigate the possibilities of integrating the Overture core into a web environment. Limitations are explored, regarding features of the Overture core to be provided in a web environment. Additionally, it is investigated how the Overture core performs with respect to concurrency.

The Abstract Syntax Tree (AST) of a VDM-SL model is required for implementing all model related features defined in chapter 3. In order to generate the AST, the VDM-SL model must be parsed and type checked. In addition, when a VDM-SL model is changed, the AST must be regenerated. Generating the AST can be computationally expensive and can therefore have an impact on the performance of the server-side application.

Using Cloud9 and Che as inspiration, the server-side application is designed as a REST API,

[4] See `https://github.com/Reactive-Extensions/RxJS`, accessed 25th of April 2016.

[5] See `http://reactivex.io/documentation/operators/debounce.html`, accessed 2nd of May 2016.

described in section 2.6.2. Additionally, the server-side application is implemented using the Play framework, evaluated in section 4.3.2. A more detailed description of the server-side applications design and implementation is provided in appendix C.2.

5.3.2.1 Evaluation of the Server-side Application

Features provided by the Overture core requires little effort to integrate on the server-side application. Features can be integrated using simple interfaces in the Overture core. However, mapping the data to a different format is required, since not all data formats are suited for HTTP communication. Using a Java framework enables code reuse from the Eclipse-based Overture Tool of features not supported by the Overture core. However, reusing features from the Eclipse-based Overture Tool is more difficult than integrating features from the Overture core. This is primarily due to the use of Eclipse platform dependencies in the implementations. This can have a negative influence on the reusability of the features, and require parts of the features to be implemented from scratch.

Concurrency issues have been observed in the Overture core implementation. The exact location of the problem has not been discovered. However, concurrent calls to the Overture core frequently generates concurrency related exceptions. The concurrency issues can make it difficult to successfully integrate the Overture core into a multi-user environment. Correcting the problems in the Overture core implementation will most likely be a significant task. This can require redesigning the Overture core. However, the concurrency issues have been mitigated to some extent in the pilot study. This is achieved by using a coarse-grain lock inside a class which wraps the interaction with the Overture cores parser and type checker [43]. The lock surrounds all communication involving parsing, type checking, and initiating the module interpreter. The lock also surrounds the preprocessing of the lists containing project files, to avoid concurrent modification[6] and index out of bounds[7] exceptions on these lists. However, this is not an ideal solution, since a coarse-grain lock sometimes surrounds more than it needs to. Hence, a coarse-grain lock can degrade the overall performance of the application, by blocking threads unnecessarily [43]. However, a load test of the server-side application has not been performed. It is therefore uncertain how the server-side application will perform under heavy loads.

As mentioned above, parsing and type checking VDM-SL models can be computationally expensive. In order to improve the performance, and attempt to mitigate the concurrency issues, a caching mechanism is used. When a VDM-SL model is parsed, type checked, and the module interpreter is initialized, the module interpreter is cached. The cached interpreter is used for all subsequent computations involving the corresponding VDM-SL model. If the VDM-SL model is changed after the caching, the VDM-SL model needs to be parsed, type checked, and initialized again.
However, the caching mechanism conflicts with the stateless principle of the REST architecture [30]. By using the cache, the server-side application is maintaining state on behalf of the client-side application. This can have a negative influence on the scalability of the server-side application. However, it has yet to be determined if the savings regarding unnecessary computation, is more valuable than the stateless principle when the server-side application experiences heavier loads.

[6] See `https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html`, accessed 10th of April 2016.

[7] See `https://docs.oracle.com/javase/7/docs/api/java/lang/IndexOutOfBoundsException.html`, accessed 10th of April 2016.

5.4. Evaluation of the Pilot Study

This section presents the evaluation results of the pilot study, according to the criteria defined in chapter 3. The evaluation of the pilot study only considers criteria which are currently supported. As a point of reference, the evaluation is compared to the evaluations of Che and Cloud9, which are the best rated web IDEs in chapter 4. In Appendix B, the evaluations of the development environment, coding assistance, runtime evaluation, and quality criteria are presented in table-form.

5.4.1 Editor

The CodeMirror editor is used in the pilot study. However, not all CodeMirror extensions are used, therefore the pilot study is rated worse than CodeMirror, as seen in Table 5.1 and Table 4.3.

Table 5.1: Evaluation results of editor criteria.

Criterion	**Web IDE**		
	Pilot study	Che	Cloud9
Annotate scrollbar	-	+	-
Auto indention	+	+	+
Close brackets	+	+	+
Code completion	+	+	+
Code folding	-	+	+
Code templates	-	+	+
Context menu	-	-	+
Edit history	+	+	+
Highlight matching brackets	+	+	+
Indent guides	-	-	+
Line wrapping	+	+	+
Linting	+	+	+
Multiple open files	+	+	+
Multiplex modes	+	+	+
Programmable gutters	+	+	+
Search	-	+	+
Search and replace	-	+	+
Syntax highlighting	+	+	+

5.4.2 Development Environment

Figure 5.1 shows that the pilot study does not perform well in the development environment criteria. As mentioned in chapter 3, these criteria are model agnostic and have therefore been given a low priority in the pilot study. Though these criteria are desirable for the web IDE, the pilot study mostly focuses on the integration with the Overture core.

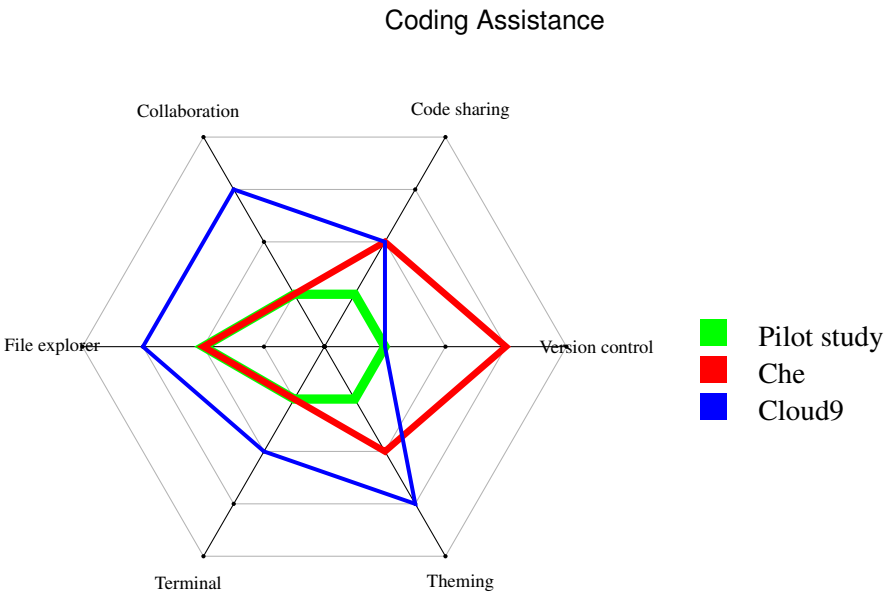


Figure 5.1: Evaluation results for development environment criteria.

5.4.3 Coding Assistance

In the coding assistance criteria, Che and Cloud9 achieves a better rating than the pilot study. However, for criteria supported by the pilot study, both the UI and integration with the Overture core is provided. Whereas, in Che and Cloud9, the rating only represents whether UI is provided for supporting the criteria.

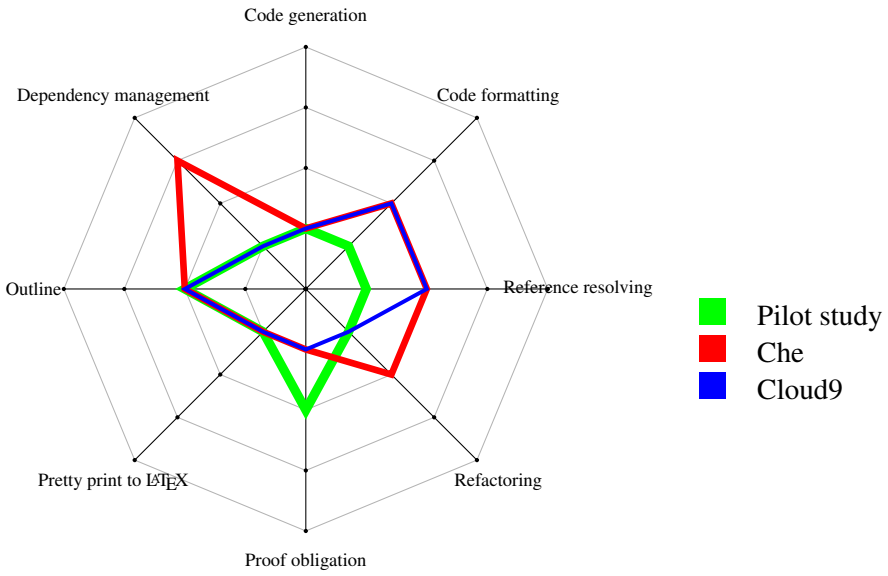


Figure 5.2: Evaluation results for coding assistance criteria.

5.4.4 Runtime Evaluation

In the runtime evaluation criteria, the pilot study and Cloud9 are equally matched, as seen in Figure 5.3. Additionally, both the pilot study and Cloud9 performs better than Che. However, as with the coding assistance criteria, in the pilot study both UI and the underlying integration with the Overture core is provided. In Che and Cloud9, only the UI elements are provided.

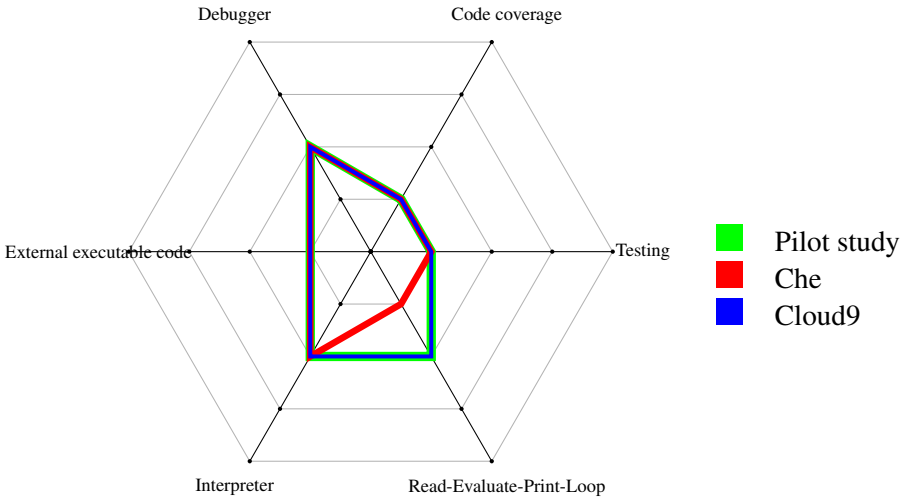


Figure 5.3: Evaluation results for runtime evaluation criteria.

5.4.5 Quality

In the evaluations on the quality criteria, the ratings for the pilot study are given as the worst rating among the frameworks and editor used in the pilot study. The evaluation on the quality criteria for the pilot study, therefore uses the evaluations of Angular2, Play, and CodeMirror from chapter 4. As a result, the pilot study achieves an better overall rating than Cloud9 in the quality criteria. The pilot study and Che achieves an equal overall rating on these criteria but differs in how the rating is distributed.

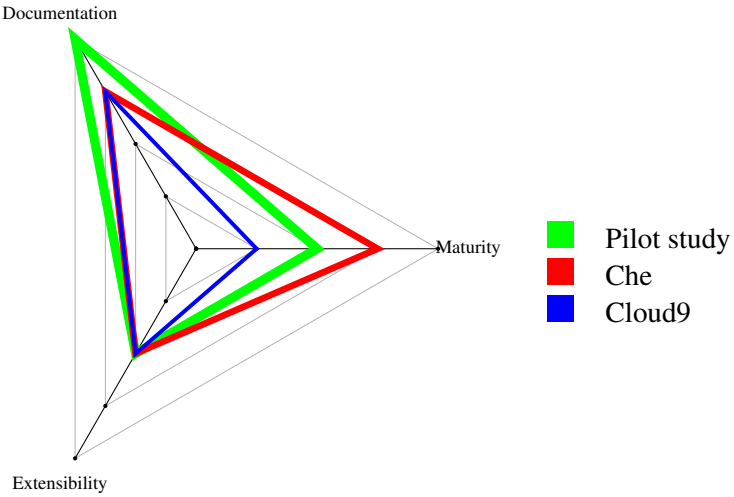


Figure 5.4: Evaluation results for quality criteria.

5.4.6 Evaluation Summary

A summary of the evaluations of the pilot study compared to Che and Cloud9 is seen in Table 5.2. The percentages and average percentages are calculated as in section 4.2.6.

Table 5.2: Summation of pilot study evaluation results.

Category	**Rating percentage**		
	Pilot study	Che	Cloud9
Editor	61	89	94
Development environment	9	45	73
Coding assistance	20	60	30
Runtime evaluation	43	29	43
Quality	71	71	43
Average percentage	**41**	**59**	**57**

As seen in Table 5.2, the pilot study project is inferior compared to Che and Cloud9 in the average percentage. However, it should be noted that the development of the pilot study spans over a period of approximately two months, while Che and Cloud9 have existed for more than two years. In addition, as mentioned above, criteria supported in the pilot study covers both UI and integration with the Overture core. Taking these two factors into account, the pilot study has come a long way within a relatively short period of time. Using well-established frameworks have aided the development to a large degree. The frameworks provide structure, and the extensive documentation and community around the frameworks is a valuable assets.

The criteria supported by the pilot study thus far, are mostly focused on integration with the Overture core. However, for the development environment criteria, it is worth looking into open-source software that can help in supporting these.

Chapter 6

Guidelines for Porting a Desktop IDE

In chapter 5, the pilot study is presented, which is conducted as part of this thesis project. In this chapter, experiences obtained during the evaluations in chapter 4 and the pilot study, are used to derive guidelines for porting a desktop IDE to a web IDE. Based on these guidelines, a set of improvements is included in the future work, which is presented in chapter 7.

6.1. Introduction

Guidelines exist for many software development disciplines and domains [44]. These guidelines describe best practices and considerations to be made during software development. Hence, the guidelines capture experiences, such that the experiences can be reused by other developers [1].

This chapter presents guidelines for porting an existing desktop IDE to a web IDE using open-source software. The guidelines represent lessons learned during this thesis project and will refer back to relevant parts of this thesis. The steps should be followed as presented, since each step builds on the preceding steps. Hence, the guidelines can also be used to assess the feasibility of porting a particular desktop IDE. If an early step cannot be fulfilled, the succeeding steps are irrelevant.

The remaining part of this chapter is structured as follows. Firstly, the limitations of web IDEs are presented in section 6.2. This is followed by a description of required properties of the existing desktop IDE in section 6.3. Then, an evaluation method for selecting the most suitable open-source software is described in section 6.4. Afterwards, the integration of the desktop IDE in a server-side application is described in section 6.5. Finally, development of the client-side application is described in section 6.6.

6.2. Limitations of Web IDEs

Knowing the limitations of a chosen platform before starting development, will help prevent time being spent on a project which cannot be completed. This section will describe the limitations of web IDEs, such that the feasibility of porting a particular desktop IDE can be assessed. Limitations of web IDEs concerns both technical limitations and disadvantages for users.

6.2.1 Browser Isolation

Client-side applications are isolated from the local machine on which it is running. This means that a web IDE cannot access the file system or software installed on the local machine [45]. Hence, applications which depend on the local file system, cannot be used together with a web IDE, such as a GUI application for version control.

In addition, external hardware connected to the local machine cannot be accessed by a web IDE. This includes hardware such as a USB connected test-bench for an embedded device. However, some exceptions exist including web cameras, microphones, and speakers which can be accessed through dedicated APIs in the browser.

6.2.2 Internet Connection

Since the file system, interpreter, and other tools run on a server, an internet connection is required to use a web IDE. This can limit the user's mobility since the user can only work in an environment with an internet connection. For instance, using a web IDE while commuting to work by train, requires a stable internet connection, as opposed to using a desktop IDE.

In addition, network latency is present between servers and clients. Network latency can degrade performance and make the web IDE unresponsive, especially on a slow internet connection. However, even with the network latency, a web IDE running on a fast server could potentially perform better than a desktop IDE running on a slow local machine, as described in section 1.1.5.3.

6.2.3 Service Provider Dependency

If a web IDE is hosted by an external service provider, the service provider constitutes an additional dependency [9]. This presents a lack of control over the web IDE for the users. Additionally, any outages at the service provider can diminish user productivity. Finally, upgrades of the web IDE cannot be controlled, and potential bugs in the web IDE are difficult to work around.

6.3. Required Properties of a Desktop IDE

A desktop IDE typically consists of two overall elements: a *GUI* and a *core* as seen in Figure 6.1. The GUI presents information to the user and allows the user to interact with the desktop IDE. The core handles the computations for the IDE features. In a web IDE the GUI is provided by the client-side application and the core is integrated in the server-side application, as described in section 1.1.5.3.

The client-side application must be developed using HTML, CSS, and JavaScript since these are the only languages that browsers can interpret. Hence, the desktop IDE's GUI cannot be reused unless it is developed using these languages. Additionally, all the features provided by the desktop IDE should be provided by the web IDE. This requires the core to be independent of the desktop IDE's GUI. However, an alternative approach is described in Appendix D. In this approach, certain features are implemented on the client-side, which provides different advantages and disadvantages.

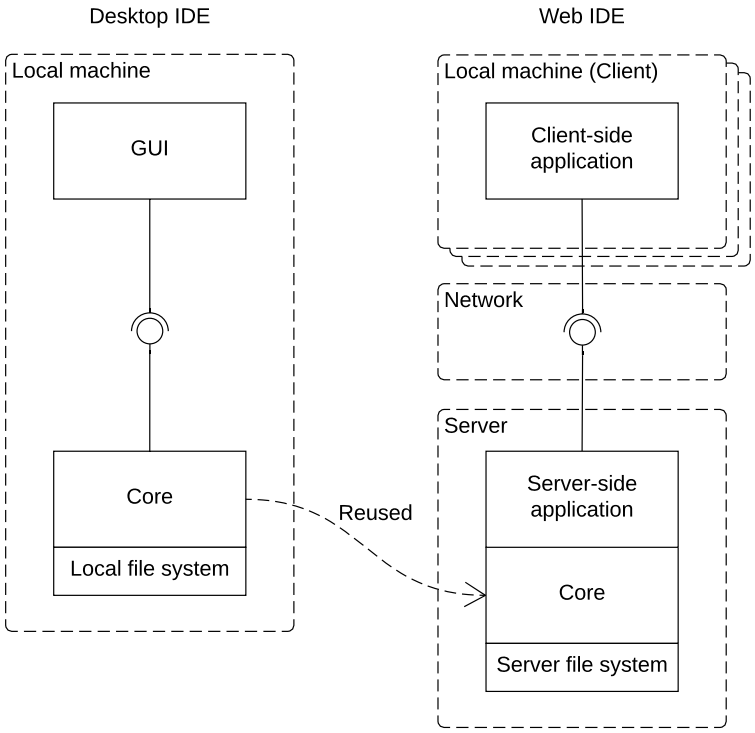


Figure 6.1: Illustration of how the core of a desktop IDE is reused in a web IDE.

In addition, concurrency properties of the core have to be considered. A web IDE imposes additional concurrency requirements on the core since multiple users can be requesting computations from the core concurrently. This needs to be taken into consideration when designing the server-side application architecture.

The pilot study reveals that the Overture core is not performing well when the client-side application is sending requests concurrently. Certain features are requested concurrently by the client-side application, which results in concurrency exceptions in the core.
The concurrency issue can be circumvented by using a queue or a locking mechanism, to process requests sequentially. However, this can potentially become a performance bottleneck when multiple users are using the web IDE.

6.4. Evaluate Software

Having determined that the core can be reused from the desktop IDE, different development approaches can be considered. Open-source software exists which can be reused for additional features of the web IDE. However, different approaches are possible regarding the granularity of software reuse. These different approaches provide different advantages and disadvantages.

The pilot study shows that, regardless of the granularity, the quality criteria are among the most important criteria defined in chapter 3. The web IDEs evaluated in this thesis project supports many criteria, but are lacking in the quality criteria. This can be an overwhelming obstacle and

extending the web IDEs can be cumbersome. Documentation is a particularly important quality criterion. The lack of proper and updated documentation can make understanding and modifying the software difficult [46, 47]. Therefore, if the documentation criterion is lacking, reusing the open-source software should be considered carefully, or possibly avoided. The issue of selecting the most appropriate open-source software for reuse can be addressed by using the QSOS method.

The QSOS method is applied in this thesis project for evaluating and selecting open-source software. The method is described in section 2.3. As mentioned, certain steps of the method are relaxed or omitted. However, the pilot study indicates that certain properties of the method are important. A summary of how the method can be applied along with lessons learned are presented in remaining part of this section.

The QSOS categories; *type of license* and *type of community*, does not need to be included in the evaluations. Instead, these categories can be considered during the search for open-source software. The software can then be rejected before the evaluation if the license hinders the reuse. Additionally, regarding the type of community, groups of developers and organizations should be favoured over sole developers. Rejecting software in the early stages of the process is intended to reduce the workload in the evaluations.

The *type of software* category, imposes a set of predefined maturity criteria. However, a subset of these criteria can be used in order to reduce the workload. Additionally, the method requires the definition of functional criteria of the software, such as the criteria defined in chapter 3. The process of defining the criteria and granularities is time consuming, and it is recommended to reuse the criteria defined in chapter 3. However, if the criteria cannot be reused it is not recommended to use QSOS. Instead, a cursory search for open-source software based on the quality criteria should be performed, followed by a pilot study.

As mentioned, the QSOS method is an iterative method. The level of detail used in the evaluations can be increased in each iteration, meaning that the software is studied in greater detail. This property of the method can be important. The existing web IDEs, which have been evaluated in this thesis project, seemed promising since they satisfied many of the functional criteria. However, the level of detail of the evaluations was increased by conducting a pilot study. The pilot study showed that extending the existing web IDEs was difficult and that the maturity and documentation were two decisive criteria. Hence, the importance of the documentation and maturity criteria should be emphasized when applying the method. This can be done by applying large weights to these criteria, giving them greater influence on the final rating. Finally, certain assumptions can be made in order to simplify the evaluations. Assumptions applied in this thesis project are described in section 4.1.

When applying the QSOS method, software of different granularities can be evaluated. However, the software evaluated will not always be comparable across the granularities. The choice of which granularity approach to select should depend mostly on how well the quality criteria are supported for the given approach.

6.5. Server-side Application

As mentioned above, different approaches can be selected regarding software granularity. Extending an existing web IDE can prove to be the best approach. In this case, the documentation and guides provided by the associated community should be followed. However, the guidelines in this section, are intended to be applied for a fine-grained approach using frameworks and components.

The development of the web IDE should start with the server-side application, which integrates the core. During integration of the core, limitations and possibilities can be discovered. This information can be used to define the interface between the client-side and server-side applications. The server-side application will then expose endpoints allowing the client-side application to utilize the core features.
The following sections provide recommendations and considerations for developing a server-side application. These include architecture, framework, resource, deployment, storage, and security.

6.5.1 Architecture

There are many architectural aspects which should be considered before development is initiated. The architecture of the server-side application can be implemented as a REST API, which is described in section 2.6. The REST architectural style has a variety of useful properties, due to statelessness and extensive use of the HTTP protocol [30]. However, as the pilot study shows, a REST API is not necessarily the most appropriate architecture for a web IDE. Compromises regarding the stateless principle can improve performance and mitigate concurrency issues. Additionally, the REST architecture separates resources conceptually, which results in separate endpoints for each IDE feature. This principle can introduce unnecessary latency, by increasing the amount of requests needed for achieving a task. A task requested by the client-side application will often require additional tasks. In the pilot study, when a model is changed, the changes are sent to the server-side application. Then, requests are sent for the new model outline, lint information, and proof obligations. This sequence of events are executed in separate requests, and are executed every time the model is changed. A single endpoint which returns outline information, lint information, and proof obligations in a single response can improve this latency issue.
Alternatively, an event-based architecture can also be considered. This architecture allows the server-side application to proactively send outline information, lint information, and proof obligations to the client-side application when a file is changed. This approach will require all communication between the client-side and server-side applications to be performed using a WebSocket. WebSockets are described in section 2.6.

6.5.2 Framework and Components

The server-side application should be developed using a framework. A framework provides domain-specific features, and additional features can often be added through extensions [14]. Additionally, reusing frameworks can help improve software quality and enhance the extensibility of the application [14]. However, many frameworks exist for server-side applications, making it difficult to select the most appropriate framework [1]. A method can be used to guide the selection process, as described in section 6.4 [13].

6.5.3 Protect Resources

Software development can involve computational intensive tasks and consume an excessive amount of memory. If not addressed, these tasks can degrade performance, or terminate the server-side application. This can necessitate that certain restrictions are enforced by the server-side application. For computationally intensive tasks, timeouts should be used to terminate the tasks, if they exceed a certain time limit. Limiting the amount of memory allocated for each user can require the use of containerization described in the following section and in section 2.6.

6.5.4 Deployment

Containerization involves wrapping the server-side application in a Docker container. This provides a number of advantages. Containerization ensures that the environment in which the web IDE is developed, tested, and deployed is the same. Additionally, certain types of software development require access to the OS environment. This is necessary in order to install dependencies and tools needed for the development. Containerization can be used to provide such an environment in a web IDE. Each user will be given a dedicated container. All client-side application requests from that particular user, is routed to the user's container. Hence, the user will be provided an isolated environment, in which additional libraries and dependencies can be installed. In addition, since the environment is isolated, changes in the environment will not affect other users' environments, nor will it affect the environment of the server hosting the server-side application. This also makes it possible to restrict the amount of resources allocated for each user, regarding memory, storage, and CPU.

Additionally, this strategy can help mitigate concurrency issues arising due to the multi-user nature of web applications. Each container will contain an instance of the server-side application including dependencies, such as the Overture core, and is therefore only accessed by one user.

A drawback of this strategy is the additional overhead associated with allocating resources, spawning, and running a dedicated container for each user [34]. Additionally, not all programming languages or modelling languages require an environment which can be customized. Therefore, it must be considered whether these advantages are needed.

6.5.5 File Storage

An inherent feature of web IDEs is the ability to store files. The files constitute user's projects, consisting of directories and files. Each user should have an isolated workspace, containing the user's projects. Two important considerations regarding the workspace are how to properly isolate the workspaces, and how to store the files safely.
Containerization can be used to isolate the workspaces, by giving each user a dedicated container. However, in this thesis project, a more lightweight approach has been used. In this approach, the isolation is handled in the code using the Commons VFS Java library[1].
In addition to providing proper workspace isolation, a file storage policy must be considered. The core may require the files to be accessible on the local file system. Hence, files will need to be stored temporally on the server on which the server-side application is running. However, backups should be stored on external cloud services, such as Amazon S3[2]. Mechanisms should therefore be added in order to ensure that files are backed up regularly.

[1] See `https://commons.apache.org/proper/commons-vfs/`, accessed 21st of April 2016.
[2] See `https://aws.amazon.com/s3/`, accessed 21st of April 2016.

6.5.6 Authentication and Authorization

Users will need to be identified in order to provide them with unique workspaces. For this purpose, it is recommended that existing protocols are used, such as the Open Authorization (OAuth) protocol [7]. The OAuth protocol allows users to reuse their existing accounts from other service providers, such as Facebook, Google, and Github. This eliminates the need for developing ad hoc authentication systems[3].

6.6. Client-side Application

This section presents general recommendations regarding the development of the client-side application of a web IDE. The recommendations include reuse of existing software, browser support, concurrency patterns, statically typed languages, and optimizing network load.

6.6.1 Frameworks

It is recommended to use a framework and either Ace or CodeMirror. No other UI components are found which satisfies the needs of the pilot study, such as a file explorer, menus, and panels. Hence, these must either be developed or reused from the pilot study. When choosing a framework, integration with the editor must be considered.

A client-side application should be able to run in many different browsers. Following the World Wide Web Consortium (W3C) standards[4], will not enable the client-side application to work in all browsers, since the browsers implement the standards slightly differently. Hence, when developing a client-side application, a set of supported browsers should be defined and tested against. Supporting all existing browsers is infeasible. In addition, it is expected that users of a web IDE tend to mostly use modern browsers. Checking browser support, of a specific part of the standard, can be done using the "Can I use" website[5]. Developers of frameworks and libraries usually define a set of browsers which are supported. Hence, a way of supporting multiple browsers is by using these frameworks and libraries.

6.6.2 Concurrency

A JavaScript application handles concurrency using an event queue, as described in section 2.6.4. An event queue model fits well for GUI applications in general, where a user can asynchronously dispatch many different events, e.g. from clicking buttons. The standard way of registering an event handler in JavaScript, is by using a callback. However, handling complex asynchronous tasks with callbacks can lead to a decrease in code readability, referred to as "callback hell" [48]. Concurrency patterns exist for addressing this issue, which including *promises* and *observables* [48]. Promises are used for handling asynchronous tasks which return a single value, such as HTTP requests. Observables are used for handling asynchronous tasks which return a stream of values, such as WebSockets. It is recommended to use these patterns for handling asynchronous tasks. In addition, promises and observables are integrated into some modern frameworks, such

[3]Though the protocol is called open authorization, it can also be used for authentication.
[4]See `https://www.w3.org/standards/`, accessed 25 of April 2016.
[5]See `http://caniuse.com`, accessed 24th of April 2016.

as Angular2 and Ember [15, 24].

In JavaScript, all events are handled on the same thread as the GUI. Hence, heavy computations in event handlers will block the GUI, giving a bad user experience. A WebWorker should be used for handling heavy computations as it runs in a separate thread.

6.6.3 Statically Typed Languages

Statically typed languages can improve development time, and make it easier for developers to add new features [49]. Additionally, statically typed languages can aid developers find and fix bugs [49]. Therefore, using a statically typed language can be beneficial for the development and continuous maintenance of a client-side application.

Most browsers can only interpret JavaScript which is not statically typed. However, multiple languages exist which can be compiled to JavaScript[6] and offers static types. TypeScript offers optional static types and syntactic sugar for commonly used JavaScript patterns [39]. Dart forces static types, as opposed to TypeScript, and can therefore do certain optimizations [40].

However, statically typed languages are not widely used in client-side applications, and developer familiarity can therefore be limited. In addition, during the evaluations in section 4.3, it was observed that libraries and frameworks for client-side applications are only beginning to adapt statically typed languages. Therefore, although there are certain arguments that speak for selecting a statically typed language for the client-side application, it should still be considered carefully.

6.6.4 Communication

Whenever a file is changed in the client-side application, the information has to send to the server-side application for storage and processing. A naive approach is to send the entire file content to the server-side application, which can then simply overwrite the existing file content. However, to minimize the information sent over the network, it is recommended to only send the changes of the file. Both CodeMirror[7] and Ace[8] includes an API for retrieving changes to a file. Only sending changes will reduce network load.

Since changes can happen multiple times per second, when a user is typing, it is recommended to batch changes before sending them to further reduce network load. This can be done by batching changes while the user is typing, and then sending the changes when the user stops typing for some small period of time, using the debounce mechanism described in subsection 5.3.1.

[6] See `https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js`, accessed 25th of April 2016.

[7] See `http://codemirror.net/doc/manual.html#event_changes`, accessed 6th of May 2016.

[8] See `https://ace.c9.io/#nav=api&api=document`, accessed 6th of May 2016.

Chapter 7

Concluding Remarks

This chapter concludes the thesis. Findings are presented and the hypothesis defined in chapter 1, is evaluated. An evaluation of whether the goals from chapter 1, have been achieved, is presented. The approach applied in the thesis project is reflected upon and discussed. Additionally, future work consisting of improvements on and investigations of the pilot study is presented.

7.1. Introduction

Following the completion of any project, achievements and approaches need to be reflected upon. The experiences should be extracted, in order to be applicable in future projects. These experiences can be valuable for people involved in the project, but also for people outside the project working in related fields. Additionally, further investigations and improvement of a project can be required. Future work of a project should therefore be described.

This thesis project has explored the possibilities of developing a web IDE supporting VDM-SL by reusing open-source software. Criteria for such a web IDE have been defined, and reusable open-source software have been evaluated. The results of the evaluations have been applied in a pilot study. The outcome of the pilot study has been used to derive a set of guidelines for others to follow. This chapter reflects on the process, presents findings, discusses achievements and mistakes, and describes future work.

The remaining part of this chapter is organized as follows. First, the thesis project scope and approach is reflected upon in section 7.2. Next, the thesis project work and the hypothesis is concluded upon in section 7.3. Afterwards, achievement of the thesis project goals are discussed in section 7.4. Then, in section 7.5 future work on the thesis project is proposed. Lastly, final remarks are provided in section 7.6.

7.2. Discussion

7.2.1 Project Scope

This thesis project deals with many different aspects of web development, including client-side software, server-side software, network communication strategies, and deployment strategies. In addition, work has been put into investigating the possibilities of integrating the Overture core in a web IDE. This means that the results are based on the full spectrum of web development. However, even with the authors' knowledge of web development from previous projects, there were still not enough time to go into detail with all the aspects. If a narrower project scope had been selected, such as focusing on integrating the Overture core in a server-side application, further progress could have been achieved on that particular aspect.

In addition, the selection process was affected by the width of the project scope. There exists a large number of languages, frameworks, and components for web development. Due to the time restrictions, a small subset of the software was selected for evaluation without examining them in detail first. This selection was based on criteria including the license, community behind the software, documentation, and general popularity. Hence, the best possible solution was not necessarily found, but the pilot study proves that building a web IDE for VDM-SL with a reuse-based approach is feasible.

7.2.2 Project Approach

The QSOS method has been applied in order to guide the evaluation and selection of open-source software. The intent behind applying a method for this process was to provide a basis on which to select among the large amount of open-source software. The method is comprehensive, and due to the limited time frame of this thesis project, compromises were made. However, applying the method in a reduced form, does not provide a comprehensive picture of the methods usefulness.
In this reduced form the method yielded software which were successfully applied in a pilot study. However, existing web IDEs were also selected based on the outcome of the method. Extending these web IDEs were aborted due to lacking quality criteria.
In its full form, the method proposes a set of maturity criteria, of which only a subset was applied in this thesis project. Had the method been applied in its full form, it is possible that the web IDEs would have received a lower rating. Hence, the web IDEs might not have been selected for the pilot study in this case.

The work conducted during the pilot study could have been more productive. A lot of time was spent on trying to integrate the Overture core with existing web IDEs, before trying the approach with frameworks and the editor. By investigating both approaches simultaneously, the best approach could possibly be found earlier, and the following work could be focused on that approach.

Some of the work conducted during the pilot study, might not have been particularly important for investigating whether a web IDE could be developed for VDM-SL with a reuse-based approach. This includes some UI components, such as a file explorer, resizable panels, and menus. In addition, much work was put into handling file system actions in the server-side application e.g. handling naming collisions when moving and creating files. However, this work was conducted to gain experience with the software and evaluate the practicality of using the software for a web IDE.

7.2.3 Published Material

When working with the newest web technologies, little published material is available, such as research articles. No articles could be found for Angular2 and Typescript, but unofficial books exist [15, 39]. Instead, online documentation had to be relied upon. Additionally, no material could be found on Che and Cloud9, other than online documentation.
Ace and CodeMirror are referenced in articles about web IDE research projects [20, 9, 50, 51, 52]. However, none of these research projects used well-established frameworks, such as Angular2 and Play, but instead developed new frameworks.

7.3. Conclusion

This thesis project investigated the possibilities of supporting VDM-SL in a web IDE using a reuse-based development approach. This was conducted by first defining criteria for a web IDE supporting VDM-SL. Then, existing software of different granularities was evaluated for reuse, based on the defined criteria. Afterwards, the best rated software was used in a pilot study to assess the feasibility of the different approaches regarding software granularity. Finally, the results from the pilot study led to the definition of general guidelines for developing web IDEs. Additionally, a description is provided on how the work conducted in this thesis project, can be continued in future projects.

The evaluations described in chapter 4, showed that existing open-source web IDEs integrated many features and can be extended to support additional languages. However, due to lack of maturity and lack of documentation, it was impractical to integrate support for VDM-SL as described in section 5.2.
Based on the results of the pilot study, it is recommended to favour quality in the evaluations, where quality includes documentation, extensibility, and maturity. For supporting these criteria, well-established frameworks can be used for the web IDE. In addition, using and developing self-contained reusable components for each individual feature of the web IDE enables reuse and extensibility.

The hypothesis for this thesis project, as defined in section 1.3, states:

It is possible to develop a web IDE supporting VDM-SL using a reuse-based development approach and open-source software.

Based on the pilot study it can be concluded that the hypothesis is true, since a web IDE for VDM-SL was developed as part of the pilot study, by reusing open-source frameworks, an editor, and the Overture core. In addition, it can be concluded that the quality criteria are critical for success. Therefore, a set of guidelines were defined in chapter 6, which includes a method for selecting open-source software for reuse for developing web IDEs.

7.4. Evaluation on the Achievement of the Goals

As part of this thesis project, three goals were defined in addition to the hypothesis. The goals were presented in section 1.3 and are shown again below:

Goal 1: Explore the possibilities and limitation of integrating the Overture core in a web IDE.

Goal 2: Investigate whether a method can be used to assess the feasibility of reusing a particular open-source software unit.

Goal 3: Provide guidelines for the development of web IDEs using a reuse-based approach.

These goals are evaluated below by explaining why each goal is considered to be achieved and which parts of the thesis that relate to the goal.

Goal 1: This goal was *successfully achieved*. As part of the pilot study, described in chapter 5, the possibility and limitations of integrating the Overture core with both an existing web IDE and a framework were explored. The exploration resulted in a proof of concept and proposals for future work, as described in section 7.5. However, the exploration also showed that the Overture core has some concurrency issues in a web environment.

Goal 2: This goal was *successfully achieved*. The evaluation, as described in chapter 4, was based on the QSOS method for evaluating open-source software for reuse. The method was applied in a pragmatic manner due to time constraints and yielded software for reuse. However, it was not clear until the pilot study, described in chapter 5, whether the high or low granularity approach was preferable. In addition, defining criteria and ratings is time consuming so these should be reused when possible. Based on experiences obtained by using this method, important properties of the method have been extracted and are presented in chapter 6.

Goal 3: This goal was *successfully achieved*. Guidelines are derived based on the evaluation, described in chapter 4, and the pilot study described in chapter 5. These guidelines describe how a desktop IDE can be ported to a web IDE. First, a set of limitations of web IDEs are described in section 6.2. These limitations are intended to be considered before the development of a web IDE is initiated. Then, requirements for a desktop IDE to be ported to a web IDE, are described in section 6.3. Afterwards, evaluation and selection of open-source software are described in section 6.4. Finally, recommendations and considerations regarding the development of web IDEs are described in section 6.5 and section 6.6.

7.5. Future work

The web IDE developed during the pilot study does not support all the criteria defined in chapter 3. The main competitors are the Eclipse-based Overture Tool and VDMPad. Currently, the pilot study web IDE exceeds VDMPad regarding supported criteria. However, the Eclipse-based Overture Tool is far ahead. An evaluation summary of these three IDEs is presented in Table 7.1. These evaluations only consider criteria which are currently supported by the IDEs. The evaluations on each category are presented in table-form in Appendix B.

Table 7.1: Summation of IDE evaluation results.

Category	Rating percentage		
	Eclipse Overture Tool	Pilot study	VDMPad
Editor	78	61	22
Development environment	64	9	0
Coding assistance	50	20	10
Runtime evaluation	86	43	43
Quality	100	71	71
Average percentage	**76**	**41**	**29**

The evaluation summary seen in Table 7.1 shows the difference between the three IDEs. VDMPad is a simple web IDE which is easy to use. Consequently, only a small number of criteria are supported. The Eclipse-based Overture Tool is located on the opposite end of the scale, since it supports almost all the criteria defined in chapter 3. However, it has a noisy UI and can therefore be difficult to use. In the middle of the scale is the pilot study web IDE. This web IDE provides a simple UI, but requires additional features in order to compete with the Eclipse-based Overture Tool.
This section describes how the remaining criteria defined in chapter 3 can be supported by the pilot study web IDE. Additionally, a description is provided on how the web IDE can be extended to support VDM-RT and VDM++. Furthermore, additional use cases are described for individual components of the web IDE.

7.5.1 Use Cases

This section describes additional use cases of VDM in a browser.

REPL on the Overture tool website: Certain official websites for programming languages, such as Python[1] and Haskell[2], have a REPL feature embedded on the front-page. This allows visitors to try the languages in the browser without installing software or creating a user account.
A REPL can be added to the Overture tool website[3], by using the REPL component and the server-side application developed in the pilot study. The server-side application depends on the workspace of an authenticated user. However, the REPL feature of the server-side application can be made stateless, meaning that the model is sent together with the expression to be evaluated. This allows anonymous users visiting the website to use the REPL feature.

Online VDM exercises: Online courses, as described in section 1.2, can be developed using components developed in the pilot study[4,5]. Online courses can include tutorials and graded exercises and integrate all the necessary modelling tools to complete the exercises. In addition, integrating automatic unit testing can give students instant feedback on the correctness of their work. Finally, instructors will be able to monitor the students' progress and understanding of the course material, since the students work is available on the server [9].

[1] See `https://www.python.org/`, accessed 14th of April 2016.
[2] See `https://www.haskell.org/`, accessed 14th of April 2016.
[3] See `http://overturetool.org/`, accessed 14th of April 2016.
[4] See `https://www.codecademy.com/en/tracks/python/resume`, accessed 31st of March 2016.
[5] See `https://tryhaskell.org/`, accessed 31st of March 2016.

7.5.2 Improvements

The evaluations seen in Table 7.1, shows that the pilot study supports fewer criteria than the Eclipse-based Overture Tool. The following three sections present improvements of the pilot study web IDE. Section 7.5.2.1 describes how to support criteria defined in chapter 3, that is not currently supported in the pilot study. Additionally, it is described how to extend the pilot study to support VDM++ and VDM-RT. Section 7.5.2.2 describes criteria beyond those supported by the Eclipse-based Overture Tool. Lastly, section 7.5.2.3 describes aspects of the pilot study, that should be investigated further.
Rough estimations of the time required to implement the features discussed in this section are provided in Appendix E.

7.5.2.1 Matching the Eclipse-based Overture Tool

Editor: Annotating the scrollbar can be implemented using an existing CodeMirror extension[6]. The scrollbar can be annotated with warning and error information, through the extension. In addition, the search and replace feature can be implemented using an existing CodeMirror extension[7].

The context menu is a UI component. This UI component is displayed at the cursor's position, when right-clicking in the editor. Hence, this feature is independent of the Overture core, and will only require extending the client-side application.

Code templates require UI support for inserting the templates into the model in the editor. In addition, UI should be provided, that allows users to define new templates. The templates should then be persisted in a database on the server-side application.

Development environment: Theming concerns the styling of the UI. Supporting this feature requires creating different themes using CSS-files. The themes will specify the colour of all UI components and styling of fonts. Additionally, UI components will be required for selecting different themes.

Coding assistance: The features; code generation and pretty print to LaTeX, have not been implemented in the pilot study. However, the code transformations and LaTeX data can be generated using the Overture core. The generated data can then be written to files in the users project directory. This makes the data accessible to the users through the file explorer.

The code formatting feature is supported in VDMPad [20]. Hence, reusing the implementation in VDMPad should be investigated.

Dependency management requires the ability to detect that a dependency is missing in a model and that the dependency is a VDM library. Additionally, after determining that the dependency is missing, the library should be imported into the user's project directory. Importing the dependency can be handled similarly to how importing example projects into the user's workspace is handled.

Reference resolving is not specified in Figure 2.4, but the feature is supported by the Eclipse-based Overture Tool. Hence, the implementation of the Eclipse-based Overture Tool should be investigated for reuse.

[6] See `http://codemirror.net/doc/manual.html#addon_annotatescrollbar`, accessed 7th of May 2016.

[7] See `http://codemirror.net/doc/manual.html#addon_searchcursor`, accessed 7th of May 2016.

Runtime evaluation: The code coverage feature requires highlighting evaluated paths of the VDM-SL model, in the editor. The code coverage information can be generated using the Overture core. The client-side application can then implement the feature by using the generated code coverage information to highlight the evaluated paths. Highlighting the evaluated paths is similar to highlighting warnings and errors in the editor and is therefore possible.

The testing feature is supported in VDMPad. Hence, reusing the implementation from VDMPad should be investigated.

Requirements for supporting external executable code needs further investigation. The feature can be implemented similarly to the Eclipse-based Overture Tool implementation. This requires JARs to be located in a library directory of a VDM-SL project. This allows the VDM-SL models to call functionality provided by the JARs. However, the JARs will sometimes implement a Java GUI, or require user input. This will not be possible in the web IDE. The JARs can only be executed on the server, therefore the functionality provided by the JARs cannot be directly accessed by a user. Hence, the JARs can only be allowed to communicate with the VDM model. Additionally, the communication between the VDM-SL model and the JARs can only be text-based.
All GUI features must be executed in the client-side application. To this end, the server-side application could provide instructions for the client-side application, on how to draw the graphics.

VDM dialects: The two additional dialects VDM++ and VDM-RT needs to be supported in the future. The features defined in chapter 3, are required for all three dialects. However, support for combinatorial testing and UML mapping are required for both VDM++ and VDM-RT. Additionally, visualization support is required for data generated by VDM-RT. This VDM-RT feature will be referred to as *Real-time Log Viewer (RLV)*.

The features implemented in the pilot study web IDE can be extended to support VDM++ and VDM-RT. This requires creating wrapper classes for parsing and type checking VDM++ and VDM-RT models. These wrapper classes will have similar responsibilities to the wrapper class for VDM-SL described in appendix C.2.4.1. The debugging feature implemented in the pilot study can be extended to support the additional dialects. This requires an additional input parameter, informing the debugging implementation which dialect is being evaluated. The REPL feature will have to be implemented from scratch for the remaining dialects. However, as mentioned in appendix C.2.4.2, the REPL feature for VDM-SL is inspired by a similar feature for VDM++.

The UML mapping and RLV features are supported by the Eclipse-based Overture Tool. Further investigation is required to determine the extent to which the implementation can be reused. However, regardless of whether the implementation can be reused, additional visualization support is required. Visualizing the RLV data must be handled by the client-side application. The current implementation in the Eclipse-based Overture Tool uses Java GUI. Hence, the existing visualization implementation cannot be reused directly. The visualization will most likely have to be implemented from scratch using HTML, CSS, and JavaScript.

7.5.2.2 Beyond the Eclipse-based Overture Tool

Editor: Code folding requires knowledge of the modelling languages. Folding a construct that spans multiple lines, requires the ability to determine where the construct starts and ends.

This feature is not supported by the Overture core. Therefore, further investigation is required for determining how this feature can be implemented. However, this feature is supported by a CodeMirror extension[8].

Indent guides can be implemented without knowledge of the modelling language. The implementation will use the number of tabs or spaces, to match lines vertically, which are indented to the same level. However, this feature is not supported by CodeMirror and needs to be investigated further.

Development environment: The terminal feature can be addressed using Docker containers. This will provide users with an isolated and customizable environment, as described in section 6.5.

Collaboration is not supported in the pilot study. Synchronization is required for enabling multiple users to simultaneously edit a VDM-SL model [53]. Changes can be pushed to users by using WebSockets. However, investigations are required for determining when to generate the AST for the lint, outline, and proof obligations features. Currently, the AST is generated every time the VDM-SL model is changed. While multiple users are editing the same VDM-SL model concurrently, the model will often not be parsable. Therefore, generating the AST every time the model is changed, is inappropriate.
Additionally, different levels of collaboration can be considered. A remote pair-programming model might to easier to implement. This model allows one user to edit the model. Other users will be limited to viewing the changes in real-time. However, users will be able to communicate using a chat feature.
Another level of collaboration is the code sharing feature defined in chapter 3. This type of collaboration will not require synchronization. One user will own the model. The model can then be shared with other users through the web IDE. However, the other users will only be able to evaluate the model, not make changes.

Version control is a language agnostic feature. The server-side application must implement logic for communicating with service providers, such as Github and Bitbucket. These service providers expose APIs for interacting with the repositories[9,10]. Repositories need to be cloned onto the servers file system, for the server-side application to evaluate the model. Additionally, UI support is required on the client-side application, allowing users to perform version control operations.

Coding assistance: Refactoring is not yet supported by the Overture core. When the feature becomes available, the integration will have to be investigated. Additional UI will be required in the client-side application, allowing users to trigger the refactoring on a specific part of a model.

7.5.2.3 Further Investigations

Network communication: The client-side and server-side applications are currently using two communication models. The debugging feature uses a WebSocket, but all other features uses HTTP requests. However, it needs to be investigated in which cases one model should be preferred over the other. In particular, the push mechanism provided by WebSockets will

[8] See `http://codemirror.net/doc/manual.html#addon_foldcode`, accessed 7th of May 2016.
[9] See `https://developer.github.com/v3/`, accessed 22nd of April 2016.
[10] See `https://confluence.atlassian.com/bitbucket/use-the-bitbucket-cloud-rest-apis-222724129.html`, accessed 22nd of April 2016.

be useful for the collaboration feature. In addition, the event-based architecture described in section 6.5, will require WebSockets for all communication.

Additionally, aspects for reducing network traffic should be investigated. When changes to a model are saved in the client-side application, a request is sent to the server-side application. This request contains the entire VDM-SL model. Instead, only the changes should be sent to the server-side application. CodeMirror supports extracting changes in the editor. Hence, it should be investigated how to handle merging the changes with the model persisted by the server-side application.

Revisit existing web IDEs: During the pilot study, existing web IDEs have been investigated. The investigation showed that these web IDEs have not yet reached an adequate maturity level. Extending the web IDEs were therefore aborted. However, the evaluation results of these web IDEs justifies a re-evaluation at a later time. If extending the existing web IDEs becomes feasible, components developed during the pilot study can possibly be reused.

7.5.2.4 Future Work Summary

The pilot study showed that features which are supported by the Overture core requires minimal effort to integrate in the server-side application. However, the concurrency issues in the Overture core must be addressed in order to improve the quality of service.
Certain features require specific UI support in the client-side application. Components developed during the pilot study can be used to support some of the missing criteria, such as the context menu. Additional components will have to be developed.

Certain features are supported by the Eclipse-based Overture Tool, but not by the Overture core. These features can be reused to some extent, but the implementation often contains Eclipse platform dependencies. This makes reusing the features more challenging, as was the case with the code completion feature. In the pilot study, features have been implemented as components, which are independent of the applied server-side framework. This will make reusing the components easier for future projects.

This section describes certain challenges involved in supporting the remaining features defined in chapter 3. However, in spite of these challenges, most of the features are possible to support in some form in a web IDE. Certain features cannot be implemented exactly as in the Eclipse-based Overture Tool. These features will need to be modified in order to fit into a web environment. However, it is possible that a web IDE can provide the same features as the Eclipse-based Overture Tool. Additionally, developing an IDE as a web service offers many additional opportunities for the users, as described in chapter 1.

7.6. Final Remarks

The results presented in this thesis sets the foundation for developing a web IDE for VDM-SL and the other VDM dialects. The results include both guidelines and a web IDE developed as part of the pilot study. The web IDE, or parts thereof, can be reused and extended for developing a complete web IDE for the VDM dialects.

During this thesis project, the web IDE was tried out by researchers in multiple countries which

provided valuable feedback. We hope that our results will be used in further development of a web IDE, and possibly other online tools, for the VDM dialects.

Bibliography

[1] I. Sommerville, *Software Engineering*. Pearson, 2010. [cited at p. viii, 1, 2, 5, 12, 13, 30, 53, 57]

[2] P. G. L. Tomohiro Oda, Keijiro Araki, "Exploratory formal specification and its support tool," *Formal Methods in Software Engineering (FormaliSE), 2015 IEEE/ACM 3rd FME Workshop on*, pp. 33–39, 2015. [cited at p. viii, 16]

[3] M. F. J. F. K. L. M. V. P.G. Larsen, N. Battle, "The overture initiative integrating tools for vdm," *SIGSOFT Softw. Eng. Notes*, vol. 35, pp. 1–6, January 2010. [cited at p. 1, 15]

[4] M. Jazayeri, "Some trends in web application development," *Future of Software Engineering(FOSE'07)*, 2007. [cited at p. 2, 11, 16, 17]

[5] B. Fitzgerald, "A critical look at open source," *IT Systems Perspectives*, pp. 92–94, 2004. [cited at p. 2, 5, 30]

[6] M. A. B. Klaas-Jan Stol, "A comparison framework for open source software evaluation methods," *OSS 2010, IFIP AICT 319*, vol. 389-394, 2010. [cited at p. 2, 13]

[7] G. Fylaktopoulos, G. Goumas, M. Skolarikis, A. Sotiropoulos, and I. Maglogiannis, "An overview of platforms for cloud based development," *SpringerPlus*, vol. 5, pp. 1–13, 2015. [cited at p. 3, 4, 5, 31, 59]

[8] C. Gacek and B. Arief, "The many meanings of open source," *IEEE Software*, vol. 21, pp. 34–40, January 2004. [cited at p. 3, 27]

[9] K. T. K. E. V. Lennart C. L. Kats, Richard G. Vogelij, "Software development environments on the web: A research agenda," p. 18, 2012. [cited at p. 3, 4, 5, 21, 54, 63, 65, 105]

[10] D. Cameron, *A Software Engineer Learns HTML5, JavaScript and jQuery*. Cisdal Publishing, 2013. [cited at p. 11, 12, 18]

[11] N. C. Tim Ambler, *JavaScript Frameworks for Modern Web Dev*. apress, 2015. [cited at p. 11]

[12] Atos, *Qualification and Selection of Open Source software (QSOS)*, version 2.0 ed., 2013. [cited at p. 11, 13]

[13] D. C. Schmidt, A. S. Gokhale, and B. Natarajan, "Leveraging application framework," *ACM Queue*, pp. 66–75, 2004. [cited at p. 12, 30, 44, 57]

[14] M. E. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Communications of the ACM*, vol. 40, pp. 32–38, 1997. [cited at p. 12, 57]

[15] N. M. C. T. Ari Lerner, Felipe Coury, *ng-book 2 The Complete Book on AngularJS 2*. gistia, 2016. [cited at p. 13, 17, 37, 46, 60, 63]

[16] J. Richard-Foy, *Play Framework Essentials*. PACKT PUBLISHING, 2014. [cited at p. 13]

[17] *Overture VDM-10 Tool Support: User Guide*, 2.3.0 ed., September 2015. [cited at p. 15, 21]

[18] J. W. J. des Rivieres, "Eclipse: A platform for integrating development tools," *IBM Systems Journal; 2004; 43, 2; ProQuest*, 2004. [cited at p. 15]

[19] D. R. Shane Caraveo, "Dbgp - a common debugger protocol for languages and debugger ui communication," 2013. [cited at p. 15]

[20] P. G. L. Tomohiro Oda, Keijiro Araki, "Vdmpad: a lightweight ide for exploratory vdm-sl specification," *2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering*, 2015. [cited at p. 15, 41, 63, 66]

[21] P. Larsen, "Ten years of historical development "bootstrapping" vdmtools¨.," *Journal of Universal Computer Science*, 2001. [cited at p. 15]

[22] D. Flanagan, *JavaScript: The Definitive Guide Activate Your Web Pages*. O'Reilly Media, Inc., 6th ed., 2011. [cited at p. 16]

[23] J. J. Garrett, "Ajax: A new approach to web applications," tech. rep., Adaptive Path, 2005. [cited at p. 16]

[24] T. Q. B. Jesse Cravens, *Building Web Apps with Ember.js*. O'Reilly Media, Inc., 2014. [cited at p. 17, 37, 60]

[25] S. S. Brad Green, *AngularJS*. O'Reilly Media, Inc., 2013. [cited at p. 17]

[26] K. P. A. M. Frolin S. Ocariza, Jr., "Detecting inconsistencies in javascript mvc applications," *IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015. [cited at p. 17]

[27] J. VepsŠlŠinen, *SurviveJS - Webpack and React From apprentice to master*. Leanpub, 2016. [cited at p. 17, 37]

[28] J. Overson and J. Strimpel, *Developing Web Components: UI from jQuery to Polymer*. O'Reilly, 2015. [cited at p. 17]

[29] C. F. X. Z. Y. W. Haijiao Kou, Zhigang Wen, "Research of restful web services for the resource sharing platform," *AIAI2010*, 2010. [cited at p. 17]

[30] M. MassŽ, *REST API Design Rulebook*. O'Reilly, 2012. [cited at p. 17, 47, 57]

[31] B. G. N. V. Pimentel, "Communicating and displaying real-time data with websocket," *IEEE Internet Computing (Volume:16, Issue: 4)*, 2012. [cited at p. 17, 45]

[32] S. S. D. Skvorc, M. Horvat, "Performance evaluation of websocket protocol for implementation of full-duplex web streams," *MIPRO 2014*, 2014. [cited at p. 17]

[33] M. K. S. Hong, Y. Park, "Detecting concurrency errors in client-side java script web applications," *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014. [cited at p. 18]

[34] R. R. J. R. W. Felter, A. Ferreira, "An updated performance comparison of virtual machines and linux containers," *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015. [cited at p. 19, 58]

[35] K. Matthias and S. P. Kane, *Docker: Up & Running*. O'Reilly, 2015. [cited at p. 19]

[36] H. H. Iyad Zayour, "How much integrated development environments (ides) improve productivity?," *JOURNAL OF SOFTWARE*, vol. VOL. 8, NO. 10, OCTOBER 2013, pp. 2425–2431, 2013. [cited at p. 21]

[37] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. [cited at p. 25]

[38] B. D. Sethanandha, B. Massey, and W. Jones, "Managing open source contributions for software project sustainability," *PICMET 2010 TECHNOLOGY MANAGEMENT FOR GLOBAL ECONOMIC GROWTH*, pp. 1–9, 2010. [cited at p. 27]

[39] S. Fenton, *Pro TypeScript: Application-Scale JavaScript Development*. Apress, 2014. [cited at p. 37, 60, 63]

[40] G. Bracha, *The Dart Programming Language*. Addison-Wesley Professional, 2015. [cited at p. 37, 60]

[41] E. Andreasen and C. S. J. P. A. J. M. M. A. M. Asger Feldthaus, Simon Holm Jensen, "Improving tools for javascript programmers," *International Workshop on Scripts to Programs*, 2012. [cited at p. 44]

[42] T. Dresher, *Reactive Extensions in Action with examples in C#*. Manning, 2016. [cited at p. 46]

[43] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. [cited at p. 47]

[44] R. J. J. V. Erich Gamma, Richard Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. [cited at p. 53]

[45] J. S. Arun Ranganathan, "File api." `https://www.w3.org/TR/2015/WD-FileAPI-20150421/`, April 2015. [cited at p. 54]

[46] K.-J. Stol and M. A. Babar, "Challenges in using open source software in product development: A review of the literature," *Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pp. 17–22, 2010. [cited at p. 56]

[47] M. Levesque, "Fundamental issues with open source software developement," 2004. [cited at p. 56]

[48] W. D. M. Kennedy Kambona, Elisa Gonzalez Boix, "An evaluation of reactive programming and promises for structuring collaborative web applications," *DYLA '13 Proceedings of the 7th Workshop on Dynamic Languages and Applications*, 2013. [cited at p. 59]

[49] S. Kleinschmager, S. Hanenberg, R. Robbes, *f*ric Tanter, and A. Stefik, "Do static type systems improve the maintainability of software systems? an empirical study," *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pp. 153–162, 2012. [cited at p. 60]

[50] S. K. K. F. Danny Yoo, Emmanuel Schanzer, "Wescheme: The browser is your programming environment," *Conference on Innovation and Technology in Computer Science Education*, 2011. [cited at p. 63]

[51] R. C. M. Max Goldman, Greg Little, "Real-time collaborative coding in a web ide," *UIST '11 Proceedings of the 24th annual ACM symposium on User interface software and technology*, 2011. [cited at p. 63]

[52] R. C. M. Max Goldman, Greg Little, "Collabode: Collaborative coding in the browser," *International Conference on Software Engineering*, 2011. [cited at p. 63]

[53] S. S. T. O. T. S. Shin-ya Katayama, Takushi Goda, "A fast synchronization mechanism for collaborative web applications based on html5," *14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2013. [cited at p. 68]

[54] M. Fowler, "Language workbenches: The killer-app for domain specific languages?," 2005. [cited at p. 105]

[55] V. P. M. Voelter, "Language modularity with the mps language workbench," *2012 34th International Conference on Software Engineering (ICSE)*, 2012. [cited at p. 105]

[56] R. N. Swamy and D. G.Mahadevan, "Event driven architecture using html5 web sockets for wireless sensor networks," *Planetary Scientific Research Center*, 2011. [cited at p. 106]

Appendices

Appendix A

Pilot Study Images

This appendix presents images and descriptions of the web IDE developed as part of the pilot study.

A.1. Google Sign In

Figure A.1, shows the first page the user meets when accessing the web IDE, it only contains a button for signing into the web IDE. By clicking the button the user will be signed in with the user's Google account. If a user has more than one google account, a choice of all accounts will be presented in a list. After signing in, the user is redirected to the web IDE, presented in the following sections.

Figure A.1: The sign in page.

A.2. File Explorer

Figure A.2 shows the file explorer which contains the projects. Each project can contain directories and files. By right-clicking a project or directory a context menu is shown which allows creating new child files and directories. In addition, the context menu for projects, directories, and files allows renaming and deleting. Finally, by dragging and dropping a file or directory to a directory or project will move the dragged item to where it is dropped.

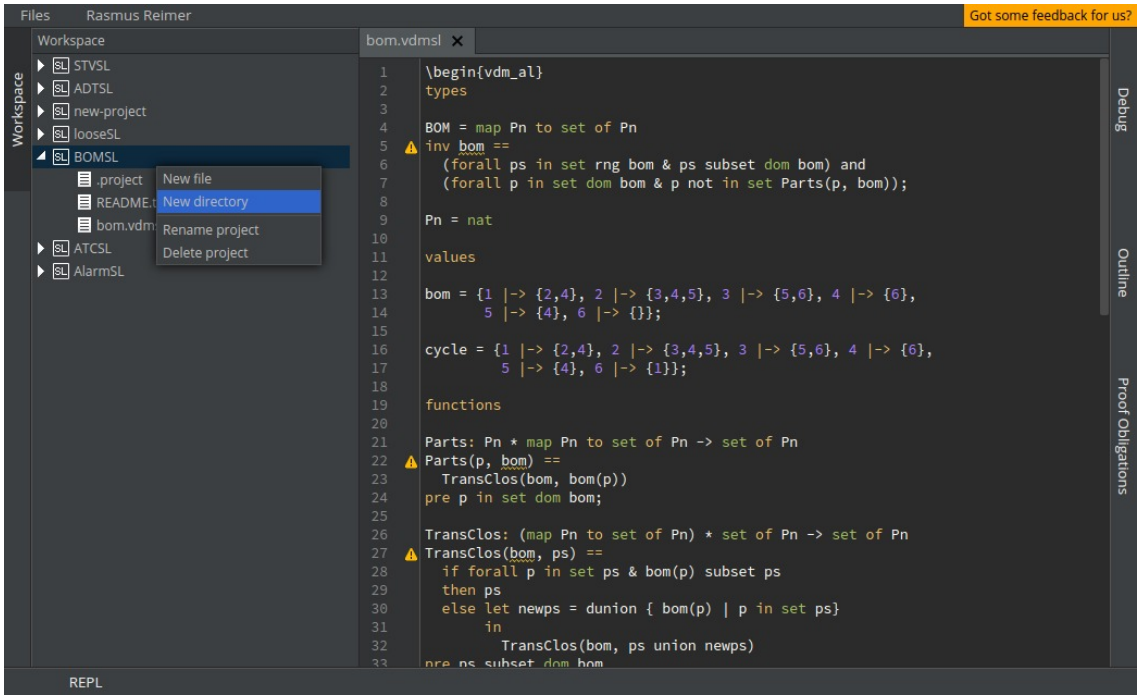


Figure A.2: The file explorer.

A.3. Import Example Projects

Figure A.3 shows the list of example VDM-SL projects. By selecting a project, more information will be presented on the right. After selecting a project, the import button can be clicked to import the project into the workspace.

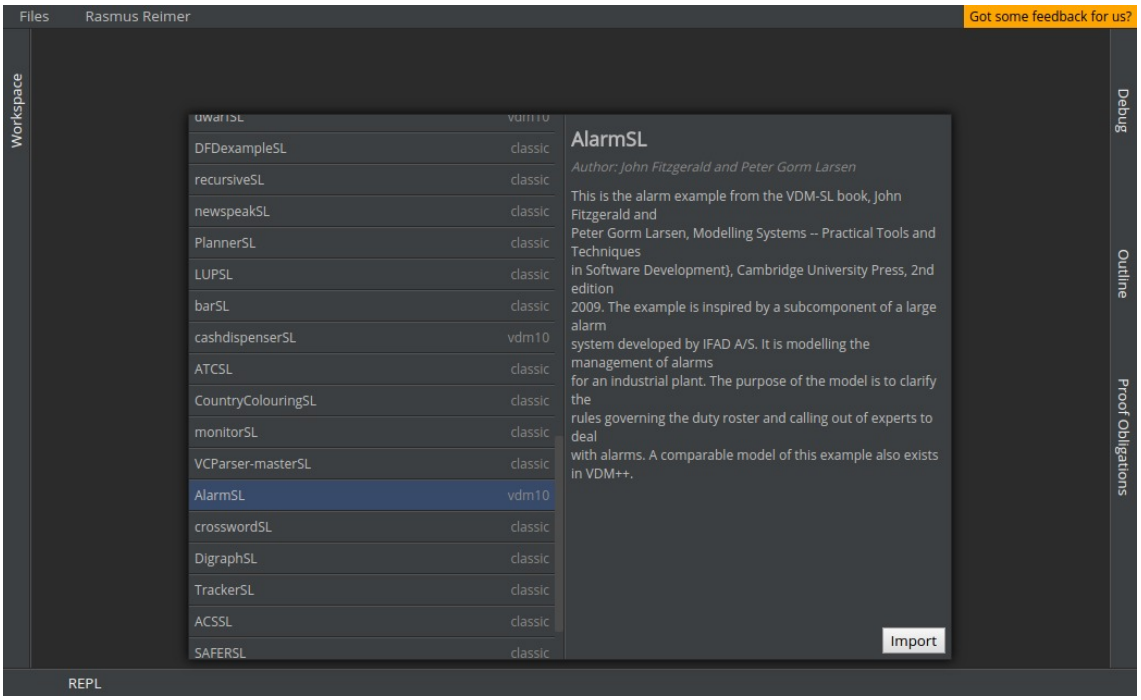


Figure A.3: The view for importing example projects.

A.4. Outline

Figure A.4 shows the panel with a list of outline items based on the currently opened file. Hovering the mouse over an item will highlight the corresponding text in the editor. Clicking an item will scroll the editor to the corresponding line.

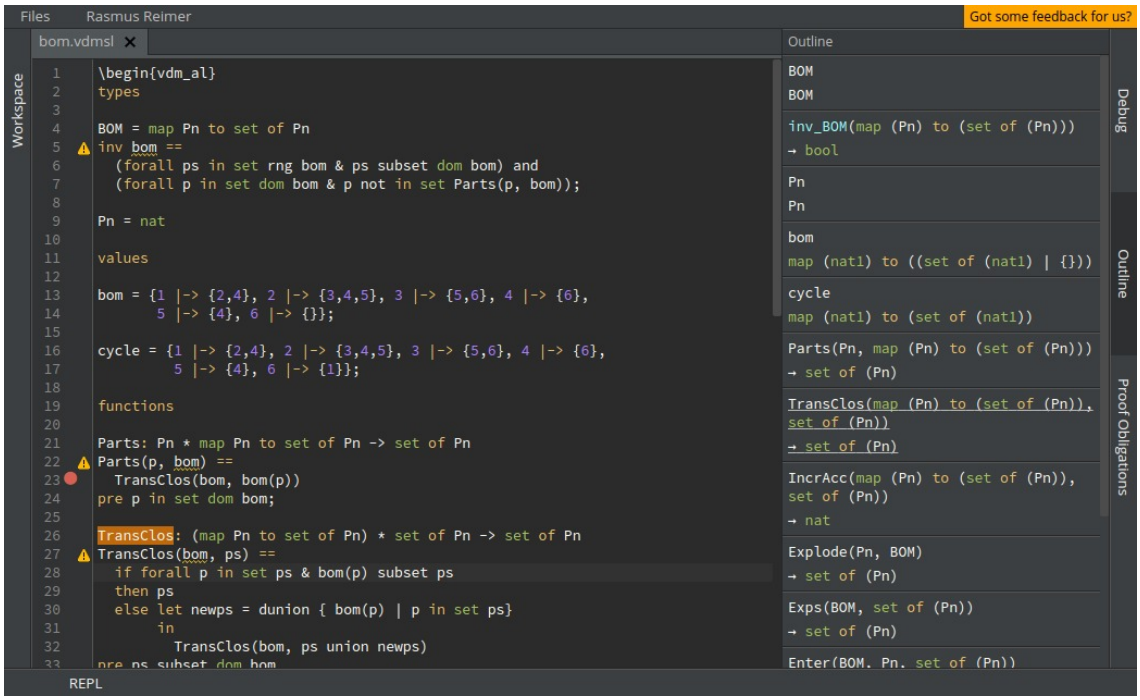


Figure A.4: The outline panel.

A.5. Proof Obligations

Figure A.5 shows the panel with a list of proof obligations based on the currently opened file. Hovering the mouse over a proof obligation will highlight the corresponding text in the editor. Clicking a proof obligation will scroll the editor to the corresponding line.

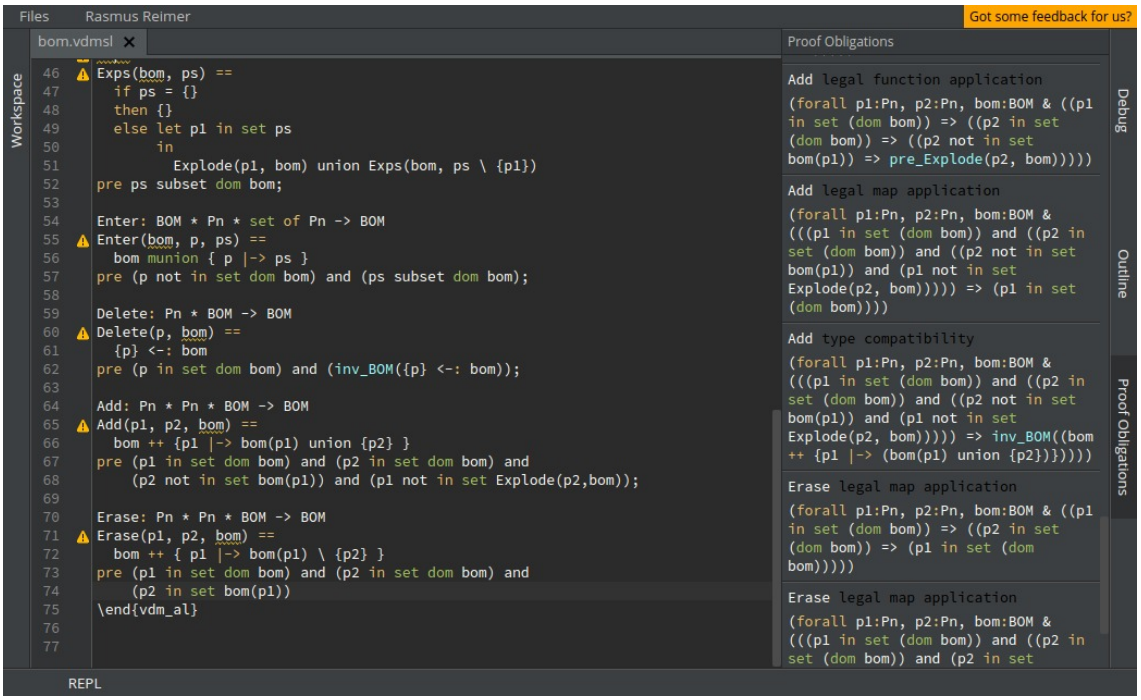


Figure A.5: The proof obligation panel.

A.6. Debug

Figure A.6 shows the panel that allows debugging the project of the currently opened file. The buttons in the top of the panel allow interacting with the debugger, such as step into, step over, and continue. In addition, stack frames are shown both in the editor and in the debug view. The stack frames can be selected to see a list of variables relating to the selected stack frame. Finally, the panel also includes a list of breakpoints. Clicking a breakpoint will open the file and scroll to the line where the breakpoint is set.

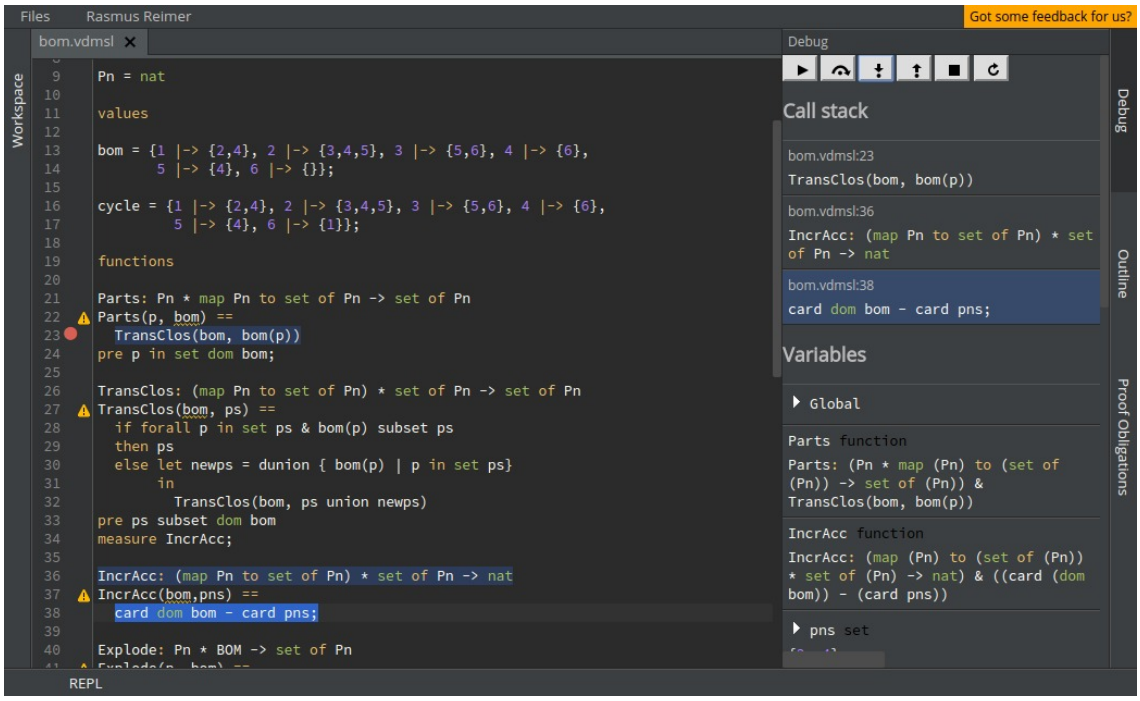


Figure A.6: The debug panel.

A.7. Read-Eval-Print-Loop

Figure A.7 shows the Read-Eval-Print-Loop (REPL) panel for evaluating VDM-SL expressions. When a file is open, the definitions in the open file is also available in the REPL as shown in Figure A.7. Using key-up and key-down allows scrolling in the history of evaluated expressions. In addition, writing $$ in an expression will be replaced with the latest result and $n will be replaced with the n'th latest result. Finally, &help can be evaluated to display a list of all available commands.

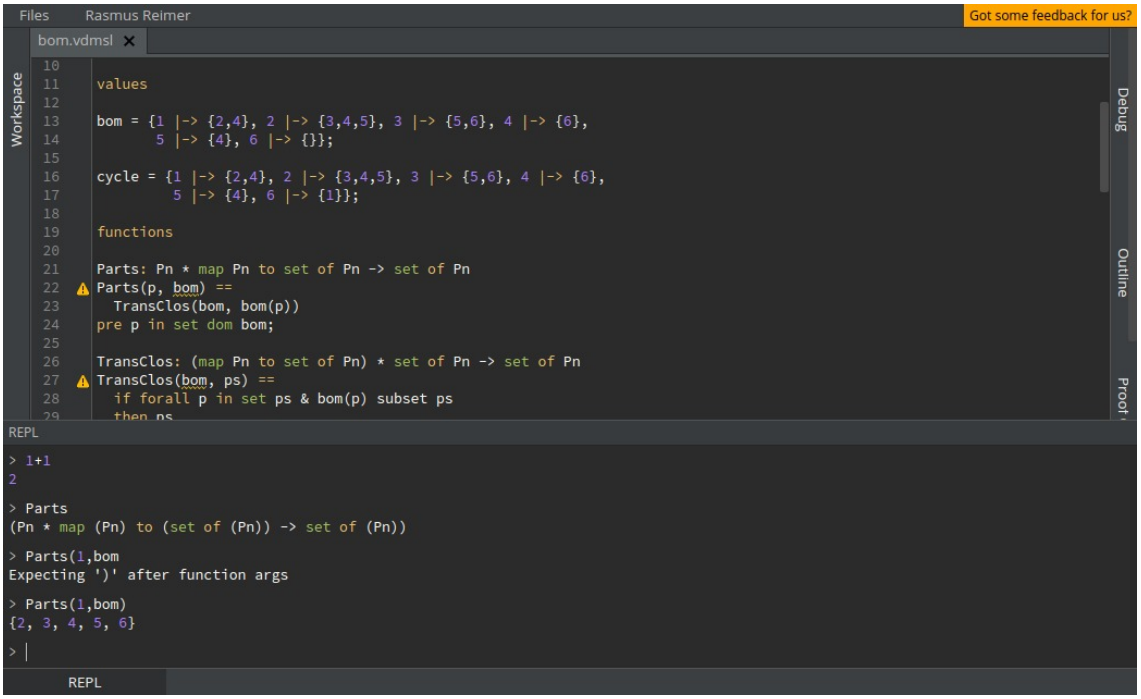


Figure A.7: The REPL panel.

Appendix B

Evaluations

This appendix presents the evaluation results of software in table-form.

B.1. Web IDEs

B.1.1 Development Environment

Table B.1: Evaluation results of development environment criteria.

Criterion	Web IDE			
	Che	Cloud9	Codebox	Orion
Code sharing	+	+	-	-
Collaboration	-	++	-	-
File explorer	+	++	++	++
Terminal	-	+	+	+
Theming	+	++	++	+
Version control	++	-	++	+++

B.1.2 Coding Assistance

Table B.2: Evaluation results of coding assistance criteria.

Criterion	Web IDE			
	Che	Cloud9	Codebox	Orion
Code formatting	+	+	-	-
Code generation	-	-	-	-
Dependency management	++	-	-	-
Outline	+	+	+	+
Pretty print to LaTeX	-	-	-	-
Proof obligation generator	-	-	-	-
Refactoring	+	-	-	-
Reference resolving	+	+	-	+

B.1.3 Runtime Evaluation

Table B.3: Evaluation results of runtime evaluation criteria.

Criterion	Web IDE			
	Che	Cloud9	Codebox	Orion
Code coverage	-	-	-	-
Debugger	+	+	+	-
External executable code	-	-	-	-
Interpreter	+	+	-	-
Read-Evaluate-Print-Loop	-	+	-	-
Testing	-	-	-	-

B.1.4 Quality

Table B.4: Evaluation results of quality criteria.

Criterion	Web IDE			
	Che	Cloud9	Codebox	Orion
Documentation	++	++	+	+++
Extensibility	+	+	+	+
Maturity	++	-	-	++

B.2. Frameworks

Table B.5: Evaluation results of evaluating client-side frameworks with quality criteria.

Criterion	Client-side frameworks		
	Angular2	Ember	React + Redux
Documentation	+++	+++	+++
Extensibility	+	+	+
Maturity	+	++	+

Table B.6: Evaluation results of evaluating server-side frameworks with quality criteria.

Criterion	Server-side frameworks	
	Play	Spring FW
Documentation	+++	+++
Extensibility	+	+
Maturity	++	++

B.3. Editors

Table B.7: Evaluation results of evaluating editors with quality criteria.

Criterion	Editor	
	Ace	CodeMirror
Documentation	+++	+++
Extensibility	+	+
Maturity	++	++

B.4. Pilot study

B.4.1 Development Environment

Table B.8: Evaluation results of development environment criteria.

Criterion	Web IDE		
	Pilot study	Che	Cloud9
Code sharing	-	+	+
Collaboration	-	-	++
File explorer	+	+	++
Terminal	-	-	+
Theming	-	+	++
Version control	-	++	-

B.4.2 Coding Assistance

Table B.9: Evaluation results of coding assistance criteria.

Criterion	Web IDE		
	Pilot study	Che	Cloud9
Code formatting	-	+	+
Code generation	-	-	-
Dependency management	-	++	-
Outline	+	+	+
Pretty print to LaTeX	-	-	-
Proof obligation generator	+	-	-
Refactoring	-	+	-
Reference resolving	-	+	+

B.4.3 Runtime Evaluation

Table B.10: Evaluation results of runtime evaluation criteria.

Criterion	Web IDE		
	Pilot study	Che	Cloud9
Code coverage	-	-	-
Debugger	+	+	+
External executable code	-	-	-
Interpreter	+	+	+
Read-Evaluate-Print-Loop	+	-	+
Testing	-	-	-

B.4.4 Quality

Table B.11: Evaluation results of quality criteria.

Criterion	Web IDE		
	Pilot study	Che	Cloud9
Documentation	+++	++	++
Extensibility	+	+	+
Maturity	+	++	-

B.5. Future Work

B.5.1 Editor

Table B.12: Evaluation results of editor criteria.

Criterion	IDE		
	Eclipse Overture Tool	Pilot study	VDMPad
Annotate scrollbar	+	-	-
Auto indention	+	+	+
Close brackets	-	+	-
Code completion	+	+	-
Code folding	-	-	-
Code templates	+	-	-
Context menu	+	-	-
Edit history	+	+	+
Highlight matching brackets	+	+	+
Indent guides	-	-	-
Line wrapping	-	+	-
Linting	+	+	-
Multiple open files	+	+	-
Multiplex modes	+	+	+
Programmable gutters	+	+	-
Search	+	-	-
Search and replace	+	-	-
Syntax highlighting	+	+	-

B.5.2 Development Environment

Table B.13: Evaluation results of development environment criteria.

Criterion	IDE		
	Eclipse Overture Tool	Pilot study	VDMPad
Code sharing	-	-	-
Collaboration	-	-	-
File explorer	++	+	-
Terminal	-	-	-
Theming	++	-	-
Version control	+++	-	-

B.5.3 Coding Assistance

Table B.14: Evaluation results of coding assistance criteria.

Criterion	IDE		
	Eclipse Overture Tool	Pilot study	VDMPad
Code formatting	-	-	+
Code generation	+	-	-
Dependency management	-	-	-
Outline	+	+	-
Pretty print to LaTeX	+	-	-
Proof obligation generator	+	+	-
Refactoring	-	-	-
Reference resolving	+	-	-

B.5.4 Runtime Evaluation

Table B.15: Evaluation results of runtime evaluation criteria.

Criterion	IDE		
	Eclipse Overture Tool	Pilot study	VDMPad
Code coverage	+	-	-
Debugger	+	+	-
External executable code	+	-	-
Interpreter	+	+	+
Read-Evaluate-Print-Loop	-	+	+
Testing	++	-	+

B.5.5 Quality

Table B.16: Evaluation results of quality criteria.

Criterion	IDE		
	Eclipse Overture Tool	Pilot study	VDMPad
Documentation	+++	+++	++
Extensibility	+	+	+
Maturity	+++	+	++

Appendix C

Pilot Study Design and Implementation

C.1. Client-side Design and Implementation

In this section, the selected framework and editor is introduced followed by a description of the web IDEs UI architecture. The code for the client-side application can be found at `https://github.com/overturetool/web-ide-frontend`.

C.1.1 Angular2

An Angular2 application consists of a tree of components and an acyclic graph of services. A component is responsible for a part of the UI and can contain other components, which gives the tree structure. A component acts as a model for its view and can update the view through data binding and react on user interactions through events. This relationship between component and view is also seen between the controller and view in the Model-View-Controller (MVC) pattern, described in chapter 2.

Each component specifies an API, which other components can use to communicate with it. A component can communicate with its parent component by emitting events, and its children with data bindings. To communicate with any other component, a component must communicate through services, which can be accessed using dependency injection.

Services in Angular2 are classes registered in the framework as an injectable class. Components and services can specify services which they depend on, and Angular2 will inject instances of the services.

Figure C.1 illustrates a simple example of a component tree and a service graph.

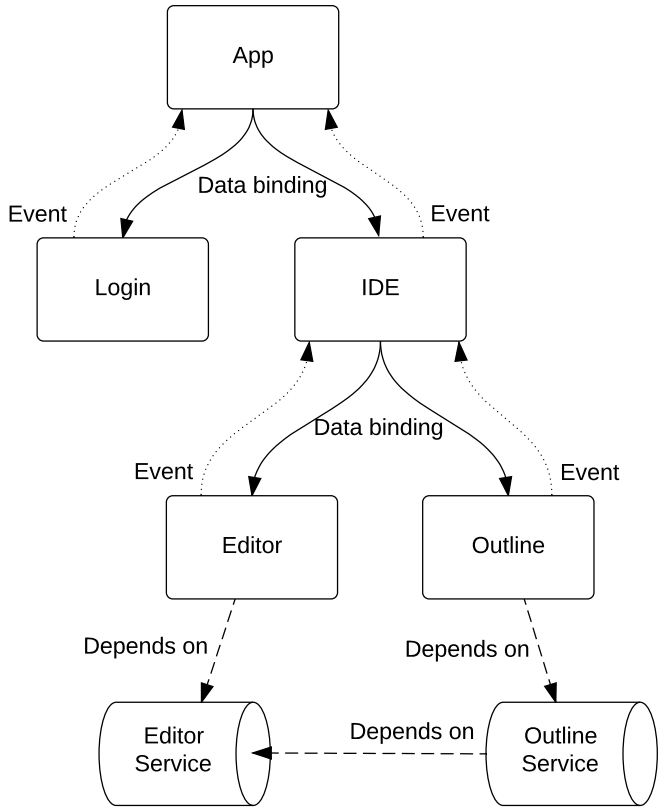


Figure C.1: Illustration of an Angular2 component tree and service graph. The rectangular boxes represent components and the cylinders represent services. Solid arrows represent data bindings, dotted arrows represent events, and dashed arrows represents dependencies.

C.1.2 CodeMirror

The CodeMirror editor is used in the client-side application. Most of the functionality of CodeMirror is provided by a subset of the approximately 40 official extensions[1].

The following extensions are used in the web IDE:

Lint: The lint extension is used to show warnings and errors in the gutter and to highlight the erroneous section of the file.

Show hint: The show hint extension is used to add a box for code completion. This box is populated by calling the server-side application.

Match brackets: The match brackets extension will highlight matching brackets.

Close brackets: The close brackets extension will insert a closing bracket after the cursor when inserting an opening bracket.

Active line: The active line extension highlights the line where the cursor is located.

[1] See `http://codemirror.net/doc/manual.html#addons`, accessed 23rd of March 2016.

Run mode: The run mode can run syntax highlighting rules on text outside the editor. This is used to syntax highlight code snippets in the outline panel, debug panel, etc.

Syntax highlighting, auto indentation, and commenting out lines for a specific language, is supported by implementing a language mode. A language mode is a JavaScript object which implements a language mode interface[2]. There exist official language modes for over 120 programming and markup languages[3], but not for VDM-SL.

In an IDE, multiple files can be open at the same time with individual undo histories and language modes. In CodeMirror, this is handled by using a document concept. Document objects can be constructed through CodeMirror's API. Documents keep track of their own content, undo history, and language mode. Documents can be swapped in and out of the editor component as needed, to allow for swapping between multiple open files.

C.1.3 Architecture

This section describes the architecture of the client-side application. The architecture will be illustrated through three diagrams, showing parts of the architecture. First, the component tree is described, then the service graph, and lastly, how components depend on the services. Figure C.2 illustrates the web IDEs component tree.

Below is a brief description of each component:

Login: The Login component uses Google sign-in[4] to authenticate the user.

IDE: The IDE component handles the structure of the UI.

App: The App component handles navigating between the login and web IDE page.

Menu: The Menu component displays the menu bar at the top of the web IDE.

Guide: The Guide component displays an interactive guide to the different panels of the web IDE.

QuickBar: The QuickBar registers a keyboard shortcut to open a search field for quickly opening files.

Examples: The Examples component displays a tool for importing example VDM-SL projects into the workspace.

EditorTabs: The EditorTabs component displays tabs for open files allowing for switching between the files.

Editor: The Editor component wraps the CodeMirror API to display a CodeMirror instance. It also handles displaying breakpoints, warnings, and errors in the gutter. It also handles highlighting sections of the document when debugging or using the outline panel.

PanelMenu: The PanelMenu component displays a menu on the sides and bottom of the web IDE to open and close panels.

[2] See `http://codemirror.net/doc/manual.html#modeapi`, accessed 23rd of March 2016.
[3] See `http://codemirror.net/mode/`, accessed 23rd of March 2016.
[4] See `https://developers.google.com/identity/sign-in/web/`, accessed 25th of March 2016.

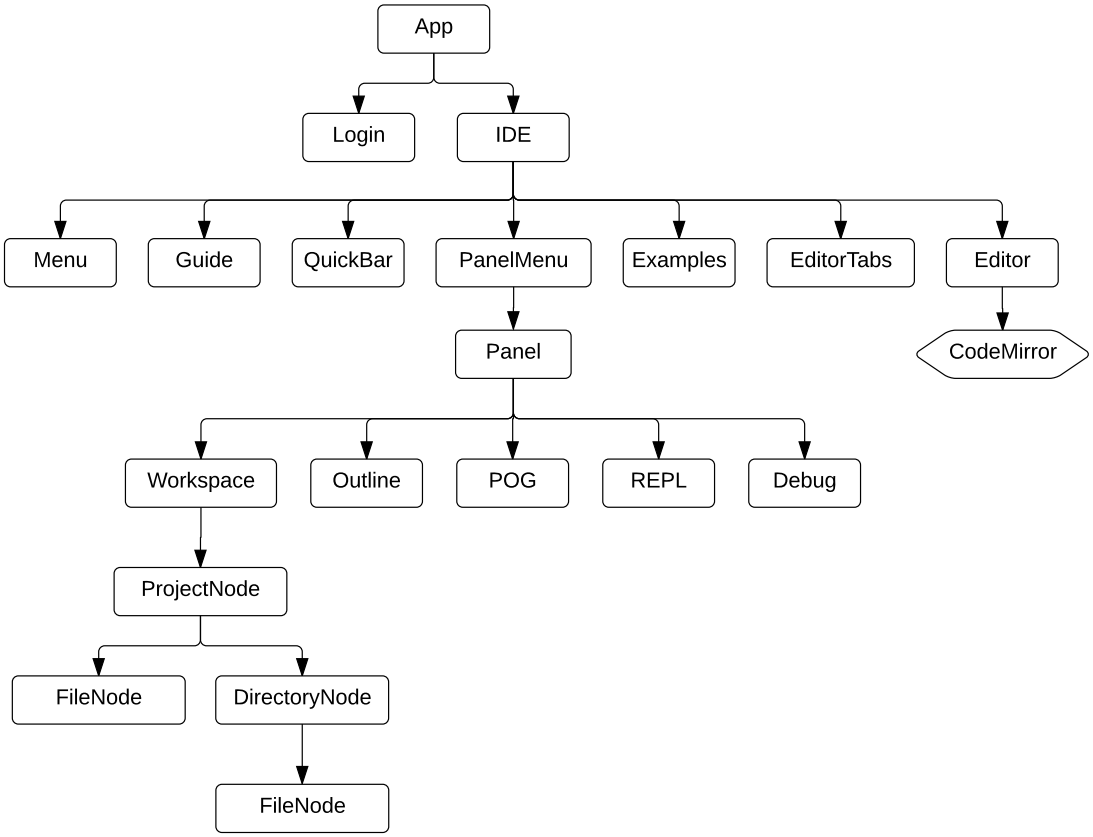


Figure C.2: Illustration of the client-side applications component tree. The rectangular boxes represent application-specific components and hexagonal boxes represent external components. Solid arrows represent component tree connections.

Panel: The Panel component is used to display a resizable panel. Other components, such as Outline and Debug, can be inserted in a Panel.

Outline: The Outline component displays a list of the types, values, and functions defined in the open document.

POG: The POG component displays a list of proof obligations for the open document.

REPL: The REPL component is used for evaluating VDM-SL expressions against the model defined in the currently opened file.

Debug: The Debug component is used for debugging the open project. When the debugger reaches a breakpoint, the Debug component shows a list of stack frames and a tree of values in the selected frame. The Debug component interacts with the Editor component by notifying about where the debugger is stopped and the currently selected stack frames.

Workspace: The Workspace component displays a file explorer consisting of ProjectNode, DirectoryNode, and FileNode components.

ProjectNode: The ProjectNode component displays a project. It handles renaming, deleting, and moving FileNode and DirectoryNodes into it.

DirectoryNode: The DirectoryNode component displays a directory. It handles renaming, deleting, and moving FileNode and DirectoryNodes into it.

FileNode: The FileNode component displays a file. It handles renaming, deleting, and opening a file.

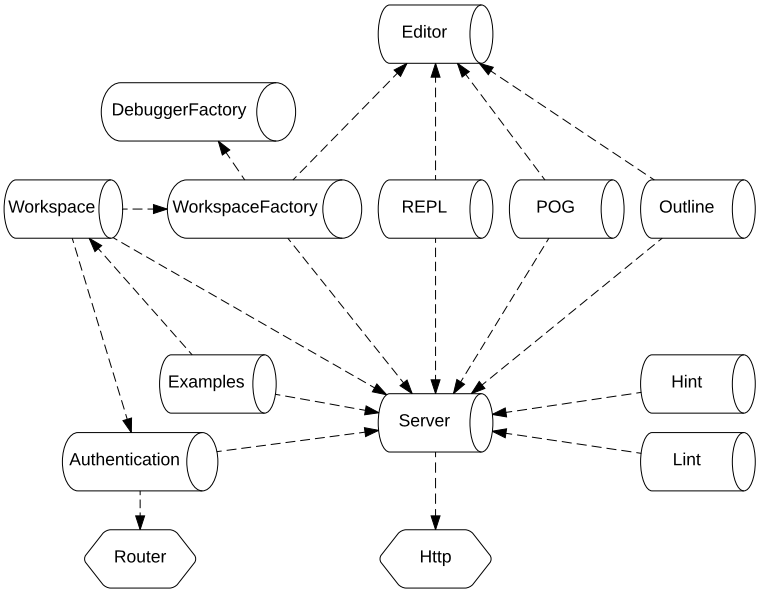


Figure C.3: Illustration of the service graph. The cylinders represent application-specific services and hexagonal boxes represent built-in Angular2 services. Dashed arrows represent dependencies. The arrows start at a service which depends on the service the arrow points at.

Figure C.3 illustrates the service graph of the client-side application. Figure C.4 illustrates how the components depend on the services. Below is a description of each service:

Server: The Server service handles communication with the server-side application. This includes both HTTP and WebSocket communication. Most other services depend on the Server service.

Editor: The Editor service allows the other services to interact with the Editor component. It also handles open files.

WorkspaceFactory: The WorkspaceFactory is used to create new files, directories, and projects. It depends on the DebuggerFactory since every project gets its own debugger when created.

DebuggerFactory: The DebuggerFactory is used to create new debugger instances. A debugger instance manages the state of debugging sessions.

Workspace: The Workspace service handles synchronizing the workspace with the server-side application.

REPL: The REPL service sends expressions to the server-side application and parses the response.

POG: The POG service listens for change events on the Editor service, and then requests the server-side application for an updated list of proof obligations.

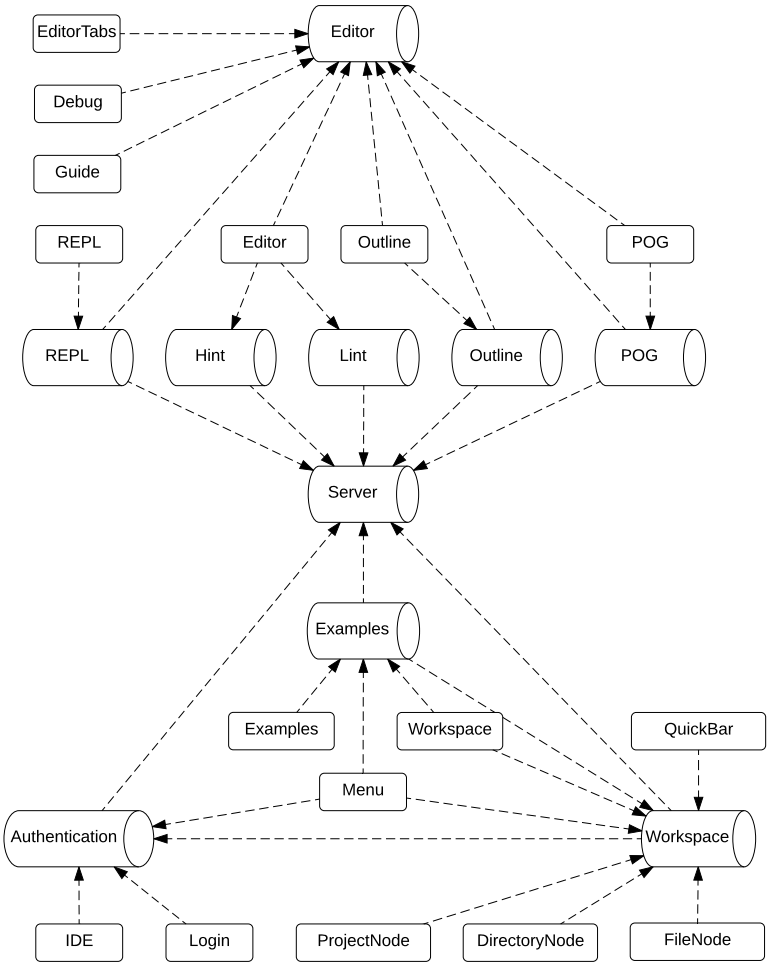


Figure C.4: Illustration of how the components depend on the services. The cylinders represent services and rectangles represent components. Dashed arrows represent dependencies. The arrows start at a component which depends on the service the arrow points at.

Outline: The Outline service listens for change events on the Editor service, and then requests the server-side application for an updated list of types, values, and functions.

Examples: The Examples service handles importing example projects.

Authentication: The Authentication service handles signing in and out.

Hint: The Hint service is used by CodeMirror to request the server-side application for code completion.

Lint: The Lint service is used by CodeMirror to request the server-side application for warning and error annotations. The POG service listens for change events on the Editor service, and then requests the server-side application for an updated list of proof obligations.

C.2. Server-side Design and Implementation

The code for the server-side application can be found at `https://github.com/overturetool/web-ide-backend`.

C.2.1 Server-side Requirements

The editor, development environment, coding assistance, and runtime evaluation criteria defined in chapter 3, represents requirements for the web IDE. In addition, certain requirements are given, which are inherent in web applications and the need to manage files. These requirements influence the architecture of the server-side application. These additional requirements are described below.

File system: The server-side application must expose a file system for managing files and directories, which includes basic file operations such as read, write, create, delete, move, and rename.

Workspace: The server-side application must provide isolated workspaces for each user, which will contain the user's project files.

Authentication and authorization: The server-side application must include authentication and authorization mechanisms.

Interface: The server-side application must define a uniform interface for communication with client-side applications.

The requirements are used to create a model of the server-side application. This model can be seen on Figure C.5, which shows the organization of the server-side application, in terms of the requirements.

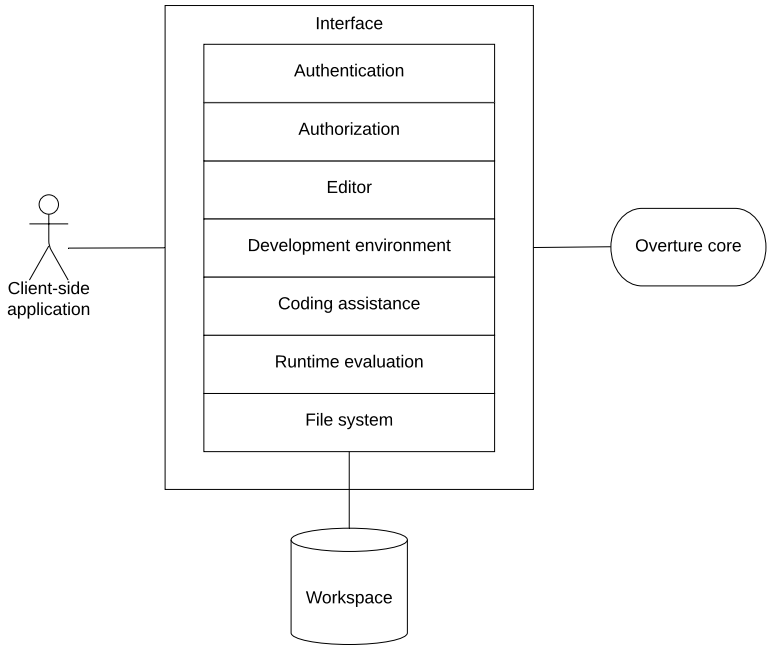


Figure C.5: Server-side model. All the features, except authentication, are accessed through the authorization feature.

The interface component defines how a client-side application can communicate with the server-side application. Hence, the interface component provided access to the feature implementations. The feature implementations include processing VDM-SL models, file system operations, authentication, and authorization. For processing VDM-SL models, the feature implementations need to utilize the Overture core to perform the tasks requested by a client-side application. Files are stored in a workspace. The concrete storage medium can be the server on which the server-side application is running or an external cloud service, such as Amazon S3[5].

C.2.2 Server Architecture

Using Cloud9 and Che[6] as inspiration, the server-side application is implemented as a REST API described in section 2.6.2. Internally the server-side application uses the MVC architecture described in section 2.2.

Since the server-side application is an API, the view is not served as a HTML document. Instead, the response is served as text or JSON, or a WebSocket connection. Requests are routed to methods in the controllers. From the methods, the requests are processed by calling domain logic implemented in the models. After the request have been processed, a response is returned to the client application. The MVC pattern as used in this pilot study is shown in Figure C.6, where the view is replaced with a resource.

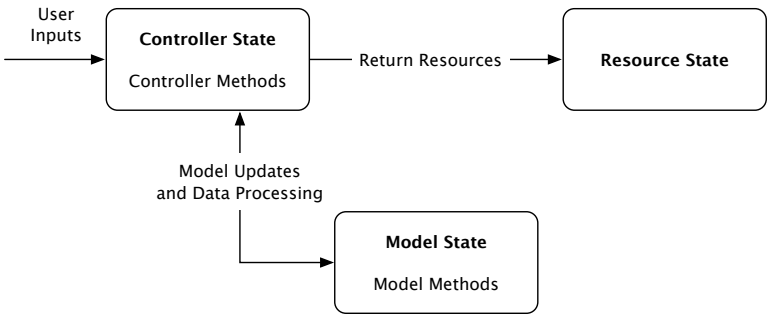


Figure C.6: MVC pattern as used in this pilot study.

The interface component in Figure C.5 is implemented as the controllers in the MVC pattern. In this pilot study, the following controllers are defined, which receives user inputs, sanitizes these inputs, and use the feature implementations to perform the tasks described below.

Auth: Used to verify the user's identity.

Codecompletion: Generates code completion based on a VDM-SL model.

Debug: Creates a WebSocket and executes a debugger instance, enabling the client-side application to debug VDM-SL models.

Evaluate: Receives VDM-SL expressions and evaluates these in the context of a VDM-SL model, and returns the result of the evaluation.

[5]See `https://aws.amazon.com/s3`, accessed 29th of March 2016.
[6]See `https://eclipse-che.readme.io/docs/`, accessed 22nd of March 2016.

Samples: Returns a list of example VDM-SL projects, which can be imported into a user's workspace.

Lint: Provides warnings and errors generated when parsing a VDM-SL model.

Outline: Provides an overview of declarations and imports in a VDM-SL model.

Proof Obligation Generator (POG): Provides proof obligations generated for a VDM-SL model.

Virtual File System (VFS): Used by the client-side application to perform file system operations, such as read, write, remove, create, move, rename, etc.

The features auth, samples, and vfs are model agnostic and the implementation of these features do not concern the Overture core. The remaining features are either supported by the Overture core or the Eclipse-based Overture Tool.

With the selection of the overall server-side architecture, the model seen in Figure C.5 can be mapped onto the MVC pattern, showing how the logical components are realized. The resulting model is seen in Figure C.7.

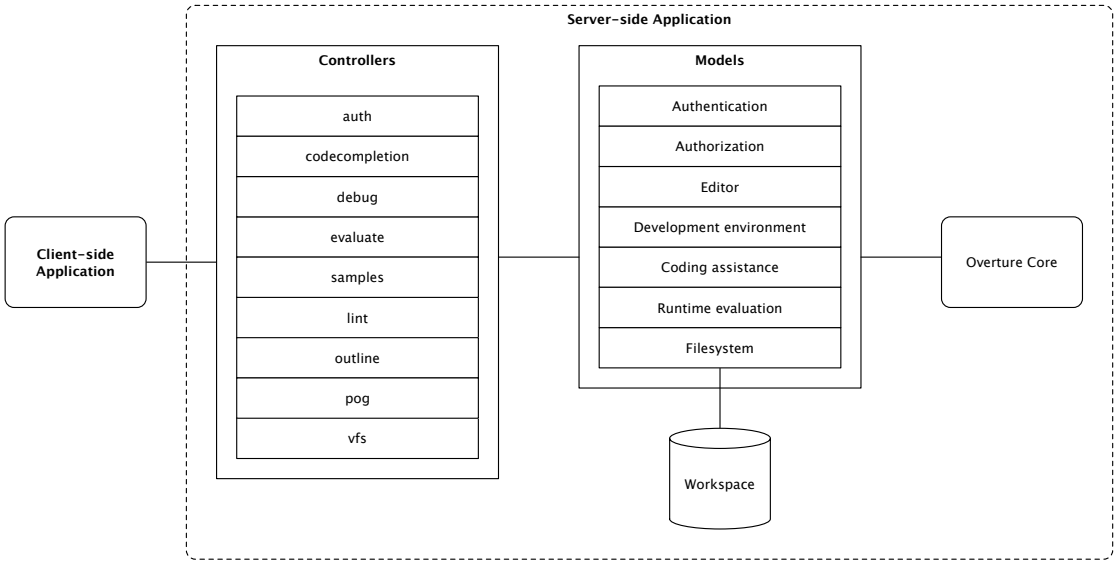


Figure C.7: Model showing how requirements are mapped onto the MVC pattern.

C.2.3 Server-side Frameworks

The server-side application is based on a framework to avoid having to implement common web application tasks, such as handling HTTP communication. The implementation of the features is kept independent of the selected framework. Decoupling implementations will make it easier to switch frameworks, if necessary. This makes the choice of framework less important in the pilot study.

In section 4.3 two frameworks supporting Java were presented. Both of these frameworks achieve good ratings in the quality category, and both are MVC-based frameworks. Selecting between these two frameworks can therefore be difficult, and additional features of the frameworks need to

be considered.

For the pilot study, Play was selected due to productivity reasons, given the limited thesis time frame. Play requires minimal configuration to get a web application up and running. Additionally, Play includes live code reload, and useful error messaging features.

C.2.4 Feature Implementations

The implementations of the features in the model layer of the MVC pattern are described in the following sections. Figure C.8 shows a diagram outlining the structure of the server-side application.

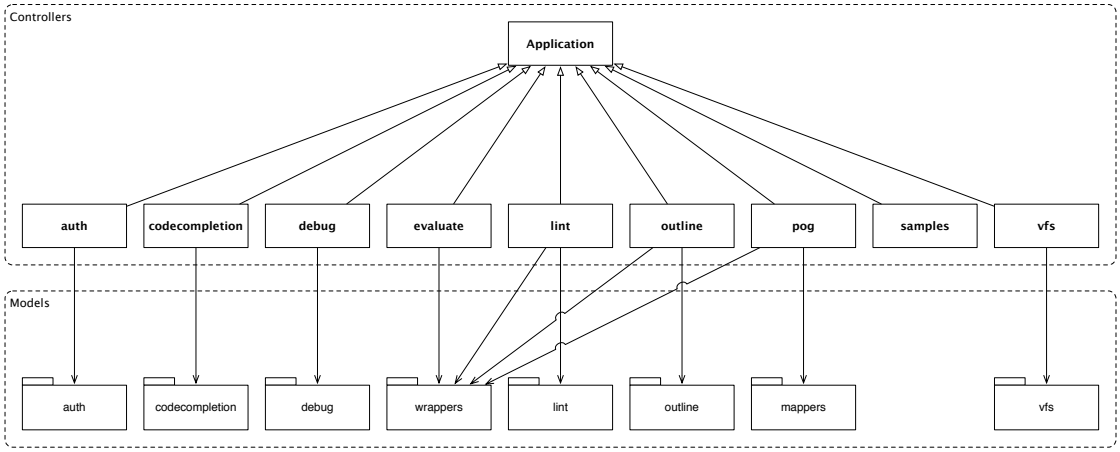


Figure C.8: Diagram showing the overall server-side structure.

The controllers inherit from the Application controller. The Application controller is annotated with a SecuredAction-class. The SecuredAction-class implements the authorization feature. This feature denies access if no valid authorization information is provided by the client-side application. The authorization mechanism is therefore enforced in all controllers, by having the controllers inherit from the Application controller.

The packages that implement the features defined in chapter 3, are placed in the model layer of the MVC pattern. The packages consist of one or more Java classes, which will not be described in detail. However, for packages integrating with the Overture core, a description of the overall implementation will be given in the following sections.

C.2.4.1 Model Wrapper

VDM-SL models are parsed and type checked using the VDMSL-class in the Overture core. When a VDM-SL model has been successfully parsed and type checked, the ModuleInterpreter-class is instantiated and initialized. The interpreter provides an access point for the AST, proof obligations, and evaluating VDM-SL expressions in the context of the AST. The VDMSL-class from the Overture core, which is responsible for instantiating and initializing the ModuleInterpreter-class, is wrapped in the ModelWrapper-class. Therefore, the ModelWrapper-class provides a single access point for VDM-SL related operations.

The parse method in the VDMSL-class takes a list of Java File objects. A VDM-SL project can

consist of one or more files and directories. However, not all files contain VDM-SL content. Since the VDMSL-class only accepts files containing VDM-SL content, preprocessing the project directory is necessary, for filtering out none VDM-SL files. This preprocessing of the projects is also handled by the ModelWrapper-class.

C.2.4.2 Evaluate

The REPL feature is defined in section 3.5. The server-side implementation of this feature receives commands as text from the client-side application. The commands are then processed and responses are returned to the client-side application. The supported operations by the REPL implementation are listed below.

Evaluate expressions: Evaluate VDM-SL expressions in the context of a VDM-SL model.

Dynamically create definitions: Create method and variable definitions and add these to the AST.

List modules: List all VDM-SL modules defined in a project.

Display default module: Display the VDM-SL module selected as the default module in the project.

Change default module: Change the default VDM-SL module to another module defined in the project.

Display REPL help information: Display the REPL commands.

Evaluating a VDM-SL expression, listing modules, displaying the default module, and changing the default module are trivially achieved using the module interpreter. No significant additional processing is necessary.

Creating new definitions and adding these to the AST is more complex. The text input to the REPL is a string consisting of a variable name, an assignment operator, and a VDM-SL expression. The input is then split into two strings, one containing the variable name, and the other containing the expression. The expression is parsed to determine the kind of expression and its type and then evaluated to generate the expression value. A value definition is then instantiated with the variable name and the expression. Lastly, the value definition is added to the AST.
The implementation of creating new definitions in the AST is inspired by an implementation of a similar feature for VDM++ provided in the Overture Command-line Tool.

C.2.4.3 Proof Obligation Generator

The proof obligations are provided by the module interpreter through the ModelWrapper-class. The only additional computation needed for this feature is to map the list of proof obligation objects to a different format. A PogMapper-class is used to extract the information from the list of proof obligation objects, and add the information to JSON resources. The JSON resources are then added to an array, which is returned to the client-side application.

C.2.4.4 Outline

The outline feature is not implemented by the Overture core. However, this feature is implemented in the Eclipse-based Overture Tool. Code from the Eclipse-based Overture Tool implementation is therefore reused, which extracts definitions and imports from the AST. Additionally, the extracted information from the AST needs to be mapped to JSON resources which are added to an array.

C.2.4.5 Lint

Linting VDM-SL models consist of providing warning and error information generated during parsing and type checking the VDM-SL models. Unfortunately, this information is not provided when parsing and type checking the models using the VDMSL-class wrapped in the ModelWrapper-class. The VDMSL-class only returns an enum type informing whether the parsing and type checking completed successfully, or not. However, the Overture core provides a ParserUtil-class and a TypeCheckerUtil-class, which return result-lists containing more elaborate warning and error information. Information from these result-lists is then mapped to JSON resources which are added to an array.

C.2.4.6 Code Completion

At the time of this writing, code completion is not available in the Overture core. The Eclipse-based Overture Tool contains very limited support for the feature, which is reused in the pilot study. Since the support for the code completion feature is very limited, and the implementation is fairly complex, very little time have been spent on this feature during the pilot study.

C.2.4.7 Debugger

The DBGPReaderV2-class in the Overture core is the debugging implementation for VDM-SL. Unfortunately, the DBGPReaderV2-class is not threadsafe. Concurrency issues can occur if multiple threads instantiate the class. Furthermore, when a debugging session finish, the DBGPReaderV2-class issues a System.exit call. This call terminates the process in which the DBGPReaderV2-class is running. The fact that the DBGPReaderV2-class issues System.exit calls, and is not threadsafe, requires that the class must be executed in a separate process. If the debugger is not executed in a separate process, the call to System.exit will terminate the JVM in which the server-side application is running. However, executing the debugger in a separate process is supported since the communication with the debugger can be achieved using a socket connection.

For each client-side application simultaneously initiating a debugging session, an available port needs to be allocated for the socket connection. Having allocated an available port, a socket server is instantiated and set to wait for a socket client to connect on the allocated port. Then, the DBGPReaderV2-class is executed in a separate process, which will connect to the socket server. This is the communication flow internally on the server-side application.

Due to the continuous communication flow between the debugger and a client-side application, a WebSocket is created for the client-side application to communicate with the debugger. Hence, the server-side application becomes a proxy for the communication between the client-side application and the debugger.

C.2.4.8 Virtual File System

Each user has a workspace containing their projects. The workspace is a local directory on the server, on which the server-side applications is running. Communication with the file system is implemented using a VFS library, called Commons VFS[7]. Commons VFS is a library for accessing different kinds of file systems.

C.2.4.9 Cache

A caching mechanism is used to store the module interpreter after the VDM-SL model have been parsed, type checked, and the module interpreter is initialized. This caching mechanism is implemented as a singleton. When a VDM-SL model have been parsed, type checked, and the module interpreter have been instantiated, the module interpreter is cached. A file object is cached together with the module interpreter, that references the project directory containing the VDM-SL model. The file object is used to identify the project, and to determine if any files in the project directory have been modified since the caching. If any files have been modified since the caching, the VDM-SL model will have to be parsed and type checked again. Otherwise, the cached version can be used, avoiding the potentially expensive computations.

[7] See `https://commons.apache.org/proper/commons-vfs/`, accessed 25th of March 2016.

Appendix D

Language Tooling

Language tools such as outline, code completion, and refactoring can be implemented in most languages on the server-side application. However, certain features can be implemented in JavaScript on the client-side application. This appendix discusses advantages and disadvantages of a client-side implementation and a server-side implementation.

D.1. Client-side implementation

The advantages of a client-side implementation include removing the network latency. In addition, the language tools will work even if the network connection is lost. Finally, it will reduce the workload of the server, which can be significant with many users. This will reduce the cost of hosting the web IDE, by reducing the required hardware resources on the server.

The disadvantages of a client-side implementation include the increase of required hardware resources on the local machine. Since JavaScript is slower than Java for parsing source code [9], this could become an issue for low-performance clients, such as tablets. However, Cloud9 successfully implements outline, basic linting, and reference resolving for JavaScript on the client-side application, which involves parsing source code.
If performance is an issue, client-side language tools could be run in a dedicated thread using a WebWorker. This is expected to increase performance, and avoid making the UI unresponsive during heavy computations.

Another possible disadvantage is that the language tools have to be implemented in JavaScript or compiled to JavaScript. It might not be possible to port existing languages tools developed in e.g Java, to JavaScript.
A possibility for implementing language tools in JavaScript is using a language workbench [54]. A language workbench can generate implementations of language tools from a language definition. Developing language tools requires significant effort which is expected to be reduced by using a language workbench. The language workbench called Meta Programming System (MPS) can generate both Java and JavaScript implementations of language tools [55].

D.2. Server-side implementation

The advantages of a server-side implementation include allowing reuse of existing language tools. Language tools running on the server does not have to be developed in any particular language, which makes reusing existing language tools more feasible. In addition, a server can have more computational resources available, especially in a cloud environment. This is expected to allow even low-performance clients, such as tablets, to use the web IDE. Finally, the runtime evaluation features, such as an interpreter and a debugger, might not be feasible to implement in JavaScript. These features will therefore have to run on the server.

The disadvantages of a server-side implementation include network latency and higher server load. The network latency can be reduced by using WebSockets for communication [56].

Appendix E

Implementation Estimates

This appendix provides rough estimates of the time required for implementing a set of features in the pilot study. These estimates are divided into the following intervals: hours, days, weeks, and months. Section E.1 presents estimates for implementing features, which enables the pilot study to match the features supported in the Eclipse-based Overture Tool. Section E.2 presents estimates for implementing features beyond the features supported in the Eclipse-based Overture Tool.

E.1. Matching the Eclipse-based Overture Tool

Table E.1: Implementation time estimations.

Criterion	Interval			
	Hours	Days	Weeks	Months
Annotate scrollbar	x			
Code coverage		x		
Code formatting		x		
Code generation			x	
Code templates		x		
Context menu	x			
Dependency management			x	
External executable code				x
Pretty print to LaTeX			x	
Real-time Log Viewer				x
Reference resolving		x		
Search	x			
Search and replace	x			
Testing		x		
Theming		x		
UML mapping			x	
VDM++ debugger	x			
VDM++ REPL		x		
VDM++ wrapper		x		
VDM-RT debugger	x			
VDM-RT REPL		x		
VDM-RT wrapper		x		

E.2. Beyond the Eclipse-based Overture Tool

Table E.2: Implementation time estimations.

Criterion	Interval			
	Hours	Days	Weeks	Months
Code folding		x		
Code sharing			x	
Collaboration				x
Indent guides		x		
Refactoring			x	
Terminal				x
Version control			x	