Design Report

Improving the VS Code VerCors IDE plugin

April 26, 2024

Supervisor: Pieter Bos (FMT)

Group 3

Pascal Bakker	2814277
Rick de Vries	2794101
Denis Asenov	2824442
Jaron Lendering	2829355
Bram Ottenschot	2778653
Lucas Lalande	2758415

Abstract

This report documents the development of the VSCode VerCors IDE, a system developed for the Design Project offered by the University of Twente as part of the bachelor Technical Computer Science. The goal of the project was:

“*To improve the VSCode VerCors IDE plugin by implementing standard IDE features and integrating graphical progress information.”*

The plugin is developed for the FMT group at the university. It is an extension to VSCode allowing users to use the VerCors tool inside the VSCode IDE. This report describes the whole design process, from requirements gathering to design choices, system testing and possible future additions.

Contents

1. Introduction

The TCS design project is one of the two final modules of the Bachelor of Technical Computer Science. After finally getting all the mandatory ECs of the study, the student can participate in this graduation project. It tests the students' capabilities in setting up, following, and finishing the design aspect of a development project, from the very first informal requirements meetings with a client to presenting and delivering a well-documented result.

The project should be done in groups of 6 persons. Together with this group, a fitting project should be chosen. From a list of around 50 projects, we narrowed it down to 5 possible projects. Finally, we were assigned the “Improve the VSCode VerCors IDE” project by the Formal Methods & Tools (FMT) chair of the University of Twente, with client Pieter Bos, who is a member of the VerCors team. This means that the communication between us and the VerCors team for example questions all goes through him. He is also our supervisor for the project, which means that he also supervises and grades our work.

The project was about improving the VSCode VerCors plugin which was merely a barebone structure. The plugin uses the VerCors tool which can verify concurrent programs with the help of annotated specification language code blocks. We set ourselves three main goals for this project which were (1) to better integrate the VerCors tool into VSCode; (2) to extend the IDE features in the editor and (3) to graphically display the output of the tool instead of textual output in the terminal.

We will start the report by making a domain analysis where we sum up and explain the basis of our project, for each domain we briefly explain what it is, and how it affects or is affected by the project. After establishing a steady base for the project we move on to the requirements of the project, which were gathered through meetings and proposals, neatly organised in the three main goals we set in this project. Then we show how we implemented these requirements in the design part, first a global overview of the whole system, and then a closer look at the design choices of each part. Furthermore, we talked about the meetings we had with our client, from short standup meetings to showcases for the whole VerCors team. After that, we talk about the testing aspect of the project, where we explain how we ensure the plugin behaves as it should. At the end of the report, we speculate what possible future improvements can be made to the plugin and we evaluate the module for ourselves on how well we organised the project and ourselves during the module.

2. Domain Analysis

2.1 The client

We will be developing the plugin for VerCors. The VerCors team is part of the FMT group of the University of Twente. Besides VerCors - which we will talk about in the next section - they also develop several tools on top of VerCors, namely Alpinist, Vesuv and VeyMont. Most of the VerCors developers use Linux as their main operating system. This might not seem like an important fact, but became quite important during the development. We will come back to that later. Most developers use IntelliJ as their preferred IDE, for which a VerCors plugin already has been developed. Therefore, we will be implementing the VSCode plugin for the few developers who use VSCode.

2.2 VerCors

VerCors - besides being a beautiful mountain arrangement in France - stands for Verification of Concurrent and distributed Software[1]. The tool reasons about programs in several languages using specifications. At the moment these languages are Java, C, OpenCL, OpenMP and PVL, but support for more is a goal for the future. PVL is a simple language developed by VerCors used for testing and demonstrating features. The specifications are contracts in the form of pre- and post conditions. The following code snippet gives an impression of what the specifications look like:

```
/*@
     ensures \result == \old(x)+ 1;
   */

public int incr(int x) {
  return x+1;
}
```

In this case, the developer stated in a post-condition that indicates that the return value of this method should be 1 higher than x was before the method call. Using these contracts, VerCors reasons about the correctness of programs, focussing on data-race freedom, memory safety and functional correctness.

[1] https://vercors.ewi.utwente.nl/wiki/#introduction

2.3 The current plugin

The client has indicated that someone already made a start on the plugin, from which we will develop further. The plugin already includes some features:

- Syntax highlighting for PVL
- A run button
- A way to add VerCors to path
- A couple of run options with checkboxes for the backend, quit mode, verbose mode, view progress, more error info and backend debug info.

The current version displays VerCors' output in the command line, just like the non-IDE version. Syntax highlighting in PVL exists but there is no syntax highlighting for other languages. Figure 1 showcases the supported command line options:

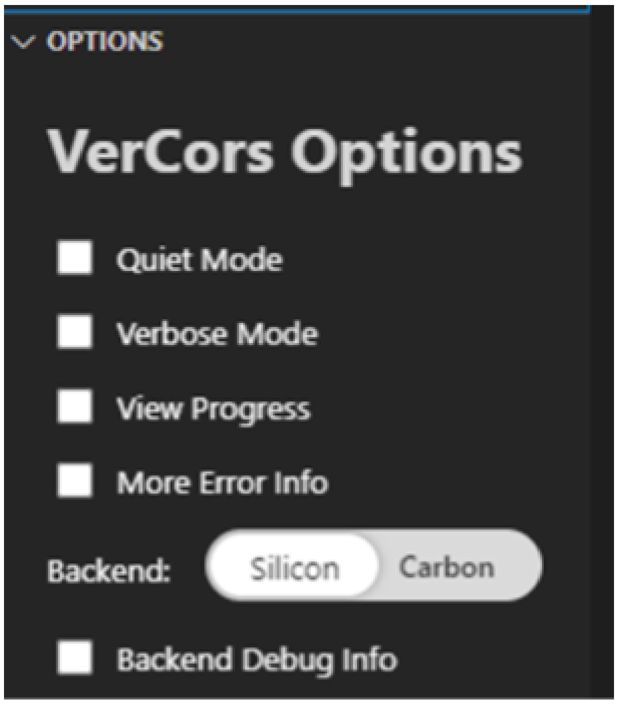


Figure 1: The old options window

2.4 Software environment

VSCode plugins are written in TypeScript and executed using NodeJS. Therefore, we used TypeScript to develop the plugin. For grammars (PVL and the specification language) we use TextMate grammars. Most of the team members were used to writing grammars in Antlr, but the VSCode API works with TextMate grammars so we were forced to use this. To implement features of an extension, VSCode provides an API to access settings or visuals in the editor.

As mentioned before, there was already a foundation for the plugin. This meant that there also was a GitHub repository owned by VerCors which we could use. We used Issues for all the user stories that we defined. In this way, we did not need an external application like Trello for project management. We followed standard Git protocols like creating a branch for each feature. Besides the plugin repository we were also added to the GitHub repository of VerCors in general, where we could ask the developers questions. Rick was also added to VerCors' skype group for quick questions. Lastly we could always contact the client via email. For internal communication, we used both WhatsApp and Discord.

2.5 VSCode Extension

VSCode extensions all follow a certain uniform layout, they revolve around three main concepts: Activation events, contribution points and the VSCode API. Activation events are events which trigger the extension to do a certain action. Contribution points are static declarations where contributions can be added to extend various functionalities, like new grammar, shortcuts, languages or system-wide commands. The VSCode API lets you interact with the existing components inside VSCode. With this API a user can change aspects of the user interface of the system or influence certain processes under the hood

By combining these, data can be gathered by activation events, the UI can be changed by using the API and functionalities can be extended by adding elements with the help of contribution points. This abstraction makes developing a tool easier since certain aspects are accessible by built-in functions, but it can be limiting when such functions are poorly documented and no real examples exist online.

3. Requirements

Before we started designing the system, the requirements were gathered. The development goals of the VerCors plugin for Visual Studio Code can be divided into three distinct goals: Extending the IDE to work with the PVL files and PVL code blocks in comment blocks; Making the command line tool work inside Visual Studio code so that it can verify supported files; And to present the textual outputs of the tool in a graphical clear way.

3.1 Extending the IDE

Extending the IDE means that built-in IDE features that are already supported for more popular languages like Java will also be supported for PVL code. This includes enabling users to click on variables to see the definition and origin of these variables and highlighting the syntax so the code is better readable. Later, code completion and refactoring support should also be made available as well as basic code generation in the form of PVL templates for common code structures like functions with parameters or for-loops. To streamline and improve the user experience, the most prominent features of the plugin could be available as easy shortcuts too.

1	**As a user, I want to be able to click on a variable to go to its definition.**	
	The plugin must contain the functionality that clicking a variable in PVL takes the user to the definition.	Must
2	**As a user, I want keywords to be highlighted with different colours.**	
	The Plugin must include keyword highlighting for PVL.	Must
	The Plugin must include keyword highlighting for the verification language inside comment blocks in supported languages other than PVL.	Must
3	**As a user, I want to generate templates with PVL code for basic code structures.**	
	The plugin should include an option where it can generate a ready-to-use template for common code structures.	Should
4	**As a user, I want to be able to refactor all occurrences of a variable or function in PVL code.**	
	The system should change all occurrences of a certain variable or keyword in PVL code or PVL comment blocks.	Should
5	**As a user, I want to be able to use shortcuts to use and operate the plugin.**	
	The plugin should include a shortcut to start a VerCors verification on the current file.	Must
	The plugin should include a shortcut to stop a VerCors verification on the current file.	Should
6	**As a user, I want to have autocomplete when writing PVL code.**	
	The plugin could give the user suggestions when writing PVL code, which the user can utilise to autocomplete code snippets	Could

3.2 VerCors in the Plugin

The plugin's main goal is to use the VersCors tool inside VSCode, an IDE. To make this work, the plugin must be able to verify Java and PVL files, and later all other supported file formats should work too. The tool in the plugin must have the same functionalities as the command line tool, to visually represent these, checkboxes must be made that resemble these options. The developers of VerCors brought up a couple of points that were helpful as well. For example, any started VerCors process should have the option to be terminated since it may take too long to finish. Because the tool is still under development, it could be handy to give the user a convenient way of selecting which version of the tool to use. The setting for each file should also be remembered so that the settings do not have to be filled in every time the user switches files. The settings should be stored across different sessions, that is, restarting VSCode should not reset any of the settings. Additionally, settings should be saved per-file.

7	**As a user, I want to be able to run VerCors on all supported program files**	
	The plugin must verify Java files.	Must
	The plugin must verify PVL files.	Must
	The plugin should verify c, OpenMP and OpenCL files.	Should
8	**As a user, I want to be able to toggle command line options.**	
	The plugin must have checkboxes for each command line option.	Must
9	**As a user, I want to be able to terminate the execution of a VerCors process.**	
	The plugin must have an option where the verification process can be stopped and terminated.	Must
10	**As a user, I want to be able to change the version of VerCors that the Plugin is using.**	
	The plugin could have an option so that different, locally installed, versions of the VerCors tool could be selected and used.	Could
11	**As a user, I want any configurations or settings to be saved.**	
	Restarting the IDE should not reset the settings	Must
	Different files should have different plugin settings saved for them.	Should

3.3 Visualising Output

One main issue of the current standalone VerCors command line tool is that the output can be difficult to read, and difficult to match with the verified file. To improve this user experience, the plugin will support a graphical representation of the tool's output. For starters, it must show the errors on the plugin view, and it should be made clear what part of the code the error is about. The plugin should inform the user which part of the code it is analysing and verifying in real-time while the tool is running so that a user knows which parts take a long time to check too. Another flaw with the command line tool is that the progress of the process is displayed in printed percentages, which do inform the user how far the tool is, but it lacks a clear graphical representation, hence a progress bar must be included.

12	**As a user, I want to be able to see a graphical representation of the errors found by the tool within the IDE.**	
	The plugin must highlight the errors identified by VerCors in the file, and link them to said error.	Must
	The plugin should show the errors identified by VerCors in a structured representation in the plugin window.	Must
13	**As a user, I want to know what part of the code is being checked while the tool is running.**	
	The plugin must include a visual representation of what part of the code, and line number, the tool is being checked.	Must
14	**As a user, I want to know how far into the tool's verification process is.**	
	The plugin must visually inform the user of the progress of the verification process through a progress bar.	Must

3.4 Changes of Requirements

During the process of developing the extension we managed to complete all the Must category but the requirements 1 & 4 which are Must and Should respectively. The reason why they have not been implemented is due to a requirement change discussed with Pieter - in order to resolve 'variables' in languages such as PVL it is necessary to parse the PVL syntax which is done internally by the Verscors tool. An additional challenge would be to create a release of the Verscors software so that we can extract the bin and point it to our extension which would be difficult to test. An ideal approach would be to be able to integrate the verscors tool via the language server so that resolution files can be passed internally without the /bin folder but this was way out of the scope of this design project. Pieter suggested that we could extract the resolution file but to do so it would be necessary to modify the source code of verscors. A suggested alternative was to just provide the interface for this implementation, and as such we decided to implement a language server, which provides hooks to compensate for the go-to definition features. In conclusion, these features are implemented partially via the language server that implements hooks that can be added and facilitate the development.

4. Meetings

During the module, our team had several meetings to ensure the quality of the product. In this section, we will discuss which kinds of meetings we had and how they influenced the process. Details on the content of the client meetings (including the meeting with the entire VerCors team) can be found in Appendix A.

4.1 Standup meetings

Every week on Tuesday at 12:15 we had a standup meeting with our team. During this meeting, we discussed the progress, since everyone was working on separate tasks. We also discussed how to tackle the assignments for reflection. Since we also had a discord in which we were actively discussing most days, we only had the need for one standup a week.

4.2 Meetings with client

Every Tuesday after the standup at 12:30 we had a meeting with the client. In the design/requirements gathering phase, we used this meeting to ask questions and gather information on what functionalities the plugin should include and what they should look like.

4.3 Meetings with supervisor

Since our client is also our supervisor, we combined the two meetings into one. However, for both the client/supervisor and us it was important to distinguish the two roles and have separate meetings. Therefore, each meeting started with a supervisor section, where we informed Pieter on our overall progress, how we've been cooperating with each other and if there have been any issues. We also discussed more of the academic side of the project - grading details, deadlines and any additional requirements.

4.4 Meeting with VerCors

Besides a weekly meeting with the client, we also conducted a demo to the VerCors team, where we showcased the features we had implemented so far, about halfway through the project. We got a ton of valuable feedback from them, thus we had to adjust some requirements.

4.5 Peer review meetings

Once every two weeks there was a peer review session where we, along with 5 other groups developing a product similar to us, got feedback on our product. We prepared a short presentation updating our peers with the newest features of our project. After the presentation we received questions and feedback. The other teams also presented their progress which was interesting but not so useful for our project.

5. Design

5.1 System overview

The goal of the project was to create an extension that integrated the VerCors command line tool into the text editor VSCode such that all functionalities of the tool could be easily used inside VSCode. The design of the plugin can be divided into three main parts, the integration of the VerCors tool inside VScode, the visualisation of the tool's output and the extension of IDE features for the Prototypical Verification Language (PVL) and the VerCors specification language.

To sketch a better overview of how a user interacts with the different parts of the extension, we will sketch a scenario in which a new user utilises the tool. It starts with a user installing the extension, then before they can use VerCors they first have to choose the locally installed version of VerCors by selecting the bin folder on their device. After this, the user can open or create a file which is either written in PVL or has VerCors specification language embedded in special annotated comments. The user can then choose the command-line-interface (CLI) options by checking the boxes next to the option and running it by pressing the start button at the top of the extension window. The user sees that a progress bar pops up in the bottom bar which informs them how far the tool is in checking the file. When the tool is finished and the progress bar has filled up, the found errors are gathered and printed in the problems tab, together with other diagnostics. This scenario is also modelled in Diagram 1.

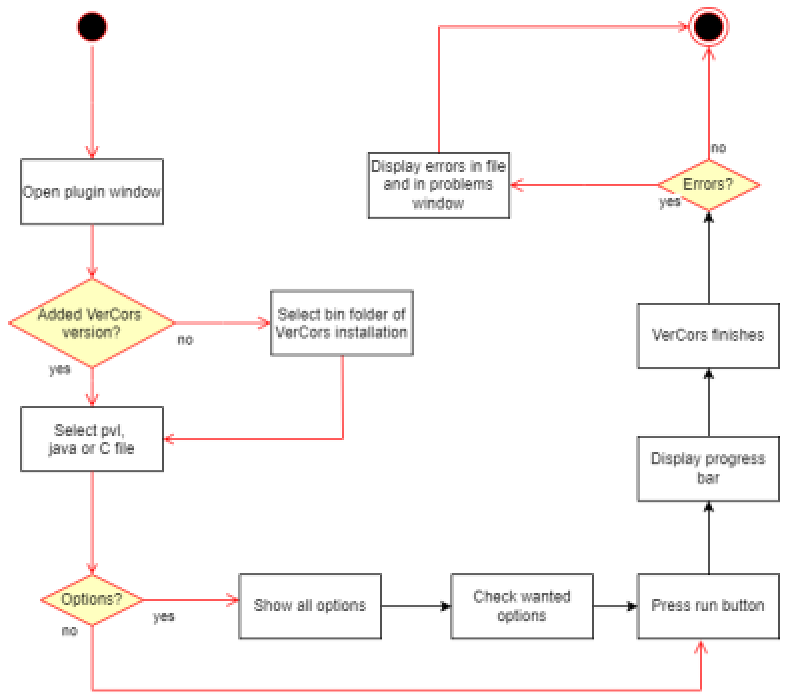


Diagram 1

The tool integration part consists of most of the features shown in the extension window, such as version selection and the CLI options. Version selection is as straightforward as clicking the “add version” button and then navigating to the location on the computer where the version is saved. Multiple versions can be added to the extension which is all saved so that a user can easily switch between versions. The CLI options can be enabled and disabled by clicking the boxes next to them. A user can also pin the options which are used more often, These pinned options will then be shown globally at the top of the options list. The selected options are saved for each file, so if a user switches to a file they have previously filled in options for, these will be put back automatically.

In the command prompt, the VerCors tool outputs the verification in plain text. To make the output better readable inside VScode the output gets parsed and displayed graphically. The output of the tool both gives insight into the progress of the verification process and finally the errors resulting from the verification. The progression is shown in the extension tab under the start/stop buttons in the form of a progression bar with the current state of verification displayed underneath. After verification, the tool also outputs the found errors. These errors are parsed and shown in the problems tab, together with the error message, and the location of the error.

The extending of the IDE was done in the form of including IDE features in the VSCode environment. In the code editor, this meant adding IDE features for PVL and the specification language used for VerCors. When typing these languages either inside a PVL file, or inside designated comment blocks, features will come up automatically. The first thing you see as a user is that the code is highlighted which makes writing the code and inspecting it more pleasant. Also while writing the plugin will make suggestions in the form of autocomplete and snippets.

5.2 Global design choices

Some of the design choices were already made for us, with the already existing initial version of the plugin - what languages we use, different views of the plugin page, etc. We built upon these choices by using VSCode native styling, and despite modifying the file structure quite a bit, the initial skeleton that was there can still be identified.

Language server was implemented later during the project due to the previously mentioned change of requirements. This change implements a client server architecture which implies two separately running instances.The server is launched via the client by creating a connection that is passed to the server. This also required a change in the way the compilation and building of the project is done as the extension needs to include the 2 components, client and server. The advantages of this design choice are explained in more detail in the language server section. The benefits are the implementation of the language server protocol via JSON RPC messages which implies that the implementation for a language server can be compatible with different IDE extension clients. Hence the same LSP could be re-used for different extensions. Additionally optimisations in the control over the file with the webhooks which facilitate the implementation of ide features such as error display.Finally it respects a fundamental concept of computer science and maintainability of code bases by providing separation of concerns where the extension can be developed with most logic in the server thus improving code quality and readability as well as prevent some bugs from being introduced.

5.2.1 Language server

In order to improve the extension we made the conscious choice to include a language server. The reasoning behind this choice was to make the code more maintainable and add easy support for autocompletion, syntax highlighting, structural snippets, as well as allow for future features like go-to definition and even a continuous verification mode.

Most of those features can be implemented separately without a language server, however the language server has benefits in terms of performance, maintainability and portability as it implements the Language server protocol[2]. As not all VerCors users use VSCode, it is within reason to believe the extension might be adapted to other IDEs in the future, and our design choice makes this a much easier task.

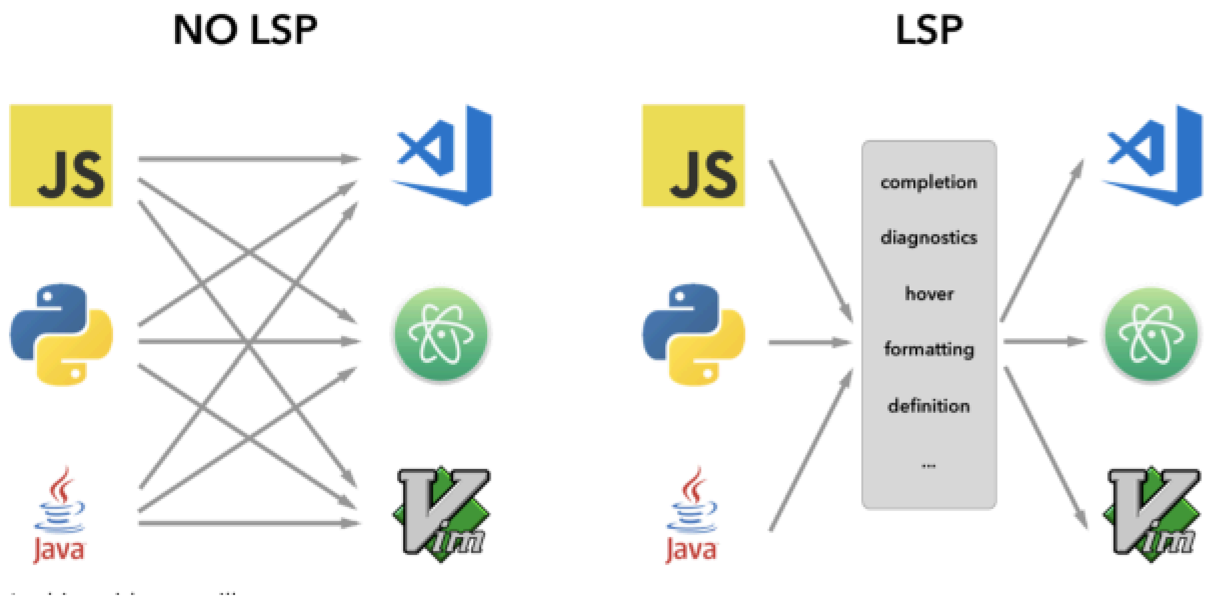


Figure 2: Language Server Protocol

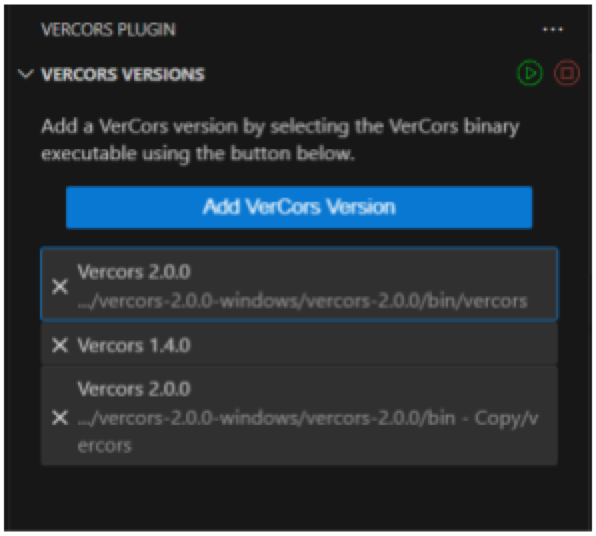
[2] https://microsoft.github.io/language-server-protocol/

5.3 Detailed design

Apart from big design choices, all smaller parts of the project have been carefully thought about as well. Hence, we will discuss the design of each part of the project here in more detail. We will talk about the design choices in the form of how certain requirements or preferences from the client shaped elements of the final product accompanied with screencaptures.

5.3.1 Version selection

One of the requirements states that the plugin should allow users to easily switch between different versions of VerCors. Therefore, the plugin contains a section in the view that is dedicated to version selection, as shown in Figure 3. Versions are added through the button, which opens a file selection dialog and lets the user select the VerCors binary executable. After selecting the executable, the corresponding version appears in the list. The version is detected by running the command with the `--version` argument. If two executables output the same version, their paths are included in the list to differentiate them more easily. The selected path is always visible when hovering over the version. The selected version is highlighted with a blue outline and the user can easily switch by clicking on another version. Every version can also be removed by using the X button. In addition to release versions, we also support versions in development, as they are added through the user selecting the binary, rather than a directory in which to look for a specific "vercors" executable.



Add a VerCors version by selecting the VerCors binary executable using the button below.

Vercors 2.0.0
.../vercors-2.0.0-windows/vercors-2.0.0/bin/vercors

Vercors 1.4.0

Vercors 2.0.0
.../vercors-2.0.0-windows/vercors-2.0.0/bin - Copy/vercors

Figure 3: VerCors versions.

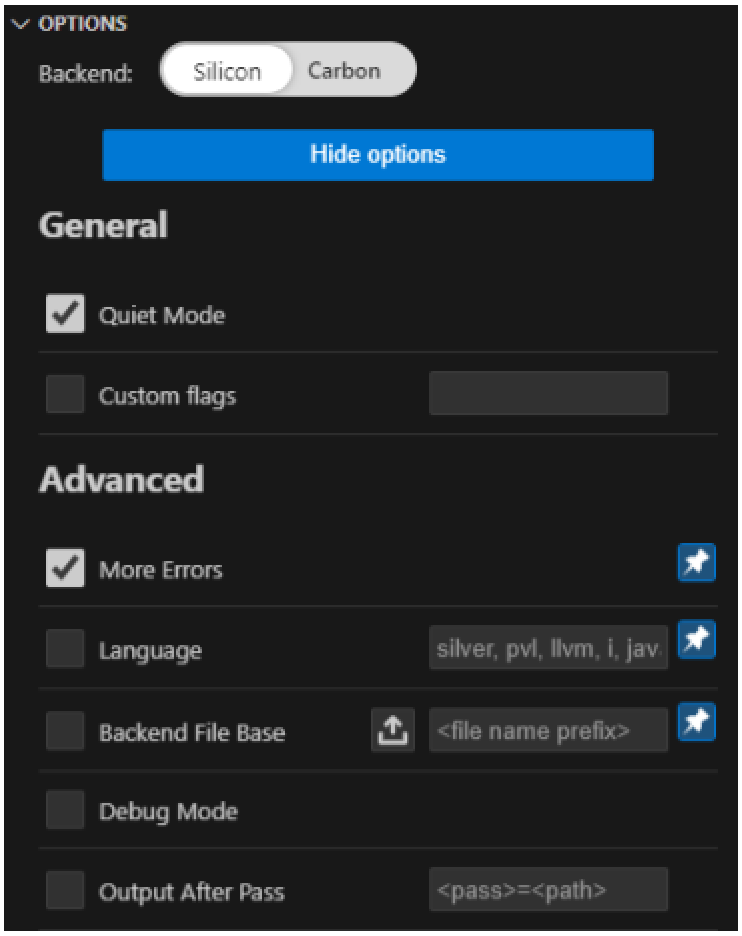
5.3.2 Snippets

The requirements state that "the plugin should include an option where it can generate a ready-to-use template for common code structures". This is how we interpreted the requirement at first - for the users to define their own snippets. After quickly realising that allowing users to define their own snippets would be significantly more difficult than providing a set of snippets ourselves, we decided to reformulate the requirement. In the VerCors Skype we asked all developers for common code structures which could be implemented as snippets. With this list we managed to add support for the most common quantifiers and keywords that have a bigger structure following them, e.g. `\forall`. An extensive list of supported snippets and their corresponding keywords can be found at Appendix D.

5.3.3 Command Line Options

As any command line tool, VerCors came with numerous command line options that alter the execution of the tool. To support this, we created a window with a list of all these options:

As Figure 4 showcases, users can freely toggle any option and the extension will run the tool with it. Since there are many options, we decided to hide them by default, and let users decide which ones should be shown through the pin feature. If the "Hide Options" button is pressed, all options that are pinned will remain visible, as well as any options currently active.


OPTIONS

Backend: [✓ Silicon] Carbon

Hide options

General

- [✓] Quiet Mode
- [ ] Custom flags

Advanced

- [✓] More Errors
- [ ] Language silver, pvl, llvm, i, jav
- [ ] Backend File Base <file name prefix>
- [ ] Debug Mode
- [ ] Output After Pass <pass>=<path>

Figure 4: CLI Options Window

Since some options take various arguments, an input field for those is added, and in some cases a file upload.

Pinning is a global feature - all pinned options will remain visible regardless of the file currently open, however which options are actually checked is file-specific. This means users can have different options for different files.

To allow for these options to be extended in the future, they are generated via a JSON file, which contains an entry for each option, alongside some setting parameters for it:

- **name** - the name of the option with which it will be shown in the extension.
- **description** - short description of the option, which will appear as a hover tooltip.
- **skip** - boolean value to determine whether the option should be skipped when generating the view; this is useful as some options are on by default and cannot be changed. This will be expanded on later.
- **arguments** - signifies that the option takes an argument so an input field will be created for it. The string value of the argument key will be used as the placeholder for the input.
- **input** - currently only used for the “--backend-file-base” feature, it creates a button with a directory selector dialogue, out of which the path will be extracted and used. It can be used for other options in the future.

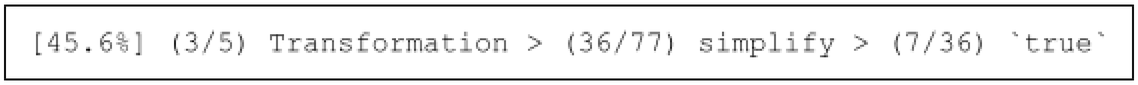
In addition to these parameters, each option belongs to a category to help group them via common features.

By default, some options are enabled and this cannot be changed - “verbose” mode and “progress” mode. They are crucial to the functionality of the extension, as without them the VerCors tool does not output any information about the execution, hence the progress bar and displaying errors would fail to work. Additionally, options like “version” and “help” would do nothing, which is why we use the “**skip**” flag. We decided to keep them in the JSON file with all the options in case future VerCors developers have other ideas.

5.3.4 Displaying VerCors output

A big part of the project was to present the output of the VerCors tool in a visual way. In the current state of the tool these textual outputs consist of progress messages, debug messages and error messages. To access, process and show these messages we had several options. One option was to alter the VerCors tool itself to output certain data structures like JSON to parse the messages in a fast and easy way. Ultimately we steered clear from this idea because this meant that next to developing a VSCode extension, we should also study, change and test the tool. Because this was out of the scope of this project, we opted to parse the messages line by line as they were, using regular expressions (regexes). Regexes were easy to use and we had some experience in working with them. Javascript and typescript have good built-in regular expression support, so it was the most apparent choice.

For the progress, The command line tool prints it in a uniform way like in Figure 5. It always starts with the global progress expressed in percentages, then the current stage of the verification process followed by the step inside the current stage. To access this we found that the best way was to use regular expressions to match the lines and extract the information using groups.

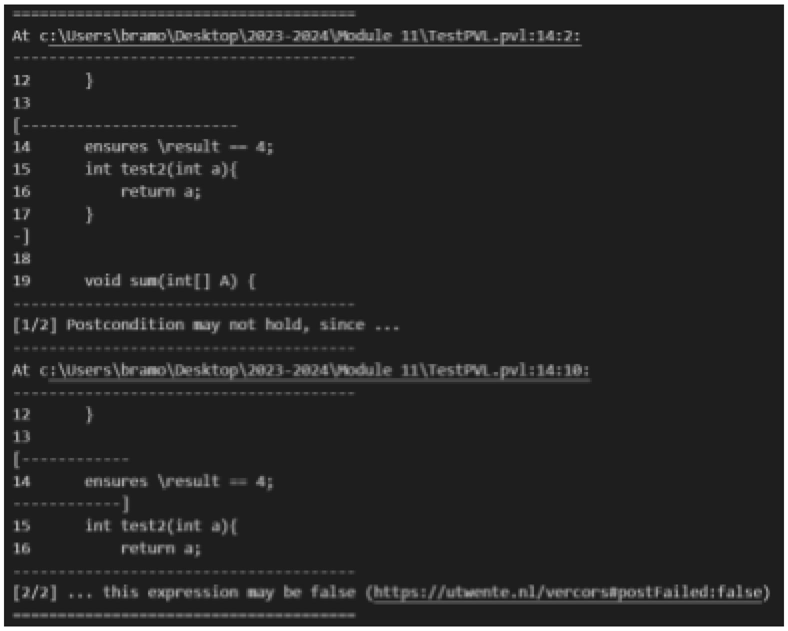


```
[45.6%] (3/5) Transformation > (36/77) simplify > (7/36) `true`
```

Figure 5: Progress output message

To display the progress we implemented a progress bar which displays the progress of the tool together with the current state it is in. More details on the progress bar can be found in subsection 5.3.6. During testing we found that parsing the progress failed on some machines, but it succeeded on others. After debugging we saw that some machines displayed decimal numbers using a comma, while others printed point-separated numbers. Luckily we could easily change the regexes inside the code to cover this problem.

Especially for displaying the errors, we considered introducing a hook to the VerCors tool. But as discussed before we decided not to, and instead opted for line-by-line parsing with regexes. To explain how we parsed them we will first look at how errors in VerCors are presented. As you can see in Figure 6, each error is printed between two '='-lines. Inside these lines, the error is displayed by multiple error parts. Each part is about a different part of the code concerning the error and consists of three separate sections separated by '-'-lines, they always follow the same structure of sections, first the location of the part, then the code piece the error shows up in followed by an error message. One VerCors error can consist of one or multiple of these error parts. Because these errors are not set structures but are parsed line-for-line we work with stages. When the parser sees a '='-line it knows it is parsing an error, so it can handle all the different parts, when it sees that the error is parsed, it combines all error parts and puts the whole error in a data structure, ready to be dealt with.

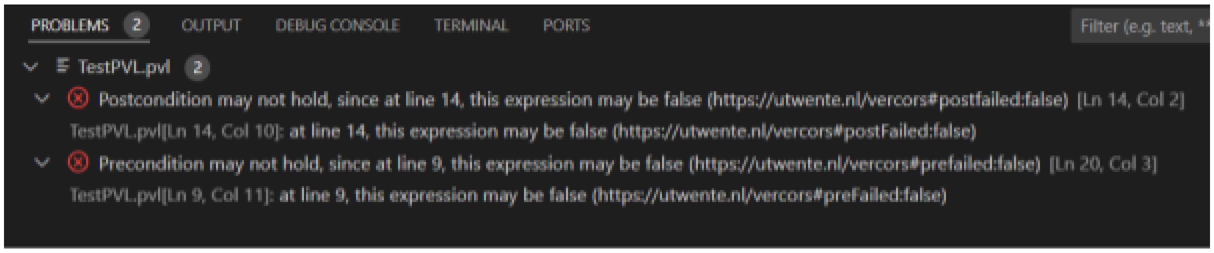


```
======================================
At c:\Users\bramo\Desktop\2023-2024\Module 11\TestPVL.pvl:14:2:
--------------------------------------
12      }
13
[-----------------------
14      ensures \result == 4;
15      int test2(int a){
16          return a;
17      }
-]
18
19      void sum(int[] A) {
--------------------------------------
[1/2] Postcondition may not hold, since ...
--------------------------------------
At c:\Users\bramo\Desktop\2023-2024\Module 11\TestPVL.pvl:14:10:
--------------------------------------
12      }
13
[-----------
14      ensures \result == 4;
-----------]
15      int test2(int a){
16          return a;
--------------------------------------
[2/2] ... this expression may be false (https://utwente.nl/vercors#postFailed:false)
======================================
```

Figure 6: A VerCors error in the terminal output

After the errors are parsed from the output, they are all saved in a data structure and they should also be displayed graphically in VSCode. We had two main ideas to do this: Displaying them in the extension view and displaying them in the diagnostic window where all the other errors and warnings are also put. We addressed the choice to the client who answered that displaying them inside the problems tab would fit the plugin better also since this was the case in the already-existent Intellij plugin.

The VSCode API offers the functionality to make a separate diagnostic collection where errors can be put in. The errors in this collection would then be published inside the problems tab in the VSCode view. Each error consists of a primary location in the file, an error message and so-called related information. Which are smaller errors with an error message and a location as well, in other words, one error item in the diagnostic collection can contain multiple locations and multiple error messages. This option fits perfectly with the VerCors errors, where an error can concern multiple places in the code.



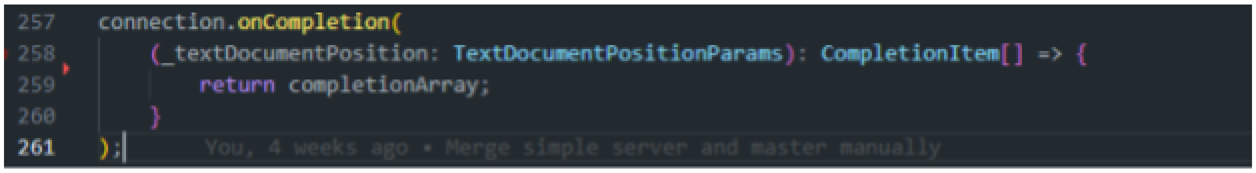
```
TestPVL.pvl 2
  Postcondition may not hold, since at line 14, this expression may be false (https://utwente.nl/vercors#postfailed:false) [Ln 14, Col 2]
    TestPVL.pvl[Ln 14, Col 10]: at line 14, this expression may be false (https://utwente.nl/vercors#postFailed:false)
  Precondition may not hold, since at line 9, this expression may be false (https://utwente.nl/vercors#prefailed:false) [Ln 20, Col 3]
    TestPVL.pvl[Ln 9, Col 11]: at line 9, this expression may be false (https://utwente.nl/vercors#preFailed:false)
```

Figure 7: Error messages in the Problems-tab

As you can see in Figure 7, The errors have one central error message, which are all error messages of the error parts concatenated together. In this example, both errors also have a secondary location in the error expressed as a related information item under it. If a user clicks on any of the errors, both main error and related information error, the user's cursor will be sent to the location in the file. Each error points to a place in the code, this can be seen in the editor similar to other diagnostic errors, by a red squiggly line underlining the error

5.3.5 Autocomplete

In order to implement the autocomplete we decided first to integrate a language server in the architecture that would also facilitate other features. This was done based on an lsp-sample[3] provided as an example from the open source Microsoft repository on Github. After this we had to modify to generate completion lists that would be called in the `onCompletion` hook:

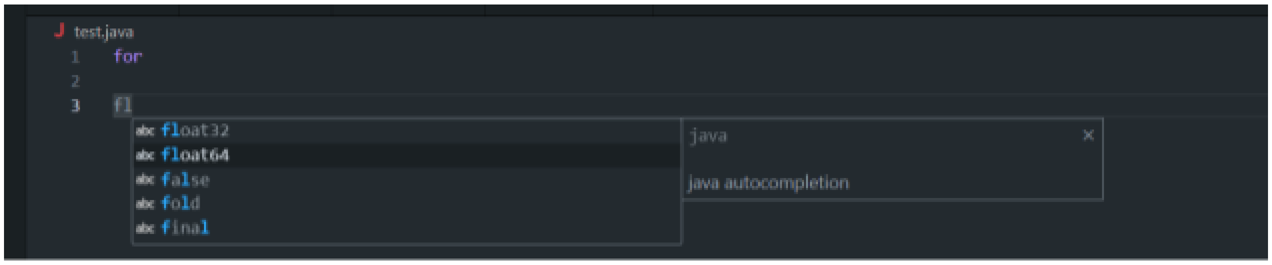


```
connection.onCompletion(
    (_textDocumentPosition: TextDocumentPositionParams): CompletionItem[] => {
        return completionArray;
    }
);      You, 4 weeks ago • Merge simple server and master manually
```

Figure 8: onCompletion handler, which fetches the list of autocompletable entries

[3] https://github.com/microsoft/vscode-extension-samples/blob/main/lsp-sample/server/src/server.ts

The JSON list was generated through the list of keywords specified by the tmLanguage that was already provided by the previous VerCors developer. It was necessary to extract the token from the regex as the language server was not supporting regex in its completion list. When the lists are generated, launching the extension provides the completion of a selected suggested item if one edits a substring in the supported languages.



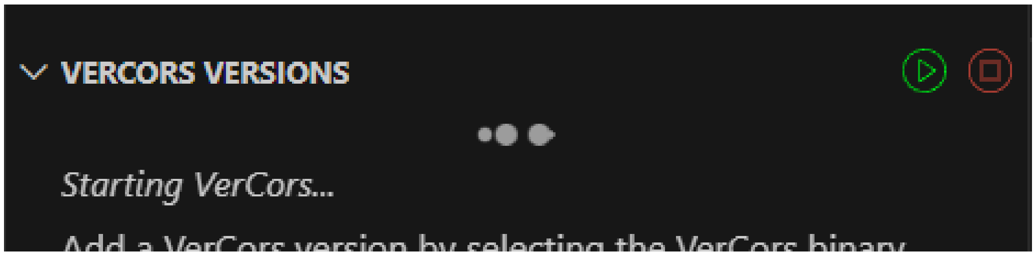
```
test.java
for

fl
float32
float64
false
fold
final
```

Figure 9: Autocompletion example

5.3.6 Progress bar

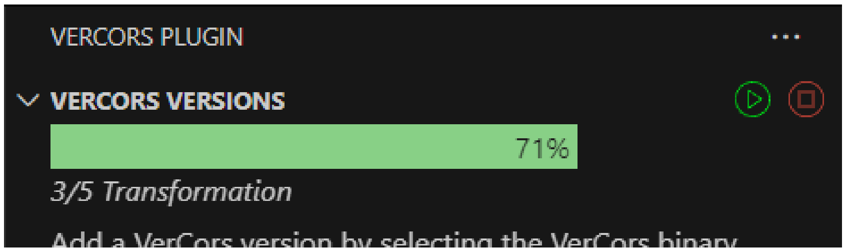
The progress percentages as given by VerCors are parsed and sent to the progress bars, where they are updated. Initialization of VerCors does not provide any percentages, so a loading animation appears while the tool is starting (see Figure 10). After the tool has begun verification, the progress bar appears and is updated live alongside the verification progress. Besides the current percentage, it displays which state the verification is in.



Starting VerCors...

Figure 10: Starting VerCors.

There are 2 progress bars. One in the plugin window (Figure 11) which can be hidden, and one in the status bar (Figure 12) which is always visible. The progress bar in the plugin window hides itself automatically after verification has completed or the process was terminated.



3/5 Transformation

Figure 11: Progress bar in the plugin window.

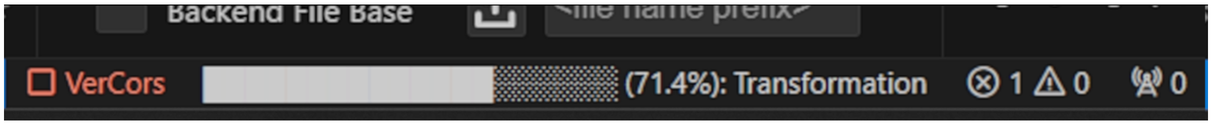


Figure 12: Progress bar in the status bar.

The VerCors tool sometimes outputs incorrect percentages. These are dealt with internally so the progress does not retrogress.

5.3.7 Shortcuts

Although Shortcuts are a small and easy to add addition to a rather big extension, they were a requirement from the client, and deserved to be thought about. There were two main actions that required shortcuts, which were starting a VerCors verification process, and terminating a process. Later we were looking at other actions in the extension and decided that a shortcut for adding new VerCors versions could be a welcome addition as well.

VSCode already has a lot of keybindings for most actions inside the editor. So after analysing all already existing shortcuts and testing loads of combinations we ended up using the ctrl+shift modifiers for the shortcuts in combination with a letter on the keyboard. If a user wants to start a verification they can simply press the button combination `CTRL+SHIFT+L`. If a user later wants to cancel the VerCors process the buttons `CTRL+SHIFT+Q` have to be pressed simultaneously. And to add a new version, `CTRL+SHIFT+J` can be used. Additionally, `CTRL+/` will create a comment block while focused on the text editor if it's in one of the supported languages.

If a user is not comfortable with these keybinds, they can change them through the VSCode settings.

5.3.8 Starting/Stopping

While implementing all the cool features to interact with vercors, running VerCors itself shouldn't be forgotten. Without it, the plugin would be useless. That is why we created quick and easy ways to run VerCors. The first way is to go to the VerCors version view and press the start or stop icon. However, to use this you need to have the VerCors plugin tab always open. That is not the most user-friendly way, so a second button is also added. On the bottom left of the status bar, there is also a start/stop button. That button is always visible, so easier starting and stopping is accomplished.

The way VerCors is started and stopped is pretty straightforward. When the start button is pressed, the language used is checked. If the language option is not used, the plugin looks at the file extension of the selected file. If the file extension is not C, PVL or Java, it will throw an error. If everything is correct, it will run VerCors with the selected options in the shell.

When VerCors has to be stopped before it is naturally finished, the tree-kill function will be called. The tree-kill function stops a certain process and every process that the main process creates. This way, all subprocesses created by VerCors will be stopped.

5.3.9 Syntax Highlighting

The extension came with some limited highlighting for PVL files. We extended this support through the grammars used by the plugin, and now syntax highlighting can be seen in PVL, Java and C files. For the latter two, it only works within a comment line or block, as the VerCors language specifications only work there.

Currently, VerCors keywords are recognised and coloured as a few different types, so the colouring is arguably somewhat limited, and as such, within the list of future improvements for the plugin, adding to the colours is one of them.

Setting up the grammars well was also crucial in later features, like Autocomplete (5.3.5), as the auto completion mechanism also relies on the grammars.

5.4 File structure

The root project folder contains a `package.json` file with the extension and language configurations and dependencies. Since the design is split into a client and server, their files have been split into different folders as well. This way, both the client and server are separated into their own packages with their own dependencies. The `package.json` file has been configured to compile both.

5.4.1 Client

The client contains a number of files. Most of which contain a single class or interface to ensure proper modularization and enhance maintainability. The files have the following purposes:

- **comparing.ts** - Utility class to compare sets and lists.
- **extension.ts** - The main initialization file that sets up the extension and runs the language server when VSCode is started.
- **output-parser.ts** - Used to parse VerCors' textual output and redirect actions to the right destinations.
- **progress-receiver.ts** - Interface used to update progress as provided by VerCors.
- **status-bar.ts** - Represents and manages the plugin's status bar section including the start/stop buttons and progress bar.
- **vercors-options-webview.ts** - Manages the plugin window tab that includes all the command line options, and manages the pinned options and selected options per file.
- **vercors-paths-provider.ts** - Manages all the added VerCors versions including their paths and which version is selected.
- **vercors-run-manager.ts** - Used to start and stop the VerCors tool.
- **versors-version-webview.ts** - Manages the plugin windows tab that includes the VerCors version selection.
- **webview-connector** - Interface used in backend to receive messages from a VSCode webview's frontend.

5.4.2 Server

The server simply consists of the **server.ts** file which starts the language server, as this is still a minimal feature of the plugin.

5.4.3 Resources

The resources folder contains various SVGs and PNGs used on the frontend, as well as the following files:

- **command-line-options.json** - Contains details about all the command line options that VerCors supports.
- **html/vercorsOptions.html** - Web page for the CLI options webview of the plugin. Uses the command-line-options.json to generate a toggleable checkbox for each option. More details in section 5.3.3
- **html/vercorsPath.html** - Web page for the paths provider webview. Details in section 5.3.1.

5.4.4 Syntaxes

This folder contains TextMate Grammars for PVL, as well as for VerCors code inside of PVL, Java and C:

- **comment-injection-java-c.tmLanguage.json** - Used to add matchers for comment blocks and lines that start with an "@", to then inject the VerCors specification inside.
- **language-configuration.json** - Contains the language configuration for PVL.
- **spec-injection-grammar.tmLanguage.json** - Injected into the comment specification for C and Java, this file has VerCors keywords to be matched and highlighted.
- **pvl.tmLanguage.json** - Language grammar for PVL.
- **pvl-language-configuration.json** - Language configuration for PVL.
- **snippets.json** - Adds autocompletion for code templates that involve bigger structures
- **language-server-pvl-matches.json** - Matchers for the Language Server's autocompletion for PVL.
- **language-server-java-c-matches.json** - same as the above, for C and Java.

6. Testing

When creating an application, testing is of the utmost importance. For creating a plugin, it is no different. However, vscode has only a little information about testing your extension. That is why we had to find ways of testing the VerCors plugin. We have divided testing into 2 main categories, manual and automated testing.

6.1 Manual testing

Manual testing is done for most of the extension. Every time a functionality was created, 2 team members who did not work on that functionality tested everything manually. The fix for the functionality itself was tested, but the earlier implemented functionalities were also tested. This was done to make sure that no functionality was broken by the creation of a new functionality. This worked quite well, but it was boring labour and not sustainable if the extension would grow. That is why we also created a system for automated tests.

6.2 Automated testing

Vscode has some standard practices for automating code, but they are pretty minimal. They are mostly meant for small unit tests. However, we wanted to be able to test big parts of the system because a lot of things work together and the created tests look more like real actions a user could take. That makes the process of going from manual testing to automated testing easier. To create such a testing environment, we decided to divide the automated tests in 3 parts: storage tests, backend tests and frontend tests.

6.2.1 Backend Testing

In this project, backend testing is defined as testing the resulting call send to the frontend after getting a call from the frontend. We defined it as such because api calls are the input and output of the backend. If you see the backend as 1 black box, a call from the frontend would be the input and a call from the backend would be the output. This way, the automated test would be as close as it can be to manual testing. It also increases the test coverage, because a lot of functions are used when you go from call to call and all of them are tested in 1 test.

While making the tests, a lot of problems occurred. Vscode extensions look like a website from a programming perspective because it has a clear backend and frontend (written in html) and they communicate using API calls. However, the backend can still use many features from the operating system. In testing, some of those operations were not possible or undesirable. That is why we decided to mock some functions. Keeping those mocks to a minimum was a main priority, mocked functions cannot be tested.

In the end, we mocked a great deal but they can be mostly divided into 2 main categories: file mocks and frontend mocks.

6.2.1.1 Function mocking

File mocking is done to ensure that as few files are used as possible. Ideally, no files would have been accessed, because that prevents accessing or destroying the wrong files, but that was not possible. To mock the file system, mock-fs is used. Mock-fs redirects any calls done with fs to a fake file system. In a similar way the workspace settings are mocked. However, one part of the system runs VerCors to check if the given VerCors version is correct. We thought it would still be valuable to test because the shell can run differently on different operating systems. By putting VerCors in a folder called fakeVercors we made sure that we could still test it. Unfortunately, this means that real files had to be accessed. To prevent accidental file manipulation of wrong files, we created 1 variable called vercorsPath that has the full path to the fakeVercors folder.

Now we can access the necessary files, but sending or receiving messages is still impossible. That is why we also mocked the frontend. Frontend mocking is done by creating a fake frontend. The frontend is passed to a WebviewViewProvider in the form of a WebviewView. A WebviewViewProvider handles all calls from and to the frontend using the WebviewView. The WebviewView has a webview object and it can send or receive messages using their respective functions. By mocking the function that sends messages, we can log those sent messages and thus check what is sent by the backend. The code is structured in a way that makes it easy to mock receiving messages because everytime a message is received, the backend calls receiveMessage with the message as an argument. So to mock receiving messages, just calling the receive function with a certain message works. Our extension uses multiple WebviewViewProviders that can all use different functions. To ensure that there is always an explicit receive message function, the WebviewConnector interface is created. Every WebviewViewProvider implements that interface.

Now we can send and receive messages, but not the entire frontend is handled by the webview. Warning and error messages are shown by vscode itself, so they are mocked by checking if the vscode api called them.

The last important mocked part is not in one of the aforementioned categories. It consists of one mocked object, namely the extension context. When the extension is started, an extension context is created. However, we couldn’t access the context while testing, because the standard testing package we used (`@vscode/test-electron`) doesn’t allow that. A better testing package was hard to find, so we tried to mock the extension context. After setting some elements, mainly the path, we had enough to use this fake context in place of the real one.

6.2.1.2 Testing

After the mocked functions were created, real tests could be made. We focused on testing the possibility of modifying and getting the VerCors path and options. Nothing else is tested in these backend tests because they test the most important components testable by backend tests. There are some frontend features, like keyword highlighting, that need different testing practices. Other features, like the progress bar, did work with calls to and from the frontend, but because of time constraints, we decided to not test it. The most important feature of the extension is running VerCors with the right options, so that is what we focused on.

Since a lot of work was done after to have a well-running backend for the tests, making the tests themselves was not extremely hard. Every test is done by sending a command to the backend and evaluating the command sent back by the backend. To test if the correct command was sent back, a new assert function was created: failOnJsonEventAbsence. Every command is structured as a JSON object, so this function checks if the right command is sent by comparing the expected JSON value with the logged JSON values.

6.2.2 Storage Testing

Vscode extensions have a small storage system to store certain values after the extension is closed, a settings file. Because it is just a file, it can be changed in unexpected ways. That is why a whole set of tests is made, only to test storing the options and path values.

To make sure that no real files were changed, the settings file was mocked by creating a dictionary, now called fake settings, that acted like the settings file and having any command changing the settings file be redirected to the fake settings.

We tested the set and got functions by attempting to set or retrieve specific values from the settings. We could observe what would happen if the settings file became polluted with incorrect values by beginning the settings at different values.

Return values could be JSON-like, array-like, or just plain text. This difference in return values prevented the use of normal equal methods. That is why we created unique equal methods and used them in unique assert.equals that accept custom equal methods. Normally, the package chai would be used for this, but chai broke constantly so we made our own equals method.

6.2.3 Frontend Testing

Although storage and backend tests were finally completed, frontend tests were excluded due to time constraints. If we had time, we would have simulated button presses with webDriverIO and seen what results they returned. With that information, tests could be made to check if all backend calls were made appropriately. The majority of the frontend's visual testing would be done by hand, however, some could be done automatically. One such automated task may be to verify whether keywords are correctly categorised for keyword highlighting.

7. Future Improvement

We have deployed the VerCors VSCode plugin as a finished product, but that does not mean it can not improve. In this chapter we discuss features that can be added in the future, and the structure we have provided to ease the development of these features. Since the system is not only available to the VerCors developers, but also to the public on the VSCode marketplace more features could be requested by the public.

7.1 Language server

As explained in Chapter 5, we have implemented a language server in the plugin. Some features in 7.2, like go-to-definition are not easy to implement and require a language server. Because we already have the language server in place, adding these features in the future becomes a much simpler task. Besides this, the language server can also be extended to other IDEs, since it works with the language server protocol.

7.2 Additional features

The following features either were part of the initial requirements but were dropped since they were out of scope for the design project or are simply ideas by us or the developers thought of during the implementation phase:

- Continuous verification: this feature was not a requirement, but was planned by the developer before us. Having this feature would allow the user to toggle continuous verification mode on and off. When it is on, running the VerCors tool on the current file would not require pressing the run button, but would happen for example after saving.
- More snippets/user snippets: the snippets that are implemented are all requested by the users, but currently it is not easy to add snippets. Although VSCode allows users to define their own snippets, it could help to make this a more accessible feature within the plugin.
- Go-to-definition: This feature, together with the next two features were requested by the client but are not included in the final product. More on the reason behind this can be found in chapter 8.1. Having this feature would mean that control-clicking a variable in PVL would show the user the point in the file where the variable was declared.
- Refactoring: Having this feature would mean that a user can refactor all occurrences of a variable. This should not be done by simply searching for the string in the file, but should only replace occurrences that have the same semantic meaning.
- Line-by-line in file: Having this feature would mean that while the tool is running you would not only see progress in the progress bar, but also in the file that is being verified itself. It would show exactly which line of the program the tool is currently verifying, which helps in more easily understanding why it gets stuck at a certain point.

- More keyword types for VerCors keywords: currently most of the keywords are all the same type, so they get coloured in the same way by the syntax highlighter. It can be beneficial to have more types, so that it’s easier to distinguish between different keywords.

8. Evaluation

8.1 Requirements

Go to definition has not been implemented for PVL and other languages within the scope of this project. The main reason being that after discussing with Pieter we discovered it required adding a parsing stage to the VerCors project, running the stage and compiling the project to a bin that we could then use within our tool. After discussing and defining that the main focus of this project is working on the VerCors IDE features and not modifying VerCors itself. We decided to not include this feature. In the future, given that a resolution stage of variables is integrated within verscors it will be relatively easy to implement using the language server.

A similar case holds for seeing the progress in the file: this is a feature that was initially included in the requirements, but was dropped later. After discussion with the client, it became clear that this feature also requires the modification of the VerCors tool itself, since now it does not provide sufficient information about which exact line in the file the tool is processing. We agreed that this is out of scope for this project since the focus should be on the plugin features and not on modifying VerCors.

8.2 Teamwork

8.2.1 Rick

I enjoyed working on this project with the team. Developing an IDE plugin was new to me and the learning curve was steep.

I implemented snippets for the verification language. I also looked into the language server before Lucas took over. In this report I wrote the following sections: Abstract, Domain Analysis, Meetings, Future Improvements and User Manual. Besides that I was in contact with the VerCors developers via Skype. I also designed slides and presented our progress at the peer feedback sessions.

8.2.2 Bram

Working on this project was a new experience for me, I have never worked on building an extension before, so the project took more research than I expected. The teamwork went well overall, prior to it I did not know half of the group, but I would say that nonetheless working together went well.

I first worked on the syntax highlighting part, and later on the error parsing, and error presentation in the VSCode editor and problems tab. For the report, I wrote the introduction, the global system overview and the detailed design description of the aforementioned project contributions.

8.2.3 Jaron

This project was the first big project I have ever done with a team. There were some challenges, like implementing a language server, but it went generally pretty well. I focused mostly on the backend part of the options and the automated tests. I liked finding ways to test it that first looked impossible to me. The teamwork in this project was exceptionally good in my opinion. We had weekly meetings where the attendance was good and everyone did what they were supposed to do. If someone had questions, they would be answered rather quickly. I think that that really increased the motivation for this project within the team.

8.2.4 Pascal

Everyone in the team was specialised in their own part and equally motivated to deliver a good product, which resulted in a good collaboration. I developed an IDE plugin before which I used during the Programming Paradigms module to run and add syntax highlighting for the ILOC language. However, that was in IntelliJ, which uses Java instead of TypeScript. I also never worked with VS Code, so the whole environment was new to me but it was relatively easy to understand. However, the VS Code API does not have very good documentation, so we had to rely a lot on example projects. I started the project by working on the part that handles version selection, including frontend and backend. This also included the progress bar, which directed me to work on the status bar too, since those two are similar features. For the backend, I ensured consistent file naming and modularisation were used, and for the frontend I made sure the menus fit within the VS Code style and worked across different themes.

8.2.5 Denis

Throughout my bachelors' I mostly worked with the same people over and over, so at first this project was a bit worrying - having half of my team be people I didn't know. I however quickly realised that it's not an issue whatsoever as we all clicked very fast and our teamwork was very good.

I initially worked on researching how the grammars for the plugin worked, and then focused on the command line options. I also somewhat helmed the organisational duties, although that was a collective effort. Everyone did their part and if someone wasn't on track, they would reach out for assistance.

8.2.6 Lucas

I am used to multidisciplinary teams due to the fact I already work in the software industry and with agile methods.This time it was really pleasing working with this group as Rick was doing a great job keeping everyone on track and I could feel that everyone was making efforts to be proactive. In addition it is one of the projects where I felt that everyone contributed significantly. The teamwork was really good and no hiccups as no one was against working when something was urgently required which was really nice. One point of improvement could be punctuality in meetings and consistency (Mainly lacking on my side) but we found ways around it when someone wasn't available. In retrospect this was a really good group !

I worked initially on the go to definition feature which then turned into implementing the language server. I also helped with debugging as I have experience with nodeJS and TypeScript in general. After implementing the language server I also worked on implementing pre-set keybinds for the extension and I also implemented auto completion in JAVA and PVL files using the language server. I usually was consulted by other teammates when something was not working within their app concerning Node or packages or when there was an issue with launching the extension.

8.3 Process

Very early on, we settled on working with an agile methodology, following some scrum practices with weekly meetings between the team and our supervisor/client. This workflow seemed the most suitable for the project, as the initial requirements were somewhat vague, so having continuous feedback from the supervisor and client was important in ensuring that the project is a success. It also helped us as a team stay up to date with what has been done over the past week - many of us had other responsibilities outside of the scope of the project so having a concrete weekly meeting to catch up on anything you might've missed was quite nice.

To establish a good workflow, we also followed many version control practices through GitHub. We started by creating issues for each requirement that we initially had, as well as other things that came up during the project. Each issue was resolved in a separate branch, before creating a pull request and having it reviewed and tested by at least one, but usually two other team members, and finally merged into the main branch, where a working version was always maintained.

As part of the workflow, we loosely divided the project into phases: Requirements gathering and evaluating - in the first week or two, we were still unsure of the specifics of the requirements, how important they were to the client and how feasible it was to implement them. As such, those first two weeks were spent focusing on researching more about the project and how each functionality is typically implemented in other, similar applications.

After that, for almost the entire rest of the project, we were in a continuous phase of implementing features, demonstrating to the client and adapting them when necessary. In the last 2 weeks we focused on finalising our tests and making sure the project can be deployed.

The communication within the team was immaculate - everyone participated and was available when in need. If a member required assistance for their assigned task, we would look at different solutions like who is best suited to help them in the case of a more complex problem, a quick response from whoever is first available if its something simpler, or reassigning a different person to the feature altogether if we decide it will benefit us all. Everyone stayed up to date with almost all features of the project.

8.4 Planning

As aforementioned, we decided to work with scrum, and early on set on two-week sprints. We figured this is a good amount of time to be able to knock down features and have somewhat of a demo for the client at the end of every sprint. If we look at the initial planning we made in the project proposal (Appendix C), we divided the tasks each sprint into the three main goals we set at the start of our project, extending IDE, Integrating VerCors and Visualising Output. We found that this division helped us with keeping a clear overview on what has to be done, and ultimately for what goal it is implemented as well as for dividing the tasks among the team. The requirements were ordered in a way such that requirements which depended on others were planned later on in the project.

In hindsight, on some parts we were a bit naive and lacked a more thorough understanding of how VSCode extensions completely worked, this caused us to wrongfully estimate the time of some requirements. Especially the requirements concerning language servers were put on the back burner since figuring out how they worked took more time than expected. Luckily our planning left room for these unforeseen setbacks since we initially left the last sprint open for any overflow of requirements from the other sprints.

Appendix A - Meetings With Client

Meeting on 27th of February

This was the first meeting where we, together with other members of the VerCors team, were discussing the project and discovering requirements. At this point, we did not have a complete view of what the project should be, so we asked the client to briefly explain what the project was about. Sequentially, we walked through all the project goals they listed in their proposal.

Meeting on 5th of March

The second meeting was primarily about discussing our project proposal and finding confirmation in the way we rated the importance of the requirements.

Meeting on 12th of March

This was the first acceptance test meeting where together with the client, we walked through all parts we have worked on in the past couple of weeks.

Meeting on 19th of March

During this weekly meeting, we did not have a lot of things to discuss, but we did have a quick question about where all the possible error messages were saved.

Meeting on 26th of March

For this meeting we arranged a gathering with the majority of the VerCors team, to do a demo of the progress so far.

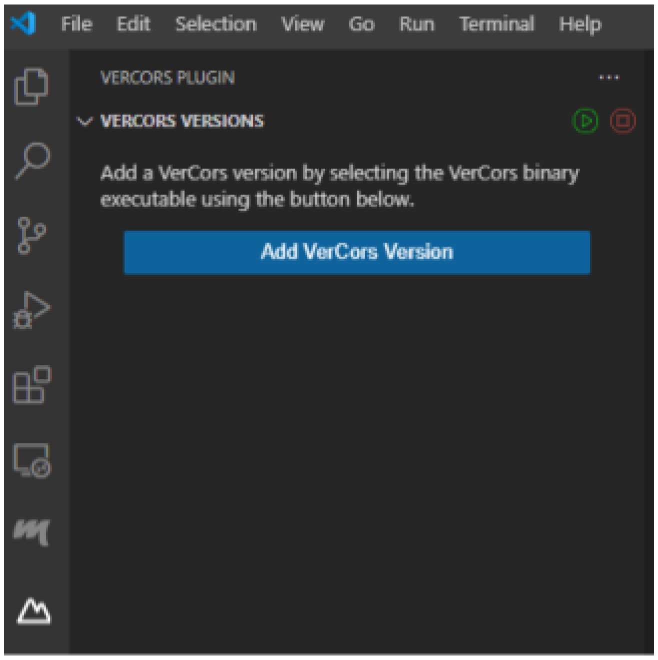
Meeting on 16th of april

This was the last meeting of the project. It was also the first meeting since the demo on 26/3, So we briefly discussed the newer requirements which came up during the demo.

Appendix B - User Manual

Version window

By selecting the VerCors icon in VSCode's Activity Bar (visible in figure 13), the VerCors side bar opens. This window provides several features. Firstly, it allows users to add and remove VerCors versions. Pressing the 'Add VerCors Version' button opens the file browser, allowing users to select the VerCors executable from their installation. This file is often contained in VerCors' bin folder and is named vercors.bat. When a version has correctly been selected, the version appears in the window like in Figure 13. Secondly, located in the top right corner are the start and stop buttons. When having a PVL, java or C file open in the editor, the start button allows users to start the verification of the open file. The start button is also located at the bottom left corner of the screen and can be run using the shortcut `CTRL+SHIFT+L`. The stop button will replace the start button in the bottom left while running and can be run using `CTRL+SHIFT+Q`.

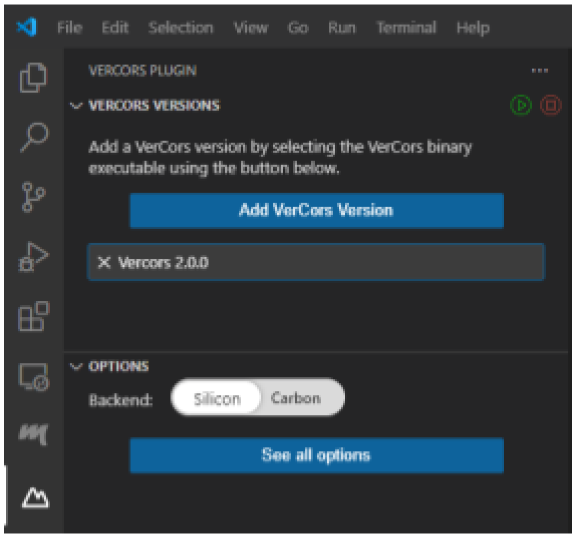


Add a VerCors version by selecting the VerCors binary executable using the button below.

Figure 13: VerCors versions

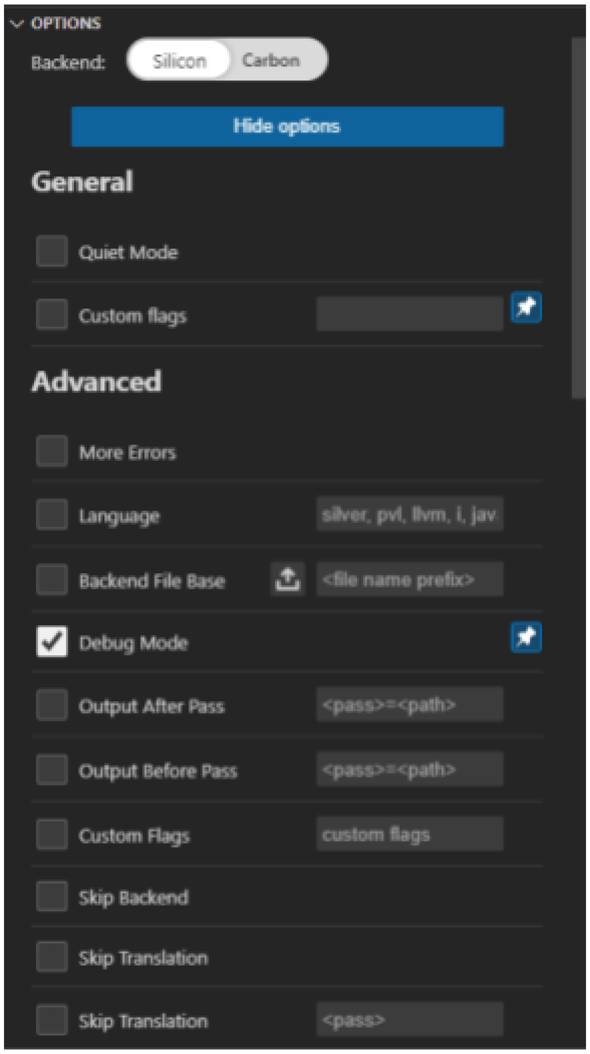
Option window

Figure 14 shows the full VerCors sidebar. The options tab initially only contains the backend option, but after pressing ‘See all options’ all available CLI options are available, as can bee seen in figure 15. Users are able to select options (e.g. debug mode) and pin them (e.g. custom flags and debug mode). The pinned options remain visible when ‘hide options’ at the bottom of the window is pressed. Pins and selected options are file specific, so each file can have a different configuration.



Add a VerCors version by selecting the VerCors binary executable using the button below.

Figure 14: The option window



General

- [ ] Quiet Mode
- [ ] Custom flags

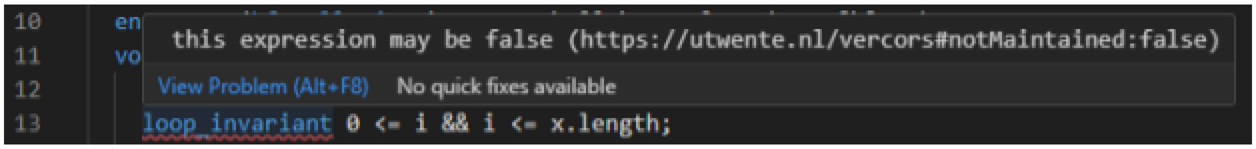
Advanced

- [ ] More Errors
- [ ] Language: silver, pvl, llvm, i, jav
- [ ] Backend File Base: <file name prefix>
- [x] Debug Mode
- [ ] Output After Pass: <pass>=<path>
- [ ] Output Before Pass: <pass>=<path>
- [ ] Custom Flags: custom flags
- [ ] Skip Backend
- [ ] Skip Translation
- [ ] Skip Translation: <pass>

Figure 15: Options window

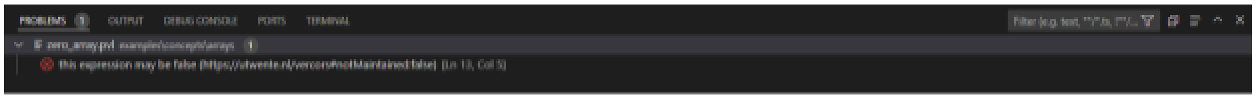
Problems window

After verifying a file, VerCors may have detected errors in the program. These errors appear both in the active file as can be seen in figure 16 and in the problems tab as can be seen in figure 17. Pressing on the error message in the problems tab places the cursor at the location in the file.



```
this expression may be false (https://utwente.nl/vercors#notMaintained:false)
View Problem (Alt+F8)   No quick fixes available
loop_invariant 0 <= i && i <= x.length;
```

Figure 16: Errors in file

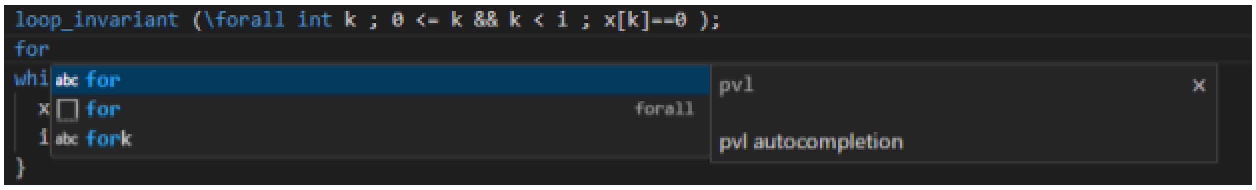


```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   PORTS   TERMINAL
zero_array.pvl examples/concepts/arrays 1
this expression may be false (https://utwente.nl/vercors#notMaintained:false) [Ln 13, Col 5]
```

Figure 17: Error window

Snippets, highlighting and autocomplete

Figure 18 shows a small part of a PVL program. Keywords like loop_invariant and int are automatically highlighted. While typing, autocomplete suggestions and snippets will appear. Snippets appear as squares and can be selected by pressing tab after selecting it with the arrows.



```
loop_invariant (\forall int k ; 0 <= k && k < i ; x[k]==0 );
for
```

Figure 18: Snippets and autocomplete

Installing from marketplace

Installing the plugin from the marketplace is currently not available. To publish an extension on the VSCode market, it must abide by a number of rules, mostly due to security. One of the requirements is to not have SVGs in the project[4], which we unfortunately have multiple of. We only found out about this at the end of the project when looking into publishing it, and as such we haven't had time to replace these SVGs. It is an involved process of reading the new image files from the backend and sending them to the frontend, as you cannot reference them directly from the HTML views.

[4] https://code.visualstudio.com/api/working-with-extensions/publishing-extension

Despite this, the `npm run build` command will create a *.vsix* file, which is used to install the plugin. The *.vsix* file is also what's published on the market, so most of the preparation for publishing has been done.

Another reason we decided to not publish the extension is because it requires a token from a Azure DevOps account and we assumed VerCors does not want their plugin to be owned by one of our accounts.

Developer manual

When you clone the repository from github and make your amazing new changes, running it immediately would result in an erroneous mess. First, some small installations have to be done.

Running

You start by running `npm install` to install all used packages. Afterwards, you run `npm run compile` to compile the project. When this is done, the extension can be executed. The extension can also be run from VS Code "Debug and Run" tab where debugging can be started (`F5`). Running this will automatically compile and run the plugin.

Building

The plugin can be built using `npm run build`. This will bundle the code and create a packaged *.vsix* file, which can be installed into VS Code using:

```
code --install-extension vercorsplugin-0.0.1.vsix
```

Testing

The tests need a working installation of VerCors. VerCors has to be installed or linked using a symlink to a folder with the path `client/src/test/fakeVercors` as shown in Figure 19. To create a symlink, for example:

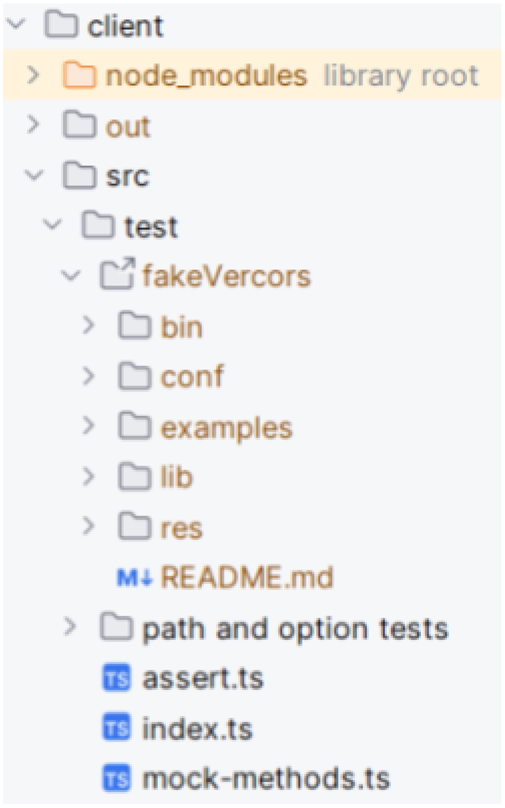
Windows (cmd.exe or prepend with `cmd /c` in PowerShell):

```
mklink /D ".../client/src/test/fakeVercors" ".../vercors"
```

Unix:

```
ln -s ".../vercors" ".../client/src/test/fakeVercors"
```

Manually changing the VerCors installation location for testing is also possible. Iit can be done by changing the variable vercorsPath in `client/src/test/mock-methods.ts`.



```
client
  node_modules  library root
  out
  src
    test
      fakeVercors
        bin
        conf
        examples
        lib
        res
        README.md
      path and option tests
      assert.ts
      index.ts
      mock-methods.ts
```

Figure 19: “fakeVercors” location for testing

Appendix C - Initial Planning

The VerCors plugin project is large. That is why we have divided the project into several tracks, as mentioned previously: Extending the IDE, integrating the VerCors tool in VSC and visualising the tool's output.

We have decided to use an agile methodology, and more specifically scrum, to develop the project, as there will be weekly meetings with the client and we'll be able to respond to feedback appropriately. Sprints will last two weeks each, with the exception of sprint 4, which will be three weeks due to the extended quarter duration. The following table shows a rough estimation of what tasks we want to have done by then.

	Sprint 1 (26/02 to 08/03)	Sprint 2 (11/03 to 22/03)	Sprint 3 and 4 (25/03 to 26/04)
Extending IDE	Syntax highlighting PVL Adding go-to definition	Refactoring PVL code Code generation for templates	Adding VerCors shortcuts Code suggestion
Integrating VerCors	Verify Java and PVL files Checkboxes for command line options Stop option for running process	Verify c, OpenMP, and OpenCL files Keeping track of file configurations	Choose VerCors version
Visualising Output	Error window in plugin screen	Link errors to the places in the code	Progress bar VerCors progress status

Sprint 1

In the first sprint we are planning to incorporate the feedback on the current project proposal, and if needed tweak the planning a bit so that new goals are added or unnecessary goals are shut down early. We plan to start by extending the IDE and integrating the VerCors tool into the IDE. The main focus for the IDE part will be adding the IDE features syntax highlighting and adding a go-to definition. To Integrate the VerCors tool in VSC we add the functionalities to verify both PVL files and Java files. We will also add checkboxes for all command line options, and add a stop option to terminate any running VerCors processes. For the visualising output part of the project, we cannot do as much since the tool is not fully integrated yet. We will start with making a designated window in the plugin screen where the errors should show up.

Sprint 2

After a good foundation is made for the whole plugin in the first sprint we can build upon this by adding options for code generation and refactoring for the IDE part. For the integration part, we plan to add support for the other languages that work with VerCors. Also to make the system that keeps track of the run configuration for the tool for each file. Now we are a bit further with the integration part, we can now explore linking the tool's verification results to the pl

Sprint 3 & 4

As scrum is an iterative process, it is hard to concretely plan what we will be working on that far down the line besides the leftover features. As such, it's mostly left as to-be-determined.

Appendix D - Snippets List

For all snippets, if the prefix is typed the fitting code structure will be generated, so called snippets. If a snippet is generated the user can fill in the predefined blank spots and with each press of the tab button, the curson moves to the next spot. In the list below these cursor spots are noted with a dollar sign ($) followed by a number, which stands for the order in which the user fills in the blanks.

Prefix	**Snippet**
`for`	`(\forall int ${1:i} ; ${2:0} <= ${1:i} && ${1:i} < $3);`
`perm`	`Perm(this.$1, $2);`
`old`	`\old($1)`
`res`	`resource $1() = $2;`
`pure`	`pure $1($2) = $3;`
`\un`	`\unfolding $1 \in $2;`
`pointer`	`\pointer($1, $2, $3);`