DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Isabelle/VSCode: Editor Improvements and Prover IDE integrations

Denis Paluca

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Isabelle/VSCode: Editor Improvements and Prover IDE integrations

Isabelle/VSCode: Editor Verbesserungen und Integration der Prover IDE

Author:	Denis Paluca
Supervisor:	Prof. Tobias Nipkow
Advisor:	M.Sc. Fabian Huch
Submission Date:	15.08.2021

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.08.2021 Denis Paluca

Acknowledgments

I would like to thank the Chair of Logic and Verification and Prof. Nipkow for giving me the opportunity to do this thesis. I would like to express my gratitude to my advisor, Fabian Huch, for all his guidance during the past four months.

Abstract

In the past few years, Visual Studio Code has risen tremendously in popularity. Therefore, many development tools and programming languages seek to provide extensions for it. For the interactive theorem prover Isabelle, an extension has already been introduced. But, this extension is still not on par with Isabelle/jEdit, the default choice for users when it comes to working with Isabelle. This is due to issues with mathematical symbols, input methods, and partially missing markup. In this thesis, we improved Isabelle/VSCode by reworking it to mend the above-mentioned issues. To do this, we implemented a new file system for the extension, added support for abbreviations and auto-completion, and added syntax highlighting for panels. Evaluation of the extension before and after the changes shows clearly that performance has been improved. Now, users do not have to wait 20 seconds everytime they open a new theory file. Based on this, the extension has been brought closer to being a valid alternative to Isabelle/jEdit.

Contents

1 Introduction

1.1 Motivation

For the interactive theorem prover *Isabelle* [1], the *jEdit IDE* [2] has been used as the basis to build the *Isabelle/jEdit prover IDE* [3]. To provide the same functionality in a more modern and popular code editor, the Isabelle extension for *VSCode* [4] (short for Visual Studio Code) was introduced. Although a good portion of the features supported in jEdit are now available for VSCode, the extension has not seen much adoption from users. In its current state, *Isabelle/VSCode* [5] is no match for its jEdit counterpart. This is due to many issues which have cropped up with the extension. The issues that mainly stop it from being a viable alternative are:

- Performance issues related to the rendering of mathematical symbols.
- Ergonomic issues while working with mathematical symbols.
- Missing input options for mathematical symbols, which are present in jEdit.
- Missing syntax highlighting for different semantic editor perspectives.

The main goal of this project is to alleviate these issues so that Isabelle/VSCode becomes an attractive option to users. In order to achieve this, the focus has been set on the following objectives:

- Add first-class support for Unicode characters. This should remove the performance and ergonomic issues with the mathematical symbols.
- Provide abbreviation and auto-completion support for the extension.
- Add the missing syntax highlighting for the semantic editor perspectives.

Of course, some features, which are available in jEdit, would still be missing. Nonetheless, this project should be a major stride towards wider user adoption for Isabelle/VSCode.

1.2 Outline

Chapter 2 starts the thesis with a short rundown of preliminary information about Isabelle, VSCode, and the Language Server Protocol.

In Chapter 3, we describe the implementation details of the project. We start with adding the syntax highlighting in Section 3.1. Then, in Section 3.2, we deal with the performance

issues related to the mathematical symbols. In the last section of the chapter, we add new ways for a user to input mathematical symbols.

In Chapter 4, we evaluate the effect of our changes on the performance of the extension. We also compare the performance of the existing solution and the new solution.

Chapter 5 concludes the thesis with a summary of the goals reached during the course of the thesis and some closing remarks on the project.

Chapter 6 describes new features which can be added to Isabelle/VSCode by future projects. Moreover, it lists other code editors for which the work done here may become relevant.

2 Background

2.1 Isabelle

Isabelle is a generic interactive theorem prover, which is mostly used for higher-order logic. Paulson introduced Isabelle in 1986 [1], the project is now being maintained and further developed by the Technical University of Munich, the University of Cambridge and countless independent contributors, the most active contributor being Makarius Wenzel. Isabelle, itself, is programmed and can be extended in Standard ML and Scala. A considerable number of theorems from mathematics and computer science have been formalized in Isabelle, many of which are stored in the *Archive of Formal Proofs*. As of 2021, the Archive of Formal Proofs contains 602 Articles from almost 400 Authors [6].

Isabelle/Isar

Isar (Intelligible semi-automated reasoning) is the formal proof language offered by Isabelle. It is inspired by the *Mizar* system. Unlike Mizar, Isar is immediately 'executable', through the Isar/VM interpreter. Interactive theory and proof development is directly supported in Isar. It makes possible writing naturally understandable proofs, although this does take skill and effort from the writer [7].

Isabelle/Scala

Scala is a statically typed programming language [8] and it is used to develop the systems around the Isabelle environment. It enables services and systems from outside to access Isabelle. *PIDE* (Prover IDE) is one of the most important services based on Isabelle/Scala and it is Isabelle's general framework for Prover IDEs. It works as a back-end for different source code editors and IDEs, by asynchronously processing source documents and providing support for editing, markup and other language features. The document model is central to the PIDE architecture, with source files being managed by PIDE and the physical file-system only playing a subordinate role [3].

Isabelle/jEdit

Isabelle/jEdit is the main front-end of Isabelle/PIDE. It is based on the original *jEdit*, which is an open source "programmer's text editor" written in Java [2]. jEdit can be extended by plugins written in languages that run on the *JVM*. Isabelle/jEdit is a slightly modified version of jEdit with an added plugin for Isabelle. Most of the language features, which are

common in modern code editors, are supported by Isabelle/jEdit for Isabelle specifically. In Isabelle/jEdit the following language edit modes are supported, as shown in Table 2.1:

edit mode	file name	content
isabelle	*.thy	Theory source
isabelle-ml	*.ML	Isabelle/ML source
sml	*.sml or *.sig	Standard ML source
isabelle-root	ROOT	Isabelle session root
isabelle-options		Isabelle options
isabelle-news		Isabelle NEWS

Table 2.1: Language edit modes supported in Isabelle/jEdit [2].

2.2 VSCode and VSCode Extensions

VSCode is a source-code editor developed by Microsoft [4]. It is a commercial distribution of the open source project *Code - OSS* which is also developed and maintained by Microsoft [9]. Since its initial release in April 2015, VSCode has managed to gain a great deal of popularity amongst developers, as illustrated in Figure 2.1.

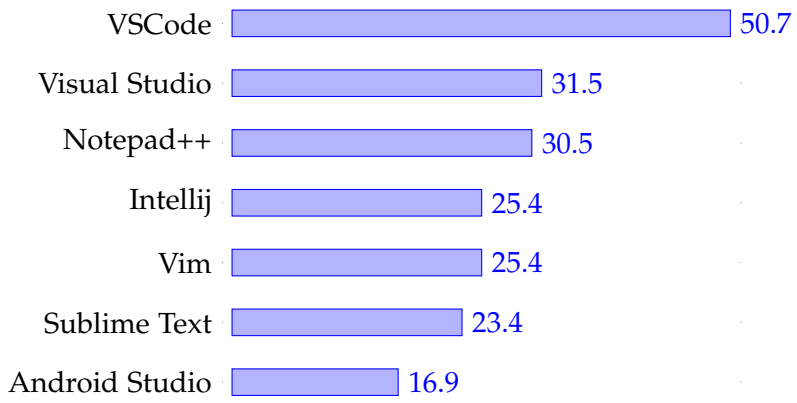


Figure 2.1: Percentage of developers who use the editor, 2019 survey [10].

VSCode is based on the *Electron* framework, which makes it easy to be ported in multiple platforms. Because it is an Electron based application, VSCode makes use of mainly web technologies.

VSCode functionality can be added via extensions, which can either be published to the VSCode Extension Marketplace or be packaged into the VSIX format and shared with other users. To develop extensions VSCode offers its *Extension API* [11], which is fairly powerful for most use cases. Many core features of VSCode are built as extensions and use the same Extension API, for example the default git integration [12].

Extensions can be developed with *TypeScript* or *JavaScript*. TypeScript being the preferred language, since most of the guides and examples are written in TypeScript. For packaging, publishing and managing extensions, VSCode offers its own command-line tool "vsce" (short for Visual Studio Code Extensions), available on *npm* [13].

In VSCode, support for programmatic language features (i.e. auto completion, error checking, jump to definition, etc) is provided through the *Language Server Protocol*.

2.3 Language Server Protocol

The Language Server Protocol was originally developed from Microsoft in collaboration with Red Hat and CodeEnvy specifically for VSCode. Now it is an open standard supported by many code editors.

Modern source code editors and IDEs support a wide range of sophisticated programmatic language features. Compilers and interpreters normally cannot provide these language features, since they are meant to work with well-formed source code. Language Servers are specifically developed to deal with these issues. They evaluate the syntactic and semantic outcomes from source code modifications and give instant feedback to the user.

Microsoft developed the Language Server Protocol to solve three common issues with Language Servers [14].

1. Language Servers are usually implemented in their native programming languages, and that presents a challenge in integrating them with VSCode, which has a Node.js runtime.

2. Language Servers can be resource intensive. For example, to correctly validate a file, a Language Server may need to parse a large amount of files, build up Abstract Syntax Trees for them and perform static program analysis. Those operations could incur significant CPU/memory usage and VSCode's performance must remain unaffected.

3. Integrating multiple language servers with multiple code editors could involve significant effort. This makes implementing language support for M languages in N code editors the work of M * N.

By standardizing the communication between a Language Server and a Language Client (source code editors/IDEs), the Language Server Protocol allows one Language Server to be reused in multiple editors. Figure 2.2 is an illustration of how this works.

The language server protocol defines a set of JSON-RPC request, response and notification messages which are exchanged. The base protocol consists of a header and a content part (comparable to HTTP). The header and content part are separated by a `\r\n`. The only supported header fields are *Content-Length* and *Content-Type*. The header is encoded using the ASCII encoding, while the content part is encoded in UTF-8. The base protocol uses a convention such that the parameters passed to request/notification messages should be of object type (if passed at all). Every processed request must send a response back to the

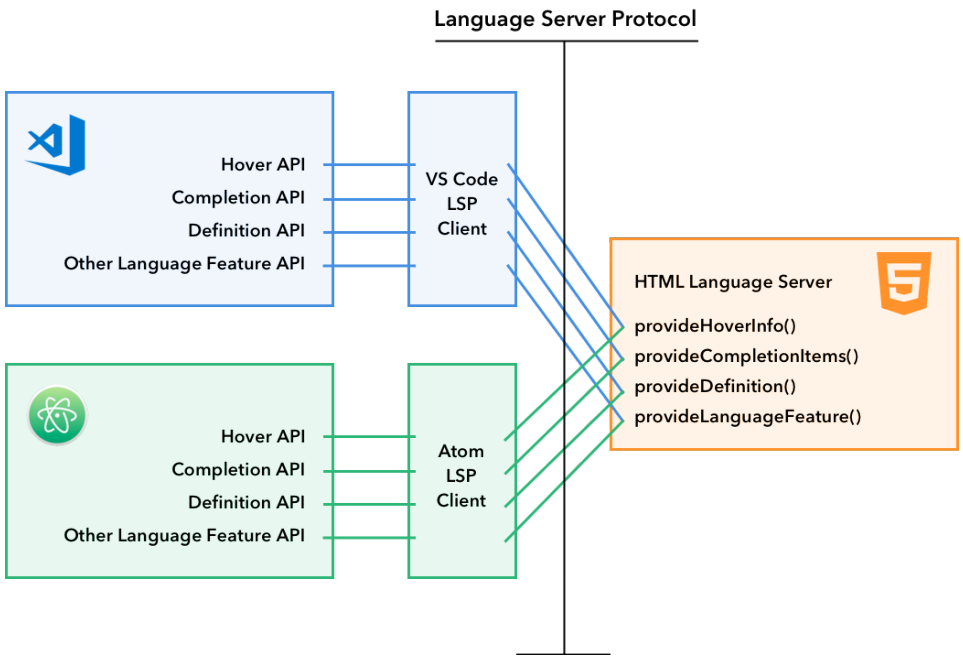


Figure 2.2: Diagram showcasing how a language server interacts with multiple editors [15].

sender of the request. If a request doesn't provide a value, the receiver of a request still needs to return a response message to conform to the JSON-RPC specification. A processed notification message must not send a response back [16].

The protocol currently assumes that one server instance serves one tool. There is currently no support in the protocol to share one server between different tools. Such a sharing would require additional protocol e.g. to lock a document to support concurrent editing.

2.4 Isabelle/VSCode

The rising popularity of VSCode also caught the attention of the prover community. Theorem provers such as *Coq* [17] and *Lean* [18] started supporting their own extensions for VSCode, namely *VSCoq* [19] and *vscode-lean* [20], as early as 2016. The Isabelle project shortly followed suit, with Wenzel developing Isabelle/VSCode [21].

The Isabelle/VSCode project consists of its VSCode extension and its server, which is instantiated through the Isabelle command-line tool. The extension and server communicate as specified by the Language Server Protocol, with some additional features/endpoints to support the idiosyncrasies of Isabelle and VSCode.

The server is an application written in Isabelle/Scala, with most of the functionality used directly from Isabelle/PIDE, but adapted to the Language Server Protocol endpoints. Communication between the client and the server happen mostly through notifications, aside from some a few endpoints which are explicitly set by the language server protocol as request/response endpoints.

3 Isabelle/VSCode: Editor Improvements and Prover IDE integrations

3.1 Syntax highlighting in state panel

This feature was used as starting point to get to know the codebase. The *State Panel* is one of the features which was ported from jEdit by Wenzel when the Isabelle/VSCode extension was first developed. The main purpose of the state panel is to show the internal proof state. The user can update the proof state by either clicking the Update State button or by enabling Auto Update, which then updates the proof state according to the current cursor position in the editor. Being able to preview the proof state is crucial to having an overall better user experience.

The jEdit state panel has syntax highlighting, which matches that of the code editor, shown in Figure 3.1. On the other hand the VSCode state panel has no syntax highlighting and it is also missing clickable items, as shown in Figure 3.2. This is problematic for user who are accustomed to working with these features.

Figure 3.1: State panel in jEdit, with the state taken at the current cursor position

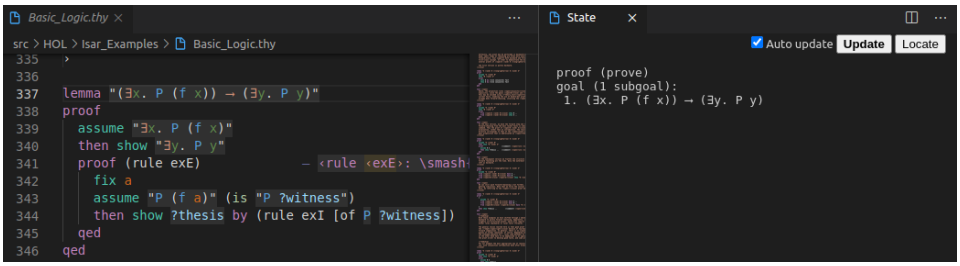


Figure 3.2: VSCode state panel missing syntax highlighting and clickable items.

Current implementation. In VSCode, the state panel is implemented as a *Webview*, which opens up as a column separately from the active editor. A message is then sent to the language server to show the proof state for the active document for the line in which the cursor is. The language server performs a query to get the state and then replies with a HTML document which is inserted directly into the Webview, in the frontend.

Solutions In the current implementation, the server sends to the state panel HTML which has the proper markup to highlight its body, but the body is inserted without markup. The most obvious solution would be to just insert the body with the markup, as shown in Figure 3.3.

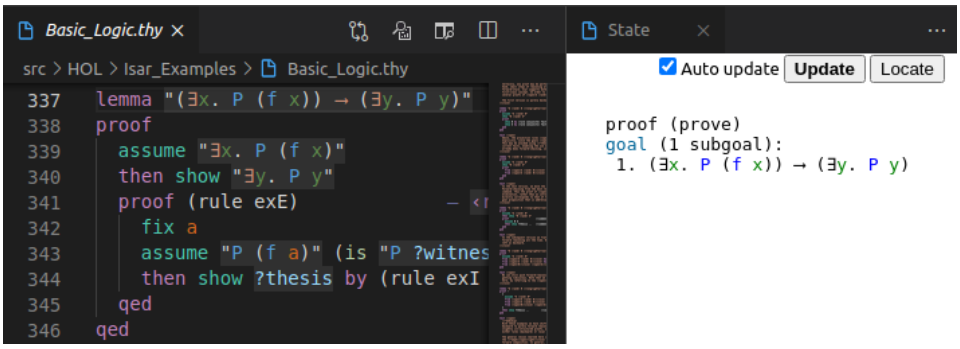


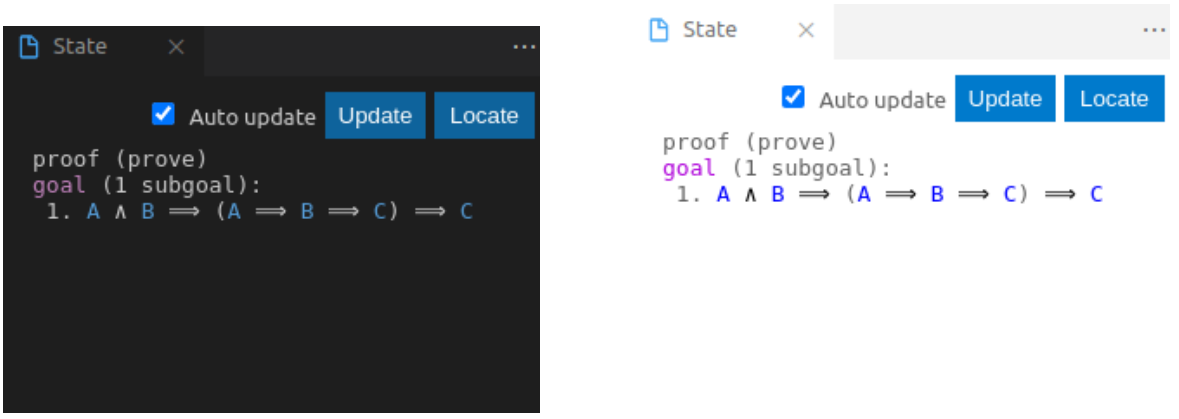
Figure 3.3: VSCode state panel with proper markup

Even though this is an easy solution and aligns perfectly with how it behaves in jEdit, it is clear that that it does not fit VSCode. The syntax highlighting does not match that of VSCode and most VSCode extensions require support for the dark theme because it is the default and by far the most popular theme. Another problem with this solution is that we build the whole HTML in the backend, which is not optimal since we have a frontend and our GUI elements seem out of place in VSCode.

The second approach would be to build the HTML for the Webview in the frontend and only ask the backend for the marked-up body. The marked-up body is obtained using the Isabelle HTML presentation elements. Then, the syntax highlighting, which is configured in the settings, is applied to marked-up body. Since the controls are now in the frontend, the state of the auto update checkbox also needs to be correctly set. Now, every update of the state panel comes with the `id`, the `auto_update boolean`, and the `body` as a `string`.

The *Output Panel*, which had the same issue, is also given the same treatment as the state panel. The only difference being that the output panel doesn't have controls (Update, Auto-Update, Locate) and is connected with the back-end through another endpoint.

Results The state panel has now syntax highlighting and clickable items. Aside from that it also matches the syntax highlighting to the currently active theme, as shown in Figure 3.4. An advantage of this implementation is that it has a higher degree of decoupling between the backend and the frontend. Also the styling of the GUI elements is consistent.



(a) State panel in dark theme.

(b) State panel in light theme.

Figure 3.4: VSCode state panel with theme appropriate markup.

3.2 Performance issues related to mathematical symbols

Since Isabelle is an automated theorem prover, its theory documents require mathematical notation. These notations and other useful symbols are saved in the form of latex-like notations, which are then rendered as Unicode symbols in jEdit and VSCode. For example, the existential quantifier ∃ is represented in text form by `\<exists>`. This process makes it easier for people who are used to mathematical notations, such as mathematicians and computer scientists, to read and work on Isabelle documents.

In VSCode, the replacement of the notations is only done visually, which has proven to be quite problematic. This has introduced a number of performance and ergonomic issues for the users, which completely stops them from using Isabelle/VSCode:

- For it to work, a third-party extension needs to be configured, which takes multiple steps in setting up your development environment.
- Every time a new document is opened, the editor becomes completely unresponsive while it is replacing the symbols.
- Inserting, deleting, selecting, replacing and navigating through the symbols is inconvenient for the user.

Based on anecdotal evidence, these and other issues make working with Isabelle on VSCode unattractive in comparison to its jEdit counterpart.

Current Implementation. When the language client is started, a request is sent to the language server asking for the symbol data (name, unicode, tag). This data is then processed by the client and a third-party extension, Prettify Symbols Mode, is started. The symbol replacements are then registered which allow the third-party extension to start replacing.

Each Isabelle or Isabelle/ML file that is opened will have its latex notations replaced with the respective Unicode symbol. This process is illustrated below in Figure 3.5.

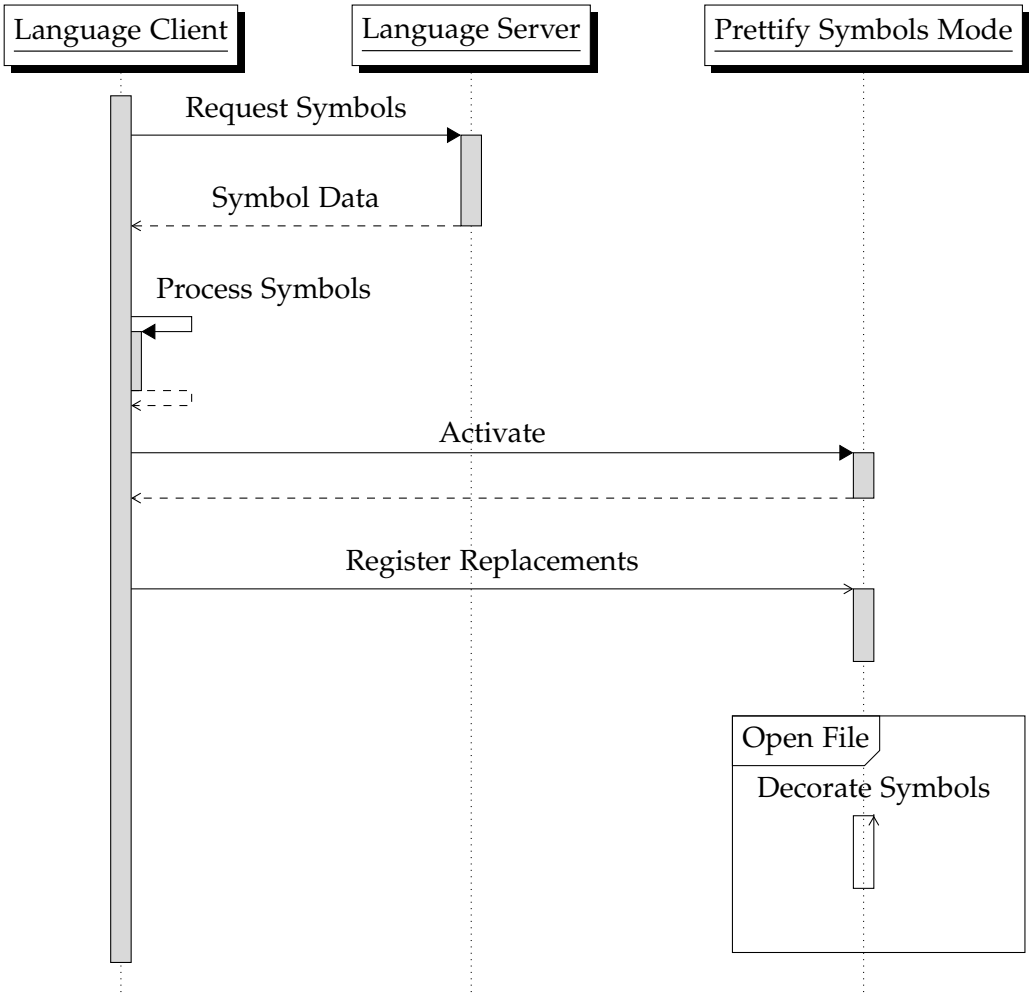


Figure 3.5: Sequence diagram representing the steps for symbol rendering with Prettify Symbols Mode.

Internally, Prettify Symbols Mode finds all the ranges in the text document with the latex notations. It manages to hide the text inside these ranges by applying CSS decorations such as making the font size really small or setting letter spacing to a negative value. Then using the CSS pseudo-element `after`, it shows the corresponding Unicode symbol after the hidden text.

Solutions. The VS Code Extension API provides a number of different endpoints to tackle this problem, but most of them are not suitable to solve it without using them in unintended ways. The following VSCode interfaces were considered:

- **Decorations.** The decorations API is used to apply CSS to different text sections. This

is the API that Prettify Symbols Mode uses to solve the symbols problem. The creator of this extension, Bell, was also trying to solve the same problem but with a different theorem prover, namely the COQ Proof Assistant. He stopped updating the extension around 2017, and since then the decorations API was not updated in any meaningful way [22]. That's why trying to use this API would be futile, since in the end it would not solve our problems, similar to Prettify Symbols Mode.

- **Virtual Document.** This API allows one to create virtual documents from sources which can then be displayed in the editor. But, these documents are unfortunately read only.
- **Custom Editor.** The custom editor could have been a suitable solution since it allows the user to manipulate the byte stream and display it to the user how you see fit. The problem with it is that we would have to implement the whole editor in HTML, which would require a lot of effort and would break the functionality of other extensions.
- **Custom Notebook.** Notebooks are more powerful versions of the custom editors. However, custom notebooks are still under development so it would be to risky to use them and they would most probably not fix the issue.
- **File System Provider.** This API is meant to enable extensions to serve files from remote places, like ftp-servers, and to seamlessly integrate those into the editor. It allows extensions to provide a file system for schemes other than "file:". Since with it we can define all the useful operations, such as `read`, `write`, etc, it was the most suitable candidate for us. The only downside is that we would have to generate an Isabelle view on the theory files and store them in memory, parallel to the original disc files.

Other potential approaches:

- **Character Encoding.** Isabelle/jEdit uses encoding to solve the symbols problem. It provides its own charset UTF-8-Isabelle, where all the latex notations are encoded to the correct symbols. In VSCode, although you can set the encoding of the document, you cannot provide your own charset.
- **Font Ligatures.** Normally, a ligature occurs when two or more letters are joined as a single glyph. An example would be when the characters a and e are joined together to form the glyph æ. We briefly considered building our own font with custom ligatures mapped to our latex notations. This approach has the same pitfalls as Prettify Symbols Mode, for example the `search and replace` function would also affect the text inside our ligatures.

The Isabelle File System

To implement our file system, we introduced the "isabelle:" scheme. The implementation of the file system consists mainly of 3 modules: *IsabelleFSP*, *SymbolEncoder* and *PrefixTree*. The relation between these modules is illustrated in Figure 3.6.

`IsabelleFSP` (FSP short for *FileSystemProvider*) implements the VSCode `FileSystemProvider` interface. `IsabelleFSP` uses the Singleton design pattern and only makes public static functions such as `register`, to register its single instance as a file system, `initWorkspace`, to setup the workspace, and other less relevant functions.
`SymbolEncoder` encodes the byte streams according to the mapping of latex notations to symbols.
`PrefixTree` stores the search tree for byte sequences and their replacement values, for fast symbol recoding.

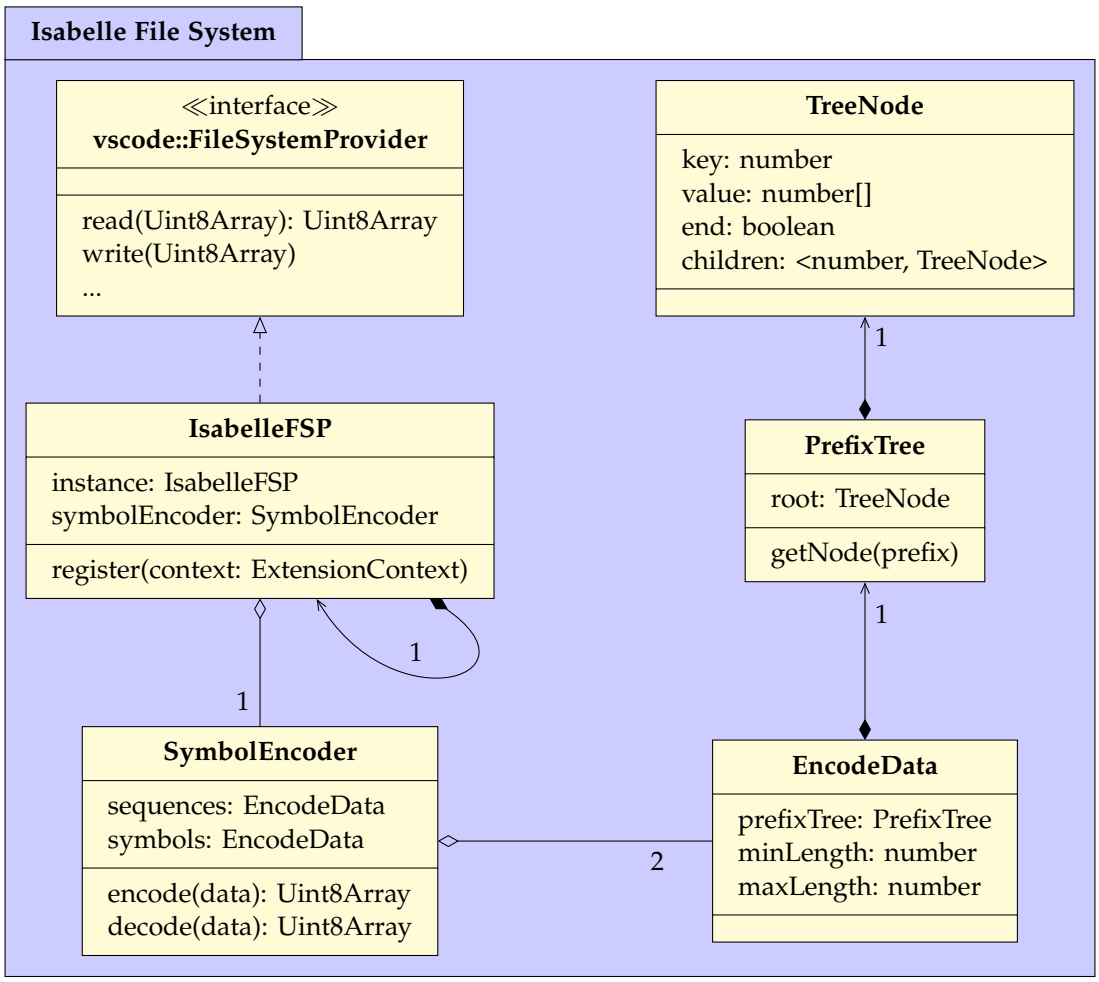


Figure 3.6: Class diagram of Isabelle file system (only important class properties shown for brevity)

When the extension is activated, the `IsabelleFSP` registers its instance to VSCode and it also creates a new workspace folder for the Isabelle memory files. This is done before the server is activated, because if the first workspace folder is added, removed or changed, the currently executing extensions (including the one that called this method) will be terminated

and restarted. If we do this after we get a response from the server, it would cause the server to restart which would lead to an even greater startup time.

`IsabelleFSP`, aside from that, also registers a function that is executed when a document has been opened. This function checks if the document is an Isabelle theory file and then creates a mirroring memory file on the Isabelle scheme. In `IsabelleFSP` a mapping of URIs is saved between the disc files and the memory files. The byte array of the old file is read and then it is encoded by the `SymbolEncoder`. When changes are written to the memory file, then they will be synced with its original file after they are decoded and vice versa.

There is only one instance of the `SymbolEncoder` which is created after we get a response from the server with the symbol data. Using the data the `SymbolEncoder` creates two `EncodeData` objects, namely `symbols` and `sequences`. These objects contain a Prefix Tree, the smallest and largest length of the words. Because in JavaScript, the byte arrays from files are represented in number arrays, it was practical to make a `PrefixTree` which works with the type of number array. Each node on the tree has a number as the `key`, a `children` map, and a `boolean end`, which tells us if it is the last node. If it is an end node, it also has the value of the number array it is supposed to be replaced with. This process is described below in algorithm 1. Inside the `SymbolEncoder`, encoding and decoding work exactly the same way as they just call the `coding` function only with a different `EncodingData` object.

Algorithm 1: Algorithm used for encoding and decoding of byte arrays.

```
Input: Number array A of length l, Object B of type EncodeData
Result: Number array C
for i ← 0 to l do
    word ← [];
    node : TreeNode;
    for j ← i to i + B.maxLength do
        word.push(A[j]);
        node ← B.prefixTree.getNode(word);
        if No node found or node is an end node then
            break;
        end
    end
    if node is defined and is end node then
        C ← C.concat(node.value);
        i ← i + word.length;
    else
        C.push(A[i]);
    end
end
```

A special case of rendering which could not be handled by Unicode was superscript and subscript text. Unicode does not have all characters in these modes. For this reason, this had to be done through CSS decorations, similarly to Prettify Symbols Mode. Since these are only 2 decoration types, one to show the text as superscirpt and one as subscript, it doesn't affect

performance the way Prettify Symbols Mode did.

Now, we have to develop a mechanism for adding files to the Isabelle file system. The first alternative is, as soon as one opens a *.thy file that file is then closed, its encoded version is added to the Isabelle file system, and then its opened/shown to the user. Although this solution is simple in theory, implementing it presents a number of problems:

- The new file has its own URI, with the scheme "isabelle:", which is not accepted by the backend. For the backend to correctly function, it requires the disc file URIs, which use the scheme "file:".
- The new file has no decorations, as a result of being marked as a draft.
- Organizing the files added to our file system in directories, that mirror the directories of the disc file system, clutters our workspace. The user would then have to navigate through multiple directories and subdirectories to find a theory file.

Because the user is now expected to work exclusively with the files on the Isabelle file system, we can completely remove communication between client and server when a normal theory file is opened/modified/etc. Instead, all requests and notifications from "isabelle:" files are now caught by a middleware and then their URI is translated to the URI of the disc file that they represent, before being sent to the server and vice-versa. This procedure works as described below in Figure 3.7. This way the server can process the Isabelle file system files and the client gets back the decorations, highlighting, and all the other features.

We still have the problem of when and which files to add to the Isabelle file system. To do this we can use the formal document model, where theory files belong to a certain Isabelle session. We can mirror the formal model on the Isabelle file system, which would be advantageous for users who are already used to this directory structure. This is also somewhat similar to the theories panel in Isabelle/jEdit.

When the extension is activated, we send to the server the current directory. Usually Isabelle projects have a ROOT or ROOTS file which is a configuration file, to mark the needed theory files for one or multiple sessions. The server then checks the directory for the ROOT file and makes a list of all the session and their theory files. This list is sent to the client which then adds all these files to the Isabelle file system in a directory with the name of the responsible session. When the user opens files that do not belong to any of the the sessions in the ROOT files, these files get added to the Draft session.

At this stage, when the user reopens a previously initialized Isabelle workspace, the workspace needs to wait for the server to start so that it can be repopulated with the appropriate files. This can take a while and the user cannot work on anything during this time. Also the user will lose the arrangement of the previously opened documents, which can be frustrating for the user to work with. To circumvent this, we need to save the state of our workspace and initialize it with this saved state while we wait for the server to start. Instead of saving the contents of the theory files, which can be pretty large, we only need to save the current configuration of our `SymbolsEncoder` and the paths of the files that we need.

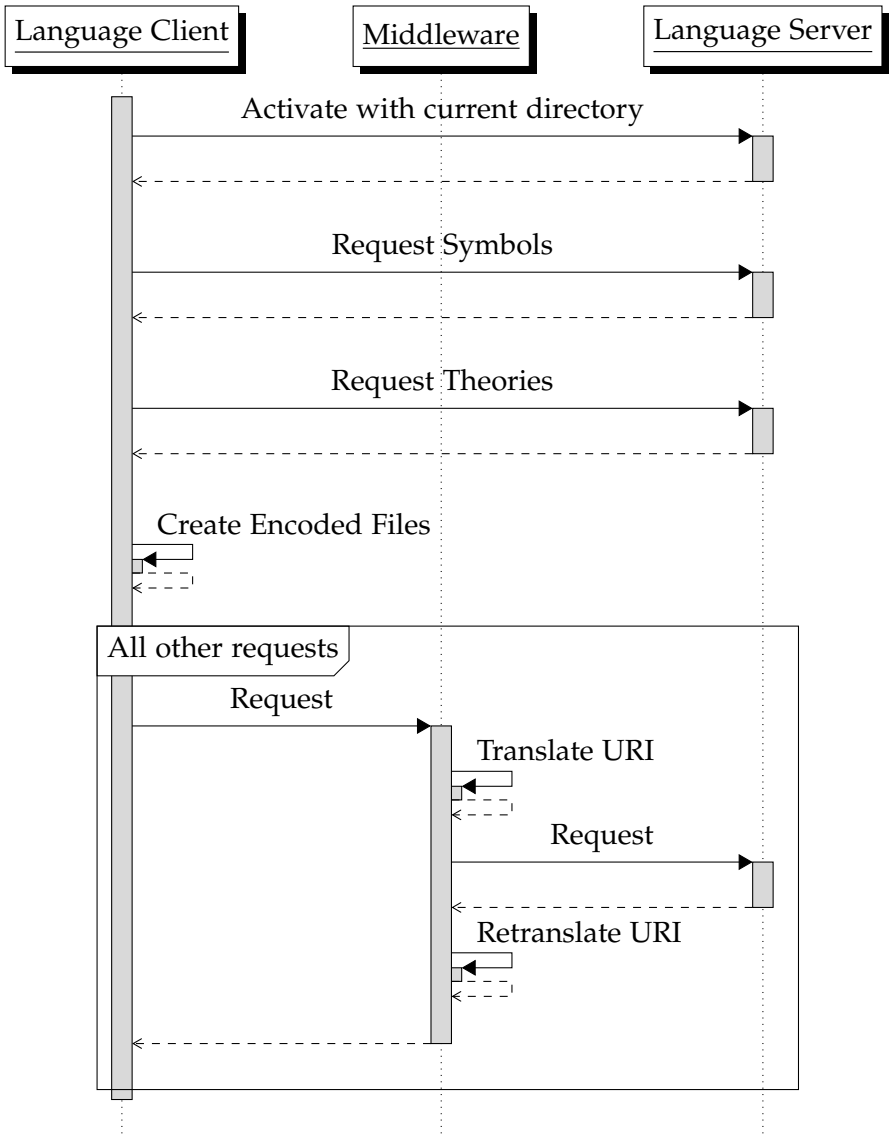


Figure 3.7: Sequence diagram describing the URI translation with the middleware.

Results

The Isabelle experience in VSCode is much closer to jEdit. Symbols can now be added, deleted, edited the same as any other characters. As opposed to Prettify Symbols Mode, where the editor would react unpredictably when manipulating symbols.

The editor freezing problem, which was also caused by Prettify Symbols Mode, is solved. For large files like *HOL/List.thy* or when trying to open imports, the extension would hang noticeably. This didn't happen only on the first time a document is opened but even when changing from one open document to another open document. Now the files are encoded once, when they are added to the Isabelle file system, and after that there are no more computations to be made. So opening or switching between windows is instantaneous.

A slight disadvantage that our solution might have is that it may use more RAM to store all the encoded theory files. Also, some users might be confused by having two workspace folders. The workspace arrangement can be seen in Figure 3.8 and Figure 3.9. To mitigate the second problem we have dialog boxes informing the user in which file system he is operating.

To keep the models consistent we had to implement a two way synchronisation between the theory files and the disc files. This means that changes to the one are reflected on the other and vice-versa. While this works perfectly fine most of the time, it breaks down when the disc file is not in the workspace. In that case only changes from the theory file will be reflected on the disc file, but not the other way around. This is due to the limitations that VSCode sets on the extensions, allowing them to only catch events for files inside the workspace.

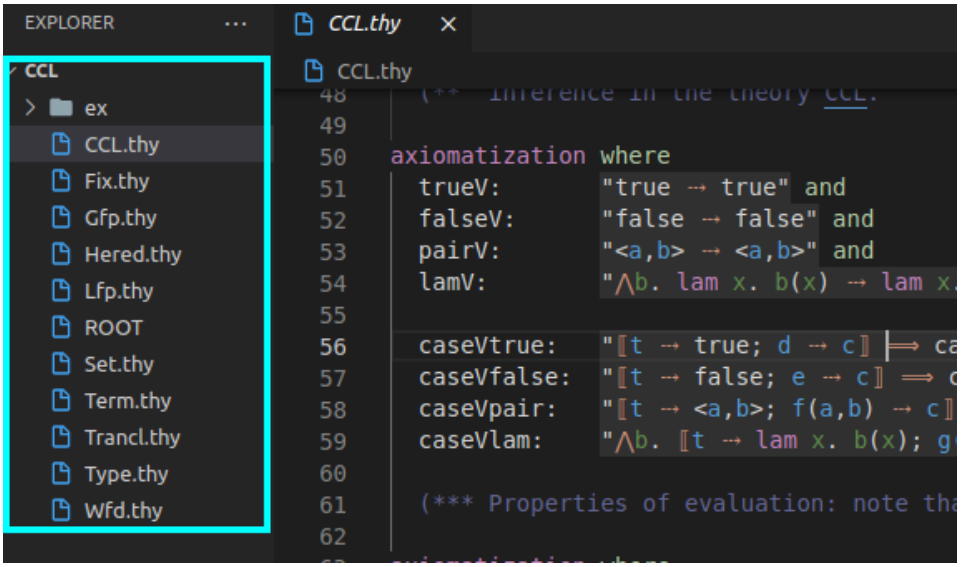


Figure 3.8: VSCode editor with one workspace folder for the "file:" file system.

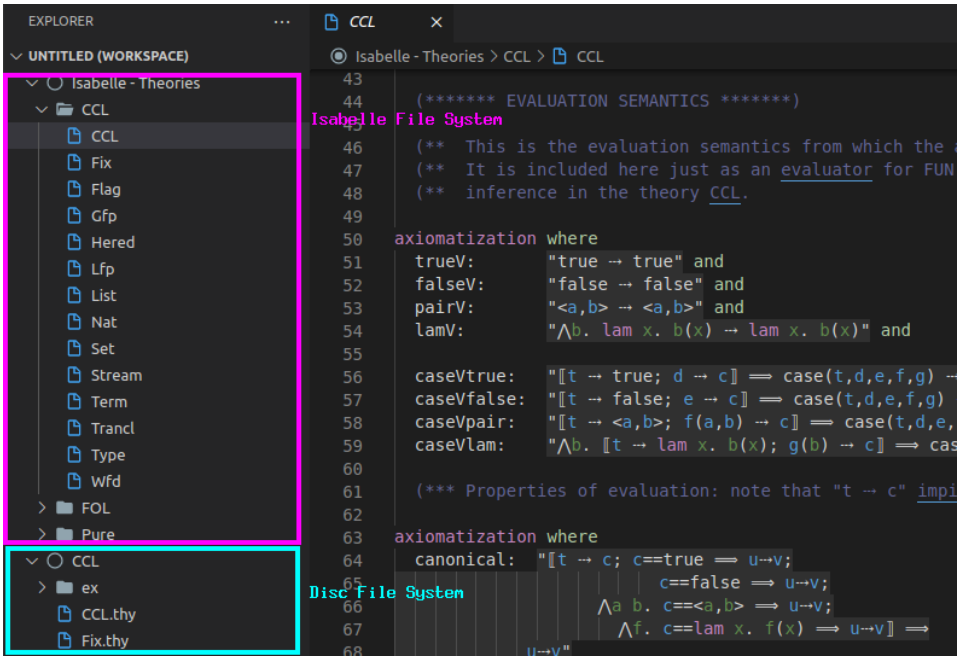


Figure 3.9: VSCode editor with two workspace folder for different file systems ("isabelle:" and "file:").

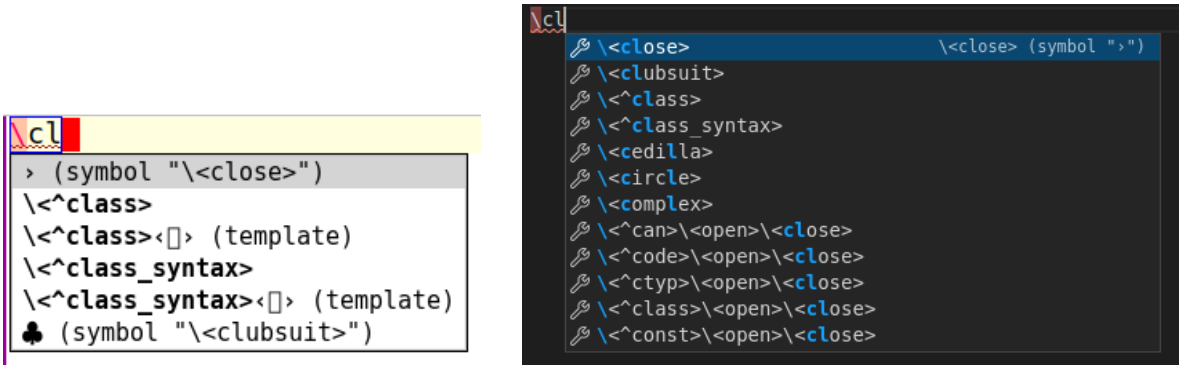
3.3 Auto-completion and Abbreviations

Even though the user can easily manipulate Unicode symbols after we implemented the Isabelle file-system, it is still hard to input them. Most users do not have keyboards made for writing mathematical notation. That is why, we need to define ways in which the user can input such symbols.

There are two ways this is achieved:

- **Auto-completion.** While the user is writing the latex notation for the symbol, a dialog pops up with the matching symbol. When the user selects one of these options the text is then replaced by the respective symbol.
- **Abbreviations.** Instead of writing the whole latex notation for a symbol, the user can use one of the preconfigured abbreviations. This abbreviation gets then replaced with the respective symbol. For example the abbreviation `-->` would get replaced by the symbol $\rightarrow$.

Current Implementation. Both abbreviations and auto-completion are implemented in jEdit. In VSCode abbreviations are not implemented. Auto-completion is already supported but only for completing the latex notation, not replacing it with the appropriate symbol. The difference between the auto-completions is illustrated below in Figure 3.10.



(a) Auto-completion example in jEdit.

(b) Auto-completion example in VSCode.

Figure 3.10: Example comparison of auto-completion between jEdit and VSCode.

Solutions. The auto-completion can be easily configured to replace the latex notation with the symbol. The notation should still be in the label of the completion, or else it wouldn't be searchable.

The abbreviations need to be implemented in the front-end, for this reason we first need to extend the symbols endpoint to include the abbreviations for each symbol. Many of these abbreviations are not unique, i.e. they are not assigned to just one symbol. For example << is an abbreviation for ⟨ and «. These abbreviations will not be automatically replaced, but the auto-completion will show the different options as suggestions. Single character abbreviations are also handled by the auto-completion. Only unique abbreviations will be automatically replaced.

To make it more customizable, we let the user choose in the extension settings when abbreviations will be replaced. With the options being:

- **none** Replacements are deactivated. No replacements are done.
- **non-alpha** Replaces all unique abbreviations that contain no alpha-numeric characters. (Abbrevitions such as `Un` for ∪ will not be replaced.) This is the default setting.
- **all** Replaces all unique abbreviations.

After we have identified all unique abbreviations, we save them all these in a prefix tree so that they are efficiently searchable. We will use the same prefix tree class that we used in the Isabelle file system. The type of the prefix tree needs to be changed to accept not only number arrays but also strings. The key for the tree nodes will also be `number` or `string`, because there is no `char` type in TypeScript. Abbreviations that are prefixes of other abbreviations are saved with a space in the end.

We also register a function that triggers when a text document is changed (i.e. user typing). It checks starting from the last typed character, backwards, if the text written is an abbreviation. For this reason we save the abbreviations in the prefix tree reversed. The text being typed is then replaced with the longest matching abbreviation. To also support

multi-cursor abbreviation replacement, we start from the last edited part of the text. Doing this helps with keeping the ranges consistent for all different parts of the text document that were changed.

One problem that emerges from using this method is that using Undo after an abbreviation has been replaced doesn't work. When the user undoes an abbreviation replacement, a `TextChange` event is emitted which in turn triggers our function that ends up replacing the abbreviation again. This is solved by not triggering our function for `TextChange` events that are replacing one character. These kind of events are the undo events, where the user replaces the symbol with the abbreviation text.

Results. The input of Unicode symbols has been highly facilitated. Now the user can input symbols through auto-completion of latex notations and through abbreviations, as shown below in Figure 3.11 and Figure 3.12. Abbreviation replacement is almost instantaneous and the auto-completion suggestion box is also relatively fast. The user can also choose through the settings what level of abbreviation replacement he wants.

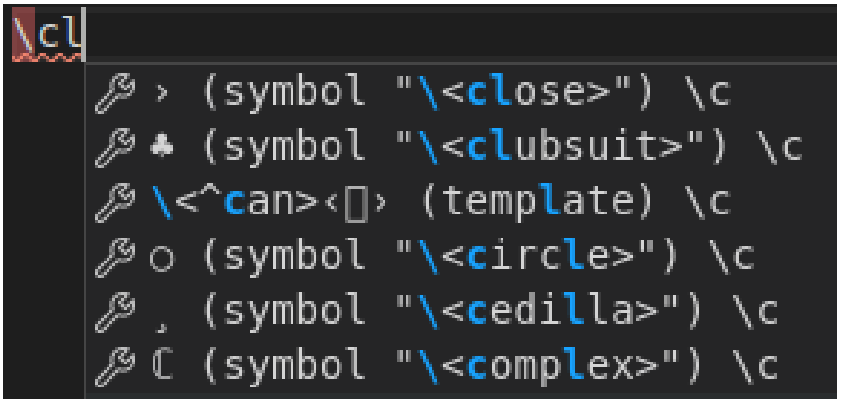


Figure 3.11: Example of latex notation auto-completion in VSCode.

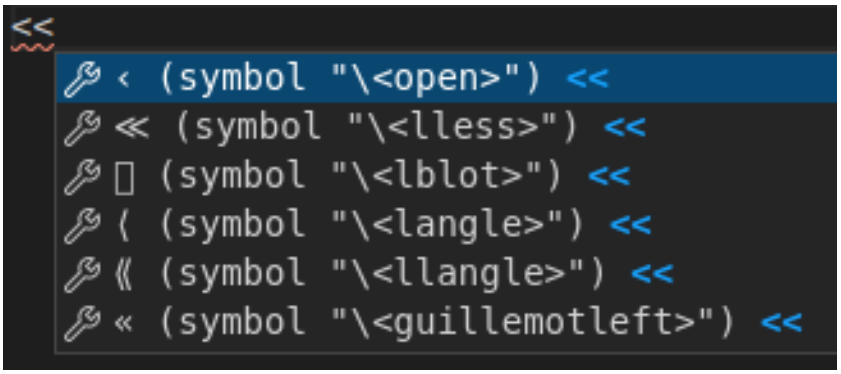


Figure 3.12: Example of auto-completion for not unique abbreviations in VSCode.

4 Evaluation

In this chapter, we evaluate the impact of our changes on the performance of Isabelle/VSCode. Performance is measured within the Isabelle/VSCode extension using the High Resolution Time API. The extension, in this case, is of course running in production mode. We also compare the performance of the extension before and after the changes.

The benchmarks are run with the following specifications, as shown below in Table 4.1:

VSCode version	1.58.2 x64
Distribution	Ubuntu 20.04 focal
Kernel	5.8.0-63-generic
Processor	Intel Core i7-8550U
RAM	8 GiB DDR4

Table 4.1: Table of specifications used for performance benchmarks.

4.1 Encoding Performance

As discussed in Section 3.2, the encoding and decoding of the files is a process that will happen frequently. We took steps to ensure that it is reasonably efficient, such as using prefix trees and running it asynchronously, to not block execution.

To benchmark the performance of encoding, we used the theory files in the Isabelle distribution. The distribution contains 1758 theory files with sizes ranging from 58 bytes to 375 kB which gives us a good representative sample. For each file we ran 1001 iterations of encoding, to extract the average and median value of the time it takes to encode a file. The results can be seen in Figure 4.1.

As expected the encode time is heavily dependent on the size of the file. Aside from the file size, encoding is also dependent on the contents of the file. Naturally, a file which contains less latex notations would take less time to encode. The highest average time is 17.18 ms for a file the size of 375 kB and the highest median time is 20.5 ms for a file size of 267 kB. Normally, theory files are not this big. In the `Isabelle/src` directory the median file size is only 8.7 kB in which case the encoding would take less then 1 ms.

With these data points, we can safely conclude that the performance of our extension will not be negatively affected from encoding.

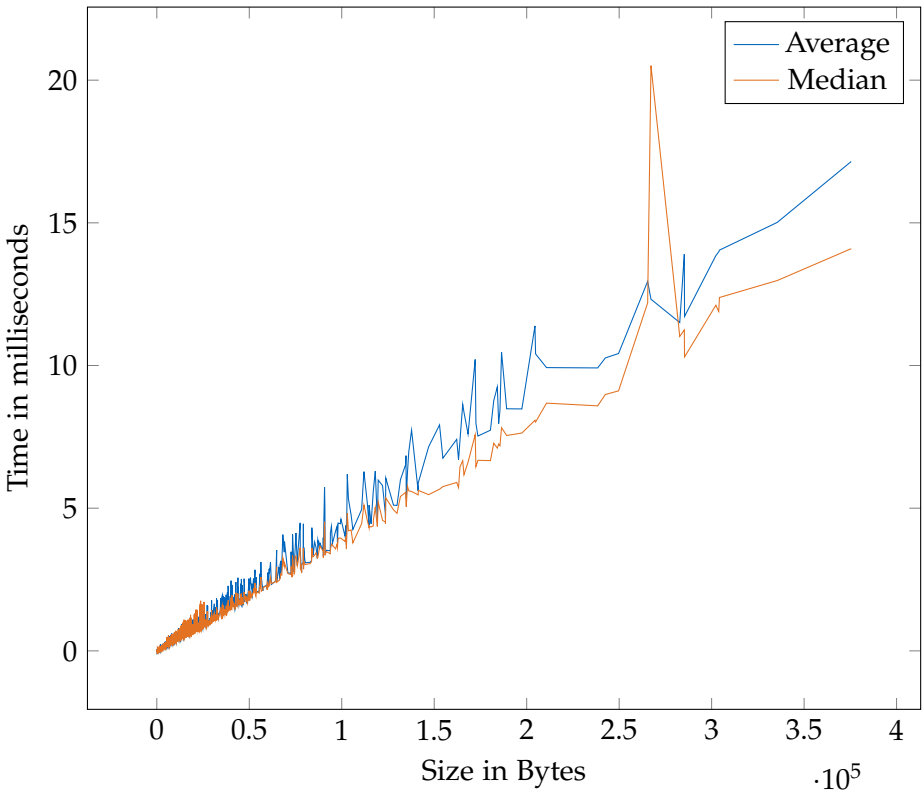


Figure 4.1: Encode time for all theories in Isabelle. Average and median calculated on 1001 iterations.

4.2 Server Startup

One area that has been affected by the changes is the server startup time. When we start the server, we also provide it with the current workspace directory, as explained in Section 3.2. If this directory contains a ROOT/ROOTS file, we get back information regarding the sessions and the theories that belong to said sessions. This process makes the startup take longer, which can potentially affect the user experience. Other changes to the server might have an influence on the startup time as well but their effect is negligible in comparison.

In this section, we measure the startup time for the server before the changes and after the changes. Since the startup time after the changes is dependent on how many theories a session has, we measure it for the following sessions: Isabelle (all sessions), Benchmarks, CCL, CTT, Cube, Doc, FOL, FOLP, HOL, LCF, Pure, Sequents, Tools, ZF. Before the changes, the startup time is not dependent on the session so we only have one case. For each case we take the average out of 10 measurements since they are quite time consuming. Figure 4.2 illustrates these measurements in a bar chart.

Most sessions are not heavily affected, with most of them being within a second of the old startup time. Only the large sessions, Isabelle, Benchmarks, DOC and HOL, are clearly disadvantaged. The old server version would takes around 12 seconds to start, while now

opening all sessions takes 25 seconds. This is a significant difference, but only occurs once at startup.

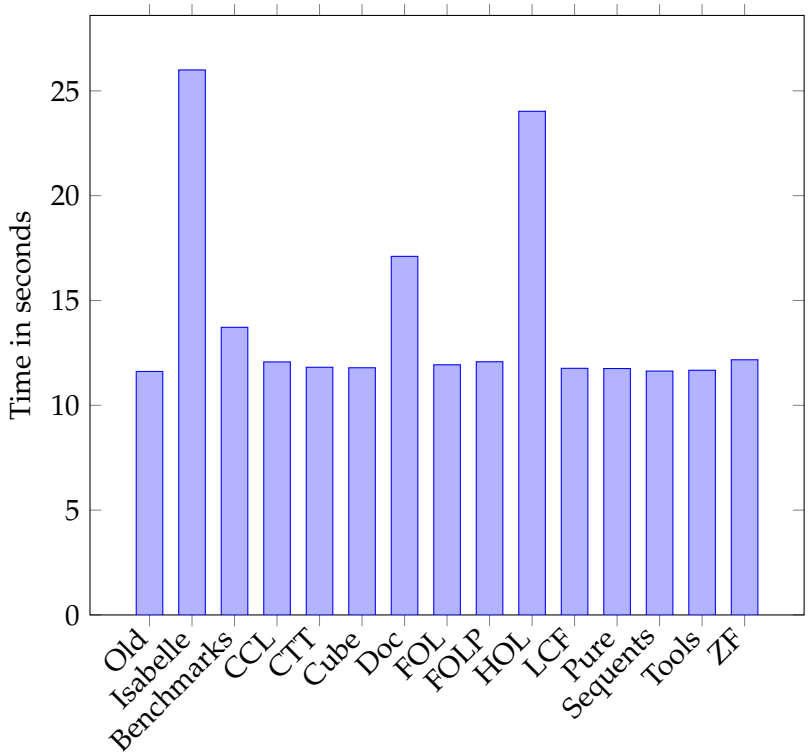


Figure 4.2: Times it takes to startup the server before the changes, and after the changes for different sessions. (The bar named Old representing the startup time before the changes.)

4.3 User Experienced Delay

The metric that we are now interested is, how long does it take before the user can start typing. In a way, this should represent how much delay the user experiences while working with the extension activated.

This metric is measured differently in the different versions of the extension. In the old version, the user cannot start working until the decorations from Prettify Symbols Mode are applied to the document, because the editor is temporarily unresponsive. To measure this, it was necessary to add timers inside the Prettify Symbols Mode extension, which was possible, since the extension is open source [23]. The measurements were conducted manually, by opening theory files and waiting for the extension to apply the decorations. We found that on average it takes 23 seconds, before the user can start working on the opened document.

In the new version of the extension, the user can start working immediately after VSCode is started if the workspace is already set up. If it is not setup, then the user would have to wait

for the server to start. This was already measured in section 4.2, therefore we do not show this case here. We now only measure how much the new Isabelle/VSCode impacts the startup of a VSCode instance. This is done by timing the `activate` function of the extension. As was the case on the server startup time, the activation time is dependent on number of theories in our sessions. For this reason, we use the same sessions that we used in section 4.2. Since this also has to be done manually, by opening and closing VSCode, we only take the average out of 10 measurements. The results of the measurements can be seen below in Figure 4.3.

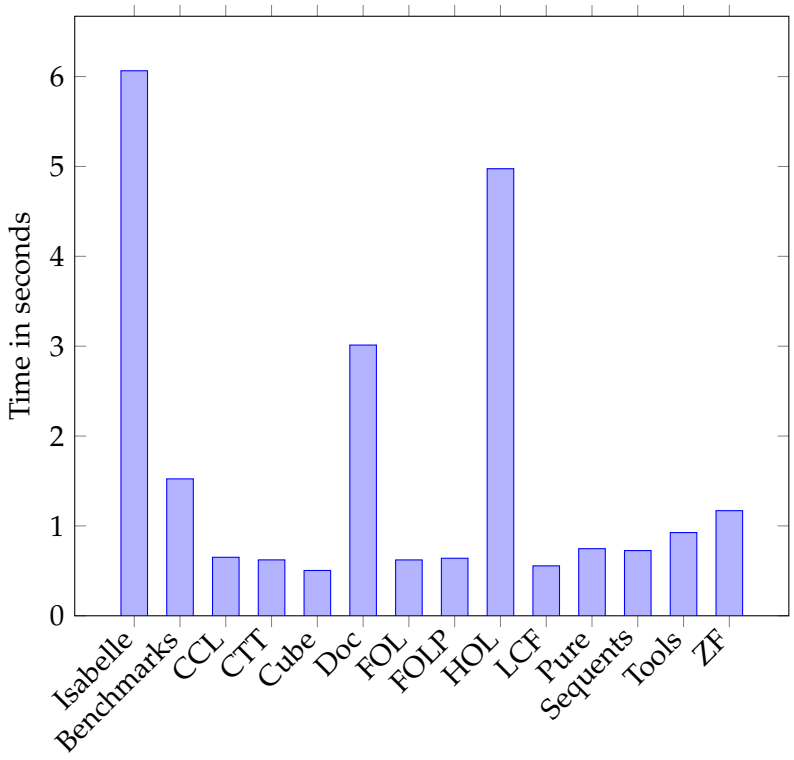


Figure 4.3: Times it takes to activate the Isabelle/VSCode extension, depending on the active session.

The activation time for the new extension is not insignificant, especially the worst case, where it takes 6 seconds to load all sessions. Regardless, it is still nowhere as impactful to the user as the delay experienced while working with Prettify Symbols Mode. It must be noted, that the delay from Prettify Symbols Mode is incurred every time a document is opened. On the other hand, the user experiences delay from the new Isabelle/VSCode only once, when VSCode is first opened, which is a significant improvement.

4.4 Threats to validity

While measuring the performance of our encoding algorithm, it quickly became evident that the algorithm itself is not the bottleneck of the process. The extension is also slowed down

by read/write operations that precede and succeed the encoding. Since these operations are completely dependent on the hardware the user is running, it was not suitable to include their performance in our measurements. In any case, we expect that computers with SSDs will run the software efficiently.

The main limitation for the startup and delay evaluation was that they had to be done manually by repeatedly opening and closing the VSCode editor while a version of the extension was active. This prevented us from running a high number of tests which in turn might have skewed the data.

In section 4.3, the measurement comparison between the two versions was obviously not a one-to-one comparison. Since the versions have very different workflows, it is difficult to do direct comparisons. Furthermore, these measurements are also dependent on the execution of other extensions, since all extensions run on a single thread in the Extension Host process [24]. These extensions are not only the extension that the user has installed but, as we established in section 2.2, core features which are built as extensions. For this reasons, the results that one might get are not always consistent.

5 Conclusion

In this chapter, we conclude the thesis by summarizing the goals reached during the course of the project. We will also compare the new solution with the current state of the existing solution.

5.1 Goals Reached

5.1.1 Syntax Highlighted Panels

The existing solution was missing syntax highlighting and clickable items for both the output and state panel. After the changes, the syntax highlighting for our panels works perfectly with two main themes, namely the dark and light theme of VSCode. The GUI elements have also been updated, to be consistent with the design of VSCode.

The solution we developed for this issue is a general solution, which can be applied to future panels as well. Consequently, future panels will be able to support all the previously mentioned features with ease.

5.1.2 Unicode Characters

As discussed in Section 3.2, the mathematical symbols in the existing solution are merely optical illusions and not true Unicode characters. We introduced the Isabelle file system, to deal with this issue. In the new file system, the Isabelle notations are encoded to represent the corresponding Unicode character.

Working with these symbols is now much easier for the user. The symbols are treated as any other character. Interacting with the symbols in the existing solution was highly unreliable, with the editor reacting unpredictably.

5.1.3 Fix Performance Issues

Since all the performance issues were directly caused by the rendering of the mathematical symbols, reworking the solution solved these issues. In Chapter 4, we showed that the new solution surpasses the existing solution in terms of performance. The user experience has definitely been improved.

We also showed in Section 4.1, that file size only affects performance marginally, which is not noticeable to the user. In contrast to that, the existing solution would cause problems with the VSCode rendering engine when a big theory file was opened.

An added benefit of the new solution is that the Prettify Symbols Mode extension does not have to be used anymore. Not only does this remove an outdated dependency, but it also simplifies the setup of Isabelle/VSCode. The user will no longer have to download and configure a third-party extension, to be able to work with mathematical symbols.

5.1.4 Abbreviations and Auto-completion

To improve the user experience even further and to bring it more in line with that of Isabelle/jEdit, we added support for abbreviations and auto-completion. With the help of these features, users can input mathematical symbols without having to write the whole Isabelle notation. They have the option to use abbreviations, which get automatically replaced with the corresponding symbol, or auto-completed Isabelle notations.

We also added the option to choose what level of abbreviation replacement the user wants. In the extension settings, the user can choose between no replacement, replacing unique non-alpha-numeric abbreviations, or replacing all unique abbreviations. Abbreviations that are not unique do not get replaced, but the user gets auto-completion suggestions.

5.2 Closing Remarks

After the comparisons made above in Section 5.1, we can say with certainty, that the developed solution has better performance than the existing solution and improves usability greatly. This is an important step towards making Isabelle/VSCode a viable alternative to jEdit. The Isabelle specific functionality, implemented for the VSCode extension, enable users to apply the same workflows that they have already established with jEdit. Moving forward, with new features being added to Isabelle/VSCode, we expect that it starts to see adoption from more users.

6 Future work

This chapter describes, what further work can be done on Isabelle/VSCode by later projects.

6.1 Features

Although several new features have been added throughout this project, there is still some work to be done, so that the functionality offered in Isabelle/VSCode is comparable to that of jEdit. In this section, we will explore new features which could be added, to improve the overall user experience.

6.1.1 Shared Server Instance

When working with multiple instances of VSCode, each of them will start their own Isabelle/VSCode server. This has the following disadvantages:

- As we stated in Section 4.2, the startup of the server takes a considerable amount of time. The user would have to wait out the server every time a new VSCode window is opened.
- Each server instance requires a lot of resources, especially memory. This makes working, with multiple VSCode windows, on low memory machines impossible.

In the future, it would be beneficial to have the client attach to an already running server instance. Implementing this feature would take some effort since the server would have to be refactored to support requests from different VSCode instances.

6.1.2 Manual Workspace Setup

With the new changes, the extension gets activated only when there is a ROOT file in the current directory. This stops users from working outside of such directories, without sessions. There are some cases where it is advantageous for the user to work without a session. In this case, the user would set up an empty workspace and then add the theories he needs to work with. All these theories would be added to the `Draft` session since the workspace was set up without a session.

To implement this feature the following steps need to be taken:

1. Add a new command to the extension, which triggers the setup of a workspace, without needing a ROOT file.

2. Give the server an option to start without the current directory.
3. Refactor the Isabelle file system to support starting without getting the sessions from the server.

This feature would be beneficial to users who only want to make quick edits and are not interested in the session structure of the project.

6.1.3 Proper Formatting for State/Output

In Section 3.1, we added the syntax highlighting for the panels. A problem that came up, was the incorrect formatting of the text in the state and output panels. The line breaks in the `WebViews` are missing and need to be added in the correct positions.

6.1.4 Extra Semantic Editor Perspectives

Not all semantic editor perspectives that are present in jEdit can be found in VSCode. Until now, only the state and output perspective have been implemented. From the missing perspectives, the most important are:

- **Sledgehammer.** This perspective is helpful for invoking the Isar `sledgehammer` command. This command applies automatic theorem provers on the current proof goal.
- **Query.** This perspective gives the user the possibility to request extra information from the prover.

The jEdit interfaces of these panels can be easily replicated in VSCode since all the necessary GUI elements are already available.

6.2 Additional Code Editors and IDEs

The use of a language server makes it possible to quickly develop plugins for other code editors and IDEs, which also use the Language Server Protocol. Furthermore, the language server of Isabelle/VSCode uses the standard endpoints of the protocol for most communications. This should reasonably speed up the development of such plugins. Normally, most editors require some level of customization in order to work properly. Below, we will explore the code editors most suitable for an Isabelle plugin, given the current state of the language server.

6.2.1 Sublime Text

Sublime Text is a sophisticated text editor for code, markup and prose [25]. The Language Server Protocol is already fully integrated into Sublime Text [26]. This, along with its popularity (see Figure 2.1), makes Sublime Text a good candidate for an Isabelle plugin.

In its current state, the language server would only provide the basic language features. More complicated features would still have to be implemented. One advantage of using Sublime Text is that there wouldn't be a need to implement a separate file system, as was necessary for the VSCode extension. Similar to jEdit, in Sublime Text, an encoding can be added programmatically from within a plugin. This makes developing a plugin for this editor, easier in comparison to VSCode.

6.2.2 IntelliJ IDEA

IntelliJ IDEA is an integrated development environment written in Java [27]. We mentioned in Section 2.1 that Scala is used to develop the systems around the Isabelle environment. IntelliJ IDEA is one of the preferred IDEs for developers to work with Scala. The language is fairly supported in IntelliJ through a plugin.

It could be feasible in the future to support a plugin for Isabelle in IntelliJ. This would be helpful for users who are already working on Isabelle/Scala with this IDE. Then IntelliJ would become a fully-featured IDE for working with Isabelle. A user wouldn't need to have an instance of jEdit or VSCode running alongside an IntelliJ instance. All operations could be conducted within only one IDE.

The Language Server Protocol is not natively supported in IntelliJ. This is potentially a big hurdle to developing the plugin for IntelliJ, but there is already a highly rated third-party plugin that adds support for the protocol [28].

List of Figures

List of Tables

Bibliography

[1] L. Paulson. "Natural Deduction as Higher-Order Resolution". In: *The Journal of Logic Programming* (1986).

[2] jEdit project. *jEdit - Programmer's Text Editor*. May 24, 2021. URL: http://www.jedit.org/ (visited on 06/01/2021).

[3] M. Wenzel. *Isabelle/jEdit*. Feb. 20, 2021. URL: https://isabelle.in.tum.de/doc/jedit.pdf.

[4] Microsoft. *Visual Studio Code - Code Editing. Redefined*. June 1, 2021. URL: https://code.visualstudio.com/ (visited on 06/01/2021).

[5] M. Wenzel. "Isabelle/PIDE after 10 years of development". In: *UITP workshop: User Interfaces for Theorem Provers* (2018).

[6] Isabelle. *Archive of Formal Proofs*. May 24, 2021. URL: https://www.isa-afp.org/index.html (visited on 06/01/2021).

[7] M. Wenzel. *The Isabelle/Isar Reference Manual*. Feb. 20, 2021. URL: https://isabelle.in.tum.de/doc/isar-ref.pdf.

[8] École Polytechnique Fédérale Lausanne. *The Scala Programming Language*. July 8, 2021. URL: https://www.scala-lang.org/ (visited on 07/08/2021).

[9] Microsoft. *Differences between Code OSS and Visual Studio Code*. Github. Mar. 21, 2020. URL: https://github.com/microsoft/vscode/wiki/Differences-between-the-repository-and-Visual-Studio-Code (visited on 05/30/2021).

[10] Stack Overflow. *Differences between Code OSS and Visual Studio Code*. Stack Overflow. 2019. URL: https://insights.stackoverflow.com/survey/2019#technology (visited on 05/30/2021).

[11] Microsoft. *Visual Studio Code Extension API*. May 5, 2021. URL: https://code.visualstudio.com/api (visited on 06/01/2021).

[12] Microsoft. *Git integration for Visual Studio Code*. Github. Oct. 28, 2018. URL: https://github.com/microsoft/vscode/tree/main/extensions/git (visited on 05/30/2021).

[13] Microsoft. *The Visual Studio Code Extension Manager*. NPM. June 1, 2021. URL: https://www.npmjs.com/package/vsce (visited on 06/01/2021).

[14] Microsoft. *Language Server Extension Guide*. May 5, 2021. URL: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide (visited on 06/01/2021).

[15] Microsoft. *Language Extensions Overview*. July 8, 2021. URL: https://code.visualstudio.com/api/language-extensions/overview (visited on 07/08/2021).

[16] Microsoft. *Language Server Protocol Specification*. May 5, 2021. URL: https://microsoft.github.io/language-server-protocol/specifications/specification-current/ (visited on 06/01/2021).

[17] G. H. Thierry Coquand. *The Coq Proof Assistant*. Mar. 5, 2021. URL: https://coq.inria.fr/ (visited on 06/01/2021).

[18] Microsoft Research. *https://leanprover.github.io/*. May 24, 2021. URL: https://leanprover.github.io/ (visited on 06/01/2021).

[19] C. Bell. *Coq Support for Visual Studio Code*. May 24, 2017. URL: https://github.com/siegebell/vscoq (visited on 06/01/2021).

[20] Microsoft Research. *Lean for VS Code*. Apr. 24, 2021. URL: https://github.com/leanprover/vscode-lean (visited on 06/01/2021).

[21] M. Wenzel. *Isabelle/VSCode in January 2017*. Jan. 26, 2017. URL: http://sketis.net/wp-content/uploads/2017/01/isabelle-vscode-jan-2017.pdf.

[22] C. Bell. *Feature request: prettify symbols mode*. Github. Jan. 27, 2016. URL: https://github.com/microsoft/vscode/issues/2402 (visited on 05/30/2021).

[23] C. Bell. *Prettify Symbols Mode*. May 7, 2018. URL: https://github.com/siegebell/vsc-prettify-symbols-mode (visited on 06/01/2021).

[24] Microsoft. *Extension Host*. Apr. 24, 2021. URL: https://code.visualstudio.com/api/advanced-topics/extension-host (visited on 06/01/2021).

[25] Sublime HQ. *Sublime Text - Text Editing. Done Right.* Aug. 9, 2021. URL: https://www.sublimetext.com/ (visited on 08/09/2021).

[26] Sublime HQ. *LSP for Sublime Text*. Aug. 7, 2021. URL: https://lsp.sublimetext.io/ (visited on 08/07/2021).

[27] JetBrains. *IntelliJ IDEA: The Capable and Ergonomic Java IDE by JetBrains*. Aug. 9, 2021. URL: https://www.jetbrains.com/idea/ (visited on 08/09/2021).

[28] G. Tâche. *LSP Support*. Aug. 9, 2021. URL: https://plugins.jetbrains.com/plugin/10209-lsp-support (visited on 08/09/2021).