SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY
– INFORMATICS –

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Improving Isabelle/VSCode: Towards Better Prover IDE Integration via Language Server

Thomas Lindae

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY
– INFORMATICS –

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Improving Isabelle/VSCode: Towards Better Prover IDE Integration via Language Server

Isabelle/VSCode Verbesserungen: Fortschritte in der Prover IDE Integration mittels Language Server

Author: Thomas Lindae
Supervisor: Prof. Dr. Stephan Krusche
Advisors: Prof. Dr. Tobias Nipkow, M.Sc. Fabian Huch
Start Date: 15.04.2024
Submission Date: 15.08.2024

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

Munich, 15.08.2024 Thomas Lindae

Acknowledgements

There are several people I would like to thank:

Prof. Dr. Tobias Nipkow for giving me the opportunity to work on a tool as big and prominent as Isabelle.

My advisor Fabian Huch for meeting with me weekly, helping me understand the inner workings of Isabelle, discussing design and implementation details and lending me his time for other silly questions.

My father Andreas Lindae for doing his best rubber duck impression and letting me waste his time by explaining the contents of this thesis to him to sort out my thoughts.

My friends Adrian Stein and Alexander Treml for their valuable feedback on various sections of this thesis and helping me with its overall structure.

Many more of my fellow student friends for joining me in my visits to the cafeteria and providing mental and emotional respite during lunch.

Abstract

The primary interface for interacting with the Isabelle proof assistant is the Isabelle/jEdit prover IDE. Isabelle/VSCode was developed as an alternative, implementing a language server for the Language Server Protocol and a language client for Visual Studio Code. However, Isabelle/VSCode did not provide a user experience comparable to Isabelle/jEdit. This thesis explores and implements several improvements to address these shortcomings by refining existing functionality and augmenting Isabelle/VSCode with new features. Key enhancements include improved completions, persistent decorations on file switch, code actions for interacting with active markup, and better formatting for state and output panels. Additionally, we implemented more granular control over symbol handling and an Isabelle system option to turn off HTML output, increasing compatibility with potential new language clients. We developed prototype language clients for the Neovim and Sublime Text code editors to evaluate the improved language server's versatility. While an Isabelle language client for these editors was previously infeasible, our enhancements made them viable. Our work not only brings Isabelle/VSCode closer to feature parity with Isabelle/jEdit, but also paves the way for future integrations with a broader range of development environments.

Contents

Acknowledgements **iv**

Abstract **v**

1 Introduction **1**
1.1 Motivation ... 2

2 Background **3**
2.1 Isabelle ... 3
2.1.1 Isabelle/Isar ... 3
2.1.2 Implementation Design ... 3
2.1.3 Output and State Panels ... 4
2.1.4 Symbols ... 5
2.1.5 Isabelle/VSCode ... 6
2.2 Language Server Protocol (LSP) ... 7
2.2.1 Initialization ... 8
2.2.2 Isabelle Language Server ... 9

3 Related Work **10**

4 Refinements to Existing Functionality **11**
4.1 Desync on File Changes ... 11
4.2 State Panel IDs ... 12
4.3 State and Output Panels ... 13
4.3.1 Correct Formatting ... 14
4.3.1.1 Isabelle's Internal XML ... 14
4.3.1.2 Using `Pretty` for Isabelle/VSCode ... 14
4.3.2 Correct Font ... 16
4.4 Completions ... 17

4.4.1 Completions Items Not Showing ... 17
4.4.2 Immediate Completions and Commit Characters ... 19

5 Enhancements and New Features **20**
5.1 Decorations on File Switch ... 20
5.2 Non-HTML Content for Panels ... 23
5.3 Symbol Handling ... 24
5.3.1 Symbol Options ... 25
5.3.2 Symbols Request ... 26
5.3.3 Symbols Conversion ... 27
5.4 Code Actions for Active Markup ... 28
5.4.1 Implementation for the Isabelle Language Server ... 30
5.5 Isabelle System Options as VSCode Settings ... 31
5.5.1 Passing Options from VSCode to the Language Server ... 32
5.5.2 Option Types ... 34
5.5.3 Extending Isabelle/VSCode's Settings ... 34

6 Evaluation **36**
6.1 Neovim ... 36
6.2 Sublime Text ... 36

7 Conclusion **37**

8 Future Work **38**

Bibliography **43**

1 Introduction

In 1911, Alwin Korselt [7] showed that Ernst Schröder's proof of the Cantor-Bernstein-Schröder theorem, initially published in 1898 [15], contained an error. While the theorem was correct, and other proofs of this theorem existed even back then [3], this is not the only instance where mathematicians tried to prove a statement, thought they conceived such proof, and later found that the proof was incorrect or incomplete. Proof assistants were developed to alleviate this issue by allowing one to formalize mathematical proofs and have the computer check for the proof's correctness. One such tool is *Isabelle*[1], originally developed at the *University of Cambridge* and *Technical University of Munich*.

To interact with the Isabelle system, the Isabelle distribution bundles the *Isabelle/jEdit* code editor [18], a modified version of the Java-based code editor *jEdit*[2]. Through that, the user is offered a fully interactive Isabelle session, in which proofs are written and checked in real time. However, Isabelle/jEdit has many accessibility shortcomings, like missing a dark theme. To tackle this problem, *Isabelle/VSCode* was built to create support for Isabelle from within the popular code editor *Visual Studio Code* utilizing the *Language Server Protocol*[3] (*LSP*) [16,10].

This protocol was originally developed by Microsoft specifically for VSCode. It consists of two main components: A language *server*, responsible for understanding the language on a semantic level, and a language *client*, which is typically the code editor itself. Later, the protocol became a standardized specification and is now widely used by many different programming languages and code editors to more easily support IDE functions like completions and diagnostics.

Unfortunately, the current state of Isabelle/VSCode is not on par with the experience of Isabelle/jEdit. There are issues with usability and missing features. Additionally, the underlying language server lacks the necessary capabilities to support the development of an Isabelle language client for another code editor.

To combat these deficiencies, we will identify the various aspects of the current Isabelle/VSCode that need improvement and evaluate potential solutions to enhance its functionality. Given that Isabelle's design is often fundamentally incompatible with the LSP spec-

[1] https://isabelle.in.tum.de/
[2] https://www.jedit.org/
[3] https://microsoft.github.io/language-server-protocol/

ification, the primary question throughout this endeavor is whether the existing LSP spec can be utilized to fit Isabelle's unique requirements, whether a completely custom solution needs to be built for each language client, or whether there is a middle ground in which the language server can take over much of the work, but custom handlers for the client are still necessary.

We will consider two primary metrics for these solutions: How closely they resemble Isabelle/jEdit and how universally applicable it is to other language clients. The former ensures consistency within the broader Isabelle system, while the latter facilitates integration with new language clients.

The primary contribution of this thesis is the implementation of several such solutions to create a more flexible language server, reducing Isabelle's reliance on jEdit and VSCode. Moreover, we extended and modified the VSCode extension to accommodate these new changes, bringing its user experience closer to that of Isabelle/jEdit, and we built two usable prototype client integrations for the *Neovim*[4] and *Sublime Text*[5] code editors to assess the new flexibility.

1.1 Motivation

Prior to this work, we attempted to build an Isabelle language client for the terminal-based code and text editor Neovim. However, it quickly became apparent that the existing language server was insufficient. For example, it only sent the content of certain panels in an HTML format. This makes it easy for an Electron-based editor like VSCode, which runs on the Chromium browser engine, allowing the editor to effortlessly and natively display HTML. However, displaying HTML content from within a terminal-based editor like Neovim is not reasonably possible, meaning an option to get non-HTML output was required.

Because of this, it was virtually impossible to build such a language client with the official Isabelle language server. Instead, we used the unofficial Isabelle fork `isabelle-emacs`[6], which includes many advancements to the Isabelle language server and fixes some of its issues to support the *Emacs*[7] code editor. While this fork enabled a usable Neovim Isabelle environment[8], e.g., by offering the aforementioned non-HTML output option, there were still many missing features compared to Isabelle/jEdit. Still, the `isabelle-emacs` fork was a strong inspiration for some of the changes made in the context of this thesis.

[4]https://neovim.io/
[5]https://www.sublimetext.com/
[6]https://github.com/m-fleury/isabelle-emacs
[7]https://www.gnu.org/software/emacs/
[8]https://github.com/Treeniks/isabelle-lsp.nvim/tree/0b718d85fd4589d27638877f8955bedb93f56738

2 Background

2.1 Isabelle

From proving the prime number theorem [1] over a verified microkernel [4] to a formalization of a sequential Java-like programming language [6], Isabelle has been used for numerous formalizations and proofs since its initial release in 1986. Additionally, the Archive of Formal Proofs[1] hosts a journal-style collection of many more such proofs constructed in Isabelle.

2.1.1 Isabelle/Isar

When one wants to write an Isabelle theory, i.e., a document containing several theorems, lemmas, function definitions, and more, Isabelle offers a proof language called *Isabelle/Isar*, allowing its users to write human-readable structured proofs [19]. An example theory and Isabelle/Isar proof can be seen in Listing 1.

The Isabelle/Isar syntax comprises three main syntactic concepts: *Commands*, *methods*, and *attributes*. Particularly relevant for us are the commands, which include keywords like `theorem` to state a proposition followed by a proof or `apply` to apply a proof method.

2.1.2 Implementation Design

Isabelle's core implementation languages are *ML* and *Scala*. Generally, the ML code is responsible for Isabelle's purely functional and mathematical domain (e.g. its LCF-style kernel [13,14]), while Scala is responsible for Isabelle's physical domain (e.g. everything to do with the UI and IO [20:Chapter 5]). Many modules within the Isabelle code base exist in both Scala and ML, thus creating an almost seamless transition between the two.

Isabelle employs a monolithic architecture. While logic is split between modules, there is no limitation on how they can be accessed within the Isabelle system. Moreover, as a JVM-based programming language, Scala effortlessly integrates into jEdit's Java code base. Due to these two facts, when using Isabelle/jEdit, Isabelle is able to offer an interactive session where the entire Isabelle system has direct access to any data jEdit may hold, and the same is true the

[1] https://www.isa-afp.org/

```
theory Example
  imports Main
begin

theorem "∄f :: 'a ⇒ 'a set. surj(f)"
proof
  assume "∃f :: 'a ⇒ 'a set. surj(f)"
  then obtain f :: "'a ⇒ 'a set" where a: "surj(f)" by blast
  from a have b: "∀A. ∃a. A = f a" by (simp add: surj_def)
  from b have c: "∃a. {(x :: 'a). x ∉ f x} = f a" by blast
  from c show False by blast
qed

end
```

Listing 1: Example Isabelle theory with Isabelle/Isar proof.

other way around. For example, Isabelle/jEdit has a feature that automatically indents an Isabelle theory. Internally, this automatic indentation uses both access to the Isabelle system and the jEdit buffer simultaneously.

Isabelle, being a proof assistant, also does not follow conventional programming language design practices. The actual Isabelle kernel is kept small to maintain correctness (albeit with performance-related additions). Many of Isabelle's systems are built within Isabelle itself, including a majority of the Isabelle/Isar syntax.

> Note that static grammar and language definitions are not ideal: Isabelle syntax depends on theory imports: new commands may be defined in user libraries.
>
> — Makarius Wenzel [16]

Even fundamental keywords such as `theorem` do not exist statically but are instead defined in user space. When editing a theory in Isabelle/jEdit, the syntax highlighting is mostly done dynamically.

2.1.3 Output and State Panels

Isabelle has a few different types of panels that give the user crucial information. The two most relevant to us are the *output* and *state* panels, as seen in Figure 1.

The output panels show messages corresponding to a given command, including general information, warnings, or errors. This also means that the content of the output panel is directly

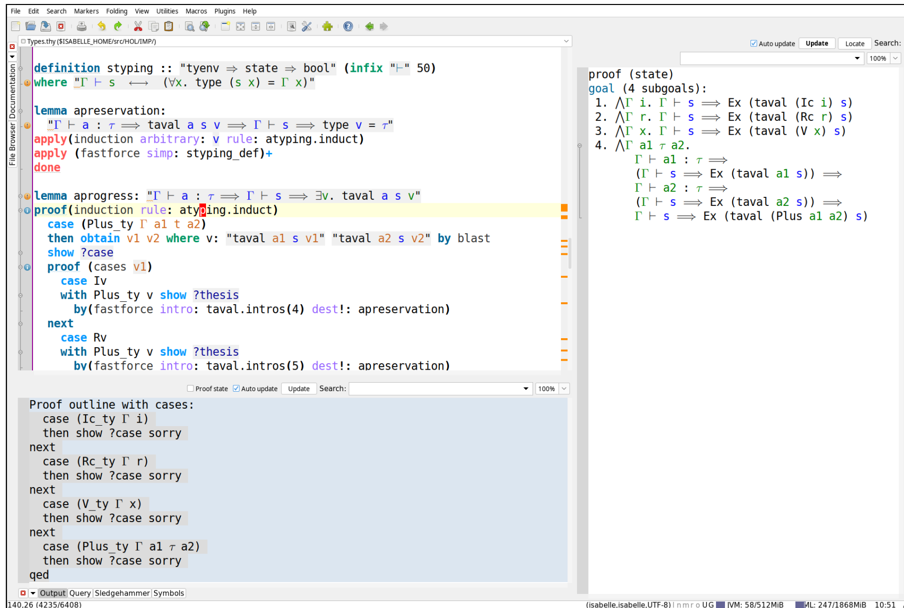

Figure 1: JEdit with both output and state panels open. Output on the bottom, state on the right.

tied to a specific command in the theory. The command is typically determined by the caret's current position.

On the other hand, state panels display the current internal proof state within a proof. Just like with output panels, it is possible to open multiple state panels, which may show states at different positions within the document. Whether moving the caret updates the currently displayed output or state depends on the *Auto update* setting of the respective panel.

2.1.4 Symbols

Isabelle uses a lot of custom symbols to allow logical terms to be written in a syntax close to that of mathematics. The concept of what an *Isabelle symbol* is exactly is rather broad. We will focus primarily on a certain group of symbols typically used in mathematical formulas.

Each Isabelle symbol roughly consists of four components: An ASCII representation of the symbol, a name, an optional UTF-16 code point, and a list of abbreviations for this symbol.

ASCII Representation	`\<Longrightarrow>`
Name	`Longrightarrow`
UTF-16 Codepoint	`0x27F9`
Abbreviations	`.>`, `==>`

Table 1: Symbol data of `⟹`.

These four are only part of the story; however, for the sake of simplicity, we will skip some details.

As an example, let us say a user writes the implication $A \Longrightarrow B$ in Isabelle. Within jEdit, they will see it written out as `A ⟹ B`; however, internally, the `⟹` is an Isabelle symbol. Its corresponding data is outlined in Table 1.

To deal with these symbols, Isabelle/jEdit uses a custom encoding called *UTF-8-Isabelle*. This encoding ensures that the user sees `A ⟹ B` while the actual content of the underlying file is "`A \<Longrightarrow> B`". However, Isabelle has no trouble dealing with cases where the actual `⟹` Unicode symbol is used within a file.

There are a few reasons why this special system exists instead of just encoding the files in UTF-16 or UTF-8. Unicode is somewhat inconsistent regarding $_{\text{subscript}}$ and $^{\text{superscript}}$ support (e.g. while the capital letters A to W exist in superscript, X, Y, and Z currently do not). Isabelle instead adds `\<^sub>` and `\<^sup>` prefixes to letters and numbers, which can also be nested. Additionally, by encoding theories with simple ASCII characters, they can be viewed with almost any font and do not require more advanced Unicode support.

2.1.5 Isabelle/VSCode

Isabelle consists of multiple different components. Isabelle/jEdit is one such component. When we mention Isabelle/VSCode, we are referring to three different Isabelle components: The Isabelle *language server*, which is a part of Isabelle/Scala, Isabelle's own patched

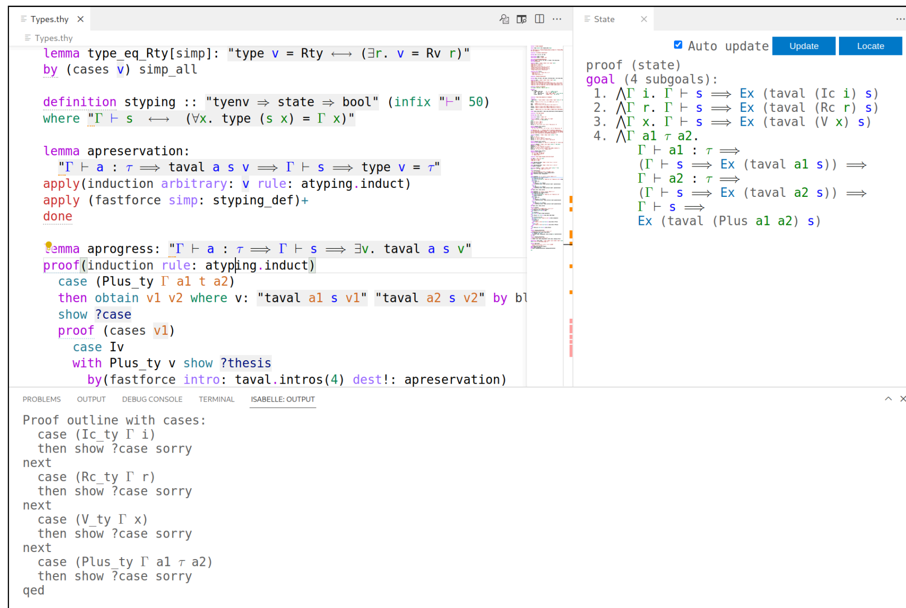

Figure 2: VSCode with both output and state panels open. Output on the bottom, state on the right.

VSCodium[2], and the VSCode *extension* written in TypeScript.[3] Note in particular that when running Isabelle/VSCode, Isabelle does not use a standard distribution of VSCode. Instead, it is a custom VSCodium package.

VSCodium is a fully open-source distribution of Microsoft's VSCode with some patches to disable telemetry and replace the VSCode branding with that of VSCodium. Isabelle adds patches on top of VSCodium to add a custom encoding mimicking the functionality of Isabelle/jEdit described in Section 2.1.4 and integrating custom Isabelle-specific fonts. Since neither adding custom encodings nor including custom fonts is possible from within a VSCode extension, these patches exist instead.

The concept of output and state panels exist equivalently within Isabelle/VSCode, as seen in Figure 2, although it is currently not possible to create multiple panels of the same type.

Generally speaking, the goal of Isabelle/VSCode is to mimic the functionality of Isabelle/jEdit as closely as possible. Many issues described and solved within this work stem from a discrepancy between the two, and Isabelle/jEdit will often serve as the reference implementation.

2.2 Language Server Protocol (LSP)

Before the introduction of the Language Server Protocol, it was common for code editors to either only support syntax highlighting for its supported languages with very basic auto-completion and semantic understanding or implement a full-fledged IDE for the language.

Now, the responsibility of semantic understanding of the language has moved entirely to the language server, while the language client is responsible for handling user interaction.

The goal is a system in which a new programming language only needs to implement a language server, while a new code editor only needs to implement a language client. In the best-case scenario, any language server and client can be used together (although, in practice, this is still not always the case). If we wanted to support N programming languages for M code editors, without the LSP, we would need $N \cdot M$ implementations of language semantics. With the LSP, this number is reduced drastically to only N language server and M language client implementations.

Microsoft [9] describes the general setup: The client and server communicate via `jsonrpc 2.0` messages. The three primary message types are:

- *Notification Messages*
- *Request Messages*
- *Response Messages*

[2] https://vscodium.com/
[3] https://www.typescriptlang.org/

As the name suggests, notification messages only exist to notify the other party. They need not send a response back. Requests are sent to the other party and require a response message to be sent back once the request has been processed. The structure of these message types is also defined within the LSP specification and can be seen in Table 2.

Notification	**Request**	**Response**
`jsonrpc: string`	`jsonrpc: string`	`jsonrpc: string`
	`id: integer | string`	`id: integer | string | null`
`method: string`	`method: string`	`result?: Any`
`params?: array | object`	`params?: array | object`	`error?: ResponseError`

Table 2: General LSP message structure.

At the time of writing, The `jsonrpc` entry of every message is always set to "`2.0`". The `id` of the request is sent to identify the associated response. Thus, the `id` in a response message must also be set appropriately. The `method` entry is an identifier for the *kind* of message at hand and dictates the shape of the `params`, `result`, and `error` entries, which in turn contain the primary data of the message.

There are many different *methods*. For example, messages dealing with text documents are sent under the "`textDocument/`" method prefix, like the "`textDocument/hover`" request, which requests for hover information, or the "`textDocument/didChange`" notification, sent by the client to keep the server informed about changes made to the document's text.

2.2.1 Initialization

Because of the LSP's server/client system, it is technically possible to use an externally running language server. Even so, in practice, the server is typically started by an IDE.

The first message between client and server is an "`initialize`" request sent by the client. The client has to wait for the server to respond to this request before sending any other messages.

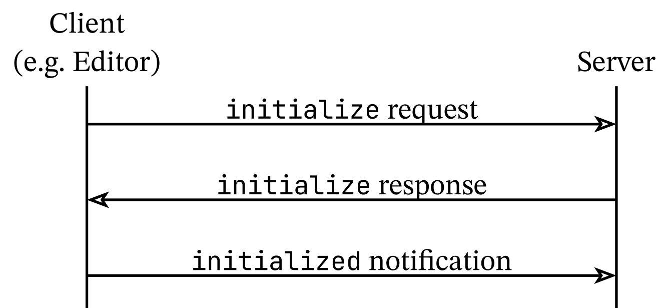

Figure 3: Visualization of the LSP initialization handshake.

Finally, the client sends an "`initialized`" notification to mark the initialization complete. This handshake is illustrated in Figure 3.

Similarly to exchanging cipher suites in a TLS handshake, the client and server send each other their capabilities within the `initialize` request and response. These capabilities describe which features of the LSP the client or server actually supports. For example, not every server supports completions. Even if it does, further information is needed, like which characters should automatically request completions. By exchanging the capabilities this early on, the client and server can exclude certain parts of messages or even skip sending some entirely, preventing expensive JSON Serialization and Deserialization for messages that the other party cannot handle.

2.2.2 Isabelle Language Server

While the LSP defines most methods required for typical language server use cases, specific language servers may extend the basic protocol by their own methods. In such cases, the corresponding client needs to define extra handlers for these new methods.

Since the standard Language Server Protocol is designed with regular programming languages in mind, it defines little for other types of languages, particularly theorem provers [5]. It is thus insufficient for Isabelle's needs. For example, to keep the output and state panels updated, the server must always know the current location of the caret. This is not a typical need for language servers of normal programming languages and is thus not built into the protocol by default.

Isabelle, therefore, extends the LSP with its own methods under the "`PIDE/`" prefix, which have to be enabled with the "`vscode_pide_extensions`" Isabelle option. For example, here are three such methods:

1. "`PIDE/caret_update`": A bidirectional notification telling the other party that the caret has been moved. Mostly sent from the client to the server.
2. "`PIDE/dynamic_output`": A notification sent from the server to the client containing the current content of the output panel.
3. "`PIDE/decoration`": A notification sent from the server to the client containing information on the dynamic syntax highlighting within the current theory.

There are several more of these methods. As a result, unlike most language servers, the Isabelle language server cannot be started from within an existing language client with the expectation that it will function correctly. Significant additional work needs to be done on the client side before an IDE can utilize the Isabelle language server.

3 Related Work

Isabelle/VSCode was created in 2017 by Makarius Wenzel [16]. Lacking features like highlighting in output and state panels, Denis Paluca [12] continued the work on Isabelle/VSCode in 2021. Since then, further improvements have been made to Isabelle/VSCode, including the introduction of a custom UTF-8-Isabelle encoding for VSCode to improve performance.

As mentioned in Section 1.1, Mathias Fleury introduced the unofficial Isabelle fork `isabelle-emacs`[1] to support the Emacs text editor, already introducing some of the features we will discuss in this thesis. While Fleury's work focuses primarily on building enough support in the language server for Emacs, this thesis's goal is to make the language server flexible enough to be usable for virtually any code editor that supports the LSP and improve more fundamental usability issues. If the changes introduced in this thesis get merged upstream into the official Isabelle distribution, the changes to the language server introduced by `isabelle-emacs` should become redundant (although the work done for the Emacs language client will not).

There are also language server implementations for other theorem provers, like VsCoq[2] for the Coq proof assistant[3], as well as `vscode-lean`[4] for the Lean theorem prover[5] [11].

Jonas Kjær Rask et al. [5] explored extensions to the LSP specification to support more types of semantic languages, including theorem provers. With these extensions implemented in both the specification and language clients, it may be possible to update the Isabelle language server to use these new LSP extensions and support other language clients with almost no additional work (i.e. no custom handlers for custom LSP messages).

---

[1]https://github.com/m-fleury/isabelle-emacs
[2]https://github.com/coq-community/vscoq
[3]https://coq.inria.fr/
[4]https://github.com/leanprover/vscode-lean4
[5]https://lean-lang.org/

4 Refinements to Existing Functionality

The work presented in this thesis on Isabelle/VSCode can be roughly categorized into two areas: The refinement of existing features and the introduction of new ones. This chapter focuses on the former. In both categories, Isabelle/jEdit serves as the primary reference implementation. Whether addressing a problem or filling a gap in functionality, the aim has been to replicate the behavior of Isabelle/jEdit closely. While it could be argued that certain features in Isabelle/jEdit also warrant improvements, this thesis does not engage with those considerations.

4.1 Desync on File Changes

While building the Neovim Isabelle client mentioned in Section 1.1, the language server frequently got out of sync with the file's actual contents. For example, it might have happened that the user wanted to write `apply auto`, but wrote `apply autt` by accident instead. If the user then corrected their mistake by removing the additional `t` and replacing it with an `o`, it could happen that the language server would think the content of the file was `apply autto`. Somewhat awkwardly, this problem *only* occurred when using Neovim; it did not happen in VSCode.

Document synchronization is done primarily through `textDocument/didChange` and `textDocument/didOpen` notifications. We will discuss the `textDocument/didOpen` notification in more detail in Section 5.3, but this desyncing issue results from the handling of the `textDocument/didChange` notifications. Its content is outlined in Listing 2.

```typescript
interface DidChangeTextDocumentParams {
    textDocument: VersionedTextDocumentIdentifier;
    contentChanges: TextDocumentContentChangeEvent[];
}
```

Listing 2: `DidChangeTextDocumentParams` interface definition [9].

The exact details of how these `contentChanges` are structured are not of interest; however, there can be multiple such content changes within a single notification. The client can decide to group multiple content changes into a single `textDocument/didChange` notification. The

desyncing problem now arises because such a list of content changes is not commutative. The LSP spec says the following about the order of application of these content changes:

> The content changes describe single state changes to the document. So if there are two content changes c_1 (at array index 0) and c_2 (at array index 1) for a document in state S then c_1 moves the document from S to S' and c_2 from S' to S''. So c_1 is computed on the state S and c_2 is computed on the state S'.
>
> To mirror the content of a document using change events use the following approach:
> - start with the same initial content
> - apply the `textDocument/didChange` notifications in the order you receive them.
> - apply the `TextDocumentContentChangeEvents` in a single notification in the order you receive them.
>
> — Microsoft [9]

The language server had code that *normalized* the `contentChanges` list, sorting them by different types of changes before applying them. Simply removing this normalization was enough to fix the original desyncing issue. We are still unsure why the issue did not occur in VSCode; however, most likely, VSCode simply groups document changes together far less frequently than Neovim.

4.2 State Panel IDs

As mentioned in Section 2.1.3, it is possible to open multiple state panels in Isabelle/jEdit. While users typically want to see the proof state at the position of their caret, there may be cases where one wants to permanently see the proof state at a different position.

The language server already supported multiple state panels (although not multiple output panels). Internally, the language server stores a `Map` from IDs to state panels. Additionally, all state-related messages must include the ID of the panel that they are referring to. For example, to disable the *Auto update* property[1] of a state panel, the client needs to send a `PIDE/state_auto_update` notification with an `id` and `enabled` field.

When starting the Isabelle language server, it does not automatically initialize a state panel. The client has to send a `PIDE/state_init` notification to create a state panel. However, the client can not define the state panel's ID within this notification. Instead, the server used the Isabelle internal `Counter` module to create a unique state panel ID.

[1]The *Auto update* property enables automatic updating of the panel's content to the caret position. If disabled, moving the caret will not change the panel's content and will only update if the user issues a manual *Update* command.

In order to keep these IDs separate between Isabelle/ML and Isabelle/Scala, this module counts in ascending order in ML and descending order in Scala. Since the language server is part of Isabelle/Scala, the state panel's IDs start at -1 and count downward with each newly created state panel. This in and of itself is not a problem; the problem was that the language server did not communicate the created IDs with the language client. Thus, the language client had to know the internal Isabelle language server ID creation logic. Furthermore, if that logic ever changes in the future, the client would need to be updated with it.

To eliminate this issue, we changed the `PIDE/state_init` message from a notification to a request. Now, when the client sends a `PIDE/state_init` request, the server sends a response back that includes the state ID of the newly created state panel. That way, we were able to decouple and future-proof the internal language server logic from the language client implementation.

An important thing to note is that Isabelle/VSCode does not support multiple state panels. While the underlying language server supports them, the Isabelle VSCode language client only supports a single state panel, therefore necessitating further work in this area.

4.3 State and Output Panels

A comparison of Isabelle/VSCode's previous panel output against Isabelle/jEdit's panel output can be seen in Table 3. Two main issues needed to be tackled:

1. The lack of formatting, particularly regarding line breaks.
2. The use of an incorrect font.

jEdit State Panel	VSCode State Panel

proof (state)
goal (4 subgoals):
 1. ⋀Γ i. Γ ⊢ s ⟹ Ex (taval (Ic i) s)
 2. ⋀Γ r. Γ ⊢ s ⟹ Ex (taval (Rc r) s)
 3. ⋀Γ x. Γ ⊢ s ⟹ Ex (taval (V x) s)
 4. ⋀Γ a1 τ a2.
      Γ ⊢ a1 : τ ⟹
      (Γ ⊢ s ⟹ Ex (taval a1 s)) ⟹
      Γ ⊢ a2 : τ ⟹
      (Γ ⊢ s ⟹ Ex (taval a2 s)) ⟹
      Γ ⊢ s ⟹ Ex (taval (Plus a1 a2) s)
	proof (state)
	goal (4 subgoals):
	 1. ⋀Γ i. Γ ⊢ s ⟹ Ex (taval (Ic i) s)
	 2. ⋀Γ r. Γ ⊢ s ⟹ Ex (taval (Rc r) s)
	 3. ⋀Γ x. Γ ⊢ s ⟹ Ex (taval (V x) s)
	 4. ⋀Γ a1 τ a2. Γ ⊢ a1 : τ ⟹ (Γ ⊢ s ⟹
	Ex (taval a1 s)) ⟹ Γ ⊢ a2 : τ ⟹ (Γ ⊢ s
	⟹ Ex (taval a2 s)) ⟹ Γ ⊢ s ⟹ Ex (taval
	(Plus a1 a2) s)

Table 3: Comparison of Isabelle/jEdit State display and previous Isabelle/VSCode State display.

4.3.1 Correct Formatting

Isabelle uses an internal module called `Pretty` to manage the formatting of content in state and output panels. Specifically, this module is responsible for adding line breaks and indentation to these outputs if the panels are not wide enough to display something in a single line. The language server did not use the `Pretty` module, meaning it was the client's responsibility to add correct line breaks, which Isabelle/VSCode did not do. Instead, it used its default word wrap line breaks.

4.3.1.1 Isabelle's Internal XML

Isabelle internally represents almost all content with untyped XML trees. An `XML.Body` is a list of `XML.Tree`, which can be an `XML.Text` containing some text or an `XML.Elem` containing some markup and a child body. Ultimately, the text of an `XML.Body` is determined exclusively through such `XML.Text` instances, essentially the leaves of XML trees. On the other hand, the markup portion of `XML.Elem` stores all kinds of semantic information Isabelle might need about its child body. This information can include what type of highlighting the text should have (e.g. `text_skolem` for Skolem variables), whether it is *active markup* that should do something when the user clicks on it, or where to go when the user initiates a *Goto Definition*. These XML bodies are so fundamental to Isabelle's inner workings that even the theory code itself is XML internally.

Output panel content is also created as such an XML body. An Isabelle theory consists of many *commands*, each with some *command result*, which is an XML body. When the caret is above a command, the content of its command result is displayed on the output panel. State panel content is similarly just an XML body.

The `Pretty` module acts primarily on these XML bodies. There are two relevant functions within the module: `separate` and `formatted`. `separate` takes an XML body and adds an XML element with a separator markup between each tree in the body. `formatted` applies the aforementioned line breaks to the XML body, as such this function also requires knowledge about the width of the area the content should be displayed in, further referred to as the *margin*. To get correctly formatted panel output, as seen in Table 3, an XML body must first go through `separate` and then through `formatted`.

Lastly, Isabelle's `XML` module includes a `content` function, which reduces an XML body down into a string by stripping all tags.

4.3.1.2 Using `Pretty` for Isabelle/VSCode

The problem with supporting correct formatting of these panels to Isabelle/VSCode is that, for `Pretty` to correctly format some output, it needs to know the margin of the panel in ques-

tion. In Isabelle/jEdit, this is not an issue since the Isabelle/jEdit UI and the `Pretty` module all exist within the same Scala codebase. With Isabelle/VSCode, a different solution is necessary.

Once again, there are several possibilities that we considered:

1. Rebuild the `Pretty` module within the VSCode extension.
2. Give access to the `Pretty` API through LSP messages.
3. Notify the language server about the current margin of each panel.

Option 1 would have required fundamentally changing the format in which the language client receives state and output content. Previously, the language client got HTML source that it could just display inside a WebView. While generating this HTML source, information stored within the XML body's markup was used to create highlighting and hyperlinks, allowing the user to click on some elements and be transported to their definition (e.g. for functions). For the language client to correctly format this content, it would need access to Isabelle's underlying XML instead. Such a change would have also required significantly more work for every Isabelle language client implementation and was thus not pursued.

Option 2 is promising because it allows a language client to use the `Pretty` module the same way Isabelle/jEdit would. However, the problem of requiring Isabelle's underlying XML content remains. Whenever the content of a panel were to change, the following would need to happen:

1. The language server sends the panel's XML body to the client.
2. The client then sends a `Pretty/separate` request to call the `Pretty` module's `separate` function on the XML body. The server calls said function and returns the resulting XML body to the client.
3. The client sends a `Pretty/formatted` request to the server, the server calls `formatted` and sends the result back.
4. The client sends an `XML/content` request to the server, and the server calls `content` and sends the result back.

In step 3, the client can easily send over the current panel's margin in its request, thus solving the original problem. This solution requires much work from the client and introduces several roundtrips for each panel. However, it also allows for the greatest flexibility for the client and gives a clear distinction between UI and logic. The language server exists exclusively for the internal Isabelle logic, while correctly displaying the internal information is the client's sole responsibility. Because of this, when the UI is modified, like changing the width of the output panel, only steps 4 and 5 need to be repeated. The language server does not need to be informed about the change in the UI, the panel's content does not need to be newly generated

and sent to the client, and the client can handle exactly *when* reformatting of the content is necessary.

Option 3 requires the least amount of work for the language client. For this option, the client only needs to inform the server about the current panel margin, and the server can decide entirely on its own whether a re-send of the panel's content is necessary. From the perspective of a language client, it is thus the simplest solution because the language server does all the actual output logic. Additionally, a client that does not notify the server about the correct margins may not have correct formatting but can still easily display the content, keeping the barrier of entry for a new Isabelle client prototype low.

Due to its simplicity, Option 3 is the option we chose to implement. To this end, a new `Pretty_Text_Panel` module was added to Isabelle/VSCode, which implements the output logic, as well as two new notifications: `PIDE/output_set_margin` and `PIDE/state_set_margin`. Both output and state internally save one such `Pretty_Text_Panel` and simply tell it to refresh whenever the margin or content has changed. The `Pretty_Text_Panel` can then decide for itself if the actual content has changed and only send the appropriate notification to the client if it did.

While this solution has worked well in practice, note that one has to be careful how to send these margin updates. In VSCode, for example, panel width can only be polled in pixels. The Isabelle language server, however, requires the margin to be in symbols, i.e., how many symbols currently fit horizontally into the panel. Since Isabelle symbols are not necessarily all monospaced, this instigates a unique problem: How do we measure symbol width? In Isabelle/jEdit, this is solved by using the test string "mix", measuring its width, and dividing that width by 3. Thus, we did the same in Isabelle/VSCode.

Additionally, we added a limit on how often the margin update notifications are sent. If we were to send this notification for every change in panel width, we would send a notification for every pixel, which is extremely wasteful. In Neovim, by its terminal-based nature, neither of these problems exist because all characters have the same width and the width of a Neovim window only exists within discrete character counts.

4.3.2 Correct Font

Isabelle/jEdit uses a variant of the *DejaVu Sans Mono*[2] font called *Isabelle DejaVu Sans Mono*. This custom font can be built using the `isabelle component_fonts` Isabelle tool. It uses the *DejaVu Sans Mono* fonts as a base and adds special Isabelle symbols, like $\Longrightarrow$ and Γ [17]. As mentioned in Section 2.1.5, part of the reason why Isabelle/VSCode adds custom patches on top of VSCodium is to add these fonts into the Isabelle/VSCode binary. That way, the user

[2]https://dejavu-fonts.github.io/

can use the *Isabelle DejaVu Sans Mono* font family within buffers without needing to install these Isabelle fonts system-wide.

Unfortunately, these patched-in fonts are not available from within VSCode extensions, and the output and state panels in Isabelle/VSCode are handled by the Isabelle extension. Therefore, to support the correct fonts for the panels, we needed to additionally include the fonts into the extension.

The Isabelle VSCode extension is built with the `isabelle components_vscode_extension` tool. Aside from calling the relevant build systems to build the extension, this tool also generates static syntax definitions to support static syntax highlighting for the most common keywords in Isabelle/Isar. We extended this tool to additionally copy the *Isabelle DejaVu Sans Mono* font family into the extension build directory and add them to the extension's manifest file to include them in the extension's build. With this newly augmented extension in place, we could refer to these fonts from within the appropriate panels, the result of which can be seen in Figure 2.

4.4 Completions

The LSP specification defines how completions are supposed to be handled. For this, it defines a `textDocument/completion` request sent by the client [9]. In particular, the choice of *when* completions are triggered is up to the client. They typically trigger them automatically for certain trigger characters that were typed, although most of them also offer a keybind to manually request completions. The client may choose which characters count as such trigger characters, and many also offer the user the option to overwrite this set in their preferences. The language server can additionally send a list of trigger characters within its capabilities during the initialization stage described in Section 2.2.1.

Completions were already implemented in the Isabelle language server. The generation of the completions used the same internal system that Isabelle/jEdit uses, meaning the completions sent by the language server are identical to those offered by Isabelle/jEdit.

4.4.1 Completions Items Not Showing

Isabelle's completions concern primarily Isabelle symbols, although completions for other things, such as keywords, are also offered. We will differentiate Isabelle's completions into three different categories:

1. Symbol completions for abbreviations.
2. Symbol completions for ASCII representations.
3. Any other types of completions.

Curiously, in Isabelle/VSCode, only categories 2 and 3 worked; category 1 did not. In Neovim, categories 1 and 3 worked; category 2 did not. Since completions are part of the standard LSP spec, handling these completions is not done within the Isabelle extensions but instead in the standard language client implementations. Although, in Neovim, the standard language client does not offer completions out of the box, so an additional plugin is required, for which we used `nvim-cmp`[3].

The core problem is that completions in the LSP are meant to work in an additive fashion. If the user writes `con` in JavaScript, it should be possible to complete to `console`. Note that the text the user wrote originally is a prefix of the completed text. This is not necessarily the case for Isabelle: In Isabelle/jEdit, the user can complete `\Longright` to `\<Longrightarrow>`, which in turn gets displayed as $\Longrightarrow$. An abbreviation like `<=` should be completed to `\<le>`, which gets displayed as $\leq$. For the language server, it may even be that the completion should insert $\Longrightarrow$ or $\leq$ directly, depending on the options described in Section 5.3.1. These non-prefix completions are called *flex* completions, which the LSP does not intend to handle.

However, the Isabelle language server sends these flex completions anyway, and it is up to the client whether it wants to display them. Clients are supposed to implement a filter for these completions, and this filter is intentionally not defined in the LSP specification to allow consistency across languages within the editor. Therefore, different clients may use vastly different approaches for these filters, causing the aforementioned inconsistency issues.

There are discussions for supporting flex completions within the LSP[4]. For our case, we made use of the `filterText` string, which can optionally be provided for a completion item. This text is supposed to be used when filtering. Setting this text to be equivalent to what the user wrote so far (`\Longright` in our previous example) should force all filters to show the completion no matter what.

This fixed the original issue but introduced a new problem specifically in Isabelle/VSCode: Now, while writing out `\Longright`, the completion overlay would fade in and out at every second keystroke. This meant that, while `\Longright` could be completed, `\Longrigh` could not. This only manifested within Isabelle/VSCode and was no problem in Neovim. To solve this, we modified some tangential values in Isabelle/VSCode's settings and its Isabelle grammar, although we could not figure out the actual source of the problem. Isabelle/VSCode is based on VSCode version `1.70.1`, released on 11 August 2022. It may be that by upgrading, the problem disappears as well. Unfortunately, we could not verify this hypothesis due to time constraints.

We also found certain other language clients reacting rather strangely to provided `filterTexts`. In Sublime Text, for example, the `filterText` is not just used during filter-

[3]https://github.com/hrsh7th/nvim-cmp
[4]https://github.com/microsoft/language-server-protocol/issues/651

ing but is also displayed to the user as the completion item itself. This means that the user is shown the choice of completing \Long with the three options \Long, \Long or \Long, even though the completions are supposed to show \<Longleftarrow> ($\Longleftarrow$), \<Longrightarrow> ($\Longrightarrow$) and \<Longleftrightarrow> ($\Longleftrightarrow$).

4.4.2 Immediate Completions and Commit Characters

Isabelle marks certain completions as *immediate*. If a completion is the only one available and marked as immediate, Isabelle/jEdit inserts the completion without further user intervention. That way, a user can write ==> and have it instantly replaced by \<Longrightarrow> ($\Longrightarrow$).

To replicate this functionality in Isabelle/VSCode, we used the LSP's commitCharacters list, which can be added to a completion item. Microsoft [9] describes this list as: "An optional set of characters that when pressed while this completion is active will accept it first and then type that character."

In the language server, we now check if the completion item is unique (i.e. there are no other completions) and immediate. If so, we set commitCharacters to a list containing *most* writeable characters (the ASCII range 0x20: Space to 0x7E: ~ to be exact). That way, while the immediate insertion of the completion that Isabelle/jEdit does is not replicated, once the user types virtually any additional character, the completion is inserted.

Note that while VSCode supports this feature, that does not seem to be the case for all language clients. Both Neovim and Sublime Text ignored this list in our testing.

5 Enhancements and New Features

This chapter focuses on significant redesigns or additions to Isabelle/VSCode. Users of Isabelle/VSCode frequently reported syntax highlighting breaking, particularly when switching files. To address this, we implemented a feature that allows manual requests for decorations.

Additionally, we added the ability to disable HTML output for state and output panels, a feature primarily motivated by the requirements of the Neovim language client prototype mentioned in Section 1.1. We also provided more granular control for the language client regarding the handling of Isabelle symbols, which enhances the server's flexibility.

Lastly, we identified two features of Isabelle/jEdit where Isabelle/VSCode had no equivalent: Active markup and the ability to set Isabelle preferences through a UI settings menu. While these features required certain compromises, the implementations in Isabelle/VSCode prioritize simplicity and compatibility, even if they deviate from the exact functionality found in Isabelle/jEdit.

5.1 Decorations on File Switch

Previously, when switching theories within Isabelle/VSCode, the dynamic syntax highlighting would not persist. It was possible to get the highlighting to work again by changing the buffer's content; however, until this was done, it never recovered by itself. This was a problem when working on multiple theory files.

To understand how Isabelle/VSCode does dynamic syntax highlighting, we will first examine the structure of the `PIDE/decoration` notifications. Recall that the primary data of notifications is sent within a `params` field. This field contains two components in this case: A `uri` field with the relevant theory file's URI and a list of decorations called `entries`. Each of these entries then consists of a `type` and a list of ranges called `content`. The `type` is a string identifier for an Isabelle decoration type. This includes `text_skolem` for Skolem variables and `dotted_warning` for text that should have a dotted underline. Each entry in the `content` list is another list of four integers describing the line start, line end, column start, and column end of the range to which the specified decoration type should be applied. Listing 3 shows an example of what a `PIDE/decoration` message may look like.

```json
"jsonrpc": "2.0",
"method": "PIDE/decoration",
"params": {
    "uri": "file:///home/user/Documents/Example.thy",
    "entries": [
        {
            "type": "text_main",
            "content": [
                { "range": [1, 23, 1, 41] },
                { "range": [5, 10, 5, 11] }
            ]
        },
        {
            "type": "text_operator",
            "content": [
                { "range": [7, 6, 7, 7] }
            ]
        }
    ]
}
```

Listing 3: Example `PIDE/decoration` notification sent by the language server.

Since this is not part of the standard LSP specification, a language client must implement a special handler for such decoration notifications. Additionally, it was not possible to explicitly request these decorations from the language server. Instead, the language server would send new decorations whenever it deemed necessary, e.g., because the caret moved into areas of the text that have not been decorated yet or because the document's content has changed.

On the VSCode side, these decorations were applied via the `TextEditor.setDecoration` API function[1], which does not inherently cache these decorations on file switch. Thus, when switching theories, VSCode did not cache the previously set decorations, nor did the language server send them again, causing the highlighting to disappear.

There were two primary ways to fix this issue:

1. Implement caching of decorations manually in the VSCode extension.
2. Add the ability to request new decorations from the server and do so on file switch.

[1]https://code.visualstudio.com/api/references/vscode-api#TextEditor.setDecorations

The main advantage of option 1 is performance. If the client handles caching of decorations, then the server will not have to calculate the decorations anew (which is a rather expensive operation), nor will another round of JSON Serialization and Deserialization have to happen. However, the trade-off is that more work needs to be done on the client side, making new client implementations for other editors potentially harder.

Because of this, we instead introduced a new `PIDE/decoration_request` notification, sent by the client to explicitly signal to the server that it should send a `PIDE/decoration` notification back.

Note that this system is atypical for the LSP. The `PIDE/decoration_request` notification is, semantically speaking, a request and intends a response from the server. Nevertheless, from the perspective of the LSP, it is a unidirectional notification, while its response is also a unidirectional `PIDE/decoration` notification. We chose this approach for two reasons:

1. There was already precedent for such behavior in the Isabelle language server, specifically with `PIDE/preview_request` and `PIDE/preview_response` notifications.
2. The `PIDE/decoration` notification is not only sent after a request. The original automatic sending behavior that existed before is still present and has not been altered. If we were to implement `PIDE/decoration_requests` as an LSP request instead, this would only result in extra implementation work on the client side because a client would need to implement the same decoration application logic for both the `PIDE/decoration` notification and the `PIDE/decoration_request` response. By defining `PIDE/decoration_requests` as notifications, the client only needs to implement a singular handler for `PIDE/decoration` notifications, which automatically covers both scenarios simultaneously.

Later, we found that client-side caching was already implemented for the Isabelle VSCode extension; however, incorrectly so. The caching was implemented with the help of a JavaScript `Map`[2]. This `Map` used `URIs`[3] as keys and the content list from the decoration messages as values. However, the `URI` type does not explicitly implement an equality function, thus resulting in an inconsistent equality check where two `URI`s referencing the same file may not have passed an equality check. Switching the key to using string representations of the URIs fixed the issue. However, we decided to keep the `PIDE/decoration_request` notification. While it may not be used by Isabelle/VSCode directly, other Isabelle language client implementations may make use of this functionality.

[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
[3]https://code.visualstudio.com/api/references/vscode-api#Uri

5.2 Non-HTML Content for Panels

The output and state panels in Isabelle/VSCode were previously always sent as HTML content by the language server. The server sends `PIDE/dynamic_output` and `PIDE/state_output` notifications with output and state content, respectively. In this section, we will focus on content for the output panel. However, everything is almost equivalently done for state panel content.

The `PIDE/dynamic_output` notification only contained a single `content` value, a string containing the panel's content. As mentioned, this content used to be HTML content that Isabelle/VSCode displayed in a WebView. However, not every code editor can natively display HTML content, and there used to be no way for a language client to get pure text content instead.

We added a new Isabelle system option called `vscode_html_output`. If disabled, the language server skips the conversion to HTML and sends text content instead. However, this poses a new problem: The conversion to HTML added highlighting to the panel content. It takes the source XML body, extracts the relevant decoration markup, and uses it to generate equivalent HTML markup. Skipping this conversion and sending pure text instead also meant the language client got no highlighting within these panels. The Neovim language client prototype mentioned in Section 1.1 had this problem, as seen in Figure 4.

```
  2
  1 lemma aprogress: "Γ ⊢ a : τ ⇒ Γ ⊢ s ⇒ ∃v. taval a s
140 proof(induction rule: atyping.induct)
  1   case (Plus_ty Γ a1 t a2)
  2   then obtain v1 v2 where v: "taval a1 s v1" "taval
  3   show ?case
  4   proof (cases v1)
  5     case Iv
NORMAL  language-server  <  isabelle  64%  140:26
 1    proof (state)
  1 goal (4 subgoals):
  2  1. ⋀Γ i. Γ ⊢ s ⇒ Ex (taval (Ic i) s)
  3  2. ⋀Γ r. Γ ⊢ s ⇒ Ex (taval (Rc r) s)
  4  3. ⋀Γ x. Γ ⊢ s ⇒ Ex (taval (V x) s)
  5  4. ⋀Γ a1 τ a2.
  6        Γ ⊢ a1 : τ ⇒
  7        (Γ ⊢ s ⇒ Ex (taval a1 s)) ⇒
--OUTPUT--                                       1:1
```

Figure 4: Neovim Isabelle client without decorations in output panel.

```
  2
  1 lemma aprogress: "Γ ⊢ a : τ ⇒ Γ ⊢ s ⇒ ∃v. taval a s
140 proof(induction rule: atyping.induct)
  1   case (Plus_ty Γ a1 t a2)
  2   then obtain v1 v2 where v: "taval a1 s v1" "taval
  3   show ?case
  4   proof (cases v1)
  5     case Iv
NORMAL  language-server  <  isabelle  64%  140:26
 1    proof (state)
  1 goal (4 subgoals):
  2  1. ⋀Γ i. Γ ⊢ s ⇒ Ex (taval (Ic i) s)
  3  2. ⋀Γ r. Γ ⊢ s ⇒ Ex (taval (Rc r) s)
  4  3. ⋀Γ x. Γ ⊢ s ⇒ Ex (taval (V x) s)
  5  4. ⋀Γ a1 τ a2.
  6        Γ ⊢ a1 : τ ⇒
  7        (Γ ⊢ s ⇒ Ex (taval a1 s)) ⇒
--OUTPUT--                                       1:1
```

Figure 5: Neovim Isabelle client with decorations in output panel.

Decorations within state and output panels are quite important, as they provide more than just superficial visuals. There are many cases when writing Isabelle proofs where a single name is used for two or more individual variables. Isabelle also often generates its own names within proofs, and that generation may introduce further overlaps of identifiers. This may create goals like `x = x` that are not provable because the left `x` is a different variable than the right `x`. The only way to differentiate these variables in these cases is by their color. If the colors are missing, the goal will look like `x = x`.

To fix this, we added an optional `decorations` value to `PIDE/dynamic_output` and `PIDE/state_output` notifications, which is only given when HTML output is disabled. The form of this value is the same as the `entries` value of the `PIDE/decoration` notifications described in Section 5.1. That way, even when the server sends non-HTML panel content, the client can apply the given decorations to the respective panel. The result of adding this functionality to Neovim's language client prototype can be seen in Figure 5.

5.3 Symbol Handling

As described in Section 2.1.4, Isabelle utilizes its own UTF-8-Isabelle encoding to deal with Isabelle Symbols. It is important to distinguish between 3 different domains:

Physical This is the contents of the file. It is essentially a list of bytes that need to be interpreted. Certainly, a theory file contains text, meaning the bytes represent some list of symbols. However, even then, the exact interpretation of the bytes can vary depending on the encoding used. For example, the two subsequent bytes `0xC2` and `0xA5` mean different symbols in different encodings. If the encoding is *UTF-8*, those two bytes stand for the symbol for Japanese Yen `¥`. If, however, the encoding is *ISO-8859-1 (Western Europe)*, the bytes are interpreted as `Â¥`. The file itself does not note the supposed encoding; without knowing the encoding, the meaning of a file's contents may be lost.

Isabelle System This is where the language server lives. Here, an Isabelle symbol is simply an instance of an internal struct whose layout is outlined in Table 1.

Editor This is where the language client lives. When opening a file in a code editor, it gets loaded into some internal structure the editor uses for its text buffers. During this loading, the editor will need to know the encoding to use, which will also affect what bytes the editor will write back to disk.

When using Isabelle/jEdit and loading a theory with the UTF-8-Isabelle encoding, the bytes of the file will be interpreted as UTF-8. Additionally, ASCII representations of symbols will be interpreted as their UTF-8 counterparts. When writing back to disk, this conversion is done in reverse; thus, if all symbols within a theory are valid Isabelle symbols, which all have ASCII representations, a file saved with the UTF-8-Isabelle encoding can be viewed as plain ASCII.

With Isabelle/VSCode, we get the additional problem of the Isabelle system not having direct access to our editor's buffer. As mentioned in Section 2.1.5, Isabelle patches VSCodium to include a new UTF-8-Isabelle encoding, so loading the file works virtually the same as in Isabelle/jEdit. However, the language server must still obtain the contents of the file.

Recall from Section 4.1 that the LSP specification defines multiple notifications for text document synchronization, like the `textDocument/didOpen` and `textDocument/didChange` notifications, both containing data that informs the language server about the contents of a file.

We will focus on `textDocument/didOpen` for now. This notification's `params` field contains a `TextDocumentItem` instance, whose interface definition is seen in Listing 4.

```typescript
interface TextDocumentItem {
    uri: DocumentUri;
    languageId: string;
    version: integer;
    text: string;
}
```

Listing 4: `TextDocumentItem` interface definition [9].

The most relevant data is the `text` field, which contains the content of the entire text document that was opened. Aside from the header, which is plain ASCII, the JSON data sent between client and server is interpreted as UTF-8; thus, the `text` string is also interpreted as UTF-8 content. The exact content of this string depends on the text editor. In Isabelle/VSCode, thanks to the custom UTF-8-Isabelle encoding, the language server will receive the full UTF-8 encoded content of the file (i.e. $\Longrightarrow$ instead of \<Longrightarrow>). However, this may not be the case for another editor. Thankfully, the Isabelle system internally deals with all types of Isabelle Symbol representations equally, so the editor is free to mix and match whichever representation is most convenient for it.

Every code editor may handle Isabelle symbols differently. Some editors may be able to add custom encodings; others may not. For example, in the Neovim code editor, it is possible to programmatically change how certain character sequences are displayed to the user using a feature called *conceal*.[4] Through this feature, Neovim can have the ASCII representation (\<Longrightarrow>) within the file and buffer and still display the Unicode representation ($\Longrightarrow$) to the user without the need for a custom encoding. All in all, the language server should not make assumptions about the implementation details of Isabelle symbols in the language client.

5.3.1 Symbol Options

There are many messages sent from the server to the client containing different types of content potentially containing Isabelle symbols: `window/showMessage` notifications sent by the server asking the client to display a particular message, text edits sent for completions, text inserts sent for code actions, content sent for output and state panels, and many more.

Previously, there was a single Isabelle option called `vscode_unicode_symbols`, which was supposed to control whether these messages sent by the server should send Isabelle symbols in their Unicode or ASCII representations. However, this option only affected a few messages

[4]https://neovim.io/doc/user/options.html#'conceallevel'

(like hover information and diagnostics). Things like completions were hard-coded to always use Unicode, as that is what Isabelle/VSCode requires.

When viewing Isabelle/VSCode in its entirety, this is not a problem. If the VSCode Isabelle client expects Unicode symbols in certain scenarios and the language server is hard-coded to do so, then it works for Isabelle/VSCode. However, this is a problematic limitation once one moves to a different client. In the case of Neovim's *conceal* feature, it would be desirable to have messages sent by the server use ASCII for consistency.

Another important consideration is that even if Neovim wants ASCII representations of symbols within the theory file, this may not necessarily be true for output and state panels. While the server sends many different types of content, it can generally be grouped into two categories: Content that is only supposed to get *displayed* and content that is supposed to be *placed* within the theory file.

To this end, we replaced the original `vscode_unicode_symbols` option with two new options: `vscode_unicode_symbols_output` for *displayed* content and `vscode_unicode_symbols_edits` for *placed* content. Additionally, we made use of these new options in the respective places within the language server code base, removing the previously hard-coded values.

5.3.2 Symbols Request

The list of existing Isabelle symbols is not static; a user may augment this list in a `$ISABELLE_HOME_USER/etc/symbols` file [18:§2.2]. The issue is that previously, there was no way for a language client to get information about which symbols exist. So, even if it is possible to hard-code the *default* set of Isabelle symbols into an Isabelle language extension, that would not be correct in light of user additions.

Isabelle/VSCode uses such a hard-coded list of symbols. This list is added to the custom UTF-8-Isabelle encoding while building the patched VSCodium. It also only includes the default set of symbols Isabelle offers out of the box; it does not include custom user additions. As this list is hard-coded, any change in the list of symbols would also require recompiling Isabelle/VSCode. This is different from Isabelle/jEdit, where the code of the UTF-8-Isabelle encoding exists within Isabelle/Scala and, therefore, has access to the complete list of symbols.

In order to eliminate the need for hard-coded lists of symbols for language clients, we added a `PIDE/symbols_request` request. When this request is sent, the language server responds with a list of all defined symbols. Note that, at the time of writing, this list *also* only includes the default set without user additions in order to be in line with the set that is used by Isabelle/VSCode. This may be worth changing in the future, which we will discuss in Chapter 8.

5.3.3 Symbols Conversion

Another issue is that different language clients may want different symbol representations within files. While the typical way of handling symbols in Isabelle is to have symbols in their ASCII representation within files, some editors may want Unicode representations instead. In order for the client to freely choose which of the two it wants to use, it would be helpful if there were some way for it to convert the symbols from one representation into the other. Within Isabelle/Scala, this is easily done with the help of the internal `Symbol` module, and to pass this functionality to the language client, we added a new `PIDE/symbols_convert_request` request.

This request gets a string it should convert, as well as whether symbols in it should be converted into Unicode or ASCII representations. The language server then converts the symbols and sends the converted string back as its response. An example conversion request and response can be seen in Listing 5.

Request

```json
1 "jsonrpc": "2.0",
2 "id": 58,
3 "method": "PIDE/symbols_convert_request",
4 "params": {
5     "text": "A \<Longrightarrow> B",
6     "unicode": true
7 }
```

Response

```json
"jsonrpc": "2.0",
"id": 58,
"result": {
    "text": "A ⟹ B"
}
```

Listing 5: `PIDE/symbols_convert_request` example request and response.

Allowing the client to request the conversion for any string enables it to offer more flexible functionality. For example, an Isabelle language extension may allow the user to select an area of the text and only convert the selected area instead of the whole file.

Both the `PIDE/symbols_request` and `PIDE/symbols_convert_request` requests are not currently used by Isabelle/VSCode. They are only offered by the language server for other language clients and have already seen use in them. For example, our current Neovim Isabelle client prototype supports a `SymbolsConvert` command to convert the symbols in the current buffer.

5.4 Code Actions for Active Markup

One Isabelle/jEdit feature that was missing entirely in Isabelle/VSCode is Isabelle's *active markup*. Active markup generally describes parts of the theory, state, or output content that is clickable. The action taken when the user clicks on an active markup can vary, as many different kinds of active markup exist. One type of active markup the user will probably come across frequently is the so-called *sendback* markup. This type of markup appears primarily in the output panel, and clicking on it inserts its text into the source theory. It appears, for example, when issuing a `sledgehammer` command.[5] When this command finds a proof, it is displayed in the output panel with a gray background (green when hovered with the mouse). The user can then click on the suggested proof, and Isabelle inserts it into the document. This example can be seen in Figure 6. As mentioned, there are other types of active markup as well, but those are out of the scope of this thesis.

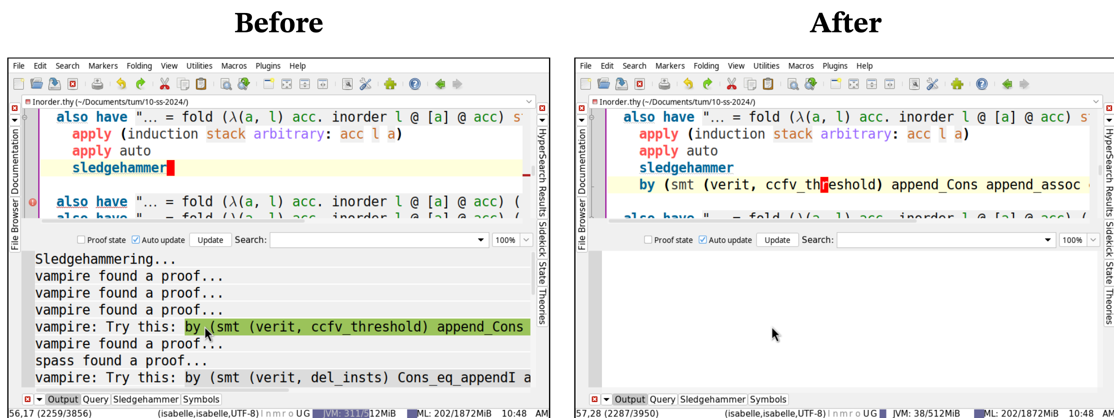

Figure 6: Active markup in Isabelle/jEdit when using sledgehammer, before and after clicking on sendback markup.

Unlike other features discussed in this work, active markup is a concept that has no comparable feature within typical code editors. Clicking on parts of code may exist in the form of *Goto Definition* actions or clicking on hyperlinks, but inserting things from some output panel into the code is unique. Hence, there is also no existing precedent on handling this type of interaction within the LSP specification. Because of this, the first question that needed to be answered is how we intend to tackle this problem in terms of user experience. That is, should the Isabelle/VSCode implementation work the same way as it does in Isabelle/jEdit (i.e. by clicking with the mouse), or should the interaction work completely differently?

[5]The `sledgehammer` command is an Isabelle command that calls different external automatic theorem provers in hopes of one of them finding a proof. Isabelle then translates the found proof back into an Isabelle proof.

There exist two major problems when trying to replicate the user experience of Isabelle/jEdit:

1. For the sake of accessibility, it is usually possible to control VSCode entirely with the keyboard. In order to retain this property, we decided it should also be possible to interact with active markup entirely with the keyboard.
2. It would need a completely custom solution for both the language server and language client, increasing complexity and the barrier of entry for new Isabelle IDEs. We would potentially need to reimagine the way that output panel content is sent to the client, and it would be very difficult to expand the functionality to other types of active markup that live within the theory instead.

Instead, we decided to deviate from Isabelle/jEdit and utilize existing LSP features where possible. Luckily, the LSP spec defines a concept called *code actions*, which are suitable for active markup.

The intended use case of code actions is to support more complicated IDE features acting on specific ranges of code that may result in beautifications or refactors of said code. For example, when using the `rust-analyzer` language server,[6] which serves as a server for the Rust programming language,[7] it is possible to use a code action to fill out the arms of a match expression, an example of which can be seen in Figure 7.

Figure 7: `rust-analyzer`'s "Fill match arms" code action in Sublime Text.

The main advantage of using code actions is that they are a part of the standard LSP specification, meaning most language clients support them out of the box. If the Isabelle language server supports interacting with active markup through code actions, there is no extra work necessary for the client.

To initiate a code action, the language client sends a `textDocument/codeAction` request to the server whose content can be seen in Listing 6. The request's response then contains a

[6] https://rust-analyzer.github.io/
[7] https://www.rust-lang.org/

list of possible code actions. Each code action may be an *edit*, a *command*, or both. For our use case of supporting *sendback* active markup, which only inserts text, the *edit* type suffices. Although to support other types of active markup, the *command* type may become necessary.

The `range` data sent in the code action request is the text range from which the client wants to get code actions. Code actions are quite dependent on caret position. Different parts of the document may exhibit different code actions. Most of the time, the `range` just includes the caret's current position. However, most clients will also allow the user to make a selection or even create multiple carets and request code actions for the selected range.

```typescript
interface CodeActionParams {
    textDocument: TextDocumentIdentifier;
    range: Range;
    context: CodeActionContext;
}
```

Listing 6: `CodeActionParams` interface definition [9].

5.4.1 Implementation for the Isabelle Language Server

When the Isabelle language server receives a code action request, the generation of the code actions list for its response is roughly done in these four steps:

1. Find all Isabelle/Isar commands within the given `range`.
2. Get the command results of all these commands.
3. Extract all sendback markup out of these command results.
4. Create LSP text edit JSON objects, inserting the sendback markup's content at the respective command's position.

Once the list of these code actions is sent to the language client, the server's work is done. The LSP text edit objects exist in a format standardized in the LSP, so the actual execution of the text edit can be done entirely by the client.

We also considered how to deal with correct indentation for the inserted text. In Isabelle/jEdit, when a sendback markup gets inserted, the general indentation function that exists in Isabelle/jEdit is called right after to correctly indent the newly inserted text. Since this internal indentation function uses direct access to the underlying jEdit buffer, we could not easily use this function from the language server. However, simply ignoring the indentation altogether results in a subpar user experience.

A proper solution would be to reimplement Isabelle/jEdit's indentation logic for the language server, which we will discuss in Chapter 8 as it exceeds the scope of this thesis. For our con-

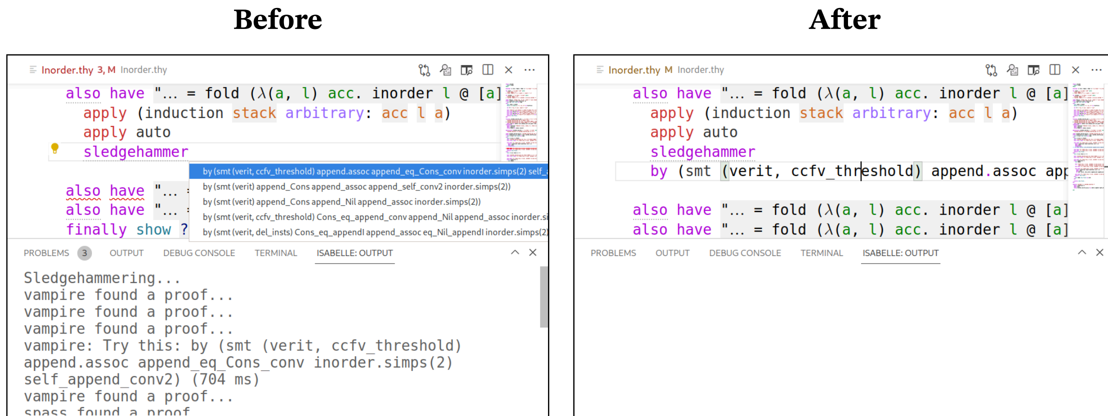

Figure 8: Active markup in Isabelle/VSCode when using sledgehammer,
before and after accepting code action. Code action initiated with “`Ctrl+.`”.

tribution, the language server instead just copies the source command’s indentation to the inserted text. This may give slightly different indentations compared to Isabelle/jEdit however, the result is acceptable in practice.

An example of the resulting implementation for Isabelle/VSCode can be seen in Figure 8.

5.5 Isabelle System Options as VSCode Settings

Isabelle has many options that can be set to adjust different aspects of the interactive sessions. For example, the option `editor_output_state` defines whether the current state should additionally be printed within the output panel.

The options, including their default values, are generally defined within `etc/options` files scattered throughout the codebase. Users can overwrite these options by adding respective entries into a `$ISABELLE_HOME_USER/etc/preferences` file. When using Isabelle/jEdit, the user will also find many of these options within Isabelle/jEdit’s settings, as seen in Figure 9. These settings and the content of the `preferences` file are kept in sync [18].

The Isabelle language server offers one additional way of overwriting Isabelle system options: Via CLI arguments. When starting the Isabelle language server with `isabelle vscode_server`, one may add additional option overwrites with `-o NAME=VAL` arguments. The order of priority for Isabelle options for the language server is then as follows:

1. CLI arguments.
2. User preferences defined in the `preferences` file.
3. Isabelle defaults.

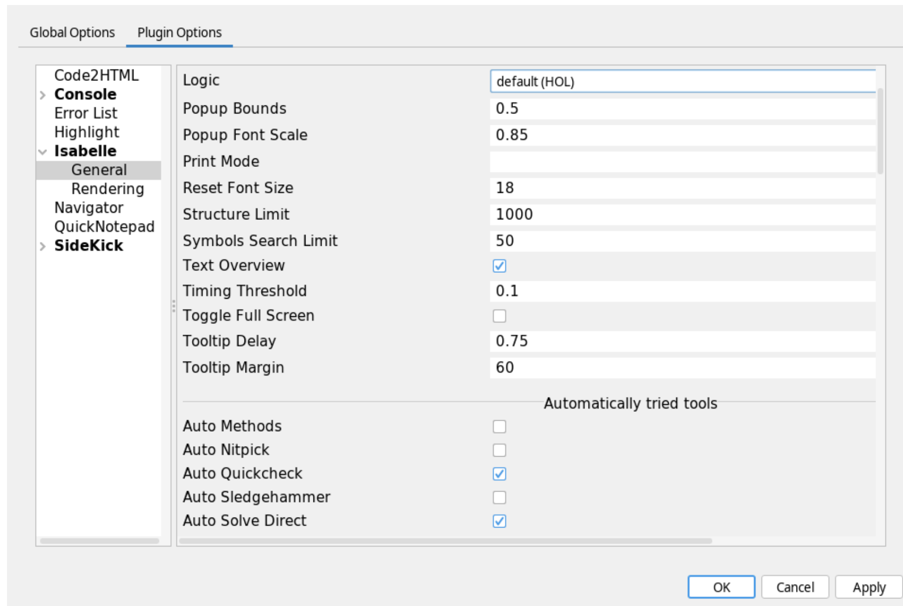

Figure 9: Isabelle options inside of Isabelle/jEdit.

The same is true for Isabelle/VSCode. When starting Isabelle/VSCode with `isabelle vscode`, the user can add option overwrites as CLI arguments. However, previously, there was no method to set Isabelle options through Isabelle/VSCode's UI. We wanted to alleviate this discrepancy between Isabelle/VSCode and Isabelle/jEdit by adding relevant options to Isabelle/VSCode to its settings.

Ideally, the settings in Isabelle/VSCode would be kept in sync with the user's `preferences` file, as Isabelle/jEdit does. However, to do so, we would need to be able to parse and understand the `preferences` file from within the VSCode extension, yet this file is supposed to be managed exclusively by Isabelle/Scala. Therefore, we instead chose to use the Isabelle/VSCode settings as pure overwrites.

5.5.1 Passing Options from VSCode to the Language Server

Isabelle/VSCode itself has no use for Isabelle system options. These options are used by Isabelle internally, not by the code editor. That means that only the language server needs to know the options set by the user.

When using Isabelle/VSCode, the user does not manually start the language server. Instead, they run `isabelle vscode`, which starts an instance of Isabelle's patched VSCodium with an Isabelle extension installed, which starts the language server once the user opens an Isabelle theory.

The `isabelle vscode` command optionally takes option overwrites as CLI arguments and converts these into an environment variable called "`ISABELLE_VSCODIUM_ARGS`", such that the extension can read this environment variable later. On top of that, the extension used to add a few hard-coded options that are needed for Isabelle/VSCode to function properly. This set of options is finally given to the language server as CLI arguments. Figure 10 shows this process.

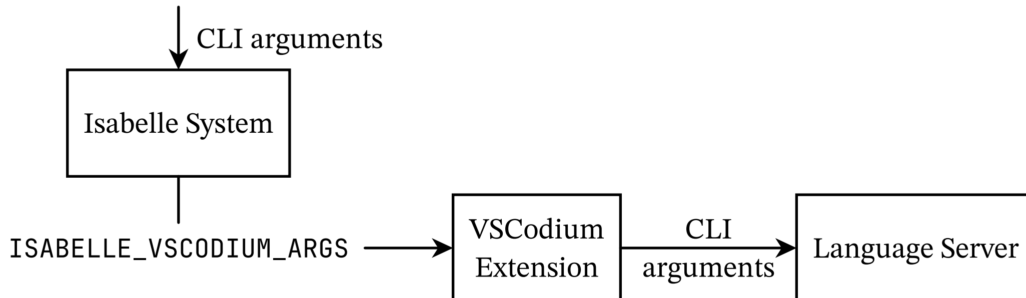

Figure 10: Previous passing of option overwrites.

The language server gets its option values by first taking the Isabelle default, overwriting those with whatever the user specified in their `preferences` file, and overwriting those again with whatever was given as CLI arguments.

In order to additionally consider VSCode settings, we must add them from within the extension, as we do not have access to the VSCode settings from within the language server nor the original Isabelle process that starts VSCodium. Therefore, the only part we can actively influence with VSCode settings is the CLI arguments sent to the server by the extension. Here, we must decide whether the user's CLI arguments or VSCode settings take precedence. This limits the possible order of priority to two different possibilities, seen in Table 4.

Option 1	**Option 2**
1. CLI	1. VSCode Settings
2. VSCode Settings	2. CLI
3. Preferences	3. Preferences
4. Defaults	4. Defaults

Table 4: Different possibilities for Isabelle system option priority order.

Of these, we chose to proceed with option 1, as CLI option overwrites are more explicit than the user's VSCode settings and should be prioritized.

Figure 11 shows the new flow of Isabelle options when starting Isabelle/VSCode. The VSCode Isabelle extension has access to both the CLI arguments given to the `isabelle vscode` command, and whatever settings are set in VSCode.

These two get merged, prioritizing the options within the `ISABELLE_VSCODIUM_ARGS` variable, and this merged set of option overwrites gets passed to the language server.

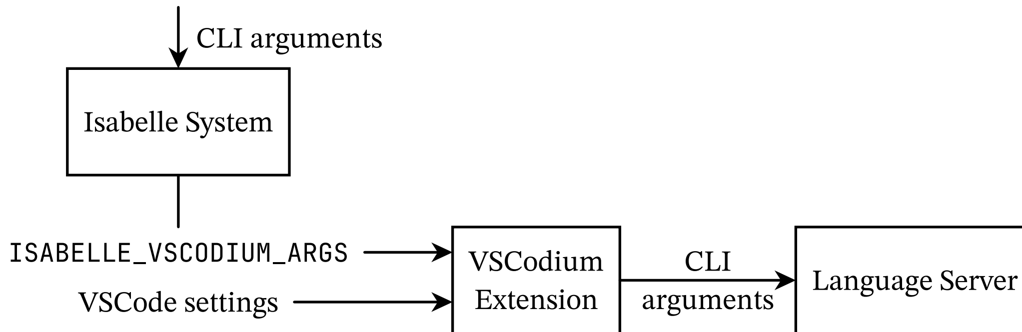

Figure 11: Passing of option overwrites with VSCode settings.

5.5.2 Option Types

Isabelle system options all have a type: `string`, `int`, `real`, or `bool`. Using the same type for the VSCode extension's settings might be tempting. However, since we ultimately want the user to be able to *overwrite* these options, this is not optimal. Taking the `editor_output_state` as an example, which is of type `bool`, the respective VSCode setting would be of type `boolean`. In the UI, this would make it a checkbox, giving it two states. However, we actually need three states: Do not overwrite, `off` and `on`. If the type of the VSCode setting were `boolean` with a default value of `off`, there would be no difference between the user not wanting VSCode to overwrite their user preferences and wanting to overwrite it with `off`.

Instead, we made all Isabelle/VSCode settings of type `string`. For Isabelle options of type `bool`, the respective VSCode setting will have possible values `""`, `"off"`, and `"on"`, meaning don't-overwrite, overwrite with `off`, and overwrite with `on`, respectively. For Isabelle options of any other type, the empty string `""` means do not overwrite, and any other value is the value the option should be overwritten with.

This system has another advantage for numerical options: The types of VSCode settings are just JavaScript types. Isabelle differentiates between `real` and `int` options, but JavaScript only has a singular `numeric` type. If the VSCode option were to take such `numeric` values, the extension would need to convert this value to a string to pass it to the language server as a CLI argument. By keeping it a string from the start, we skip potential conversion errors that may occur otherwise.

5.5.3 Extending Isabelle/VSCode's Settings

Many Isabelle options are annotated with a tag, thus creating groupings of similar options. For example, the `content` tag includes options such as `names_long`, `names_short`, and `names_unique`, which affect how names (like function names) are printed within output and state panels.

Many of the options Isabelle exposes are not relevant for Isabelle/VSCode. For example, one of the option tags available is the `jedit` tag, which, as the name suggests, includes options relevant specifically to Isabelle/jEdit.

The first options we deemed relevant are those specifically designed for Isabelle/VSCode and the language server. These options are defined within `src/Tools/VSCode/etc/options`. We added and assigned a new `vscode` option tag to these options to allow easy access.

The second set of relevant options are options tagged with the aforementioned `content` tag.

The third set are manually chosen options helpful for Isabelle/VSCode but not included in the previous two tags. The list of options we chose is:

- `editor_output_state`
- `auto_time_start`
- `auto_time_limit`
- `auto_nitpick`
- `auto_sledgehammer`
- `auto_methods`
- `auto_quickcheck`
- `auto_solve_direct`
- `sledgehammer_provers`
- `sledgehammer_timeout`

To add custom settings to VSCode with a VSCode extension, one can add a `contributes.configuration` entry into the extensions `package.json` file [8]. Since the options available in the given tags may change in the future, simply adding them manually to the `package.json` file was unsatisfactory. Instead, the options are dynamically added while building the extension with `isabelle component_vscode_extension`. To do so, the `package.json` file includes a `"ISABELLE_OPTIONS": {},` marker, which is replaced with the appropriate JSON format of the given options by the Isabelle system during build.

Additionally, we gave the options that were previously hard-coded into the extension a respective default value during this build process instead. That way, the user can change these settings if they want, which was not possible before.

6 Evaluation

Our original goal included enhancing the Isabelle language server's flexibility and enabling the development of language clients for other code editors. In order to assess the server's new flexibility with our advancements, we built two prototype language clients for the Neovim and Sublime Text code editors.

6.1 Neovim

Originally the motivation behind this thesis, our Neovim language plugin has seen significant usability improvements. We found the greatest boon in user experience to be the highlighting in output and state panels. Especially state content has become significantly easier to read thanks to proper highlighting.

While Isabelle/VSCode and our Neovim Isabelle plugin still differ in the details, they are now comparable in functionality. Other users have also already used the plugin with success, and it can be found publicly on GitHub.[1]

6.2 Sublime Text

Utilizing the Sublime LSP package[2], we found that a working Sublime Text Isabelle language package was doable in only ~200 lines of Python code. This package supports dynamic highlighting, a working output panel with highlighting, active markup via code actions, completions for symbols, and conversion of symbols from ASCII representations to Unicode—all that on top of more typical LSP features like diagnostics. Although still rough around the edges, we found this package already usable for basic theories.

[1] https://github.com/Treeniks/isabelle-lsp.nvim
[2] https://lsp.sublimetext.io/

7 Conclusion

The primary objective of this thesis was to refine and enhance the functionality of Isabelle/VSCode, trying to mimic Isabelle/jEdit while also adding features that make supporting Isabelle in other code editors possible. We discussed different ways of implementing the unique features required by Isabelle into a language server/client setup.

In the process, we discovered that, by migrating away from Isabelle/jEdit's exact functionality, certain features can be built without needing any custom handlers in the language client, like how we used LSP code actions to implement Isabelle's active markup.

We also found existing issues in the language server, like frequent desyncs of theory contents or limited options regarding symbol handling, and implemented improvements to those areas to make the language server more robust. That way, we found that usage of the language server outside of Isabelle/VSCode became viable, like with our Neovim and Sublime Text language client prototypes.

Lastly, it became apparent that great care must be taken in the handling of Isabelle symbols, as it consistently proved challenging to deal with. We extended the language server to allow for more granular control over how symbols are sent, making it more flexible. Thus, a language client now has more freedom when choosing how to deal with these symbols.

All in all, Isabelle is a uniquely monolithic system. This is both its greatest strength and its greatest weakness. It gives it the power to build a large set of tools, which are all consistent with one another, and do things that seem magical. However, once you want to venture outside its system, it becomes all the more difficult to integrate. Nevertheless, our work shows that it is not impossible.

8 Future Work

Isabelle/VSCode is still far from perfect, and there are many things that future projects may improve upon. To name just a few:

More Clients Now that the language server offers enough flexibility for other clients, it would be interesting to build Isabelle support for many more editors.

- **Sublime Text** We already mentioned a working prototype for Sublime Text. However, this prototype needs a lot more work before it can be reliably used. It does not retain decorations when switching files; in fact, it often completely breaks when switching files. There is also no way to open state panels.

- **Zed**[1] Developed by the creators of Atom[2] and Tree-sitter[3], Zed is a relatively new code and text editor. At the time of writing, the capabilities of extensions are still fairly limited, although language extensions, in particular, are now possible [2]. It could be interesting to investigate a possible Isabelle language extension for Zed.

- **Pulsar**[4] While the once-popular code editor Atom has been abandoned, a community-led fork called Pulsar has emerged. While the user base is pretty small, it is a very extensible code editor for which an Isabelle client is certainly conceivable.

- **Helix**[5] A modern terminal-based text editor with support for the LSP out of the box. Its plugin system is still in the works[6], so an Isabelle client is currently not realistically doable. However, once it is, it may be a nice candidate.

Minor Advancements Many further small improvements could be made to the language server. These are things that are relatively easy and obvious to build but just require some additional work.

- **Multiple State Panels for VSCode** We already mentioned in Section 4.2 that Isabelle/VSCode does not currently support using multiple state panels. This would require some work in the Isabelle VSCode extension. However, since most users tend to use only one auto-updated state panel, this feature is not of particularly high priority.

- **Update VSCode Version** As previously noted, Isabelle/VSCode is based on a rather old version of VSCode: `1.70.1` released on 11 August 2022. Because Isabelle adds its own patches, upgrading Isabelle/VSCode to a newer version is not entirely trivial. Users may run into compatibility issues with newer extensions in the meantime.

Custom Symbols The `PIDE/symbols_request` notification currently does not relay custom symbols defined by the user. Instead, it only sends the default set of symbols defined in the Isabelle distribution. This makes it virtually impossible for a language client to support such custom symbols and should be changed. Ideally, Isabelle/VSCode would also support user symbols. However, that is much more difficult to achieve due to the use of the UTF-8-Isabelle encoding. In this context, it may be worth exploring other ideas for symbol handling in Isabelle/VSCode that do not rely on a custom encoding.

More Active Markup Currently, the only active markup supported with code actions is sendback active markup. However, there are also other kinds of active markup: `browser`, `theory_exports`, and `simp_trace_panel`, to name a few. The latter, for example, is supposed to open a window that shows Isabelle's simplifier trace. Some of these other active markup types may also be possible to support with the language server, but further work is needed.

Indentation Support As mentioned in Section 4.3.1, Isabelle/jEdit has an internal function to automatically indent a theory document. This function uses jEdit buffers and other internal data and is not usable outside of Isabelle/jEdit. However, the actual indentation logic could be extracted, and the feature could be added to the language server. The LSP specifies a `textDocument/formatting` client request for formatting source files, which could be used here. Once such an indentation function is available within the language server, correct indentation for sendback active markup would also be possible.

Updating Options Through Language Server One potentially useful API to offer a client is the permanent changing of Isabelle system options. That way, the client could send notifications to the server, which then writes the appropriate entries into the user's `preferences` file. It could also offer a request to get the value of certain options. That way, synchronization of Isabelle/VSCode settings and Isabelle preferences would become feasible.

[1]https://zed.dev/
[2]https://atom-editor.cc/
[3]https://tree-sitter.github.io/tree-sitter/
[4]https://pulsar-edit.dev/
[5]https://helix-editor.com/
[6]https://github.com/helix-editor/helix/pull/8675

List of Figures

Figure 1: JEdit with both output and state panels open. Output on the bottom, state on the right. 4
Figure 2: VSCode with both output and state panels open. Output on the bottom, state on the right. 6
Figure 3: Visualization of the LSP initialization handshake. 8
Figure 4: Neovim Isabelle client without decorations in output panel. 23
Figure 5: Neovim Isabelle client with decorations in output panel. 23
Figure 6: Active markup in Isabelle/jEdit when using sledgehammer, before and after clicking on sendback markup. 28
Figure 7: `rust-analyzer`'s "Fill match arms" code action in Sublime Text. 29
Figure 8: Active markup in Isabelle/VSCode when using sledgehammer, before and after accepting code action. Code action initiated with "`Ctrl+.`". 30
Figure 9: Isabelle options inside of Isabelle/jEdit. 31
Figure 10: Previous passing of option overwrites. 33
Figure 11: Passing of option overwrites with VSCode settings. 33

List of Tables

Table 1: Symbol data of $\Longrightarrow$. 5
Table 2: General LSP message structure. 8
Table 3: Comparison of Isabelle/jEdit State display and previous Isabelle/VSCode State display. 13
Table 4: Different possibilities for Isabelle system option priority order. 33

List of Listings

Listing 1: Example Isabelle theory with Isabelle/Isar proof. 3
Listing 2: `DidChangeTextDocumentParams` interface definition [9]. 11
Listing 3: Example `PIDE/decoration` notification sent by the language server. 21
Listing 4: `TextDocumentItem` interface definition [9]. 25
Listing 5: `PIDE/symbols_convert_request` example request and response. 27
Listing 6: `CodeActionParams` interface definition [9]. 30

Bibliography

[1] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. 2006. A formally verified proof of the prime number theorem. Retrieved from https://arxiv.org/abs/cs/0509025

[2] Max Brunsfeld. 2024. Extensible Language Support in Zed - Part 1. Retrieved August 12, 2024 from https://zed.dev/blog/language-extensions-part-1

[3] Richard Dedekind. 1887. Ähnliche (deutliche) Abbildung und ähnliche Systeme. In *Gesammelte mathematische Werke*, Robert Fricke, Emmy Noether and Øystein Ore (eds.). Friedr. Vieweg & Sohn Akt.-Ges., 447–449. Retrieved from http://resolver.sub.uni-goettingen.de/purl?PPN23569441X

[4] Kevin Elphinstone, Gerwin Klein, and Rafal Kolanski. 2006. Formalising a High-Performance Microkernel. In *Verified Software: Theories, Tools and Experiments (Microsoft Research Technical Report MSR-TR-2006-117)*, August 2006. 1–7.

[5] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. 2021. The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions. *Electronic Proceedings in Theoretical Computer Science* 338, (August 2021), 3–18. https://doi.org/10.4204/eptcs.338.3

[6] Gerwin Klein and Tobias Nipkow. 2006. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.* 28, 4 (July 2006), 619–695. https://doi.org/10.1145/1146809.1146811

[7] Alwin Korselt. 1911. Über einen Beweis des Äquivalenssatzes. (Mit 1 Figur im Text). *Mathematische Annalen* 70, (1911), 294–296. Retrieved from http://eudml.org/doc/158498

[8] Microsoft. Visual Studio Code Extension API. Retrieved August 1, 2024 from https://code.visualstudio.com/api

[9] Microsoft. 2022. Language Server Protocol Specification - 3.17. Retrieved August 1, 2024 from https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

[10] Klaus Miesenberger, Walther Neuper, Bernhard Stöger, and Makarius Wenzel. 2023. Towards an Accessible Mathematics Working Environment Based on Isabelle/VSCode.

Electronic Proceedings in Theoretical Computer Science 375, (March 2023), 92–111. https://doi.org/10.4204/eptcs.375.8

[11] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, 2021. Springer International Publishing, Cham, 625–635. Retrieved from https://lean-lang.org/papers/lean4.pdf

[12] Denis Paluca. 2021. Isabelle/VSCode: Editor Improvements and Prover IDE integrations. Retrieved from https://www21.in.tum.de/students/past/vscode_plugin_improvements/assets/Isabelle_VSCode_Thesis.pdf

[13] Lawrence C. Paulson. 2000. Isabelle: The Next 700 Theorem Provers. Retrieved from https://arxiv.org/abs/cs/9301106

[14] Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. 2019. From LCF to Isabelle/HOL. *Form. Asp. Comput.* 31, 6 (December 2019), 675–698. https://doi.org/10.1007/s00165-019-00492-1

[15] Ernst Schröder. 1898. Über zwei Definitionen der Endlichkeit und G. Cantor'sche Sätze. *Abhandlungen der Kaiserlichen Leopoldinisch-Carolinischen Deutschen Akademie der Naturforscher 71*, 303–362. Retrieved from https://www.biodiversitylibrary.org/item/45265

[16] Makarius Wenzel. 2017. *Isabelle/VSCode in January 2017*. Retrieved from https://sketis.net/wp-content/uploads/2017/01/isabelle-vscode-jan-2017.pdf

[17] Makarius Wenzel. 2018. [isabelle-dev] NEWS: Isabelle DejaVu fonts. Retrieved August 12, 2024 from https://mailman46.in.tum.de/pipermail/isabelle-dev/2018-November/008132.html

[18] Makarius Wenzel. 2024. Isabelle/jEdit. Retrieved from https://isabelle.in.tum.de/doc/jedit.pdf

[19] Makarius Wenzel. 2024. The Isabelle/Isar Reference Manual. Retrieved from https://isabelle.in.tum.de/doc/isar-ref.pdf

[20] Makarius Wenzel. 2024. The Isabelle System Manual. Retrieved from https://isabelle.in.tum.de/dist/Isabelle2024/doc/system.pdf