Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Improvements in Ciao Prolog's Development Environment: Developing A Rich Development Environment for Ciao on Visual Studio Code

Autor: Marco Ciccalè Baztán
Tutor: José Francisco Morales Caballero
Cotutor: Pedro López García

Madrid, Junio 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado es Ingeniería Informática

Título: Improvements in Ciao Prolog's Development Environment: Developing A Rich Development Environment for Ciao on Visual Studio Code

Junio 2024

Autor: Marco Ciccalè Baztán
Tutor: José Francisco Morales Caballero
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid
Cotutor: Pedro López García
Consejo Superior de Investigaciones Científicas

Acknowledgements

First off, a huge thanks to my family for unconditionally supporting me throughout my studies. Your encouragement and belief in me have meant the world.

I owe a big debt of gratitude to my advisors, Jose, Manuel, and Pedro. Your guidance and patience were key throughout this work. We had a lot of laughs along the way, and I couldn't have asked for better mentors.

A special thank you to Aitana for all her love and support. Your presence and encouragement have been invaluable.

Additionally, I want to extend my thanks to everyone else who has supported me during this journey, whether through academic assistance, moral support, or simply being there when needed.

Marco

Abstract

The software industry has consistently experienced rapid and transformative changes since its beginning. A significant aspect of this evolution lies in the tools and utilities available for creating software solutions. These tools enable developers to focus on crafting solutions while minimizing time spent on non-essential tasks.

Among these tools, the text editor is arguably one of the most crucial. In recent years, Visual Studio Code has emerged as the most widely used text editor globally. It allows users to extend its functionality through extensions implemented using modern web technologies.

This project aims to explain the design and implementation of a Visual Studio Code extension for the Ciao programming language. The extension integrates core tools such as the Ciao preprocessor (CiaoPP) and the autodocumenter for (C)LP systems (LPdoc) in an intuitive and comprehensive way inside a modern and accessible text editor. This integration makes the extension suitable for developers of all expertise levels in logic programming and software verification.

Resumen

La industria del software siempre se ha caracterizado por su rápida y continua evolución desde sus inicios. Una parte importante de esta evolución radica en las herramientas empleadas para crear soluciones software. Estas herramientas permiten a los desarrolladores concentrarse en idear soluciones, minimizando el tiempo perdido en tareas ajenas a este objetivo.

De todas estas herramientas, el editor de texto es posiblemente una de las más importantes. En los últimos años, Visual Studio Code, un editor de texto moderno y altamente configurable usando tecnologías web modernas, se ha convertido en el editor de texto más usado en todo el mundo.

Este trabajo tiene como objetivo explicar el diseño y la implementación de una extensión de Visual Studio Code para el lenguaje de programación Ciao. Esta extensión integra herramientas centrales del ecosistema de Ciao como el preprocesador (CiaoPP) y el autodocumentador para sistemas (C)LP (LPdoc) de una manera intuitiva e integral en un editor de texto moderno y accesible. Esta integración convierte a la extensión en una herramienta apta para desarrolladores con cualquier nivel de experiencia en programación lógica y verificación de software.

Contents

1 Introduction **1**
1.1 Background 1
1.2 Motivation 2
1.3 Objectives 2

2 State of the Art **3**
2.1 The Importance of Software Analysis and Verification 3
2.2 Ciao Development Environments 4
2.3 Prolog Visual Studio Code Extensions 6

3 Technology Stack **7**
3.1 Visual Studio Code 7
3.2 TypeScript 7

4 Implementation **9**
4.1 Visual Studio Code 9
4.1.1 User Interface 9
4.1.2 Extensions 10
4.2 Ciao Top Level 11
4.2.1 Integrating the Ciao Top Level Inside Visual Studio Code . . 12
4.2.1.1 Initial Approach 12
4.2.1.2 Final Approach 12
4.2.1.3 Architecture 13
4.2.1.4 `CiaoTopLevel` 14
4.2.1.5 `CiaoPTY` 15
4.2.1.6 `CProc` 16
4.2.2 Ciao Top Level Usage Inside Visual Studio Code 18
4.2.3 Highlighting Errors and Warnings in Ciao Source Code . . . 19
4.2.4 Ciao Testing Integration 20
4.2.5 Ciao Debugger Integration 21
4.2.6 Ciao Generic Menus Inside Visual Studio Code 22
4.3 Ciao Language Integration 23
4.3.1 Language Configuration 23
4.3.2 Syntax Highlighting 24
4.3.3 Snippet Completion 25
4.4 LPdoc Integration 25

4.4.1 Preview Documentation 27
4.4.2 Generate and Save Documentation 27
4.5 CiaoPP Integration 29
4.5.1 Statically Analyze 30
4.5.2 Check Assertions 30
4.5.3 CiaoPP Menu 30
4.6 *On-the-fly* Analysis 33
4.6.1 Ciao `Flycheck` Integration 33
4.6.2 Language Server Protocol 33
4.6.3 Ciao Language Server 34
4.7 Ciao Playground Integration 36
4.8 Ciao Installation and Version Management 38
4.9 Multiplatform Support 40
4.9.1 Operating Systems 40
4.9.2 Cloud Development Environments 40
4.10 Bundling and Publishing the Extension 42

5 Conclusions and Future Work **43**

6 Impact Analysis **45**

Bibliography **47**

Appendices **53**

A Source Code for CProc **53**

B Source Code for Parsing Ciao Error Messages **59**

C Source Code for Marking Ciao Debugger Steps **61**

D Source Code for CiaoPP Menu Webview Panel **65**

E Source Code for Ciao Language Server **69**

F Script for Bundling the Extension **73**

Chapter 1

Introduction

1.1 Background

Many software engineers spend nearly half of their daily working session completing coding-related activities such as writing and reviewing code, bug fixing, testing and generating documentation [1]. These activities typically involve spending time in repetitive and non-relevant tasks such as manually compiling the code, finding errors, and running tests among others.

As a solution to these problems, *Integrated Development Environments* (*IDEs*) where created, a programming language-sensitive tool that aims to reduce the time wasted in the previously mentioned tasks by automating them, enabling engineers to focus on what matters. The common features included in most *IDEs* are: syntax and diagnostics highlighting, debugging, and compiling [2].

However, these advantages also come with a substantial set of drawbacks. One of them lies in the rigidity inherent in such solutions. *IDEs* typically rely on a non-extensible core, providing a strongly opinionated and inflexible user experience that restricts users from extending their functionalities. Furthermore, *IDEs* typically come with significant default resource demands, which, when combined with the previously mentioned rigidity, will definitely strain the CPU of less powerful computers.

Ciao [3] is a general-purpose, multi-paradigm programming language in the Prolog family, created in the nineties, as a continuation of the &-Prolog system [4, 5]. It integrates the flexibility of dynamically typed languages with the speed and safety of statically typed languages. The Ciao ecosystem includes a rich set of powerful tools and utilities. For example, CiaoPP is a tool for program analysis, verification, debugging, and optimization of Ciao programs, and it has been extended to other programming languages such as *C*, *Java*, or even smart contracts [6, 7]. Another tool, LPdoc, is an automatic documentation generator for (C)LP systems, capable of generating comprehensive documentation and manuals for Ciao programs [8]. Additionally, Ciao provides two development environments: the Ciao *Emacs* mode and the Ciao Playground, both of which offer robust support for interacting with all the tools of the Ciao ecosystem.

1.2 Motivation

Emacs was released in the early eighties as an extensible text editor with a robust and minimalist core, allowing users to customize their experience to meet their specific needs by adding custom functionalities using the *Elisp* (*Emacs Lisp*) programming language. These snippets are then processed by the *Emacs* interpreter, applying the intended changes to itself. This feature makes *Emacs* one of the most powerful and extensible text editors and, in fact, one of the most versatile general IDEs available. However, this great power comes at a cost: it causes *Emacs* to have a comparatively steeper learning curve and overall less intuitive interface, especially for beginners [9].

In contrast, Visual Studio Code (*VS Code*), a recent addition to the realm of text editors, not only offers a robust, minimalist and extensible core like *Emacs*, but also offers an arguably more intuitive and accessible user experience for developers of all experience levels [10]. This has led to a very rapid increase in its popularity: according to the 2023 Stack Overflow developer survey, 73.71% of developers now prefer using *VS Code* as their main text editor, compared to the 4.69% of developers who prefer using *Emacs* [11].

Ciao provides an *IDE* experience inside *Emacs*, integrating features like: syntax and diagnostics highlighting, debugging, test running, integrated top level, *on-the-fly* verification and documentation generation among others. Yet, the steep learning curve of *Emacs* may negatively affect Ciao developers without prior experience with it, potentially slowing down their workflow and decreasing productivity, at least until they become familiar with the environment.

1.3 Objectives

The main objective of this project is to design and develop a Ciao Language Support *VS Code* extension, aiming to bridge the gap between the powerful and innovative features of Ciao and the intuitive user experience provided by *VS Code*. By doing so, this extension seeks to enhance the development workflow of Ciao programmers with any level of prior experience with text editors. Additionally, through the integration of a Ciao Language Support Extension for *VS Code*, the language will become more accessible and discoverable to a wider audience of developers, facilitating easier adoption and exploration of its capabilities. More concretely, the objectives of this work are:

- Explore the state of the art – Chapter **2**.
- Select all the necessary tools to develop the extension – Chapter **3**.
- Implement all the necessary functionalities – Chapter **4**.
- Analyze the current state of the project and suggest possible next steps – Chapter **5**.
- Review the impact of the project – Chapter **6**.

Chapter 2

State of the Art

2.1 The Importance of Software Analysis and Verification

Society heavily relies on software in almost every aspect of life, from leisure activities to critical areas such as medical assistance, spacecraft, automobiles, and banking systems, where the creation and maintenance of entirely reliable and fault free systems is crucial.

Ciao offers advanced tools for software engineers to develop systems that are reliable and correct by construction. One such tool is CiaoPP, an analysis, verification, debugging, and optimization tool with native support for (C)LP programs the enables developers to perform static analysis on their entire system, inferring properties such as types, determinacy, computational cost, and correctness, among others [6, 7].

Given this necessity for reliable and correct software systems, some professors at the Technical University of Madrid (UPM) are progressively introducing these advanced topics to their students. They encourage the use of these techniques and tools as a first step towards software verification and inference of properties as a formal method of verifying software correctness, unveiling the necessity for an intuitive environment where students can focus on the concepts rather than the tools themselves.

```
:- module(qsort, [qsort/2], []).

% Example: ?- qsort([5,2,3,4,1],X).

qsort(Data, Out) :-
    qsort_(Data, Out, []).

qsort_([], R, R).
qsort_([X|L], R, R0) :-
    partition(L, X, L1, L2),
    qsort_(L2, R1, R0),
    qsort_(L1, R, [X|R1]).

partition([],_,[],[]).
partition([X|L],Y,[X|L1],L2) :-
    X =< Y, !,
    partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
    partition(L,Y,L1,L2).
```

```
Ciao 1.23 [DARWINx86_64]
?- use_module('/Users/marcociccalebaztan/clip/examples/qsort.pl').

yes
?- qsort([5,2,3,4,1],X).

X = [1,2,3,4,5] ?

yes
?-
```

Figure 2.1: Ciao Emacs Mode Interface

2.2 Ciao Development Environments

The first development environment for Ciao was derived from the 1993 version of the *Emacs* mode[1] for &-Prolog [5] by Manuel Hermenegildo, itself derived from the original mode for Prolog (`prolog.el`) by Masanobu Umeda [12]. The *Emacs* mode for Ciao provides a complete *IDE* that includes syntax highlighting, auto-indentation, context-sensitive help, etc., and integrates the *top-level*, the preprocessor (CiaoPP), the autodocumenter (LPdoc), and the (source-level) debugger. It also features buttons and custom keymaps associated with commands that act as shortcuts for interacting with the *IDE* (loading, running, testing or analyzing a Ciao program). The latest improvements include integration with `flycheck` for *on-the-fly* verification and `company` for auto-complete features [13].

The Ciao *top-level* is a key component of the *Emacs* mode for Ciao, and it will be further detailed in Section 4.2, but it is important to note that most commands of the Ciao *Emacs* mode are based on sending *queries* to the integrated *top-level* and processing its output, providing a convenient way of executing repetitive tasks such as loading, testing, debugging, documenting or even statically analyzing a program (see Figure 2.1).

However, due to the comparatively steep learning curve of *Emacs* and the time it takes for users to create a customized and tailored workspace [9], developers that do not want to spend time mastering *Emacs* and just want to use it as a regular text editor without customizing it, may find it too hard to climb this learning curve, which can be frustrating.

As a solution to this issue, the design of this interface has been taken as refer-

[1] An *Emacs* mode specifies the editing behaviour of the current buffer based on the filetype and is commonly used to remap or map new key binds for custom functionality.

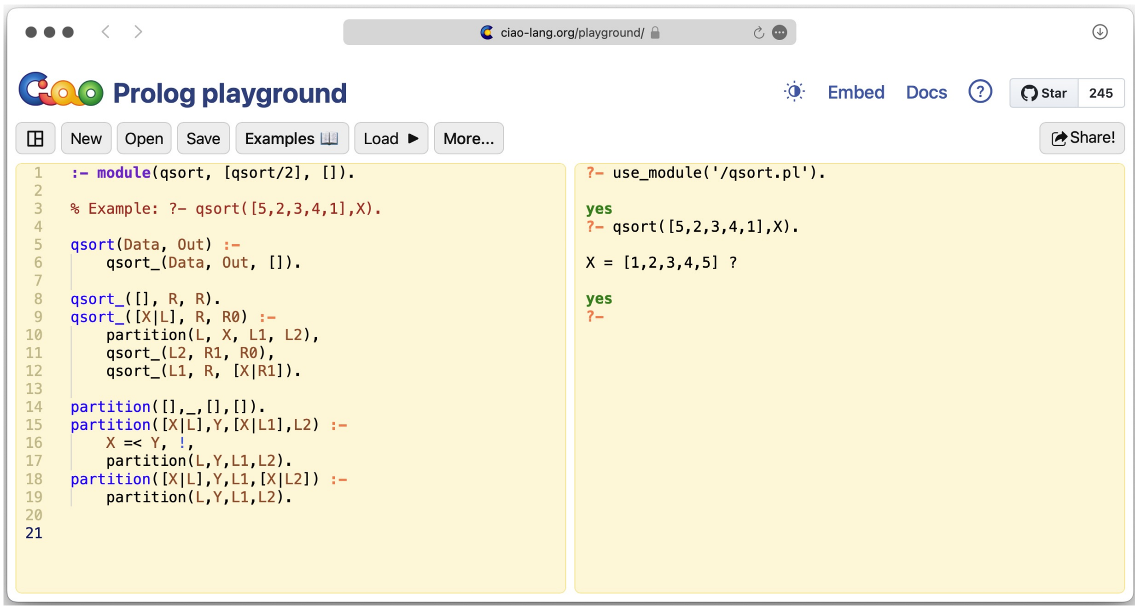

Figure 2.2: Ciao Playground Interface

ence for the implementation of another development environment for Ciao, the Ciao Playground, a beginner-friendly, web-based environment that allows users to immediately start working with the Ciao ecosystem without prior local configuration nor installation, as it runs entirely in the browser [14] (see Figure 2.2).

This simpler environment is what makes the Ciao Playground the perfect choice for beginners and students, which in fact was the first step towards *Active Logic Documents* (*ALDs*) a new and innovative approach for teaching Prolog to beginners that leverages the Ciao Playground core to create interactive and runnable Ciao documents [15]. However, as the whole system runs inside a browser, it is more limited than, and lacks the flexibility of, a local development environment.

Notice that both development environments adhere to a common standard that ensures a consistent high-level user experience and a unified low-level architecture. This standard includes integrating a Ciao *top-level* as a first-class citizen, which requires establishing bidirectional communication between the environment and the integrated *top-level* to programmatically send *queries*, processing its output and providing real-time feedback inside the editor. Additionally, both environments incorporate buttons for interacting with the environment, serving as simple shortcuts for sending *queries* to the integrated *top-level*. Moreover, by sharing the same low-level architecture, these environments ensure consistency and simplicity for Ciao maintainers across multiple platforms. This unified approach guarantees that once users have learned to work with one Ciao development environment, they can easily transition to and be productive in any other environment within the Ciao ecosystem. This standard for Ciao development environments will be the basis for the environment described in this work.

2.3 Prolog Visual Studio Code Extensions

As mentioned before, *VS Code* has rapidly gained significant popularity among developers, emerging as the preferred text editor in recent years. Consequently, many tools and utilities have been ported or developed for this environment, aiming to enhance the development experience of various programming languages, including Prolog. Among the available Prolog extensions for *VS Code*, only one of them really provides *IDE* functionalities similar to the ones present in both Ciao development environments. The *VSC-Prolog* extension released on 12/08/2017 and last updated on 29/11/2018, provides basic Prolog language support mainly tailored for *SWI Prolog* [16, 17].

The key features of the extension are:

- Syntax highlighting.
- Snippets completion.
- Load active source file and query goals inside a shell.
- Experimental debugger for *SWI Prolog* only.

While the *VSC-Prolog* extension offers a valuable functionality for Prolog development — particularly for *SWI Prolog* developers — Ciao developers may find themselves unable to comfortably access the core tools such as CiaoPP and LPdoc inside *VS Code* when using this Prolog extension, resulting in a completely different and downgraded experience compared to the dedicated Ciao development environments discussed above.

Chapter 3

Technology Stack

3.1 Visual Studio Code

As mentioned in Section 1.2 *VS Code* is an extensible text editor that allows developers to customize their workspace to meet their specific needs. Built upon *Electron.js*, a framework that enables the creation of desktop applications for Linux, macOS, and Windows using web technologies like *HTML*, *CSS*, and *JavaScript* (*JS*) [18], *VS Code* offers a familiar environment for many developers, who can extend and customize their workspace without needing to learn a new set of tools to do so, as *HTML*, *CSS*, and *JS* are among the most widely used languages by professional developers in recent years [11].

3.2 TypeScript

The *TypeScript* programming language (*TS*) — developed by *Microsoft* — emerged as a strongly-typed *superset* of *JS* that provides several tools to write scalable and maintainable code, and catch possible type errors during is compilation to *JS* process.

To illustrate the differences between *JS* and *TS*, take this trivial adding function defined in *JS*:

```
1 const add = (a, b) => a + b;
```

If the function is called with two arguments of the type `number`, it should work as expected:

```
1 add(2, 3); // 5 (type number)
```

However, as *JS* has a dynamic and weak type system, it is totally possible and valid to call this function with arguments of any valid type in the language (`string`, `array`, `object`...), which can lead to an unexpected and weird behaviour that could crash an entire system:

```
add(2, '3'); // '23' (type string)
add(2, [3]); // '23' (type string)
add(2, { '3': 3 }); // '2[object Object]' (type string)
```

TS tries to solve this problem by adding automatic type inference, optional type annotations, generic types, type assertions, and many other features present in statically and strongly typed languages such as *Java*.[1] These features make *TS* a much more robust and reliable programming language than *JS*, as it can be statically analyzed for detecting possible errors using the techniques mentioned in Section 2.1.

The `add` function could now be defined using *TS* to only accept arguments of type `number` as follows:

```
const add = (a: number, b: number): number => a + b;
```

With these type annotations, the *TS* compiler can now detect possible type errors during compile time:

```
add(2, 3); // 5 (type number)
add(2, '3'); // TypeScript Error
add(2, [3]); // TypeScript Error
add(2, { '3': 3 }); // TypeScript Error
```

Given the additional safety that *TS* provides to web applications, many web-based tools are being built with it; a great example is *VS Code*. This decision made *TS* the language of choice for building *VS Code* extensions, as the *API* is entirely written in *TS*.

[1]The reader may notice that this is the same role that the Ciao assertion language and the CiaoPP preprocessor play in the context Prolog. Alhough the Ciao approach covers a wider range of properties, one can say that in many ways Ciao is to Prolog, like TypeScript is to JavaScript. In this sense, the Ciao system is a precursor to the many modern “gradual typing”-oriented systems.

Chapter 4

Implementation

This chapter aims to provide a comprehensive insight into the development process of the Ciao Language Support Extension for *VS Code*. It provides a brief overview of *VS Code* and its user experience and a detailed explanation of each feature of the extension.

4.1 Visual Studio Code

4.1.1 User Interface

Each window of *VS Code* is called a workspace. It can hold a collection of one or more opened folders in the editor, providing easy access to all the contents of the folder of a particular project (see Figure 4.1).

A workspace in *VS Code* has four main areas that can be displayed or hidden by the user:

- **Editor**: where open files are accessible.
- **Sidebar**: containing the file explorer and extension marketplace[1].
- **Integrated Terminal**: which offers a conventional terminal within *VS Code*.
- **Status Bar**: providing information on the editor and workspace.

In *VS Code*, users interact with the editor by executing commands. These commands cover a wide range of actions, such as opening a new file, running tasks, or accessing editor settings. These commands are accessible through the *command palette* (see Figure 4.2), which serves as a search-based interface within *VS Code*. Through this *command palette*, users can easily locate and execute commands. Additionally, many frequently used commands have default key bindings assigned to them. For instance, pressing `Ctrl+o` opens a file, and `Ctrl+P` opens the command palette. However, users have the flexibility to customize or create their own key bindings according to their preferences.

[1]Similar to a package registry where users can find new extensions to enhance the capabilities of the editor.

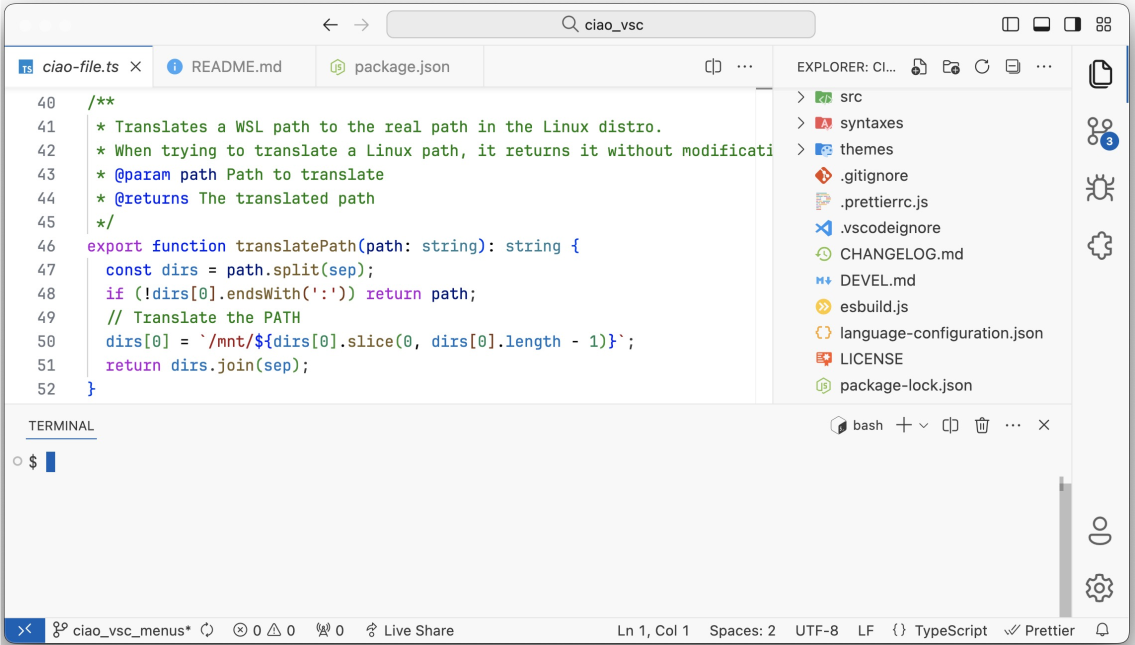

Figure 4.1: Visual Studio Code Workspace.

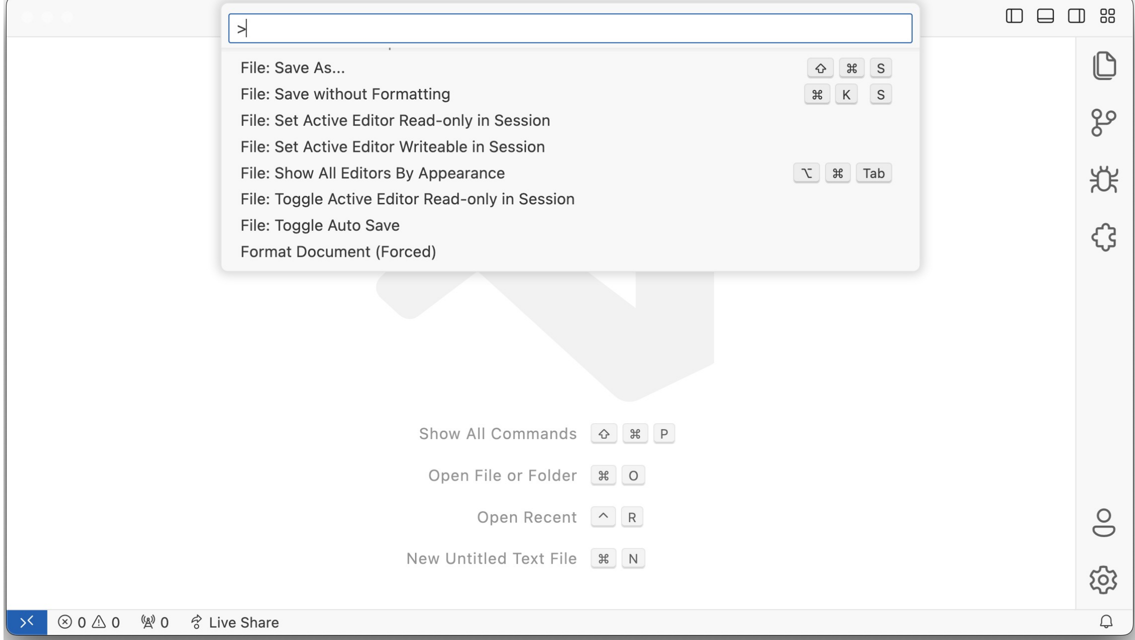

Figure 4.2: Visual Studio Code *Command Palette*.

4.1.2 Extensions

VS Code is a highly extensible text editor, mostly due to its support for extensions. These extensions can enhance and customize the functionality and behaviour of *VS Code* to better meet the needs of individual users and specific software development tasks.

Developers can create extensions using the *VS Code API* and *Node.js*.[2] Given the *event-driven* nature of *JS* and *TS*, the *VS Code API* provides a natural and convenient way for developers to interact with the editor, manipulate its features, and extend its capabilities. An example of a *VS Code* extension mentioned in Section 2.3 is *VSC-Prolog*, which aims to enhance the editor's functionalities to offer a more convenient way of developing Prolog programs, especially *SWI-Prolog* ones. These extensions are published in the extension marketplace, a community-driven repository where *VS Code* users can easily find and install them.

4.2 Ciao Top Level

As mentioned before, the Ciao *top-level* is an interactive programming environment that provides users with the ability to load and run programs. It also enables seamless interaction with the other tools within the Ciao ecosystem, allowing for debugging, testing, documentation generation, and analysis of programs. By offering these capabilities, it acts as the cornerstone of the entire Ciao ecosystem.

The Ciao *top-level* has a shell-like interface where users can execute internal commands or *queries* (a goal or a sequence of goals). Users can provide valid Prolog terms as input to the *top-level* whenever the prompt (`?-`) is printed.

After executing a *query* or an internal command, the possible answers of the *top-level* are:

- `no` if the execution has failed.
- `yes` if the execution has succeeded, together with all the variable bindings as a sequence, allowing the user to keep generating new solutions when the prompt (`?`) is printed.

By default, the predicates available in the *top-level* are the classic set of built-in predicates available in traditional Prolog systems (which in Ciao are actually in libraries that are loaded by default) plus a number of Ciao-specific predicates. In addition, if the user wants to interact with a program, the corresponding Ciao file(s) can be loaded into the *top-level* to make available all the predicates declared therein.

Given the importance of the *top-level* in the Ciao ecosystem, it is essential that any Ciao *IDE* includes a comprehensive and fully functional *top-level* within the editor. For example, the *top-level* inside the Ciao *Emacs IDE* includes not only all the previously features mentioned before, but also syntax highlighting that improves the development experience when working with the *top-level*. Figure 4.3 illustrates the Ciao *top-level* inside the Ciao *Emacs IDE*.

[2] Node.js is a *JS* runtime that allows JavaScript code to run outside the browser

```
Ciao 1.23 [DARWINx86_64]
?- member(a, [b, c]).

no
?- member(a, [b, a]).

yes
?- member(X, [a|L]).

X = a ? ;

L = [X|_] ?

yes
?-
```

Figure 4.3: The Ciao Top Level running inside Emacs.

4.2.1 Integrating the Ciao Top Level Inside Visual Studio Code

4.2.1.1 Initial Approach

The first approach for integrating the Ciao *top-level* in *VS Code* involved registering a *VS Code* command that spawned a shell that would subsequently start a Ciao *top-level* process, such as `ciaosh` [19] inside a new terminal in the integrated terminal panel (see Figure 4.1).

While straightforward, this method presented significant limitations. The primary issue was the inability to read from the output of the integrated terminal panel itself, and hence, the inability to read from the output of the integrated *top-level*. This limitation is inherent to the design of *VS Code*, as its integrated terminal *API* [20] does not support retrieving output data [21]. As mentioned in Section 2.2, implementing bidirectional communication between the environment and the integrated *top-level* is required for any new Ciao development environment. Consequently, due to this significant limitation, this approach was discarded in favour of alternative solutions that can meet the requirements.

4.2.1.2 Final Approach

Given the necessity of reading from the output of the *top-level*, a more flexible and less opinionated solution was required. *VS Code* provides a relatively simple *API* for defining pseudo terminals (`pty`) attached to the integrated terminal panel that can be controlled by extensions [20].

The `pty` *API* allows developers to create and register custom handlers for reading from and writing to the integrated terminal, which is exactly what is needed for the integrated Ciao *top-level*. However, all the logic of implementing a bidirectional communication channel between the user or programmatic input and the Ciao *top-level* process had to be designed and implemented from scratch.

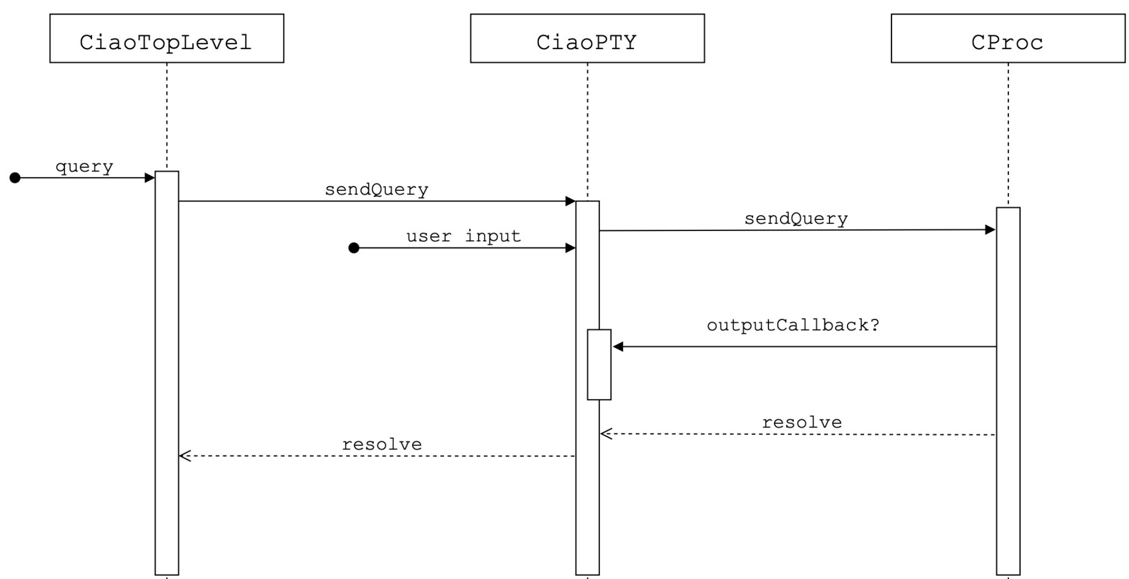

Figure 4.4: Sequence Diagram of the Integrated Ciao *top-level*.

This approach, while requiring much more development effort, covers the requirement of reading the output of the integrated *top-level*, together with the possibility of enhancing the Ciao development experience inside *VS Code* by customizing the *top-level* behaviour and style to match the integrated *top-levels* present in the rest of Ciao development environments.

4.2.1.3 Architecture

After having discussed the final approach for creating the Ciao integrated *top-level* inside *VS Code*, in this section every component of the architecture will be explained.

The final architecture is composed of three classes: `CiaoTopLevel`, `CiaoPTY`, and `CProc`. These classes have different responsibilities, ensuring a modular architecture that is both scalable and decoupled. For instance, the `CiaoTopLevel` class manages the lifecycle of the *top-level*, the `CProc` class acts as a wrapper for a Ciao *top-level* process, such as `ciaosh`, by managing its standard input and output, and the `CiaoPTY` class handles all the logic of capturing and forwarding user input to the `CProc`, and processes and displays the output in the integrated terminal panel of the `CiaoTopLevel`.

An important aspect of this architecture is that all the classes implement the `sendQuery` method with the same interface. This method allows sending programmatic *queries* to the upper layer of the integrated *top-level*, `CiaoTopLevel`, which then propagates the call to the `sendQuery` methods of the lower layers. The `sendQuery` method is asynchronous because it needs to wait for the deepest layer, `CProc`, to complete the execution of the *query*; but once this *query* has finished, the execution is resolved with the output of the *query* as the return value. This asynchronous model has been implemented to enable sequential

query execution without race conditions, allowing each *query* to wait for the previous one to be resolved before proceeding. Figure 4.4 represents the sequence diagram of the architecture, with two main entry-points: **user input** represents the user input using the pty, and **query** represents the programmatic input. For example, the following *TS* source code represents a sequential programmatic communication with the CiaoTopLevel architecture:

```
// Ensure a CiaoPP top-level is started
await ensureTopLevelStarted(CiaoTopLevelKind.CiaoPP);
// Send a query to load the module
await topLevel.sendQuery("use_module('qsort.pl').");
// Send a query to analyze the module
await topLevel.sendQuery("auto_analyze('qsort.pl').");
```

As mentioned in Section 2.2, this architecture is based on the previous *top-level* architectures developed for the *Emacs* mode and the Ciao Playground, ensuring a consistent experience for both Ciao users and maintainers.

4.2.1.4 CiaoTopLevel

```
declare class CiaoTopLevel implements vscode.Disposable {
  /**
   * Create a new CiaoTopLevel instance.
   * @param {CiaoTopLevelKind} kind - Ciao, CiaoPP or LPdoc top-level.
   */
  constructor(kind: CiaoTopLevelKind);

  /**
   * Start the top-level. Create and start a CiaoPTY instance.
   * @param {Object.<string, boolean>} opts - Configuration options on start.
   * @returns {Promise<CiaoTopLevel>} - A promise that resolves to the CiaoTopLevel.
   */
  start(opts?: { hidden?: boolean }): Promise<CiaoTopLevel>;

  /** Restart the CiaoPTY without disposing the terminal. */
  restart(): Promise<void>;

  /** Dispose the terminal and all its resources associated. */
  dispose(): void;

  /**
   * Send a query to the CiaoPTY and returns a promise that will
   * resolve when the query has terminated or reached a goal.
   * @param {string} query - The query string.
   * @param {boolean} [muted=false] - Whether to mute the output or not (default false).
   * @returns {Promise<string>} - A promise that resolves to the query output.
   */
  sendQuery(query: string, muted: boolean = false): Promise<string>;
}
```

The CiaoTopLevel component defined above serves as a high-level interface for managing the lifecycle of the integrated *top-level* via the start, restart and dispose methods. For instance, when the CiaoTopLevel start method is invoked,

it creates a new `CiaoPTY` instance, injecting it with all necessary dependencies like the type of *top-level* (`ciaosh`, `ciaopp`, or `lpdoc`). Additionally, it stores a reference to this `CiaoPTY` instance as an attribute, enabling communication with subsequent layers via the `sendQuery` method.

4.2.1.5 CiaoPTY

```
declare class CiaoPTY implements vscode.Pseudoterminal {
  /**
   * Create a new CiaoPTY instance.
   * @param {CommandRing} commandRing - CommandRing to use.
   * @param {CiaoTopLevelKind} kind - Ciao, CiaoPP or LPdoc top-level.
   */
  constructor(commandRing: CommandRing, kind: CiaoTopLevelKind);

  /**
   * Open the pty. Create and start CProc instance.
   * @param {vscode.Dimensions} dimensions - The terminal initial dimensions.
   * @returns {Promise<CiaoPTY>} - A promise that resolves to the CiaoPTY.
   */
  open(dimensions?: vscode.TerminalDimensions): Promise<CiaoPTY>;

  /** Close the pty and terminates the Ciao top-level process. */
  close(): void;

  /**
   * Callback executed when new input is received.
   * @param {string} data - Input received.
   */
  handleInput(data: string): void;

  /**
   * Send a query to the CiaoPTY and returns a promise that will
   * resolve when the query has terminated or reached a goal.
   * @param {string} query - The query string.
   * @param {boolean} [muted=false] - Whether to mute the output or not (default false).
   * @returns {Promise<string>} - A promise that resolves to the query output.
   */
  sendQuery(query: string, muted: boolean = false): Promise<string>;
}
```

The `CiaoPTY` component defined above handles user interaction with the `pty` such as inserting and deleting characters, moving the cursor within the current line using arrow keys and common key bindings like `Ctrl+A` and `Ctrl+E`, clearing the screen with `Ctrl+L`, and browsing through previous commands using a command ring and the arrow keys.

Additionally, it manages the logic of sending user or forwarding programmatic *queries* to the `CProc` component and processing its output. This includes waiting for the *query* to terminate and then processing its output by applying syntax highlighting or marking the current step of the debugger if the user is debugging a Ciao program (further explained in Section 4.2.5).

The methods `open`, `restart`, and `close` are responsible for managing the `pty` itself.

For example, when the open method is called, the CiaoPTY instance creates and stores a new CProc instance, injecting it with all necessary dependencies, such as the type of *top-level* and a callback to be executed by CProc whenever there is data available in its buffers. This callback will apply syntax highlighting to the output by wrapping some of its sections with special pty control sequences, and will mark the current debugging step — if needed.

The handleInput method processes new input for the pty, implementing the usual terminal interface behaviours and sends a *query* when the user presses the Enter key. Also, storing a reference to the CProc instance enables establishing bidirectional communication with the deepest layer of the architecture via the sendQuery method, asynchronously forwarding *queries* to the CProc component.

4.2.1.6 CProc

```
declare class CProc {
  /**
   * Create a new CProc instance.
   * @param {CiaoTopLevelKind} kind - Ciao, CiaoPP or LPdoc top-level.
   * @param {(output: string) => void} outputCallback - Callback to execute once output
   ↪  data is available.
   */
  constructor(
    kind: CiaoTopLevelKind,
    outputCallback: (output: string) => void
  );

  /**
   * Start the Ciao top-level process.
   * @returns {Promise<CProc>} - A promise that resolves to the CProc.
   */
  start(): Promise<CProc>;

  /** Terminate the Ciao top-level process. */
  exit(): void;

  /**
   * Send a query to the CiaoPTY and returns a promise that will
   * resolve when the query has terminated or reached a goal.
   * @param {string} query - The query string.
   * @param {boolean} [muted=false] - Whether to mute the output or not (default false).
   * @returns {Promise<string>} - A promise that resolves to the query output.
   */
  sendQuery(query: string, muted: boolean = false): Promise<string>;
}
```

The CProc component defined above is a custom wrapper for a Ciao *top-level* process, such as ciaosh, managing interactions by sending *queries* and buffering output. The constructor of CProc requires the type of *top-level* to spawn and a callback from CiaoPTY to be executed when a *query* has finished in order to display its output in the pty.

The sendQuery method writes a *query* to the Ciao *top-level's* standard input and

```
import { spawn } fron "node:child_process";
import { markErrorsOnSource } from "./ciao-utils";

class CProc {
  constructor(outputCallback?: (output: string) => void) {
    this.outputCallback = outputCallback;
    this.buffer = "";
    this.muted = false;
    this.resolve = undefined;
    this.cproc = undefined;
  }
  sendQuery(query: string, muted: boolean = false): Promise<string> {
    // Return a new Promise and store the resolve function
    // to call it whenever the query has finished
    return new Promise<string>((resolve) => {
      this.resolve = resolve;
      // Write the query in the stdin of the process if started
      this.cproc?.stdin?.write(`${query}\n`);
    });
  }
  start() {
    // Spawn the top-level process
    this.cproc = spawn("ciaosh");
    // Register an event handler for new data on stdout
    this.cproc.stdout.on("data", (data: Buffer) => {
      // Buffer data
      this.buffer += data;
      // If the query has finished
      if (this.buffer.endsWith("?- ")) {
        // If the query is not muted, execute the outputCallback to display
        // the output in the pty and mark possible errors on source
        if (!this.muted) {
          this.outputCallback(this.buffer);
          markErrorsOnSource(this.buffer);
        }
        // If there's a query Promise that needs to be resolved
        if (this.resolve !== undefined) {
          // Resolve the promise with the output of the query
          this.resolve(this.buffer);
        }
        // Reset variables
        this.resolve = undefined;
        this.buffer = "";
      }
    });
  }
}
```

Listing 1: Simplified version of the `CProc` implementation.

returns a `Promise`.[3] This `Promise` is resolved when the prompt appears in the standard output, indicating that the *query* has finished or reached one goal. This method ensures asynchronous handling of *queries*, allowing the system to wait for a *query* to complete before proceeding with subsequent actions.

[3]An object that represents the eventual completion or failure of an asynchronous operation together with its resulting value.

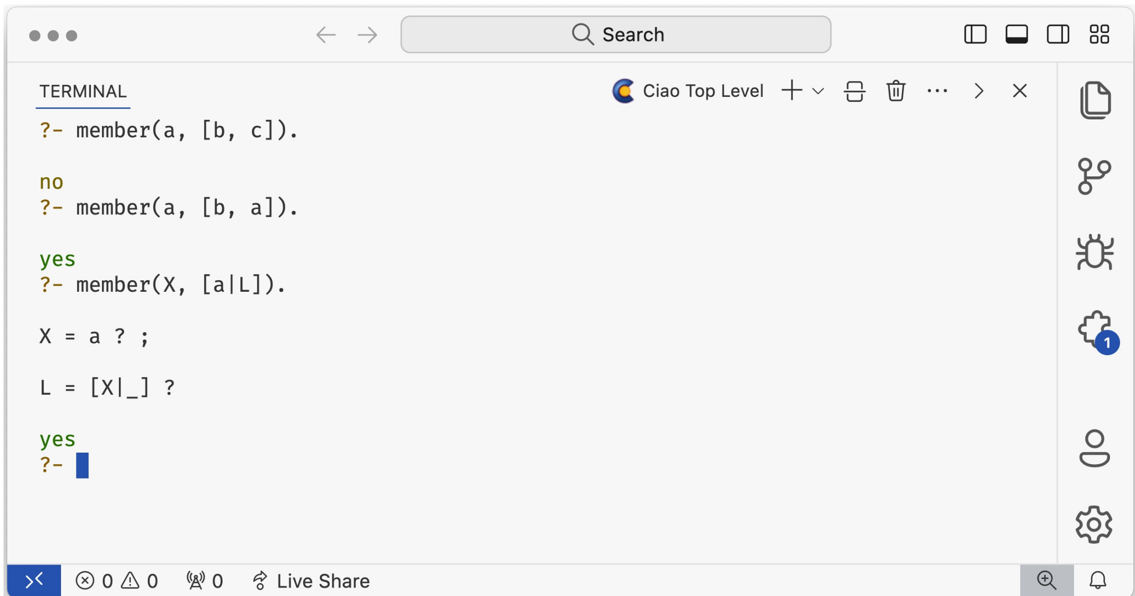

Figure 4.5: Integrated Ciao *top-level* inside *VS Code*.

The `start` method sets up event handlers for standard output, standard error, and close events. These handlers manage the data flow and process lifecycle. For example, the standard output event handler continuously buffers data from the process, and, when it detects the prompt, the handler resolves the `Promise` with the buffer's content. Additionally, it executes a function to mark possible syntax errors in the Ciao source code in case the output includes such messages. If the *query* is not muted, the handler also runs the `outputCallback` to display the output in the `pty` and mark the current debugger's step — if needed. The *TS* source code in Listing 1 is a simplified version of the `CProc` implementation that includes the logic explained above (see Appendix A for the full implementation).

4.2.2 Ciao Top Level Usage Inside Visual Studio Code

With this architecture, the Ciao Language Support Extension for *VS Code* provides an enriched Ciao *top-level* that provides many extra features like an *API* for sending *queries* and waiting until their completion, syntax highlighting for the *top-level* output, or processing and marking error and debugging messages in a Ciao source file.

A command can be executed to start an integrated Ciao *top-level* inside *VS Code*. The *top-level* will be started in a new window inside the integrated terminal panel. Inside this window, users can start interacting with the *top-level* by writing and executing *queries* as in other Ciao development environments (see Figure 4.5). Additionally, as in the existent Ciao development environments, users can also interact with the *top-level* by using custom Ciao buttons inside the *VS Code* user interface. These buttons are shortcuts for executing the most used commands such as start a *top-level* or load the current Ciao file in the *top-level* among others. Moreover, if the *top-level* receives an `EOF` or a `SIGINT` signal, it

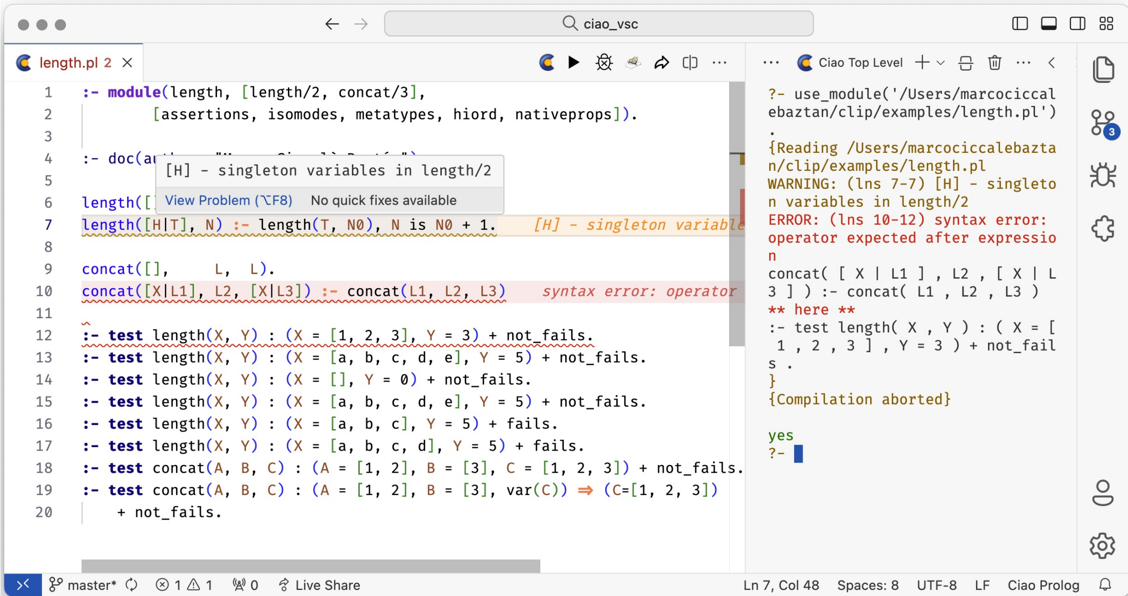

Figure 4.6: Highlighted Errors and Warnings on Ciao Source in *VS Code*.

can be restarted inside the same integrated terminal panel, also recovering the previous command ring. In following sections, the possible use-cases of the integrated *top-level* will be detailed.

4.2.3 Highlighting Errors and Warnings in Ciao Source Code

Each time a Ciao program is loaded inside the *top-level*, the Ciao compiler will check for and output possible syntax errors or warnings that will then be displayed in the *top-level*. By parsing these errors or warnings, it is possible to obtain information such as the specific line range and a brief description of the issue, which can then be utilized to highlight the problematic code within the editor.

As mentioned previously in Section 4.2.1.3, when CProc processes the standard output of the Ciao *top-level* process, it will call a parser that will capture and return all the possible errors generated by the Ciao compiler in a convenient data structure. This data structure will be then processed by another function that will make use of *VS Code's API* to highlight these errors in the active editor. The *TS* source code below contains a pseudo-implementation of the error marking function (see Appendix B for full implementation), and Figure 4.6 illustrates an example of how errors and warnings related to the current file are highlighted after having processed the output inside the integrated *top-level*.

Figure 4.7: Executing Ciao Tests in *VS Code*.

```
const markErrorsOnCiaoSource = (filePath: string, msgs: CiaoDiagnostics) => {
  // Creating errors
  const errors = msgs.errors.map((m) => new Diagnostic(m.lines, m.msg, 'error');
  // Creating warnings
  const warnings = msgs.warnings.map((m) => new Diagnostic(m.lines, m.msg, 'warning');
  // Set diagnostics for file
  vscode.setDiagnosticsForFile(filePath, [...errors, ...warnings]);
}
```

4.2.4 Ciao Testing Integration

Users can execute a *VS Code* command from the command palette for executing all the tests inside their Ciao module. Internally, the command simply sends a set of *queries* to an active integrated *top-level*. The *TS* source code below implements a simplified version of the command, and Figure 4.7 represents the result of executing this command.

```
const runTestsInModule = async (filePath: string) => {
  // Create and start a Ciao top-level
  await ensureTopLevelStarted(CiaoTopLevelKind.CiaoSH);
  // Load unittest library and run tests in the specified file
  await topLevel.sendQuery("use_module(library(unittest)).");
  await topLevel.sendQuery(`run_tests_in_module('${filePath}')`);
};
```

Figure 4.8: Ciao Debugger in *VS Code*.

4.2.5 Ciao Debugger Integration

The Ciao debugger allows users to follow the execution steps of a Ciao program within the *top-level* interface, including backtracking and control flow. The debugger is essential for understanding the sequence of predicate calls and variable bindings.

Once the debugger is activated for a particular module and a *query* is executed in that module, the debugger will trace the program's execution step by step. With every step the debugger takes, it outputs two lines to the standard output of the *top-level* process. The first line contains information about the current state of the debugger, such as the name of the predicate that is being executed and a range of lines where this predicate is located. The second line includes the current state of the variables and their bindings. Below is an example of the output generated by one step of the Ciao debugger:

```
In /Users/marcociccalebaztan/clip/examples/qsort.pl (2-6) qsort_-1
2  2  Call: qsort:qsort_([5,2,3,4,1],Out,[])
```

The first line of the output means that the debugger is at the first appearance of the `qsort_` atom between lines 2 and 6 of the `qsort.pl` module, while the second line shows the current state of the program.

Existing Ciao development environments process the first line of every step to determine the current position of the debugger. This information is omitted from the *top-level* interface and instead processed to visually indicate the exact position of the debugger with a light blue line and a blue rectangle surrounding the current atom.

This behaviour is achieved by locating the exact position of the nth occurrence of

the given atom within a specified range of lines from the Ciao file. To accomplish this, it is best to search through the tokens of the file. However, in *VS Code*, it is impossible to access the generated tokens for the current file using its native *API* [22]. Therefore, a custom Ciao tokenizer function was implemented for this purpose.

A Ciao debugging session inside *VS Code* can be started by using a custom button or by executing the associated command. This command will start a *top-level* if one has not already been started and will send the necessary *queries* for debugging the current module to it. When `CiaoPTY` displays a debugging step, it processes and omits the first line of the output to handle the debugging information appropriately by highlighting each step the debugger takes in the source code (see Figure 4.8 for an example and Appendix C for the implementation).

4.2.6 Ciao Generic Menus Inside Visual Studio Code

Many tools within the Ciao ecosystem, such as CiaoPP or LPdoc, can be customized to meet the specific needs of each user. To facilitate this customization, the standard library of Ciao includes a dedicated package, `library(menu)`, designed for defining and generating interactive Ciao menus. This package can be utilized from the *top-level*, serving as the *backend* component for any graphical interface that acts as the *frontend* of the menu. With this client-server architecture, Ciao menus can be created in any Ciao development environment that can interact with a *top-level* process.

For example, the *Emacs* mode includes a text-based CiaoPP menu that allows users to configure all the flags for CiaoPP. Behind the scenes, this text-based interface sends *queries* to the *top-level*, which tracks the state of the flags being modified by the user.

In an initial iteration of the Ciao Playground, the Ciao Development Team created a generic web-based menu interface using *HTML*, *CSS*, and *JS* that could communicate with a *top-level* process. Given that *VS Code* is a web-based application, as mentioned in Section 3.1, it made sense to reuse this menu interface for generating Ciao menus within *VS Code*.

The *JS* source code was developed prior to the ECMAScript 2015 (ES6) major revision for *JS* [23]. Consequently, it was adapted to this new version of *JS*, ensuring compatibility with the source code of the extension.

With this revised version of the Ciao generic menu interface for web-based environments, it is possible to render it inside *VS Code* using its Webview *API* that allows extensions to register custom views embedded in *VS Code's* user interface using web technologies. In Section 4.5, the process for creating a Webview for the CiaoPP menu is detailed.

```
{
  "comments": {
    "lineComment": "%",
    "blockComment": ["/*", "*/"]
  },
  "brackets": [
    ["{", "}"],
    ["[", "]"],
    ["(", ")"]
  ],
  "autoClosingPairs": [
    { "open": "{", "close": "}" },
    { "open": "[", "close": "]" },
    { "open": "(", "close": ")" },
    { "open": "'", "close": "'", "notIn": ["string", "comment"] },
    { "open": "\"", "close": "\"", "notIn": ["string"] },
    { "open": "/*", "close": "*/", "notIn": ["string"] }
  ],
  "folding": {
      "markers": {
        "start": "^\\s*%\\s*region\\b",
        "end": "^\\s*%\\s*endregion\\b"
      }
  }
}
```

Listing 2: Language Configuration File for Ciao

4.3 Ciao Language Integration

In this section, all the necessary steps for integrating Ciao as a programming language within *VS Code* will be detailed. These steps include defining a set of declarative language features, registering a custom grammar, and creating custom snippets for Ciao files.

4.3.1 Language Configuration

The first step towards integrating Ciao within *VS Code* is defining some language features and characteristics that configure the editing experience for Ciao files. These features are declared inside a *JSON* file that must follow a schema defined by *VS Code* for this purpose.

In this file, it is possible to define the way of commenting code inside Ciao files for *VS Code* to know how to comment and uncomment Ciao files when the shortcut for commenting a portion of code is executed. It is also possible to define *auto-closing pairs* such as `(...)` or `[...]`, folding patterns for hiding portions of the code when needed, and basic indentation rules.

Listing 2 includes some parts of the language configuration file for Ciao in *VS Code*.

```
[
  {
    "name": "entity.name.function.ciao",
    "match": "^\\w+([_-]?\\w+)*"
  },
  {
    "name": "variable.parameter.named.ciao",
    "match": "[A-Z][\\w\\-\\.']*"
  },
  {
    "name": "keyword.predefined.operator.ciao",
    "match": "(=<|>=|=|>|<|->|-|\\+|\\*)"
  }
]
```

Listing 3: Some *TextMate* rules for Ciao

4.3.2 Syntax Highlighting

One of the core features of any *IDE* is syntax highlighting, a basic yet important feature that improves code readability and comprehension by highlighting tokens[4] with different colours. In Ciao, syntax highlighting is particularly crucial due to its use of *domain-specific languages* (*DSLs*) and language extensions such as assertions or functional syntax.

The process of highlighting code involves tokenizing the source file and identifying different tokens to then apply the same colour to all tokens of the same type. To achieve this, the tokenization engine of *VS Code* supports *JSON TextMate Grammars* [24], a structured collection of regular expressions written as a *JSON* file that defines the grammar of a programming language.

Listing 3 includes the definition of three types of Ciao tokens: the definition of a predicate, a variable and some predefined operators, which are part of the *TextMate* grammar defined for Ciao.

However, this grammar will only be used by the *VS Code* tokenization engine for generating the tokens, not to apply colours to them. This second step of applying the colours to the tokens in *VS Code* is managed by *color themes*, extensions that affect all *workspaces*. This prevents extensions to apply custom colours to certain tokens, which can be a problem for Ciao and other programming languages that use fixed colours for some of its syntax. In the case of Ciao, the status of an assertion has its own colours that must not change across themes, such as `checked`, `false`, or `check`.

Curiously, users can define some rules in their *VS Code* settings to customize the colour of a specific token, this being the only way of specifying the colour of a token without having to change the colour theme. Leveraging this option, the extension provides a command that automatically adds a set of rules to add custom colours only for Ciao files, these colours are different for light and dark themes, so the user can specify what type of theme they are using in order to

[4]Smallest unit of meaningful data in source code.

```
:- module(_, [nrev/2], [assertions,fsyntax,nativepro

% Naive reverse with some complex assertions.
% The system flags a (cost) error reminding us that
% nrev/2 is quadratic, not linear.
% (Requires cost-related domains.)

:- pred nrev(A,B) : (list(num,A), var(B)) => list(B)
   + ( det, terminates, steps_o( length(A) ) ).

nrev( [] )     := [].
nrev( [H|L] ) := ~conc( ~nrev(L),[H] ).


:- pred conc(A,B,C) + ( det, terminates, steps_o(len

conc( [],    L ) := L.
conc( [H|L], K ) := [ H | ~conc(L,K) ].
```

```
:- module(_1,[nrev/2],[assertions,fsyntax,nativeprop

%% %% :- check pred nrev(A,B)
%% %%    : ( list(num,A), var(B) )
%% %%    => list(B)
%% %%    + ( det, terminates, steps_o(length(A)) ).

:- checked calls nrev(A,B)
   : ( list(num,A), var(B) ).

:- checked success nrev(A,B)
   : ( list(num,A), var(B) )
   => list(B).

:- false comp nrev(A,B)
   : ( list(num,A), var(B) )
   + ( det, terminates, steps_o(length(A)) ).

:- true pred nrev(A,B)
   : ( mshare([[B]]),
       var(B), ground([A]), list(num,A), term(B) )
   => ( ground([A,B]), list(num,A), list(num,B) )
```

Figure 4.9: Ciao Syntax Highlighting inside *VS Code*

match their preferences. Figure 4.9 shows examples of syntax highlighting of some Ciao code inside VS Code.

4.3.3 Snippet Completion

Another valuable feature for enhancing the Ciao development experience in *VS Code* is the use of code snippets. Snippets are predefined code templates that can be quickly inserted into a file using a prefix. These snippets can significantly improve productivity by reducing the need to type repetitive code segments manually.

The extension registers a set of predefined snippets for Ciao that aim to reduce the time spent writing boilerplate for assertions and module declarations. *VS Code's* snippets are particularly powerful, as one can specify some *breakpoints* inside the snippet so the user can *tab* through those *breakpoints* and fill the slots.

Listing 4 illustrates some of the snippets already registered when the user installs the Ciao extension.

4.4 LPdoc Integration

LPdoc is a key tool within the Ciao ecosystem designed to generate documentation manuals directly from the source code files of (C)LP systems. These manuals can be exported in various formats, such as `HTML`, `PDF`, and `Info`. Users can interact with LPdoc via its command-line interface using the `lpdoc` command:

```
lpdoc [Options] Input
```

```
{
  "rule": {
    "prefix": "rule",
    "body": "${1:functor}(${2:Args}) :- \n\t${3:% Code}.",
    "description": "Rule",
    "scope": "source.ciao"
  },
  "module": {
    "prefix": "module",
    "body": ":- module(${1:Module}, [${2:Exports}], [${3:Imports}]).",
    "description": "Module",
    "scope": "source.ciao"
  },
  "test": {
    "prefix": "test",
    "body": ":- test ${1:functor}(${2:Args})\n\t: (${3:Precondition})\n\t=>
    ↪  (${4:Postcondition})\n\t+ (${5:GlobalProperties})\n\t# \"${6:Comment}\".",
    "description": "Unit test assertion",
    "scope": "source.ciao"
  },
  "pred": {
    "prefix": "pred",
    "body": ":- pred ${1:functor}(${2:Args}) : ${3:Precondition} =>
    ↪  ${4:Postcondition}\n\t# \"${5:Comment}\".",
    "description": "Predicate assertion",
    "scope": "source.ciao"
  }
}
```

Listing 4: Set of Ciao Snippets

Additionally, LPdoc has its own *top-level* process that, by default, loads all the internal predicates it uses, providing greater flexibility and control over documentation generation. Consequently, it has been the preferred option for integrating LPdoc into the existing Ciao development environments, and, to maintain consistency with these environments, the Ciao *VS Code* extension will also integrate LPDoc using its *top-level process* instead of its command-line interface.

For generating large and complex manuals, such as the Ciao system manual, LPdoc makes use of a special file: `SETTINGS.pl`. Inside this file the user can specify advanced configuration settings for generating manuals that consist of multiple modules or packages. While documenting a single module with LPdoc, a `SETTINGS.pl` file may not be necessary, it becomes essential for managing large projects or complex directory structures.

In following subsections, all the *VS Code* commands related to LPdoc, which happen to be the most frequent use cases of LPdoc, will be discussed and explained. These commands will follow a common structure:

1. Ensure an integrated LPdoc *top-level* is started and running inside *VS Code*, as the following steps will involve programmatically interaction with it.
2. Execute external functionality such as: creating a temporary directory or checking the existence of a `SETTINGS.pl` file.

3. Send the necessary *queries* to the LPdoc *top-level* for generating or displaying the documentation.

4.4.1 Preview Documentation

This command allows users to quickly examine an HTML version of the documentation manual generated from their current Ciao file inside a temporary directory, preventing the final and intermediate files generated by LPdoc from cluttering the current directory. The external functionality required by this command involves creating a temporary directory and creating a symbolic link to the current Ciao file inside this temporary directory, allowing access to the source file from the new directory. The following source code implements this functionality in a simplified manner:

```
1  const previewDocumentation = async () => {
2    // Create a temporary directory
3    const tmpDir = createTmpDir();
4    // Create a symbolic link to the original source file inside the temporary directory
5    symlink('/path/to/foo.pl', tmpDir);
6    // Ensure a LPdoc top-level is started
7    await ensureTopLevelStarted(CiaoTopLevelKind.LPdoc);
8    // Set the cwd of the top-level to the temporary directory
9    await lpdocTopLevel.sendQuery(`working_directory(_,'${tmpDir}')`);
10   // Generate the documentation
11   await lpdocTopLevel.sendQuery(`doc_cmd('foo.pl',[],gen(html)).`);
12   // Display the documentation
13   await lpdocTopLevel.sendQuery(`doc_cmd('foo.pl',[],view(html)).`);
14 };
```

4.4.2 Generate and Save Documentation

This command allows users to generate and save the documentation manual generated from their current Ciao module inside their current working directory in the specified format. By default, only the module source file will be used to generate the documentation. However, if a SETTINGS.pl file exists in the current working directory, LPdoc will use this file to generate the documentation. The external functionality required for this command involves searching for a possible SETTINGS.pl file inside the current working directory. The following source code implements this functionality in a simplified manner:

```
1  const generateAndSaveDocumentation = async (format: string) => {
2    // Ensure a LPdoc top-level is started
3    await ensureTopLevelStarted(CiaoTopLevelKind.LPdoc);
4    // Set the file that LPdoc will be using (SETTINGS.pl or source)
5    const file = existsSettingsFile() ? 'SETTINGS.pl' : 'foo.pl';
6    // Generate the documentation
7    await lpdocTopLevel.sendQuery(`doc_cmd('${file}',[],gen(${format})).`);
8    // Display the documentation
9    await lpdocTopLevel.sendQuery(`doc_cmd('${file}',[],view(${format})).`);
10 };
```

Figure 4.10 represents the *VS Code* workspace after previewing the documentation for the current Ciao file, and Figure 4.11 shows the generated documentation for that file.

```
length.pl                                    Preview documentation for file
1  :- module(length, [length/2, concat/3],
2          [assertions, isomodes, metatypes, hiord, nativeprops]).
3
4  :- doc(author, "Marco Ciccalè Baztán").
5
6  length([],    0).
7  length([_|T], N) :- length(T, N0), N is N0 + 1.
8
9  concat([],     L,  L).
10 concat([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).
11
12 :- test length(X, Y) : (X = [1, 2, 3], Y = 3) + not_fails.
13 :- test length(X, Y) : (X = [a, b, c, d, e], Y = 5) + not_fails.
14 :- test length(X, Y) : (X = [], Y = 0) + not_fails.
15 :- test length(X, Y) : (X = [a, b, c, d, e], Y = 5) + not_fails.
16 :- test length(X, Y) : (X = [a, b, c], Y = 5) + fails.
17 :- test length(X, Y) : (X = [a, b, c, d], Y = 5) + fails.
18 :- test concat(A, B, C) : (A = [1, 2], B = [3], C = [1, 2, 3]) + not_fails.
19 :- test concat(A, B, C) : (A = [1, 2], B = [3], var(C)) => (C=[1, 2, 3])
20    + not_fails.
```

```
LPdoc Top Level
lpdoc ?- working_directory(_,'/var/folders/zm/xm4_h9gd1bb49bph2w69n8b40000gn/T/length.pl_MYFtED').

yes
lpdoc ?- doc_cmd('length.pl',[],gen(html)).
{Generating html documentation for /private/var/folders/zm/xm4_h9gd1bb49bph2w69n8b40000gn/T/length.pl_MYFtED/length}
{Resolving references globally}
{Running html backend for length}

yes
lpdoc ?- doc_cmd('length.pl',[],view(html)).

yes
lpdoc ?-
```

Ln 1, Col 40 Spaces: 8 UTF-8 LF Ciao Prolog Prettier

Figure 4.10: Preview Documentation Command inside *VS Code*.

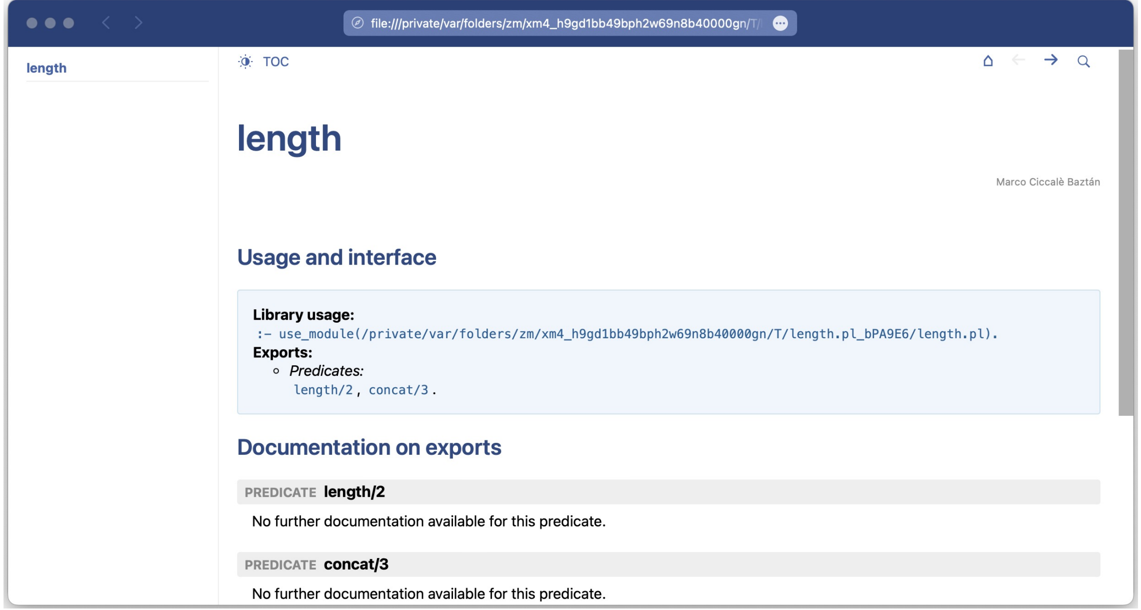

Figure 4.11: Documentation Generated using LPdoc.

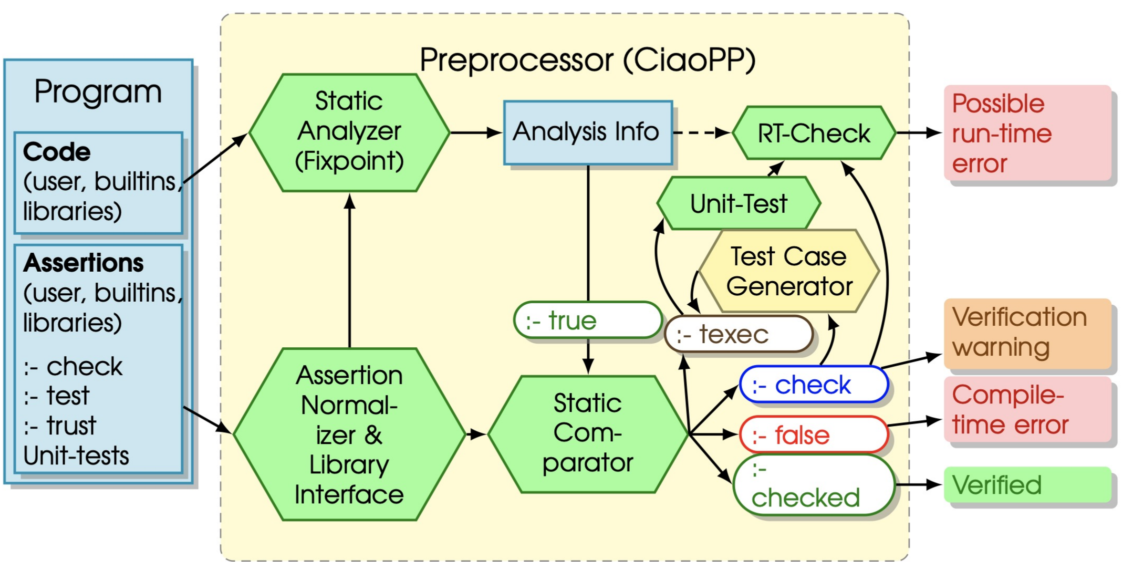

Figure 4.12: High-level Architecture of the CiaoPP Framework

4.5 CiaoPP Integration

Global static analysis and verification tools serve as a resource that software engineers can leverage to detect high-level errors or ensure their absence in complex software systems. This is especially relevant for declarative and dynamic languages such as Prolog where inferring certain properties such as types, modes, non-failure, determinacy, computational cost, and variable sharing can contribute to designing safe and complex systems. These analyses are performed over code specifications in the form of assertions contained within the source code.

The CiaoPP framework natively supports Prolog and various (C)LP systems, along with any intermediate Horn clause-based representation of a system developed using other programming languages not included in the (C)LP paradigm. CiaoPP can identify semantic errors that extend far beyond the traditional compiler's error reporting capabilities. Furthermore, CiaoPP emits statically-obtained certificates that ensure program properties do not need to be checked at runtime, thus eliminating certain runtime assertion tests.

CiaoPP is powered by an efficient fixpoint engine that can perform incremental analysis, reducing the number of reanalyses performed across a software system that can be made up of a single module or a collection of separate modules. Figure 4.12 represents the architecture of the CiaoPP framework.

Program verification is first performed at compile-time by inferring the previously mentioned properties via abstract interpretation-based static analysis, and comparing the results against the the assertions inside the source code.

Figure 4.13: CiaoPP Analysis Inside *VS Code*

4.5.1 Statically Analyze

Users can analyze their current Ciao file using a simple command. This command will: ensure a CiaoPP *top-level* is started, send a *query* to it for analyzing the current file using the `auto_analyze/1` predicate from the CiaoPP framework, mark possible diagnostics on source, and open the analysis output file besides the current file (see Figure 4.13). The following source code implements this functionality in a simplified manner:

```
1 const analyzeFile = async () => {
2   // Ensure a CiaoPP top-level is started
3   await ensureTopLevelStarted(CiaoTopLevelKind.CiaoPP);
4   // Analyze file and get output file
5   const outputFile = await ciaoppTopLevel.sendQuery("auto_analyze('foo.pl').");
6   // Display the output file
7   vscode.openFileInEditor(outputFile);
8 };
```

4.5.2 Check Assertions

Users can check if the assertions defined in their Ciao program are satisfied or violated using a simple command. This command will have the same structure as the `analyzeFile` command, but it will send a different *query* and not display an output file.

4.5.3 CiaoPP Menu

The actions performed by CiaoPP can be configured by a set of flags. Depending on the flags selected by the user, additional flags may become available for

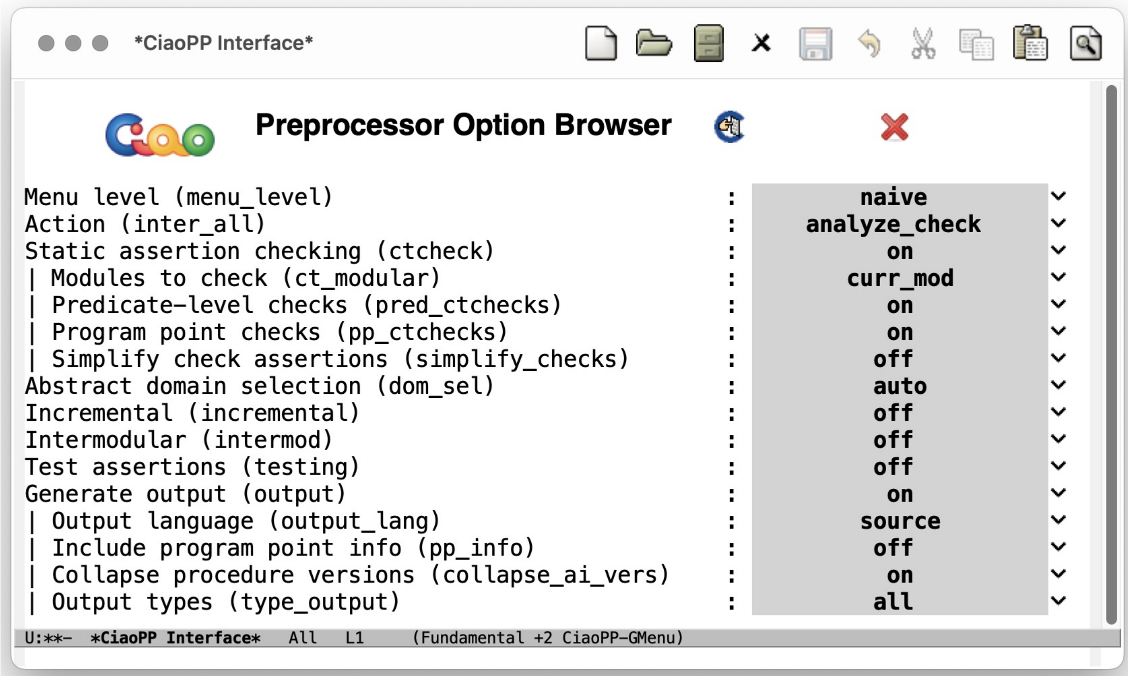

Figure 4.14: *Emacs* Text-based CiaoPP Menu

further customization. After customizing CiaoPP's behaviour, users can use the preprocessor to analyze their Ciao program according to this custom set of flags.

These flags can be manually configured by using the `set_menu_flag/3` predicate inside the CiaoPP *top-level*. However, the *Emacs* mode for Ciao provides an interactive, graphical, text-based CiaoPP menu where users can customize any flag of the preprocessor (see Figure 4.14). Each time the user changes the value of a flag using this menu, the selected value is sent to the underlying CiaoPP *top-level*. The *top-level* then saves the value and reevaluates the state of the menu to enable or disable other flags for further customization.

To bring this functionality to the Ciao *VS Code* extension, the initial CiaoPP menu definition in *JSON* is obtained by sending the following *query* to the CiaoPP *top-level*:

```
menu_to_json(all,_I), json_to_string(_I,_S), write_string(_S).
```

Once the initial definition of the CiaoPP menus is retrieved, the Webview *VS Code API* is leveraged to create a custom web container. This container integrates the updated Ciao generic web-based menu interface discussed in Section 4.2.6, acting as the *frontend* for the CiaoPP *top-level*, which serves as the *backend* using the `library(menu)` Ciao standard package for creating interactive menus.

The Webview panel is created using the initial menu definition and a simple *HTML* structure that includes minimal *CSS* styles and a script to dynamically generate the configurable menu flags. Additionally, a message listener is registered within the panel to facilitate communication between the Webview panel and the Ciao extension (see Appendix D for full implementation).

The script loaded inside the *HTML* template integrates the Ciao generic web-based menu interface. This script is responsible for rendering the menu options based on the initial *JSON* menu definition and dynamically updating the menu as users interact with it. It captures user inputs, such as selecting flags, and executes an injected, context-specific `callback` when these events are triggered. In the case of the CiaoPP menu inside *VS Code*, the `callback` sends a *query* to the integrated CiaoPP *top-level*, updating the value of the flag.

Below is a schematic implementation of creating a CiaoPP menu using the Ciao generic web-based menu interface (`ciao-menu-html`).

```
import CiaoPPMenu from '../ciao-utils/ciao-menu-html';

const createCiaoPPMenu = (menudef: string) => {
  try {
    const menu = new CiaoPPMenu(
      // Parse the menu definiton from JSON string to JS Object
      JSON.parse(menudef),
      // Callback to execute when a flag's value is changed
      () => {
        // Send a message that will be captured by the Webview panel message listener
        vscode.postMessage({
          command: 'submit',
          text: /* Query for updating the flag in the top-level */,
        });
      }
    );
  } catch {
    vscode.postMessage({ command: 'error' });
  }
}
```

With this setup, the CiaoPP interactive menu can be opened with a simple *VS Code* command, which starts a hidden integrated CiaoPP *top-level* along with the custom Webview. Additionally, as the *queries* sent to the *top-level* can be extensive, they are sent silently, ensuring that the user does not see these *queries* within the integrated *top-level* interface. Figure 4.15 shows the CiaoPP menu integrated within *VS Code*.

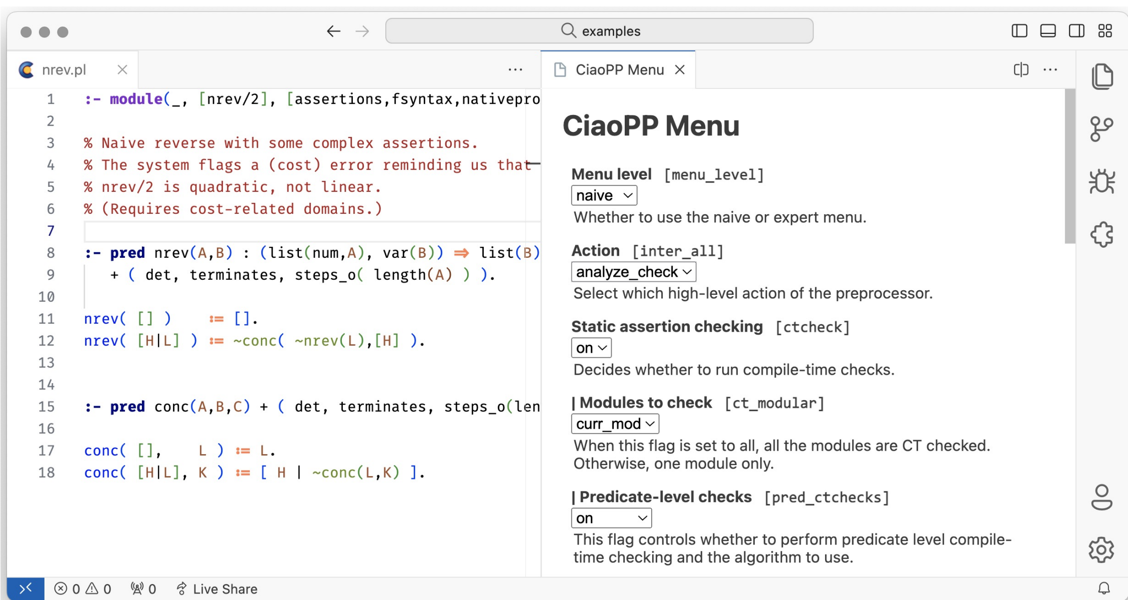

Figure 4.15: Integrated CiaoPP Menu Inside *VS Code*.

4.6 *On-the-fly* Analysis

On-the-fly analysis refers to the real-time checking and validation of source code as it is being written. This process helps developers catch errors early, receive immediate feedback, and improve code quality by running various checkers and analyzers continuously in the background. This approach significantly enhances the development experience and productivity.

4.6.1 Ciao `Flycheck` Integration

`Flycheck` is an *on-the-fly* analysis package for *Emacs* that provides real-time feedback within the current buffer by running background processes called checkers [25]. These checkers report errors or warnings in the content of the current buffer, which are then displayed in the current buffer. `Flycheck` continuously runs the checkers to ensure a responsive experience for the developer. The Ciao development team integrated `Flycheck` into the *Emacs* mode for Ciao to enhance the coding experience by providing immediate feedback on syntax errors and assertion checking (*VeriFly*) [13]. Behind the scenes, the `Flycheck` integration for Ciao creates a temporary file to track modifications made to the original file without automatically saving it. The `Flycheck` checkers analyze this temporary file and send any diagnostics directly to the current buffer.

4.6.2 Language Server Protocol

Language Server Protocol (*LSP*) is a `JSON-RPC`-based protocol definition developed and maintained by *Microsoft* that tries to standardize the communication between a text editor or *IDE* and a language server that provides programming language specific functionalities, such as autocompletion, go to definition and

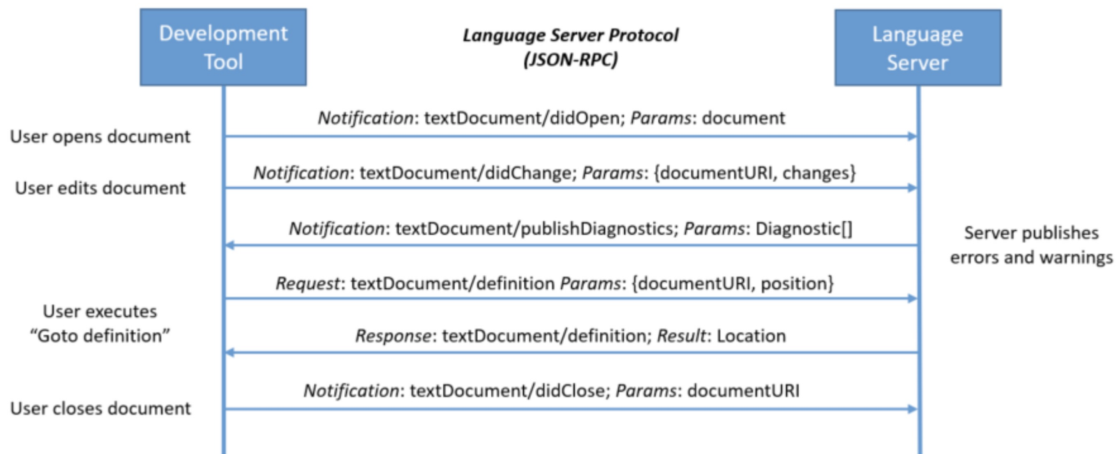

Figure 4.16: Communication Between Editor and Language Server via *LSP*.

diagnostics reporting [26]. This way, a language server that complies with the *LSP* standard can be used in any text editor or *IDE* that supports communication with language servers via *LSP*.

Before explaining how *LSP* works, it is crucial to understand the `JSON-RPC` protocol. A *Remote Procedure Call* (`RPC`) is defined as the action of executing a specific procedure, function, or subroutine on a remote host (another computer), abstracting the developer from specifying the details of the remote interaction. `JSON-RPC` is a protocol for encoding these `RPC`s using *JSON*.

LSP is the standard for providing *on-the-fly* analysis inside *VS Code*. It leverages `JSON-RPC` for encoding the messages exchanged between the text editor and the language server. These messages include requests from the editor for language-specific features, and responses from the language server providing the requested information (see Figure 4.16). Note that when a user opens a document, the editor loads the content into memory, making it the ground truth instead of the disk contents. By keeping an in-memory copy, the editor can track changes to the document without automatically saving them.

LSP has proven to be a great tool for facilitating the development process for both language and editor implementers, particularly for languages with *domain-specific languages* (*DSLs*), such as Ciao [27].

4.6.3 Ciao Language Server

The Ciao development team had already considered creating a Ciao language server [13], and, in this work, a minimal implementation of the first Ciao language server that provides *on-the-fly* verification with CiaoPP and diagnostics reporting is presented.

For the sake of simplicity, the Ciao language server will be implemented using a *Node.js* library that simplifies the development of language servers by providing utilities and abstractions to handle tasks such as `JSON-RPC` communication.

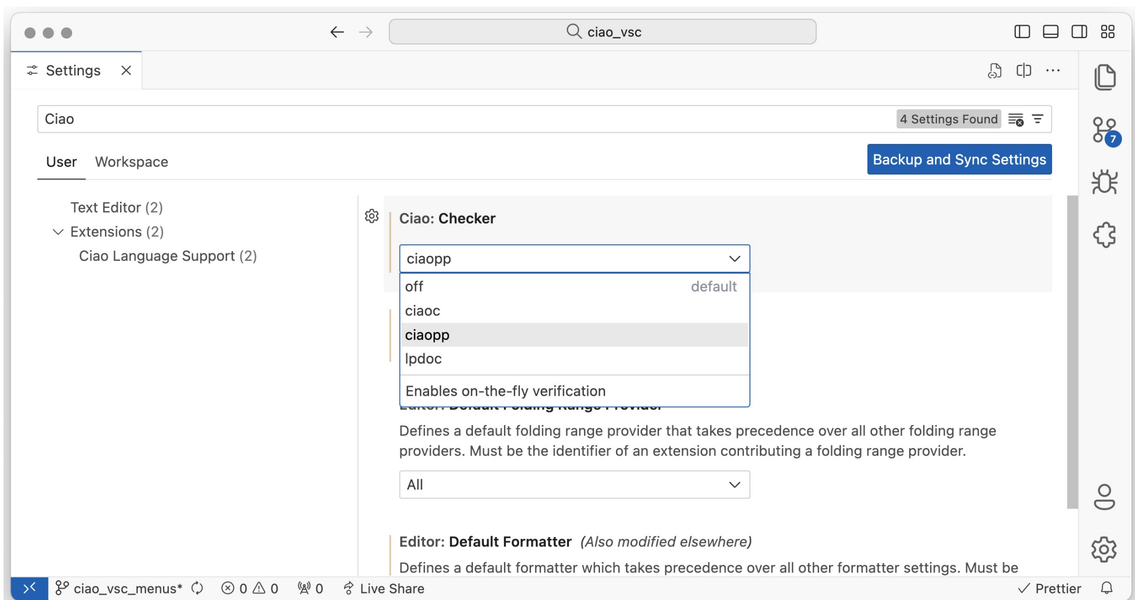

Figure 4.17: Ciao Checkers Available in *VS Code*.

When a user changes the document in the editor, the language server performs the following steps:

1. **Document Change Detection:** The language server detects changes in the in-memory representation of the file using an event listener that triggers when the content is modified.
2. **Temporary File Creation:** Since Ciao only supports compilation and analysis by providing an actual file, not just the contents as an argument, the language server creates a temporary file. This temporary file has the same name as the original file, appended with a special suffix. This suffix allows Ciao's module system to recognize that, despite the differing file names, the temporary file corresponds to the same module as the original. The temporary file tracks modifications without affecting the original file, as the changes received by the language server are in the form of a string containing the content of the document.
3. **Running a Ciao Checker:** Users can specify what Ciao checker they want to use for *on-the-fly* analysis inside *VS Code's* settings (see Figure 4.17). The available options are: **`off`** (disables *on-the-fly* analysis), **`ciaoc`** (checks for syntax errors), **`ciaopp`** (checks assertions), and **`lpdoc`** (checks for documentation errors). The selected checker will be executed to process the temporary file previously created, and output possible diagnostics.
4. **Send Diagnostics to Client:** If diagnostics are generated, the language server will process and convert them into diagnostics messages that are then sent back to the editor using `JSON-RPC`, where they are displayed to the user in real-time.

Figure 4.18 shows the Ciao language server in action, and Listing 5 is its schematic implementation (see Appendix E for full implementation).

Figure 4.18: Ciao *On-the-fly* Analysis Using the Ciao Language Server.

```
// Register an onChange event that triggers when the in-memory document has changed
documents.onDidChangeContent((change) => {
  validateTextDocument(change.document, ciaoChecker);
});
const validateTextDocument = async (document, ciaoChecker) => {
  // Create a temporary file with the flycheck suffix
  // that has the contents of the modified file
  const tmpFilePath = getTmpPath(document.path);
  writeFileSync(tmpFilePath, document.getText());
  // Spawn the CiaoChecker and obtain the stderr
  const { stderr } = spawnSync(ciaoChecker.executable, ciaoChecker.args);
  // Retrieve the parsed diagnostics
  const { errors, warnings } = parseErrorMsg(stderr.toString());
  // Send the computed diagnostics to the language client (VS Code)
  connection.sendDiagnostics({
    uri: document.uri,
    diagnostics: [...errors, ...warnings],
  });
};
```

Listing 5: Schematic implementation of the Ciao language server.

4.7 Ciao Playground Integration

The Ciao Playground, discussed in Section 2.2, is a web-based Ciao development environment where users can write, run, debug, document, and analyze Ciao programs all within their browser with no prior configuration or setup. From the Ciao *VS Code* extension, users can open their current file in a Ciao Playground window, facilitating collaborative development, lectures and code sharing as the whole program is inserted as a query parameter in the Ciao Playground URL. Figure 4.19 shows the button to share the current Ciao file and the browser tab

```
/** @returns Ciao Playground URL with the current **Ciao Prolog** file embedded */
export function createPlaygroundURL(): string | undefined {
  // Get Active File Content
  const code = getActiveCiaoFileContent();
  if (!code) return;
  // Encode the File Content and Insert it as a Query Parameter
  const url = `https://ciao-lang.org/playground/?code=${encodeURIComponent(code)}`;
  // Check if the URL Does not Exceed the Maximum Size
  const maxUrlLength = 2048;
  if (url.length <= maxUrlLength) {
    return url;
  }
  window.showErrorMessage(
    `ERROR: The file length exceds the maximum limit accepted by most
     browsers: ${maxUrlLength}.`
  );
}
```

Listing 6: Function to send code to playground.

that was opened after sharing the current file. Listing 6 shows the implementation of this functionality.

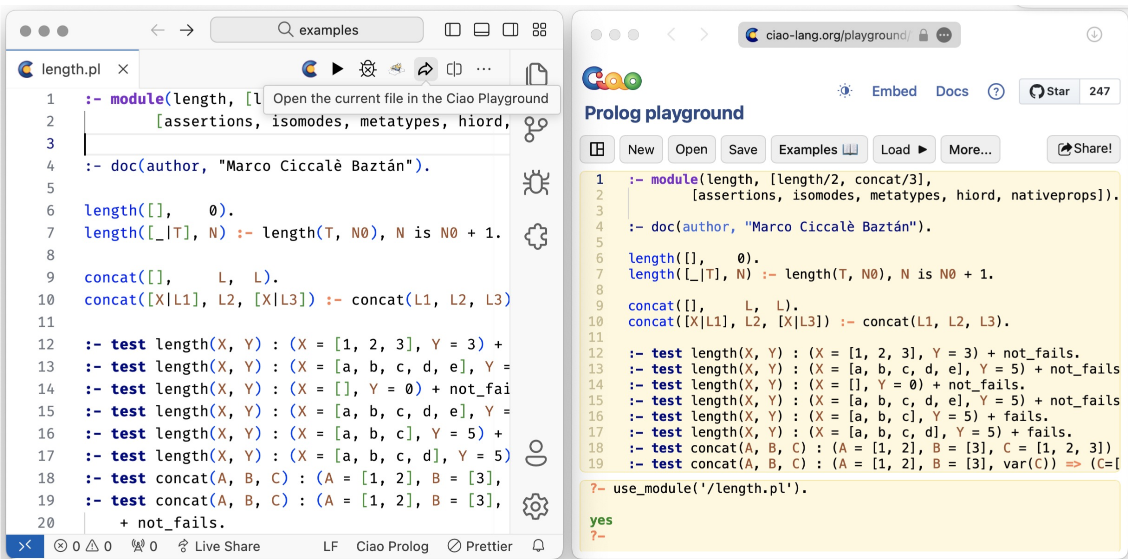

Figure 4.19: Ciao Playground Window after Sharing from *VS Code*.

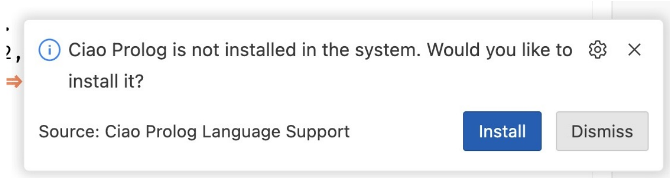

Figure 4.20: Pop Up Alert when Ciao is not Installed.

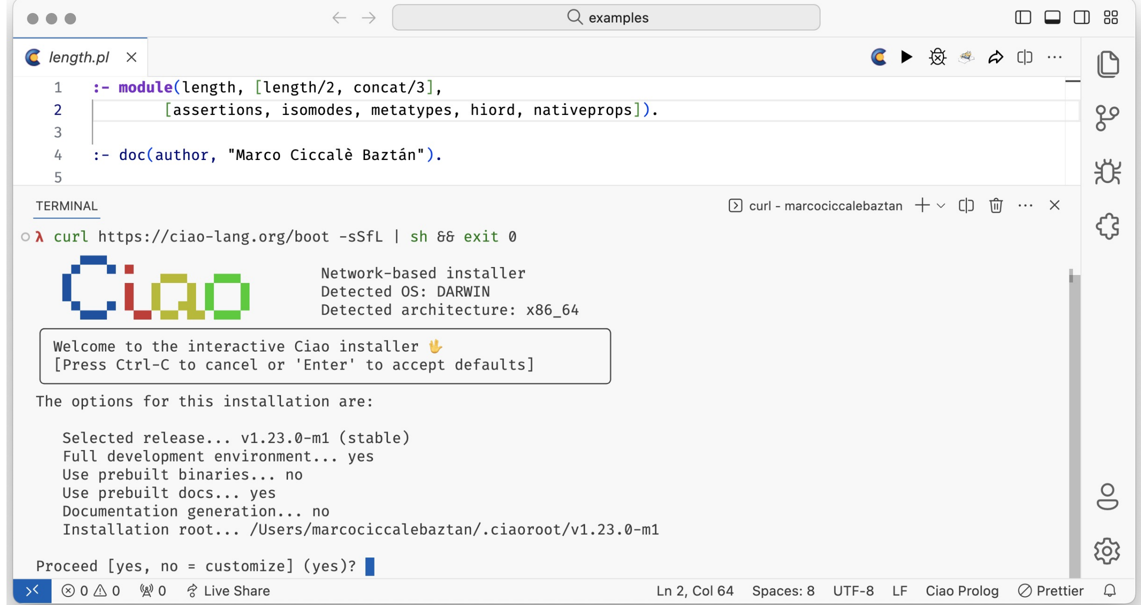

Figure 4.21: Ciao Installer Inside *VS Code*.

4.8 Ciao Installation and Version Management

Novice software engineers often encounter issues when installing new software tools or managing their versions [28]. To address this, the Ciao Development Team has worked diligently to simplify the installation process of the Ciao ecosystem. Despite these efforts, some beginners, particularly students, still struggle when installing the Ciao ecosystem. To further ease this process, users can now download the Ciao toolchain and manage different Ciao versions directly from the Ciao *VS Code* extension. It is important to note that this feature is currently available only in the *VS Code* Ciao development environment, but there are plans to extend it to the *Emacs* mode for Ciao also.

When a user installs the Ciao *VS Code* extension and opens a `.pl` file, the extension will check for the `ciao` command inside the `PATH` environment variable (by default, when Ciao is installed, it adds its executables directory to the `PATH`). If it is not found, an alert within *VS Code* will pop up indicating that the user does not have Ciao installed on the system (see Figure 4.20). This alert will also include a button for downloading the Ciao system. If clicked, a new terminal will be spawned with the Ciao interactive installer process running, allowing users to follow the installation instructions all within *VS Code* (see Figure 4.21).

Figure 4.22: Ciao Version Picker Inside *VS Code*.

Additionally, users with multiple Ciao versions installed on their system can easily select which version to use when working with the integrated *top-level* in *VS Code*. This can be done using a simple *VS Code* command that allows users to choose their preferred version. By default, this command will list the Ciao version found in the `PATH`, as well as the versions located in the default installation directory `$HOME/.ciaoroot` (see Figure 4.22). Once a version is selected, the `ciao-env` process from the binaries folder of that version is executed to obtain the necessary environment variables needed to use that version, which are then stored in *VS Code's* local storage. Finally, when spawning any integrated *top-level*, if the local storage contains any custom environment variables, they are loaded during the *top-level* process initiation (see the source code below).

```
import { spawn } from "node:child_process";

// If there is a version of Ciao specified, retrieve its environment variables
// If not, simply return an empty object.
const ciaoEnv = getCiaoEnvVariables() ?? {};

// Load environment variables when spawning the top-level process
const cproc = spawn('ciaosh', {
  env: {
    // Keep the current environment variables
    ...process.env,
    // Load ciaoEnv to the environment variables of the process
    // If ciaoEnv is empty, no changes are made
    ...ciaoEnv,
  }
});
```

Moreover, if users have Ciao versions installed in non-standard locations, they

can register custom versions using *VS Code's* settings file by providing a name and a path to that version. These user defined versions will also be considered when listing all Ciao versions in the system.

4.9 Multiplatform Support

4.9.1 Operating Systems

The Ciao system supports multiple UNIX-based systems, including various *Linux* distributions such as *Debian*, *Ubuntu*, and *Arch Linux*, as well as *macOS*. Regarding *Windows*, Ciao does not fully support it natively at the moment. This could be seen as a shortcoming, given that *Windows* is the most widely used operating system for both personal and professional purposes among software engineers [11]. However, Ciao does offer partial support via *MSYS2/MinGW* and full support via *Windows Subsystem for Linux* [29] (*WSL*), developed by *Microsoft*, which allows the execution of a *Linux* distribution directly on *Windows* and thus provides *Windows* users with a UNIX-based development environment within their system. *VS Code* in turn offers a *WSL* extension that sets up a *WSL*-based development environment within the editor.[5] This enables developers to use *VS Code* to interact seamlessly with the *WSL* system.

The Ciao *VS Code* extension provides the same functionality for both native UNIX-based operating systems and *Windows* with *WSL*, including handling the logic of opening external applications like the browser.

4.9.2 Cloud Development Environments

Cloud Development Environments (*CDEs*) offer preconfigured, remote development environments hosted in the cloud, eliminating the need for developers to configure *on-premise* environments. Their popularity is expected to rise significantly in the coming years due to their ease of use [30].

Examples of *CDEs* include *GitHub Codespaces*, *Gitpod*, *AWS Cloud9*, *Google Cloud Shell*, *Repl.it*, *Codeanywhere*, and *Eclipse Che*. While not all *CDEs* are based on *VS Code*, those like *GitHub Codespaces* and *Gitpod* leverage *VS Code*, making them compatible with *VS Code* extensions.

The Ciao *VS Code* extension also supports *VS Code*-based *CDEs*, such as *GitHub Codespaces* or *Gitpod*. For example, users can create a *GitHub Codespace* directly from Ciao's GitHub repository.[6] After creating the *Codespace*, it will be built as a Docker image, which includes the latest stable version of Ciao together with the Ciao *VS Code* extension installed by default. This setup ensures that developers can immediately start working in a fully configured *VS Code*-based remote environment for Ciao (see Figures 4.23 and 4.24).

[5] `https://code.visualstudio.com/docs/remote/wsl`

[6] `https://github.com/ciao-lang/ciao`

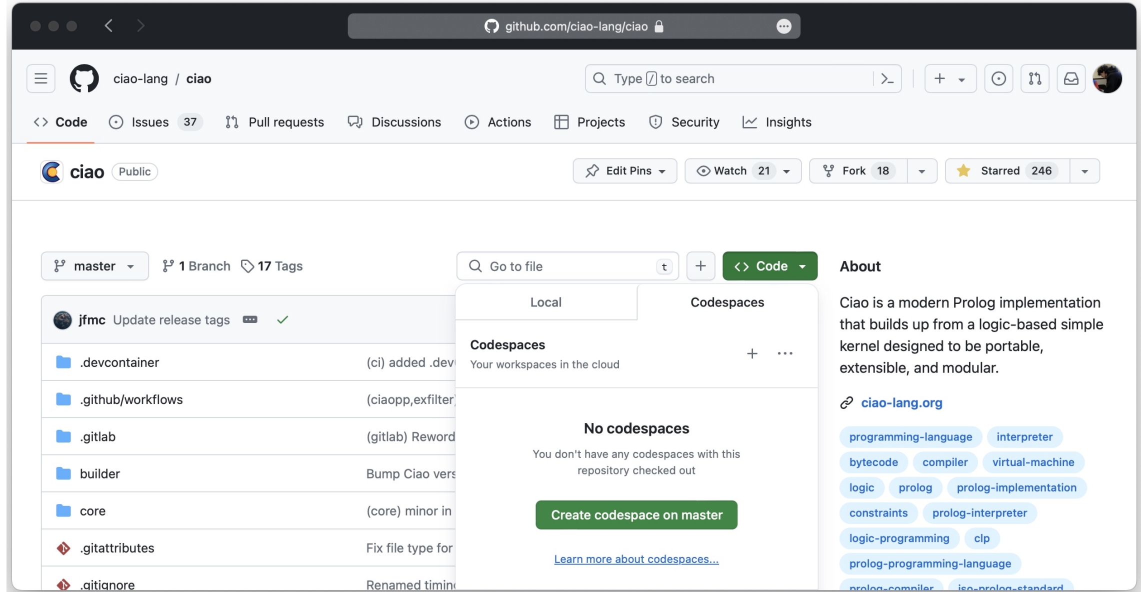

Figure 4.23: *GitHub Codespace* creation from Ciao's GitHub Repository.

```
cautious-funicular-qg7pwrqrr53xpg9.github.dev
ciao [Codespaces: cautious funicular]
length.pl U                                  Ciao Top Level
examples > length.pl
 1  :- module(length, [length/2, concat/3],   ?- use_module('/workspaces/ciao/exampl
 2          [assertions, isomodes, metatypes   es/length.pl').
 3
 4  :- doc(author, "Marco Ciccalè Baztán").   yes
 5                                            ?- length([a,b,c,d,e],L).
 6  length([],    0).
 7  length([_|T], N) :- length(T, N0), N is   L = 5 ?
 8
 9  concat([],     L,  L).                    yes
10  concat([X|L1], L2, [X|L3]) :- concat(L1,  ?-
11
12  :- test length(X, Y) : (X = [1, 2, 3], Y
13  :- test length(X, Y) : (X = [a, b, c, d,
14  :- test length(X, Y) : (X = [], Y = 0) +
15  :- test length(X, Y) : (X = [a, b, c, d,
16  :- test length(X, Y) : (X = [a, b, c], Y
Codespaces: cautious funicular   master*   0 0   0   UTF-8   LF   Ciao Prolog   Layout: U.S.
```

Figure 4.24: Ciao *GitHub Codespace* environment.

4.10 Bundling and Publishing the Extension

An important part of the development process of the Ciao *VS Code* extension is the packaging and bundling process. This step is essential for installing and testing the extension locally, ensuring it matches the version that end users will eventually install. Moreover, only bundled extensions can run in *CDEs*.[7]

Bundling involves merging multiple source code files, including *TS* and *JS* files, into a single *JS* file. The goal of bundling is to reduce the number of requests made by the web browser. Bundlers can also compile *TS* code to a specific version of *JS*, ensuring compatibility with various browsers and runtimes.

Concretely, the Ciao *VS Code* extension is bundled using `esbuild`, an extremely fast *JS* bundler that is easy to configure. The bundled version of the Ciao *VS Code* extension consists of one *JS* file for the extension, one *JS* file for the language server, and a *JS* file together with a *CSS* file for the webview panel (see Appendix F for the bundling script).

After the extension is bundled, it can be published to the *VS Code* Marketplace, a public repository where developers can publish their extensions or download extensions created by the community. The Ciao *VS Code* extension was published under the **`ciao-lang`** verified organization using the `vsce` command-line tool for packaging and publishing *VS Code* extensions.[8] Moreover, for Gitpod's *CDE*, the extension was also published to the OpenVSX Registry, an open-source alternative to *VS Code's* marketplace where *VS Code*-compatible extensions can be published.

[7] `https://code.visualstudio.com/api/working-with-extensions/bundling-extension`
[8] `https://github.com/microsoft/vscode-vsce`

Chapter 5

Conclusions and Future Work

In this work, a completely new, innovative, and modern development environment for Ciao has been created as a *VS Code* language extension. This extension seamlessly integrates the Ciao ecosystem into one of the most popular text editors, providing a powerful and comprehensive yet user-friendly tool for developers of all experience levels with Ciao. It offers features such as custom syntax highlighting, an enriched integrated Ciao *top-level* with a powerful *API*, Ciao debugging integration with custom marks, CiaoPP integration with a graphical menu to configure its flags, LPdoc integration for generating documentation manuals of Ciao programs, and *on-the-fly* analysis with multiple checkers provided by a minimal Ciao language server among others. The development of this extension involved writing nearly 3,000 lines of code. Despite the extensive functionality, the extension is implemented in a way that keeps the number of lines as small as possible while delivering a robust set of features, making the extension more maintainable and scalable.The extension is published in *VS Code*'s Marketplace[1], and the full source code of the extension is publicly available in a GitHub repository.[2]

Arguably, the Ciao system has become significantly more discoverable and accessible due to the engagement and positive reception of the extension since its publication. With over 1,150 downloads, it is clear that the extension has filled a crucial need within the community, promoting broader adoption and exploration of Ciao's capabilities. Furthermore, students at UPM have used the extension for completing their Declarative Programming course's assignments, demonstrating its practical application and effectiveness in an academic setting.

While already very capable, there is certainly still room for improvement in the Ciao *VS Code* language extension. Below are a few possible *lines of future work*:

Enhance the `sendQuery` *API*: While the `sendQuery` *API* of the `CiaoTopLevel` component already includes the output of the *query* upon completion, meeting a critical requirement for the extension, there is still room for improvement. By

[1] `https://marketplace.visualstudio.com/items?itemName=ciao-lang.ciao-prolog-vsc`
[2] `https://github.com/ciao-lang/ciao_vsc`

incorporating additional information about the state of the *query*, such as variable bindings, the `sendQuery` *API* can be made more robust. This enhancement would facilitate the creation of a general-purpose *API* for interacting with the Ciao ecosystem from *TypeScript*.

Implement a complete Ciao language server: In this work, and initial version of the Ciao language server was implemented using *TypeScript*. However, developing the language server using the Ciao language itself would offer invaluable advantages. This approach would enable the provision of comprehensive programming language features to any text editor that integrates with *LSP*.

Integrate the Ciao debugger using the Debug Adapter Protocol (DAP): The Ciao debugger has been deeply integrated within *VS Code* according to the other Ciao development environments, ensuring a consistent user experience across all of them. However, in recent years, a debugging abstract protocol has emerged with the same purpose of the *LSP*, to standardize the interaction between a text editor and a debugger [31]. The *Debug Adapter Protocol* (*DAP*) is natively supported in *VS Code*, and it already powers the debugging experience of other languages like *Java* or *Node.js*. Further work is needed to fully leverage the *DAP* and provide an overall better experience when debugging Ciao programs inside *VS Code* or any text editor that supports *DAP*.

Chapter 6

Impact Analysis

After having explored the goal and results of the Ciao Language extension for *VS Code*, it can be assured that the Ciao ecosystem will be much more accessible for software engineers with any level of expertise around the globe, since the whole set of tools of the Ciao ecosystem are now natively supported in the most popular code editor in the industry, Visual Studio Code. Furthermore, since Ciao is taught to students who may lack expertise in Declarative Programming, providing an accessible environment adapted to a well-known editor may positively affect their learning process by reducing the time spent in learning how to use a new tool such as *Emacs*.

Nevertheless, *IDEs* may negatively affect inexperienced users [32]. For example, they may become dependent on the *IDE* buttons and facilities and not truly deepen in the manual interaction with the Ciao ecosystem. However, as most features provided in the extension are just interacting with the integrated *top-level*, if the user wants to find out what is happening behind the scenes, they can simply read and explore the *top-level* history of queries and results.

Taking a look at the objectives of the Agenda 2030, this project could be framed in Objective 9, supporting research and increasing the development of technology. In particular, this work will make CiaoPP, the Ciao preprocessor, more accessible to software engineers by providing a seamless interface to work with it, making it easier for programmers to implement optimization and analysis techniques in their development workflow. This can actually represent a contribution to reducing the energy footprint of IT: in fact, the static analyses that CiaoPP performs make it possible to infer resource consumption metrics from the source code without executing it, verifying that a system conforms to its energy specifications or detecting energy consumption violations [33]. This also allows programmers to optimize complex systems to reduce their resource consumption. By making the use of CiaoPP more convenient, the Ciao *VS Code* extension can contribute to the reduction of the energy footprint of IT and can thus help comply with the sustainable development objective of the Agenda 2030.

Bibliography

[1] A. N. Meyer, E. T. Barr, C. Bird, and T. Zimmermann, "Today was a good day: The daily life of software developers," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 863–880, 2019.

[2] I. Zayour and H. Hajjdiab, "How much integrated development environments (ides) improve productivity?" *JSW*, vol. 8, no. 10, pp. 2425–2431, 2013.

[3] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, and G. Puebla, "An Overview of Ciao and its Design Philosophy," *TPLP*, vol. 12, no. 1–2, pp. 219–252, 2012. [Online]. Available: http://arxiv.org/abs/1102.5497

[4] M. Hermenegildo, "An Abstract Machine for Restricted AND-parallel Execution of Logic Programs," in *ICLP'86*, ser. LNCS, vol. 225. Springer-Verlag, 1986, pp. 25–40.

[5] M. Hermenegildo and K. Greene, "The &-Prolog System: Exploiting Independent And-Parallelism," *New Generation Computing*, vol. 9, no. 3,4, pp. 233–257, 1991.

[6] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia, "Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)," *Science of Computer Programming*, vol. 58, no. 1–2, pp. 115–140, October 2005.

[7] M. V. Hermenegildo, F. Bueno, G. Puebla, and P. Lopez-Garcia, "Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor," in *1999 Int'l. Conference on Logic Programming*. Cambridge, MA: MIT Press, November 1999, pp. 52–66.

[8] M. V. Hermenegildo, "A Documentation Generator for (C)LP Systems," in *International Conference on Computational Logic, CL2000*, ser. LNAI, no. 1861. Springer-Verlag, July 2000, pp. 1345–1361.

[9] M. Birkenkrahe, "Teaching data science with literate programming tools," *Digital*, vol. 3, no. 3, pp. 232–250, 2023.

[10] Microsoft, "Visual studio code," 2015. [Online]. Available: https://code.visualstudio.com

[11] S. Overflow, "Stack overflow developer survey 2023," 2023. [Online]. Available: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-integrated-development-environment

[12] M. Umeda, "prolog.el," Online, 1986. [Online]. Available: https://github.com/emacs-mirror/emacs/blob/master/lisp/progmodes/prolog.el

[13] M. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, and M. V. Hermenegildo, "VeriFly: On-the-fly Assertion Checking via Incrementality," *Theory and Practice of Logic Programming*, vol. 21, no. 6, pp. 768–784, September 2021.

[14] G. Garcia-Pradales, J. Morales, and M. V. Hermenegildo, "The Ciao Playground," Technical University of Madrid (UPM) and IMDEA Software Institute, Tech. Rep., 2021. [Online]. Available: http://ciao-lang.org/ciao/build/doc/ciao_playground.html/ciao_playground_manual.html

[15] J. Morales, S. Abreu, D. Ferreiro, and M. Hermenegildo, "Teaching Prolog with Active Logic Documents," in *Prolog - The Next 50 Years*, ser. LNCS, D. S. Warren, V. Dahl, T. Eiter, M. Hermenegildo, R. Kowalski, and F. Rossi, Eds. Springer, July 2023, no. 13900, ch. 14, pp. 171–183. [Online]. Available: http://cliplab.org/papers/ActiveLogicDocuments-PrologBook.pdf

[16] A. Wang, "Vsc-prolog," 08 2017. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=arthurwang.vsc-prolog

[17] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[18] Electron.js, "Electron.js," 2013. [Online]. Available: https://www.electronjs.org

[19] F. Bueno, M. Carro, M. V. Hermenegildo, P. Lopez-Garcia, and J. M. (Eds.), "The Ciao System. Ref. Manual (v1.22)," Tech. Rep., April 2023, available at http://ciao-lang.org. [Online]. Available: http://ciao-lang.org

[20] Microsoft, "Visual studio code api," Online, 04 2015. [Online]. Available: https://code.visualstudio.com/api/references/vscode-api

[21] ——, "Get output from integrated terminal "sendtext"," Online, 11 2018, gitHub Issue #59384. [Online]. Available: https://github.com/microsoft/vscode/issues/59384

[22] ——, "Obtain tokens of current file," Online, 11 2015, gitHub Issue #580. [Online]. Available: https://github.com/microsoft/vscode/issues/580

[23] E. International, "ECMAScript 2015 Language Specification," 2015. [Online]. Available: https://www.ecma-international.org/ecma-262/6.0/

[24] MacroMates, "Textmate language grammars," 10 2004. [Online]. Available: https://macromates.com/manual/en/language_grammars

[25] S. Wiesner, "Flycheck: Modern on-the-fly syntax checking for gnu emacs," 2013. [Online]. Available: https://www.flycheck.org

[26] Microsoft, “Language server protocol,” 2016. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[27] H. Bünder, “Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages.” in *MODELSWARD*, 2019, pp. 129–140.

[28] L. Salerno, C. Treude, and P. Thongtatunam, “Open source software development tool installation: Challenges and strategies for novice developers,” 2024.

[29] Microsoft, “Windows subsystem for linux documentation,” 2016. [Online]. Available: https://learn.microsoft.com/en-us/windows/wsl/

[30] L. Perri, “What’s new in the 2023 gartner hype cycle for emerging technologies,” August 2023. [Online]. Available: https://www.gartner.com/en/articles/what-s-new-in-the-2023-gartner-hype-cycle-for-emerging-technologies

[31] Microsoft, “Debug adapter protocol,” 2017. [Online]. Available: https://microsoft.github.io/debug-adapter-protocol/

[32] E. Dillon, M. Anderson-Herzog, and M. Brown, “Teaching students to program using visual environments: Impetus for a faulty mental model?” *Journal of Computational Science Education*, vol. 5, no. 1, pp. 1–2, 2014.

[33] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo, “Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption,” *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification*, vol. 18, no. 2, pp. 167–223, March 2018, arXiv:1803.04451. [Online]. Available: https://arxiv.org/abs/1803.04451

Appendices

Appendix A

Source Code for CProc

```
1  import { ChildProcessWithoutNullStreams, spawn } from 'node:child_process';
2  import { workspace } from 'vscode';
3  import { EXE, PROMPTS } from '../constants';
4  import { markErrorsOnCiaoSource } from './ciao-file';
5  import { isDebuggerLine } from './ciao-dbg';
6  import {
7    type Resolver,
8    type OutputCallback,
9    type CiaoEnvVars,
10   type CiaoUserConfiguration,
11   CiaoTopLevelKind,
12 } from '../types';
13 import { parseErrorMsg } from './ciao-parse';
14 import { getGlobalValue } from '../managers/context-manager';
15
16 const getCwd = (): string =>
17   workspace.workspaceFolders ? workspace.workspaceFolders[0].uri.fsPath : '/';
18
19 const getExecutableInfo = (
20   procKind: CiaoTopLevelKind
21 ): { exe: string; args: string[] } => EXE[procKind];
22
23 export class CProc {
24   private outputCallback: OutputCallback;
25   private resolveCommand: Resolver | undefined;
26   private cproc: ChildProcessWithoutNullStreams | undefined;
27   private procKind: CiaoTopLevelKind;
28   private commandOutputBuf: string;
29   private stdoutBuf: string;
30   private stderrBuf: string;
31   private errors: string;
32   private muted: boolean;
33   private flushTimeout: NodeJS.Timeout | undefined;
34
35   /**
36    * Create a new CProc instance.
37    * @param {CiaoTopLevelKind} kind - Ciao, CiaoPP or LPdoc top-level.
38    * @param {(output: string) => void} outputCallback - Callback to execute once output
      ↪  data is available.
39    */
```

```
40    constructor(
41      procKind: CiaoTopLevelKind,
42      outputCallback: OutputCallback
43    ) {
44      this.outputCallback = outputCallback;
45      this.procKind = procKind;
46      this.commandOutputBuf = '';
47      this.stdoutBuf = '';
48      this.stderrBuf = '';
49      this.errors = '';
50      this.muted = false;
51    }
52
53    /**
54     * Start the Ciao top-level process.
55     * @returns {Promise<CProc>} - A promise that resolves to the CProc.
56     */
57    start(): Promise<CProc> {
58      const cwd: string = getCwd();
59
60      // Getting executable info
61      const { exe, args } = getExecutableInfo(this.procKind);
62
63      // Obtain ENV variables of the Ciao Version
64      const ciaoEnv: CiaoEnvVars = getGlobalValue('CIAO-ENV', {});
65
66      // Spawn the process
67      this.cproc = spawn(exe, args, {
68        cwd,
69        env: {
70          ...process.env,
71          ...ciaoEnv,
72        },
73      });
74
75      // Return a promise that sets all the listeners
76      return new Promise<CProc>((resolve) => {
77        this.cproc?.on('exit', this.handleExit);
78        this.cproc?.stderr.on('data', this.handleStderr);
79
80        // Setup a 'once' listener to treat differentely the initial
81        // prompt that is printed by 'ciaosh'
82        this.cproc?.stdout.once('data', (data: Buffer) => {
83          this.stdoutBuf += String(data);
84          // Printing the first prompt of the Ciao Top Level
85          this.outputCallback(this.stdoutBuf);
86          this.stdoutBuf = '';
87          // Setup the regular listener for subsequent data
88          this.cproc?.stdout.on('data', this.handleStdout);
89          resolve(this);
90        });
91      });
92    }
93
94    /** Flush the current output buffer. */
95    flush(): void {
96      if (!this.stdoutBuf) return;
97      this.outputCallback(this.stdoutBuf);
98      this.commandOutputBuf += this.stdoutBuf;
99      this.stdoutBuf = '';
```

```
100   }
101
102   /**
103    * Send an interrupt signal to the Ciao process.
104    * @returns {Promise<string>} - A promise that resolves to the top-level exit menu.
105    */
106   interrupt(): Promise<string> {
107     return new Promise<string>((resolve) => {
108       this.resolveCommand = resolve;
109       this.cproc?.kill('SIGINT');
110     });
111   }
112
113   /**
114    * Send a query to the Ciao process.
115    * @param {string} query - The query string.
116    * @param {boolean} muted - Whether to mute the output.
117    * @returns {Promise<string>} - A promise that resolves to the query output.
118    */
119   sendQuery(command: string, muted: boolean = false): Promise<string> {
120     this.muted = muted;
121     return new Promise<string>((resolve) => {
122       this.resolveCommand = resolve;
123       this.cproc?.stdin?.write(`${command}\n`);
124     });
125   }
126
127   /**
128    * Check if the top-level is still running.
129    * @returns {boolean}
130    */
131   isRunning(): boolean {
132     return (
133       !!this.cproc &&
134       this.cproc.exitCode === null &&
135       (this.cproc.signalCode === null || this.cproc.signalCode === 'SIGINT')
136     );
137   }
138
139   /** Terminate the Ciao top-level process. */
140   exit(): void {
141     this.cproc?.kill('SIGQUIT');
142   }
143
144   private isWaitingForQuit = (): boolean =>
145     this.stdoutBuf.endsWith(PROMPTS.PROMPT_QUIT.text);
146
147   private isWaitingForResponse = (): boolean =>
148     this.stdoutBuf.endsWith(PROMPTS.PROMPTVAL.text);
149
150   private isWaitingForInput = (): boolean =>
151     this.stdoutBuf.endsWith(PROMPTS[this.procKind].text) ||
152     this.isWaitingForResponse() ||
153     this.isWaitingForQuit();
154
155   private handleExit = (_code: number, _signal: string) => {
156     this.outputCallback('\r\nCiao Listener finished\r\n\n');
157   };
158
159   private handleStdout = (buffer: Buffer): void => {
```

```
160       // Buffering the data
161       this.stdoutBuf += String(buffer);
162
163       // Buffering all the output of the command
164       this.commandOutputBuf += String(buffer);
165
166       // Clear the previous flushTimeout
167       clearTimeout(this.flushTimeout);
168
169       // If there's data in STDERR, send it and reset buffer
170       if (this.stderrBuf.length > 0) {
171         this.errors = this.stderrBuf;
172         if (!this.muted) this.outputCallback(this.stderrBuf);
173         this.stderrBuf = '';
174       }
175
176       // Split the stdout in the last '\n'
177       const lines = this.stdoutBuf.split('\n');
178
179       // Do not add an additional new line character if the buffer only has one line,
180       // or if the previous line is a debugging information line.
181       const rest = `${
182         lines.length === 1 || isDebuggerLine(lines[lines.length - 2]) ? '' : '\n'
183       }${lines.pop()}`;
184       const data = lines.join('\n');
185
186       // Buffer the rest of data or restart buffer
187       this.stdoutBuf = rest ?? '';
188
189       // Send the data to the PTY
190       if (!this.muted) this.outputCallback(data);
191
192       // Set the flushTimeout in case there is data in stdout without newlines
193       this.flushTimeout = setTimeout(this.flush.bind(this), 300);
194
195       // When the command is finished, reset the buffer and resolve the promise
196       if (this.isWaitingForInput() && this.resolveCommand) {
197         // Print the promt
198         if (!this.muted) this.outputCallback(this.stdoutBuf);
199
200         // Check if the user wants to mark errors on save
201         const userConfiguration = workspace
202           .getConfiguration('ciao')
203           .get<CiaoUserConfiguration>('checker');
204
205         if (userConfiguration === 'off' && !this.muted) {
206           markErrorsOnCiaoSource(parseErrorMsg(this.errors));
207         }
208
209         // Resolve the promise
210         this.resolveCommand(this.commandOutputBuf);
211         // Resetting variables
212         this.resolveCommand = undefined;
213         this.commandOutputBuf = '';
214         this.stdoutBuf = '';
215         this.stderrBuf = '';
216         this.errors = '';
217         this.muted = false;
218       }
219     };
```

```
220
221   private handleStderr = (buffer: Buffer): void => {
222     // Buffering the data
223     this.stderrBuf += String(buffer);
224     // Buffering all the output of the command
225     this.commandOutputBuf += String(buffer);
226   };
227 }
```

Appendix B

Source Code for Parsing Ciao Error Messages

```
import type { CiaoDiagnosticInfo, CiaoDiagnostics } from '../types';
/**
 * Parses the compilation messages generated by the Ciao top-level.
 * @param msgs
 * @returns An object containing the parsed messages.
 */
export function parseErrorMsg(msgs: string): CiaoDiagnostics {
  const warnings: CiaoDiagnosticInfo[] = [];
  const errors: CiaoDiagnosticInfo[] = [];
  const regexp =
  ↪  /{[^{}]*\b(WARNING|ERROR|Reading|In|Compiling|Checking|Loading)\b([^^]+)}/g;
  const w_regexp = /(Reading|In|Compiling|Checking|Loading)/g;
  msgs.match(regexp)?.forEach((e) => {
    let lines: string | undefined;
    let msg: string | undefined;
    if (/{SYNTAX (ERROR|WARNING)/.test(e)) return;
    if (w_regexp.test(e)) {
      e.split('\n')
        .filter((line) => line.includes('WARNING') || line.includes('ERROR'))
        .forEach((line) => {
          const errmsg = line.slice(line.indexOf(':') + 2);
          if (line.includes('lns')) {
            lines = errmsg.slice(errmsg.indexOf('(') + 5, errmsg.indexOf(')'));
            msg = errmsg.slice(errmsg.indexOf(')') + 2);
          } else {
            lines = undefined;
            msg = errmsg;
          }
          if (line.includes('WARNING')) {
            warnings.push({ lines, msg });
          } else if (line.includes('ERROR')) {
            errors.push({ lines, msg });
          } else {
            return;
          }
        });
    } else {
```

```
37        const errmsg = e.slice(e.indexOf(':') + 2);
38        if (e.includes('lns')) {
39          lines = errmsg.slice(errmsg.indexOf('(') + 5, errmsg.indexOf(')'));
40          msg = errmsg.slice(errmsg.indexOf(')') + 2, errmsg.indexOf('}') - 1);
41        } else {
42          lines = undefined;
43          msg = errmsg.slice(0, errmsg.indexOf('}') - 1);
44        }
45        if (e.includes('WARNING')) {
46          warnings.push({ lines, msg });
47        } else if (e.includes('ERROR')) {
48          errors.push({ lines, msg });
49        }
50      }
51    });
52    return { errors, warnings };
53  }
```

Appendix C

Source Code for Marking Ciao Debugger Steps

```
import type { DebugMark } from '../types';
import { translatePath } from './ciao-file';

const dbgMarkRegex = / {0,9}In (.*) \(([0-9]+)-([0-9]+)\) (.*?)-([0-9]+)/;

/**
 * Determines wether the line is a debugging line or not
 * @param line Output line to check
 * @returns `true` if it is, `false` otherwise
 */
export function isDebuggerLine(line: string): boolean {
  return dbgMarkRegex.test(line);
}

/*
 * Useful regexes?
 *
 *     [/^   [0-9]+  [0-9]+  Call: /m, 'comment'],
 *     [/^   [0-9]+  [0-9]+  Exit: /m, 'comment'],
 *     [/^   [0-9]+  [0-9]+  Redo: /m, 'comment'],
 *     [/^   [0-9]+  [0-9]+  Fail: /m, 'comment'],
 *
 */

/**
 * Parses a debugger message and returns the information
 * @param msg Debugger message to parse
 * @returns Object containing all the extracted information
 */
export function parseDbgMsg(msg: string): DebugMark | undefined {
  const match = msg.match(dbgMarkRegex);
  if (!match) return;
  const [, srcFile, startLine, endLine, predName, nthPred] = match;
  return {
    predName,
    srcFile: translatePath(srcFile),
    nthPred: Number(nthPred),
```

```
38      startLine: Number(startLine) - 1,
39      endLine: Number(endLine) - 1,
40    };
41  }
42
43  /**
44   * Sets the current debugger mark in source.
45   * @param Object containing the parsed info of the debug mark.
46   */
47  export function markDbgMarksOnCiaoSource({
48    srcFile,
49    startLine,
50    endLine,
51    predName,
52    nthPred,
53  }: DebugMark): void {
54    const activeEditor = window.activeTextEditor;
55    // Check if the active editor is not the same as the file being debugged
56    if (activeEditor?.document.fileName !== srcFile) return;
57    // Creating Range
58    const range = new Range(
59      activeEditor.document.lineAt(startLine).range.start,
60      activeEditor.document.lineAt(endLine).range.end
61    );
62    // Get the chunk of code from source to parse
63    const code = getActiveCiaoFileContent(range);
64    // Check if there's code
65    if (!code) return;
66    // Tokenize the chunk of code
67    const tokens = ciaoTokenize(code);
68
69    // Search the line of the nth predName
70    let count = 0;
71    let predLine = -1;
72    let row = -1;
73
74    for (const token of tokens) {
75      // Found an atom including the name
76      if (token.kind === 'atom' && token.text.includes(predName)) {
77        count += 1;
78        if (count === nthPred) {
79          // The count is finished
80          row = token.position.row;
81          predLine = startLine + token.position.line;
82          break;
83        }
84      }
85    }
86
87    // Check if the line was found
88    if (predLine === -1 || row === -1) return;
89
90    // Create one line range
91    const lineToMark = new Range(
92      activeEditor.document.lineAt(predLine).range.start,
93      activeEditor.document.lineAt(predLine).range.end
94    );
95
96    const atomToMark = new Range(
97      new Position(predLine, row),
```

```
98      new Position(predLine, row + predName.length)
99    );
100
101   // Mark the line on source
102   activeEditor.setDecorations(debuggerDecorationType, [lineToMark]);
103   // Mark the specific atom on source
104   activeEditor.setDecorations(debuggerDecorationAtom, [atomToMark]);
105   // Focus the line at the center of the source file
106   activeEditor.revealRange(lineToMark, TextEditorRevealType.InCenter);
107 }
```

Appendix D

Source Code for CiaoPP Menu Webview Panel

```
1  import {
2      window,
3      Disposable,
4      ViewColumn,
5      WebviewPanel,
6      Webview,
7      Uri,
8  } from 'vscode';
9  import { getNonce, getUri } from '../utils';
10 import { CiaoTopLevel } from '../ciao-utils/ciao-top-level';
11 import { WebviewMessage } from '../types';
12
13 export class CiaoPPMenuPanel {
14     public static currentPanel: CiaoPPMenuPanel | undefined;
15     private readonly _panel: WebviewPanel;
16     private _disposables: Disposable[] = [];
17     private _topLevel: CiaoTopLevel;
18
19     private constructor(
20         panel: WebviewPanel,
21         extensionUri: Uri,
22         menudef: string,
23         topLevel: CiaoTopLevel
24     ) {
25         this._panel = panel;
26         this._panel.onDidDispose(() => this.dispose(), null, this._disposables);
27         this._panel.webview.html = this._getWebviewContent(
28             this._panel.webview,
29             extensionUri,
30             menudef
31         );
32         this._topLevel = topLevel;
33         this._setWebviewMessageListener(this._panel.webview, this._topLevel);
34     }
35
36     public static render(
37         extensionUri: Uri,
```

```
38            menudef: string,
39            topLevel: CiaoTopLevel
40        ): void {
41            if (CiaoPPMenuPanel.currentPanel) {
42                CiaoPPMenuPanel.currentPanel._panel.reveal(ViewColumn.Beside);
43                return;
44            }
45            const panel = window.createWebviewPanel(
46                'ciaopp-menu-panel',
47                'CiaoPP Menu',
48                ViewColumn.Beside,
49                {
50                    // Enable JavaScript inside the Webview
51                    enableScripts: true,
52                    // Restrict the webview to only load resources from the
                  ↪  `out/client/src/webview`
53                    localResourceRoots: [
54                        Uri.joinPath(
55                            extensionUri,
56                            'out',
57                            'client',
58                            'src',
59                            'webview'
60                        ),
61                    ],
62                }
63            );
64            CiaoPPMenuPanel.currentPanel = new CiaoPPMenuPanel(
65                panel,
66                extensionUri,
67                menudef,
68                topLevel
69            );
70        }
71
72        public dispose(): void {
73            CiaoPPMenuPanel.currentPanel = undefined;
74            this._panel.dispose();
75            while (this._disposables.length > 0) {
76                const disposable = this._disposables.pop();
77                if (disposable) {
78                    disposable.dispose();
79                }
80            }
81        }
82
83        private _getWebviewContent(
84            webview: Webview,
85            extensionUri: Uri,
86            menudef: string
87        ): string {
88            const nonce = getNonce();
89            const webviewUri = getUri(webview, extensionUri, [
90                'out',
91                'client',
92                'src',
93                'webview',
94                'webview.js',
95            ]);
96            const stylesUri = getUri(webview, extensionUri, [
```

```
97              'out',
98              'client',
99              'src',
100             'webview',
101             'styles.css',
102         ]);
103         return /*html*/ `<!DOCTYPE html>
104 <html lang="en">
105 <head>
106   <meta charset="UTF-8">
107   <meta name="viewport" content="width=device-width, initial-scale=1.0">
108   <meta http-equiv="Content-Security-Policy" content="default-src 'none'; style-src
    ↪  ${webview.cspSource}; script-src 'nonce-${nonce}';">
109   <link rel="stylesheet" href="${stylesUri}">
110   <title>CiaoPP Menu</title>
111 </head>
112 <body>
113   <h1>CiaoPP Menu</h1>
114   <span id="menudef">${menudef}</span>
115   <script type="module" nonce="${nonce}" src="${webviewUri}"></script>
116 </body>
117 </html>`;
118     }
119
120     private _setWebviewMessageListener(
121         webview: Webview,
122         topLevel: CiaoTopLevel
123     ): void {
124         webview.onDidReceiveMessage(
125             (message: WebviewMessage) => {
126                 const { command, text = '' } = message;
127                 switch (command) {
128                     case 'submit': {
129                         if (text) {
130                             topLevel.sendQuery(text);
131                         }
132                         return;
133                     }
134                     case 'error': {
135                         window.showErrorMessage(
136                             text || 'CiaoPP Menu could not be generated succesfully.'
137                         );
138                         return;
139                     }
140                     default:
141                 }
142             },
143             undefined,
144             this._disposables
145         );
146     }
147 }
```

Appendix E

Source Code for Ciao Language Server

```
1   'use strict';
2
3   import * as path from 'node:path';
4   import { fileURLToPath } from 'node:url';
5   import { writeFileSync } from 'node:fs';
6   import { spawnSync } from 'node:child_process';
7   import {
8       createConnection,
9       _Connection,
10      TextDocuments,
11      Diagnostic,
12      DiagnosticSeverity,
13      ProposedFeatures,
14      TextDocumentSyncKind,
15      InitializeResult,
16  } from 'vscode-languageserver/node';
17  import { TextDocument } from 'vscode-languageserver-textdocument';
18  import type {
19      CiaoDiagnosticInfo,
20      CiaoDiagnostics,
21      CiaoChecker,
22      CiaoUserConfiguration,
23  } from './types';
24  import { parseErrorMsg } from './ciao-utils/ciao-parse';
25
26  const flycheckSuffix = '_flycheck_tmp_co';
27  const connection = createConnection(ProposedFeatures.all);
28  const documents: TextDocuments<TextDocument> = new TextDocuments(TextDocument);
29
30  // Timer to wait for the user to stop typing
31  let compileTimer: NodeJS.Timeout | undefined;
32
33  connection.onInitialize(() => {
34      const result: InitializeResult = {
35          capabilities: {
36              textDocumentSync: TextDocumentSyncKind.Incremental,
37          },
```

```
38      };
39      return result;
40  });
41
42  documents.onDidChangeContent((change) => {
43      // If the user opens a tmp file, do not check it
44      if (
45          path
46              .basename(fileURLToPath(change.document.uri))
47              .includes(flycheckSuffix)
48      ) {
49          return;
50      }
51      // Minidelay so it starts when the user stops typing
52      clearTimeout(compileTimer);
53      compileTimer = setTimeout(() => {
54          validateTextDocument(change.document);
55      }, 200);
56  });
57
58  function createDiagnostics(
59      msgs: CiaoDiagnosticInfo[],
60      severity: DiagnosticSeverity
61  ): Diagnostic[] {
62      return msgs.map(({ lines, msg }) => {
63          // If the lines are not specified, hardcode it to the top of the file
64          const [startLine, endLine] = lines ? lines.split('-') : ['1', '1'];
65          const diagnostic: Diagnostic = {
66              severity,
67              message: msg ?? '',
68              range: {
69                  start: {
70                      line: Number(startLine) - 1,
71                      character: 0,
72                  },
73                  end: {
74                      line: Number(endLine) - 1,
75                      character: 20_000,
76                  },
77              },
78          };
79          return diagnostic;
80      });
81  }
82
83  async function validateTextDocument(textDocument: TextDocument): Promise<void> {
84      const { dir, ext, name } = path.parse(fileURLToPath(textDocument.uri));
85
86      const tmpFilePath = path.join(dir, `${name}${flycheckSuffix}${ext}`);
87
88      const checker: CiaoChecker | undefined = await getCiaoChecker(tmpFilePath);
89      if (!checker) {
90          return;
91      }
92
93      writeFileSync(tmpFilePath, textDocument.getText());
94      const { stderr } = spawnSync(checker.executable, checker.args);
95
96      // Parse messages
97      const { errors, warnings }: CiaoDiagnostics = parseErrorMsg(String(stderr));
```

```
98      // Send the computed diagnostics to VSCode.
99      connection.sendDiagnostics({
100         uri: textDocument.uri,
101         diagnostics: [
102             ...createDiagnostics(errors, DiagnosticSeverity.Error),
103             ...createDiagnostics(warnings, DiagnosticSeverity.Warning),
104         ],
105     });
106 }
107 
108 const ciaoCheckerTable: { [k in CiaoUserConfiguration]: CiaoChecker } = {
109     off: {
110         executable: 'false',
111         args: [],
112     },
113     ciaopp: {
114         executable: 'ciaopp',
115         args: ['-op', flycheckSuffix, '-V'],
116     },
117     lpdoc: {
118         executable: 'lpdoc',
119         args: ['-t', 'nil', '-op', flycheckSuffix],
120     },
121     ciaoc: {
122         executable: 'ciaoc',
123         args: ['-c', '-op', flycheckSuffix],
124     },
125 };
126 
127 async function getCiaoChecker(
128     filePath: string
129 ): Promise<CiaoChecker | undefined> {
130     const userConfiguration = <CiaoUserConfiguration>(
131         await connection.workspace.getConfiguration('ciao.checker')
132     );
133     const checker = ciaoCheckerTable[userConfiguration];
134     if (checker.executable === 'false') {
135         return;
136     } else {
137         // Append the filePath to the arg list
138         checker.args.push(filePath);
139         return checker;
140     }
141 }
142 
143 connection.onDidChangeWatchedFiles(() => {});
144 
145 documents.listen(connection);
146 
147 connection.listen();
```

Appendix F

Script for Bundling the Extension

```
const path = require('node:path');
const { build } = require('esbuild');
const { copy } = require('esbuild-plugin-copy');

///@ts-check
/**@typedef {import('esbuild').BuildOptions} BuildOptions **/

/**@type BuildOptions */
const baseConfig = {
    bundle: true,
    minify: process.env.NODE_ENV === 'production',
    sourcemap: process.env.NODE_ENV !== 'production',
};

/**@type BuildOptions */
const extensionConfig = {
    ...baseConfig,
    platform: 'node',
    mainFields: ['module', 'main'],
    format: 'cjs',
    external: ['vscode'],
};

/**@type BuildOptions */
const clientConfig = {
    ...extensionConfig,
    entryPoints: [path.resolve(__dirname, 'src', 'extension.ts')],
    outfile: path.resolve(__dirname, 'out', 'client', 'src', 'extension.js'),
};

/**@type BuildOptions */
const serverConfig = {
    ...extensionConfig,
    entryPoints: [path.resolve(__dirname, 'src', 'server.ts')],
    outfile: path.resolve(__dirname, 'out', 'server', 'src', 'server.js'),
};

```

```
38 /**@type BuildOptions */
39 const webviewConfig = {
40     ...baseConfig,
41     target: 'es2020',
42     format: 'esm',
43     entryPoints: [path.resolve(__dirname, 'src', 'webviews', 'main.ts')],
44     outfile: path.resolve(
45         __dirname,
46         'out',
47         'client',
48         'src',
49         'webview',
50         'webview.js'
51     ),
52     plugins: [
53         copy({
54             resolveFrom: 'cwd',
55             assets: {
56                 from: [
57                     path.resolve(__dirname, 'src', 'webviews', 'styles.css'),
58                 ],
59                 to: [
60                     path.resolve(__dirname, 'out', 'client', 'src', 'webview'),
61                 ],
62             },
63         }),
64     ],
65 };
66
67 (async () => {
68     try {
69         await Promise.all(
70             [clientConfig, serverConfig, webviewConfig].map(build)
71         );
72         console.log('Build complete!\n');
73     } catch (err) {
74         console.error(err);
75         process.exit(1);
76     }
77 })();
```