OST
Eastern Switzerland
University of Applied Sciences

`VHDL formatter`

A standalone CLI tool for formatting VHDL code

by

David Bachmann · Dominik Bühler

supervised by

Dominic Klinger · Felix Morgner

2025-12-19
Revision: `a9cf898`

Semester Thesis · Autumn 2025

EASTERN SWITZERLAND UNIVERSITY OF APPLIED SCIENCES

Abstract

VHDL is an established hardware description language used in specialized digital design domains, particularly where reliability and determinism are essential. Despite its continued use, tooling support for VHDL remains limited compared with many other programming languages. In particular, there is no formatter that enables consistent coding conventions or seamless integration into automated development workflows.

This project introduces a standalone command-line formatter for VHDL, based on a structured parsing and pretty-printing pipeline. ANTLR is used as the parser generator, providing a CST derived from the publicly available antlr grammar for VHDL. The CST is translated into an AST. This representation avoids the complexity of full semantic analysis with a focus on providing everything necessary for deterministic formatting.

The final output is produced using a document-based pretty-printing strategy inspired by Philip Wadler's layout algebra. This method models formatting as the composition of layout primitives, such as indentation, line breaks and groups. To guarantee safety, the system implements a rigorous post-formatting verification layer that compares the semantic token streams of the input and output, ensuring that the code's logic is preserved bit-for-bit. As a result, the formatter produces consistent and structurally clean code, independent of the quality of the input.

Recognizing the diversity of VHDL styles, the formatter is designed to be configurable. This allows users to define project-specific rules for indentation, keyword casing, and alignment strategies, ensuring the tool supports existing organizational style guides rather than enforcing a single fixed standard.

While full VHDL-2008 support is still in progress, the formatter already handles a substantial subset of the language. Future work aims to provide cross-platform binaries and editor integration to support streamlined and maintainable VHDL development.

CONTENTS

1 Introduction

1.1 Problem Statement & Motivation

In modern software development, a robust tooling ecosystem is crucial for developer productivity and code quality. Essential tools like language servers, linters, and formatters have become standard practice, automating tasks, reducing cognitive overhead, and ensuring consistency across a codebase. This uniformity is especially vital for collaborative projects, as it makes code more readable and maintainable for all team members.

However, languages for specialized applications, such as VHDL used to program devices like FPGA, often lack this advanced tooling. A significant issue resulting from VHDL's case-insensitivity, which could lead to inconsistent code formatting between different developers. Such stylistic variation reduces readability, slows down development, and can introduce errors. This project is motivated by the clear need to address this aiming to create a robust VHDL formatter to standardize code quality, and bring the benefits of modern software development practices to hardware design.

1.2 Vision

Our vision is to develop a reliable and comprehensive CLI tool for formatting VHDL code. This tool will be simple to integrate into any project's workflow, whether for an individual developer or as part of a CI pipeline. While the tool is for VHDL, its underlying architecture should be designed to be language-independent, allowing it to semantically understand the code and handle complex syntax correctly. Ultimately, the goal is to create an open-source tool that not only standardizes code formatting but also serves as a foundational component for a more complete VHDL development environment, making the hardware design process more efficient and collaborative for the entire community.

1.3 State of the Art

The current VHDL tooling ecosystem lacks an efficient and practical code formatter suitable for modern development workflows. Although progress has been made in language server support, formatting functionality remains largely absent or insufficient. Existing tools either lack essential features, suffer from poor performance, or raise concerns regarding maintainability and transparency. As a result, no solution currently meets the requirements for seamless editor integration or reliable use in larger projects.

- The VHDL Language Server provides basic language server features such as diagnostics and code navigation. However, it does not include any formatting capabilities [1].
- An online formatter written in TypeScript offers only basic formatting functionality. The project is no longer maintained, has seen no activity for over four years, and is not suitable for multi-file projects or automated CI workflows [2].
- Another available formatter is distributed as a closed-source binary. While it provides basic functionality, the lack of source code limits transparency, makes long-term maintenance uncertain, and prevents community contributions. In addition, the binary shows runtime errors when executed without arguments, suggesting that development may have been discontinued before completion. The last release, dated 2024-05-26, further raises concerns about ongoing maintenance [3].
- The open-source project *vhdl-style-guide* focuses on enforcing coding conventions and consistency, with a strong emphasis on code formatting [4]. Although the tool is mature and feature-rich, it exhibits significant performance limitations. These issues make it impractical for interactive editor features such as *format-on-save* and for formatting larger source files.

2 Requirements

2.1 Specification

This section outlines the functional and non-functional requirements for the VHDL formatter, derived from analysis of existing tools and the specific needs of VHDL development workflows. The specification remains flexible to accommodate insights gained during implementation and testing phases.

2.1.1 Functional Requirements

- **Parse VHDL Code:** The tool must accurately parse VHDL code from single files or directory structures. The parser must support VHDL-2008 as the baseline standard, given its widespread industry adoption.
- **Apply Consistent Formatting:** The tool must enforce a standardized set of formatting rules including proper indentation, alignment of port maps and generic clauses, consistent whitespace usage, and configurable casing conventions for keywords, identifiers, and user-defined types.
- **Preserve Code Semantics:** The formatter must preserve the semantic equivalence of the code. No syntactic or behavioral changes should be introduced during the formatting process.
- **Reject Invalid Syntax:** The formatter must refuse to process syntactically incorrect code. When parsing fails, the tool should provide a clear diagnostic message indicating the nature of the error.
- **Normalize Casing:** The tool must address VHDL's case-insensitive nature by applying consistent casing rules throughout the codebase (e.g., uppercase keywords, lowercase user identifiers), improving code readability and maintainability.

- **Support Flexible Configuration:** The tool must support flexible configuration, allowing users to customize formatting via a configuration file (JSON/YAML) or command-line flags. Configurable options include indentation style, line width limits, and casing conventions.
- **Provide Command-Line Interface:** The tool must operate primarily as a CLI application to enable seamless integration with build systems, CI pipelines, and automated development workflows.
- **Flexible Output Options:** The formatter must support multiple output modes: writing to a specific file or to the standard output.
- **Enable Editor Integration:** The tool should be designed for integration with popular editors and IDEs (VS Code, Vim, Zed, etc.), supporting format-on-save functionality.

2.1.2 Non-Functional Requirements

- **Performance:** The formatter must provide responsive performance, formatting typical VHDL files (less than 1000 lines of code) in under $100\,\mathrm{ms}$. It should also be capable of handling project-scale directories (less than 100 files, each under 1000 lines of code) within $12\,\mathrm{s}$ on mid-range hardware.
- **Cross-Platform Compatibility:** The tool must run natively on major operating systems (Windows, macOS, Linux) without external runtime dependencies.
- **Modular Architecture:** The codebase must maintain clear separation between parsing, formatting logic, configuration management, and I/O operations to facilitate maintenance, testing, and future enhancements.
- **Extensible Design:** The architecture must accommodate future extensions including additional formatting rules and configuration options.
- **Comprehensive Observability:** The tool must provide detailed logging and clear error diagnostics.

3 System Architecture

This chapter outlines the high-level design and architectural components. The system is structured as a multi-stage pipeline, following a compiler-inspired architecture that transforms raw source code into a formatted output while ensuring semantic integrity.

The architecture is divided into four primary stages:

- **Frontend:** Analyzes raw VHDL source code and transforms it into a clean, type-safe representation.
- **AST:** Serves as the central semantic representation of the code, enriched with trivia such as comments and whitespace.
- **Backend:** Employs a document algebra to resolve layout, indentation, and line wrapping dynamically.
- **Verification:** A safety stage that performs a token-based comparison to guarantee that the formatting process has not altered the code's meaning.

The class diagrams presented throughout this chapter are simplified versions intended for demonstration with a focus on core structural relationships and architectural patterns.

3.1 Technology Stack

Architectural decisions for this project are guided by two primary technology choices:

- **Language:** C++23 was chosen for its performance and memory management, leveraging modern features like variants and "deducing this".
- **Parser Generator:** `ANTLR4` was chosen to generate the lexer and parser from a community-maintained VHDL grammar. The choice of parser generator directly influences the frontend architecture and is discussed further in Section 3.2.1.

Supporting infrastructure (build system, dependency management, testing framework, and CI) is detailed in Chapter 5.

3.2 The Frontend: Analysis and Abstraction

The Frontend transforms raw VHDL source code into a clean, type-safe AST, abstracting away the complexities of the grammar and parser implementation.

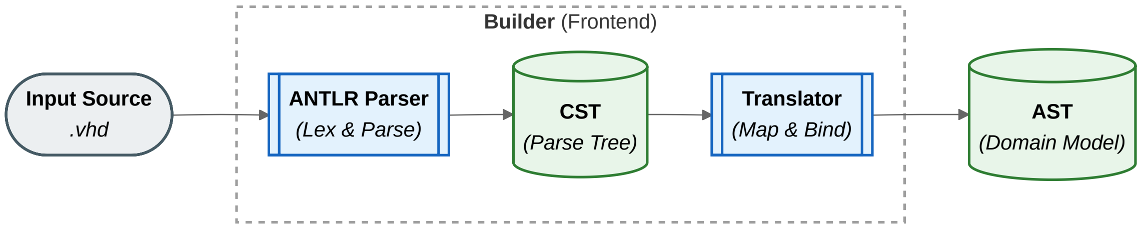

Figure 3.1: Frontend Pipeline: From Input Source to AST

As illustrated in Figure 3.1, the analysis process operates in two stages:

- **Parsing:** The source code is processed by a lexer and a parser to create a CST. This tree contains every detail of the original code, including punctuation and the full syntactic structure.
- **Translation:** The translation layer filters the detailed tree into a semantically focused AST. During this stage, the system identifies and binds comments to the correct nodes to ensure they are preserved for formatting.

3.2.1 Parsing Decision

Three parser generators were considered: `ANTLR` [5], a PEG-based generator (specifically `cpp-peglib` [6]), and `Tree-sitter` [7]. Given the 14-week project timeframe, grammar stability and strong support for VHDL-2008 were prioritized over a potentially more performant parser.

Among the candidates, ANTLR is the most suitable choice, a formally tested grammar covering VHDL-93,VHDL-2002 and most of VHDL-2008 already exists for the tool [8]. In contrast, the grammars available for `cpp-peglib` and `Tree-sitter` are incomplete or experimental. The `ANTLR` grammar therefore provides a solid foundation and avoids the overhead of debugging or extending a complex grammar.

While ANTLR is robust, its standard parsing mode can be performance-intensive. To mitigate this, the system employs a **Hybrid Parsing Strategy** (SLL parsing with LL parsing fallback), the implementation of which is detailed in Section 4.1. This parser produces a concrete representation that serves as the entry point for the transformation into a more manageable structure.

3.2.2 The Translator

Typically, a parser produces a detailed CST containing many nodes that are not needed for formatting. To simplify this structure, the `Translator` converts the CST into the high-level AST described in Section 3.3. Figure 3.2 illustrates the relationships between the translator, the node builder, and the trivia binder.

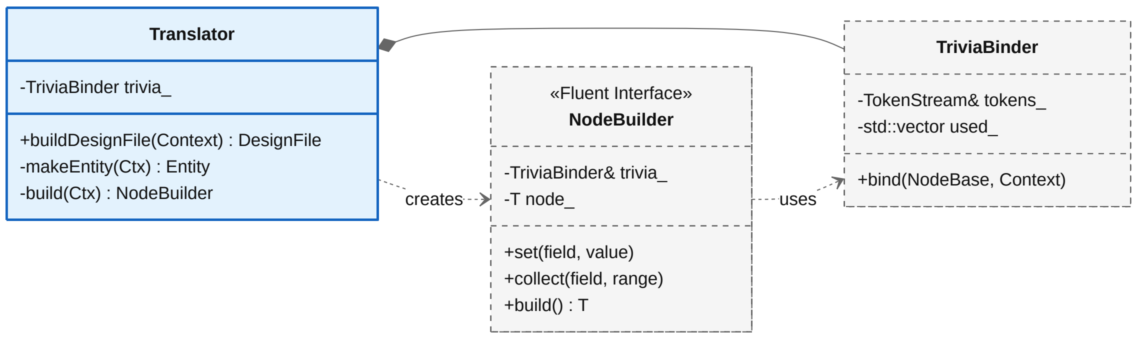

Figure 3.2: Translator Class Design: Utilizing a Fluent Builder for AST Construction

No semantic analysis or validation is performed by the `Translator`. Its sole responsibility is to map the syntactic structure of the CST into the semantic structure of the AST, preparing it for the formatting stages that follow.

3.2.3 Node Builder

Construction of nodes is streamlined by the `NodeBuilder` class. It uses a fluent interface to construct nodes via chained method calls. Listing 1 shows how the builder creates a waveform node.

```cpp
// src/builder/translators/statements/waveform.cpp
auto Translator::makeWaveform(vhdlParser::WaveformContext &ctx) -> ast::Waveform
{
    return build<ast::Waveform>(ctx)
      .set(&ast::Waveform::is_unaffected, ctx.UNAFFECTED() != nullptr)
      .collect(&ast::Waveform::elements,
               ctx.waveform_element(),
               [this](auto *el) { return makeWaveformElement(*el); })
      .build();
}
```

Listing 1: Fluent Builder Pattern in Action: Constructing a Waveform Node

Crucially, it automatically attaches comments through the `TriviaBinder`. Details on the implementation of this interface are provided in Section 4.2.

3.2.4 Trivia Binder

Collection of comments and blank lines from the hidden token channels of the parser is encapsulated within the `TriviaBinder`. These channels are separate streams where the lexer stores tokens that do not affect the logic of the code. The binder then attaches these tokens to the correct AST nodes as they are created. This process ensures that the original comments and intentional spacing are preserved, so the final output stays true to the user's intent.

Specific algorithms used to bind these tokens and handle vertical spacing heuristics are detailed in Section 4.3. Figure 3.3 illustrates how the lexer separates the token stream into separate channels (e.g., `DEFAULT`) and to which component each token belongs.

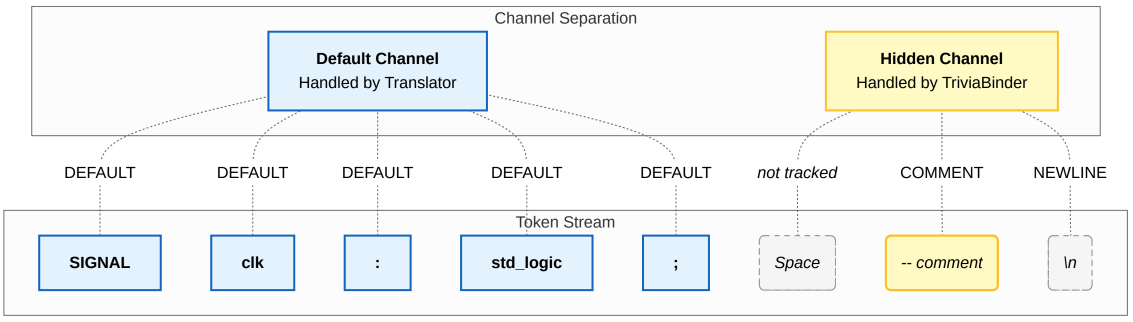

Figure 3.3: Token Channel Separation: Code vs. Trivia

3.2.5 Rationale

Translation of the CST to AST offers three key benefits:

- **Decoupling:** The core formatting logic depends solely on the custom AST, not on ANTLR. This ensures that the parser generator can be replaced in the future without requiring changes to the formatter.
- **Simplification:** The translation process filters out noise from the grammar, providing the formatting engine with a clean, semantic view of the code structure.
- **Type Safety:** Unlike the generic, string-based nature of the CST, the custom AST utilizes strong C++ types and `std::variant`. This leverages the C++23 compiler to enforce exhaustive handling of node types in the visitor, preventing invalid state access during formatting.

With the CST transformed into the semantic representation, the complexity of the parser is fully encapsulated within the frontend.

3.3 The Abstract Syntax Tree

Representing the Domain Model of this project, the AST is composed of nodes that model the semantics of VHDL code. Every node inherits from `NodeBase`, which is a base class that provides common storage for metadata and comments, also known as trivia. This structure is shown in Figure 3.4.

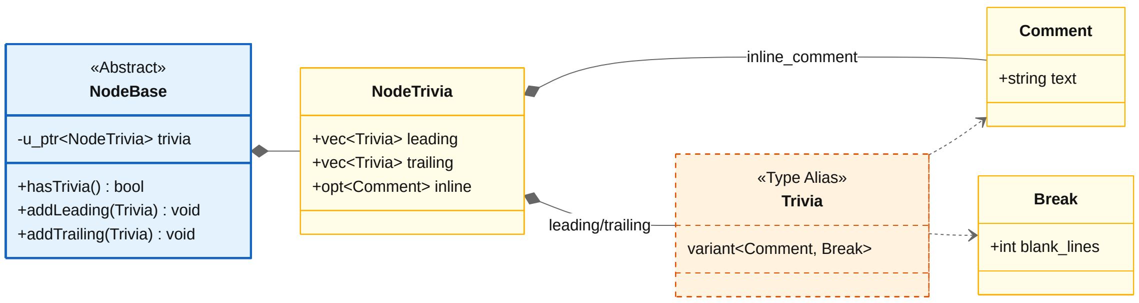

Figure 3.4: NodeBase Class Hierarchy with Trivia Storage

The design moves away from traditional virtual functions and instead uses C++17's `std::variant` for polymorphism. This allows related AST nodes to be grouped into unions, for example as shown in Listing 2, a `LibraryUnit` is a variant of either an `Entity` or an `Architecture`. This approach removes the speed penalty of virtual function calls, and it allows for better compiler optimizations.

```cpp
// src/ast/nodes/design_units.hpp
using LibraryUnit = std::variant<Entity, Architecture>;
```

Listing 2: Grouping related nodes using variants

3.3.1 Trivia Preservation

As described in Section 3.2.4, comments and blank lines are preserved by mapping them from the parser to the AST nodes. Each node inherits from `NodeBase` and can store a `NodeTrivia` structure. Trivia is split into three types:

- **Leading trivia:** Comments and blank lines appearing before the node.
- **Trailing trivia:** Comments and breaks appearing after the node.
- **Inline comment:** A single comment on the same line as the node's final token.

This categorization allows the AST to carry all relevant formatting information alongside the code structure, keeping the users intended comment positioning intact.

3.3.2 Memory Efficiency

The AST uses lazy allocation to save memory. The `NodeBase` class uses a `std::unique_ptr` for trivia, which means memory is only allocated for nodes that actually have comments. Additionally, recursive nodes like expressions use a `Box` alias for `std::unique_ptr`. This is a requirement for recursive types, and it also prevents unnecessary data copying.

3.3.3 Visitor Pattern

A generic `VisitorBase` template is used to traverse the tree. Using C++23 “deducing this”, the visitor achieves static polymorphism without the complex code of older patterns. This allows the base visitor to call methods in derived classes, such as the `PrettyPrinter` (Figure 3.6), at compile time. The visitor uses `std::visit` to find the active type in a variant and then calls the correct logic for it.

```cpp
// src/ast/visitor.hpp
template<typename ReturnType = void>
class VisitorBase {
  public:
    template<typename Self, typename T, typename... Args>
    auto visit(this const Self &self, const T &node, Args &&...args)
      -> ReturnType; // Handles concrete node types

    template<typename Self, typename... Ts, typename... Args>
    auto visit(this const Self &self, const std::variant<Ts...> &node, Args &&...args)
      -> ReturnType; // Dispatches variants to the correct overload
};
```

Listing 3: Visitor base class using C++23 "deducing this"

3.3.4 Compile-Time Safety

Using `std::variant` with the visitor pattern provides a major safety benefit. In traditional programming, forgetting to handle a new class might only cause an error at runtime. With this design, the compiler checks that every type in a variant is handled by the visitor. If a new node is added to a variant but the visitor is not updated, the code will not compile. This ensures that the formatter always knows how to process every node.

Finally, once the AST and its trivia are ready, the backend uses the visitor interface to generate the final formatted output.

3.4 The Backend: Layout and Generation

The Backend processes the AST to generate the final formatted output.

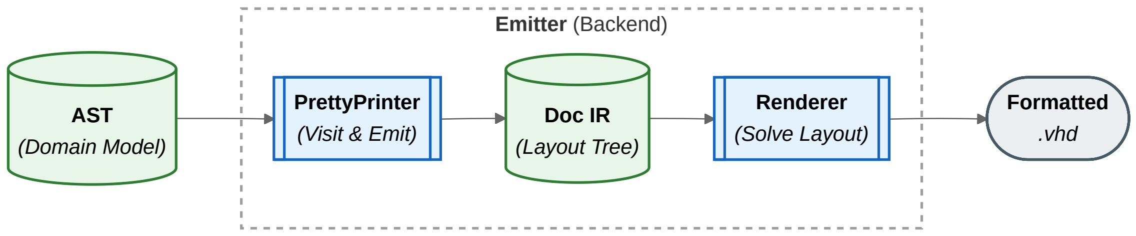

Figure 3.5: Backend Pipeline: From AST to Formatted Output

As illustrated in Figure 3.5, the formatting process operates in two stages:

- **Emission:** The `PrettyPrinter` moves through the AST using the Visitor pattern to build a `Doc`, also called the IR.

- **Rendering:** The `Renderer` processes the `Doc` tree, manages the current column position, and follows the `Config` for indentation and width.

3.4.1 Formatting Strategy

The formatting engine implements a document algebra based on the approach described by Philip Wadler in *"A Prettier Printer"* [9]. This strategy separates the definition of the output structure from the logic used for line wrapping and indentation.

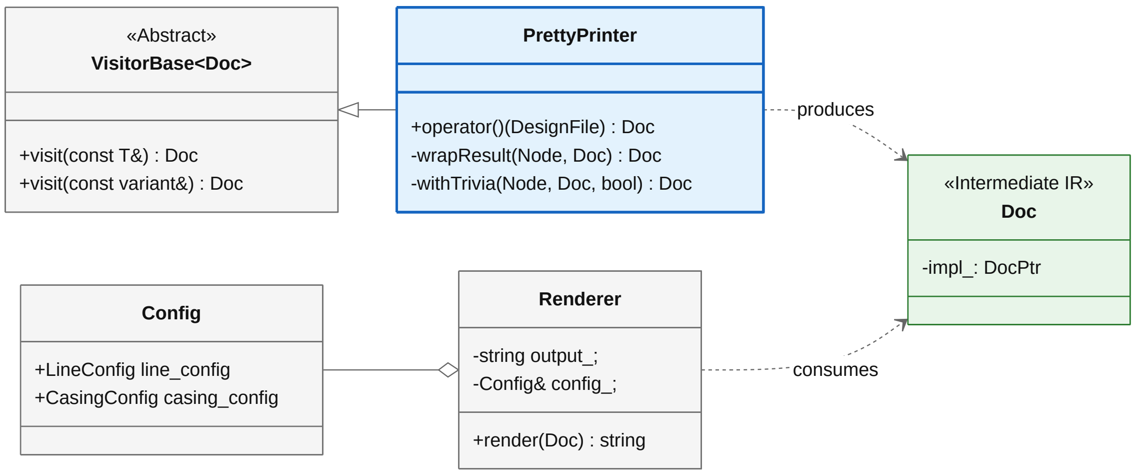

Figure 3.6: PrettyPrinter: Visiting AST Nodes and Generating Doc IR

As shown in Figure 3.6, the `PrettyPrinter` implements `VisitorBase<Doc>` to transform AST nodes into the `Doc` intermediate representation.

3.4.2 Document Class Overview

The `Doc` class is the central data structure for the backend, and it acts as an immutable handle that uses the Pointer to Implementation (PImpl) pattern.

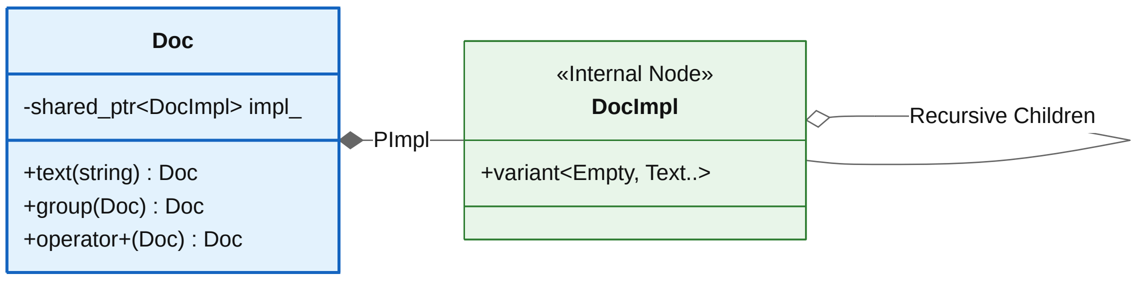

Figure 3.7: Doc: Document Class Diagram

Following Figure 3.7, it uses `std::shared_ptr` to share common parts of the document tree and keep memory usage low. The immutability allows the system to make layout decisions safely without changing any of the original data.

3.4.3 Document Algebra Primitives

The document algebra provides a set of building blocks that define how the code should be formatted. These primitives are stored in a `std::variant` and represent different layout rules. Each component has a specific purpose:

- **Text and Keyword:** These primitives store literal strings of text. The `Keyword` type also keeps track of casing information.
- **SoftLine:** This is a flexible break that changes based on the available width. It prints as a single space when the line is short enough, but becomes a newline if the content exceeds the available width.
- **HardLine:** This represents a mandatory newline. When a `HardLine` is used inside a group, it is forced to break into multiple lines.
- **Concat:** This primitive joins two document objects into a single one.
- **Nest and Hang:** These components manage the indentation of the code. `Nest` increases the indentation by a fixed amount, while `Hang` aligns the next lines with the current cursor position.
- **Union:** The most important part of the algebra. It holds two different versions of the same code, a flat version and a multi-line version. The renderer then picks the best option based on the configuration.

3.4.4 Optimization and Construction

The tool uses smart constructors to optimize the tree as it is built. For example, the system automatically removes empty nodes during concatenation to keep the tree small. To make the code easier to read, the tool uses C++ operator overloading as a DSL. These operators are summarized in Table 3.1.

Operator	Semantics
`a + b`	Direct concatenation
`a & b`	Space-separated concatenation
`a / b`	Softline-separated (breaks if needed)
`a | b`	Hardline-separated (always breaks)
`a << b`	Nested softline: `a + (line + b).nest()`

Table 3.1: Document Algebra Operators

3.4.5 Algebra Extensions

The algebra includes custom nodes like `Align` to handle common VHDL formatting styles. When the renderer sees an `Align` node, it calculates the necessary spacing to keep text vertically aligned. Listing 4 shows an example of a port declaration with and without this alignment.

```vhdl
-- Without alignment
port (
    clk : in std_logic;
    data_o : out std_logic_vector(7 downto 0)
);

-- With alignment
port (
    clk    : in  std_logic;
    data_o : out std_logic_vector(7 downto 0)
);
```

Listing 4: Port Declaration with and without Alignment

Following the emission stage with its produced document tree, the `Renderer` uses the information provided to produce the formatted output.

3.4.6 The Rendering Stage

The `Renderer` uses a solver that primarily focuses on the `Union` node. It tries to print the single-line version of a node first. If that version fits within the remaining width, the renderer selects it. If it does not fit, the multi-line version is used instead. This decision process is illustrated in Figure 3.8.

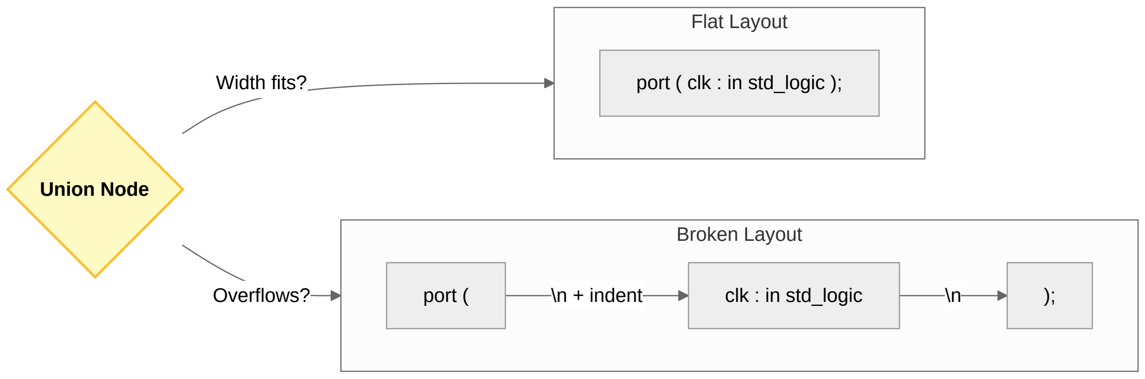

Figure 3.8: Union Node: Choosing Between Flat and Broken Layouts

3.4.7 Design Evaluation

The separation between emission and rendering provides several technical advantages. By using a document algebra, the implementation remains focused on the structure of the code rather than the details of the output format.

The architecture provides the following key benefits:

- **Decoupling:** The `PrettyPrinter` only describes the desired layout, while the `Renderer` handles the complex logic for line wrapping and indentation.
- **Maintainability:** Formatting rules are defined using a high-level Domain Specific Language, which makes the code readable and easy to update.
- **Efficiency:** Immutable document handles allow for structural sharing, which minimizes memory usage when building large trees.
- **Optimal Layout:** The layout solver resolves `Union` nodes in linear time to ensure the code always fits within the configured page width.

With the Backend architecture defined, the system is capable of producing consistent and high-quality VHDL code. To ensure that the formatting process did not alter the code's meaning, a verification stage is employed, which is discussed in the next section.

3.5 Verification

The verification step ensures that formatting has not altered the semantic meaning of the original source code. This is crucial, as any change in logic could introduce bugs that are difficult to find. Figure 3.9 illustrates the workflow of this verification process.

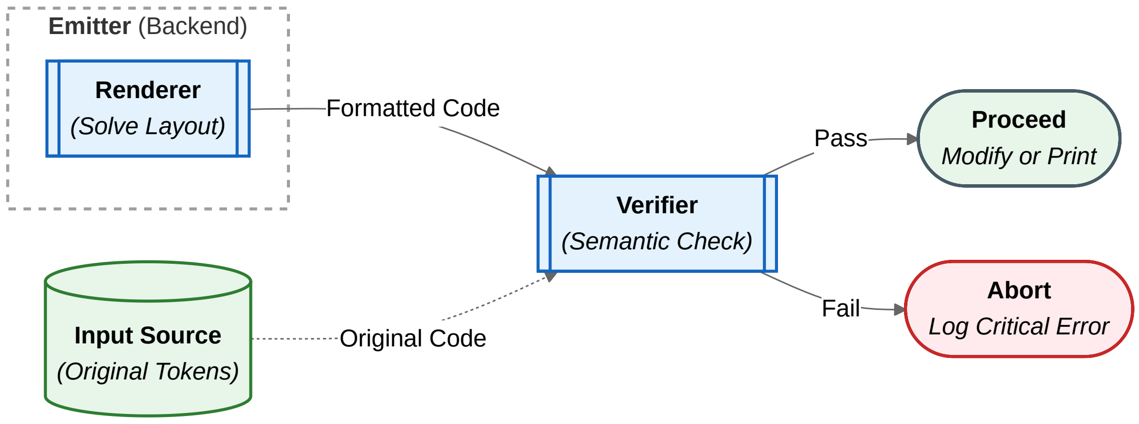

Figure 3.9: Verification Overview

3.5.1 Semantic Token Comparison

A token-based comparison algorithm is used instead of a simple string equality check. This allows the formatter to modify whitespace and capitalization while ensuring the underlying logic remains identical. The process consists of three stages:

1. **Re-Lexing:** The formatted string is passed through the ANTLR lexer to create a new stream of tokens.
2. **Filtering:** Both the original and the new token streams are filtered to remove trivia tokens like whitespace and comments.
3. **Equivalence Check:** The sequences of semantic tokens are compared pair-wise using three rules:
   - **Type Match:** The token types must be identical.
   - **Text Match:** The token text must match case-insensitively.
   - **Stream Completeness:** The formatted stream must not have any missing or extra tokens compared to the original code.

The technical implementation of this comparison is further detailed in Section 4.11.

3.5.2 Feedback on Verification Failure

If the system finds a mismatch, the verification fails immediately and the tool aborts the process. This prevents the formatter from writing corrupted code to the disk. Listing 5 shows an example of the log output when a semantic error is detected during comparison.

```
[2025-12-12 14:12:15.066] [critical] Formatter corrupted the code semantics.
[2025-12-12 14:12:15.066] [critical] Token Type Mismatch!
Original:  'proc' (Type: 118, Line: 42)
Formatted: ';'    (Type: 138, Line: 40)
[2025-12-12 14:12:15.066] [info] Aborting write to prevent data loss.
```

Listing 5: Safety Verification: Aborting on Semantic Corruption

The logs provide the token types, text, and line numbers for the first point of divergence. This detailed feedback helps the developer understand why the verification failed and ensures that the formatter is a reliable tool for professional hardware development.

3.5.3 Conclusion

By validating the semantic results before any files are modified, this verification step acts as a final safeguard for the system. The architecture guarantees that the VHDL logic remains identical while the code style is improved.

4 Implementation Details

Following the architecture established in Chapter 3, this chapter explains the technical implementation of each stage in the formatter pipeline. The design prioritizes two critical software engineering qualities:

- **Modularity:** The system explicitly decouples the parsing, translation, and rendering stages. This separation ensures that each component can be tested in isolation and allows the tool to evolve without one stage tightly coupling to another.
- **Performance:** To support interactive workflows, the implementation minimizes runtime overhead and memory usage. By employing a hybrid parsing strategy and a linear-time layout solver, the tool prevents exponential complexity and scales efficiently even with large input files.

4.1 Hybrid Parsing Strategy

VHDL grammar contains ambiguities that require the full LL parsing prediction mode in ANTLR for correct parsing. However, this mode is slow because it handles conflicts through complex resolution and arbitrary lookahead. To balance speed and accuracy, the tool implements a two-stage hybrid strategy:

- **Stage 1 (Fast SLL Parsing):** The parser first attempts to build the tree using SLL parsing mode. It uses a strict error strategy that aborts immediately if it finds an ambiguity or a syntax error.
- **Stage 2 (Robust LL Fallback):** If the first stage fails, the system catches the exception and resets the parser state. It then retries the operation using the more powerful LL parsing mode to resolve complex grammar rules correctly.

This approach optimizes for the common case while ensuring the tool can still process complex designs. As shown in the performance evaluation in Section 6.2, this hybrid strategy reduces parsing time by over $87\,\%$ for typical files. This optimization is highly effective because $90\,\%$ of real-world VHDL files are small and structurally simple (Section 6.1).

4.2 Fluent AST Construction

Converting the CST into the custom AST involves repetitive tasks such as null checks, memory allocation, and container transformations. To reduce boilerplate code and ensure type safety, the project uses a `NodeBuilder` class that implements a fluent interface pattern.

4.2.1 Technical Implementation

The builder uses C++23's "deducing this" to enable efficient method chaining. Before this feature, developers had to write duplicate code to handle temporary objects correctly. This feature allows the builder to work efficiently in all situations with a single implementation.

Listing 6 shows the implementation of the `set` method, which serves as the core primitive for the builder.

```cpp
// src/builder/node_builder.hpp
template<typename Self, typename Field, typename Value>
auto set(this Self &&self, Field T::*field, Value &&value) -> Self &&
{
    self.node_.*field = std::forward<Value>(value);
    return std::forward<Self>(self);
}
```

Listing 6: The `set` primitive using C++23 Deducing This

This function uses three specific techniques to achieve high performance:

- **Deducing This (`this Self &&self`):** This syntax allows the compiler to detect if the builder is a temporary object or a named variable at the call site.
- **Pointer to Member (`T::*field`):** This pointer allows the builder to assign values to specific fields without needing to know their names in advance.

- **Perfect Forwarding (`std::forward`):** This moves data if it is a temporary value or copies it if it is a persistent variable, which prevents unnecessary memory operations.

4.2.2 Automatic Trivia Binding

As seen in Listing 7 the builder manages the association of comments and whitespace automatically. The constructor accepts a `TriviaBinder` which attaches any surrounding hidden tokens from the parsing context to the new AST node.

```cpp
// src/builder/node_builder.hpp
template<typename Ctx>
explicit NodeBuilder(Ctx &ctx, TriviaBinder &trivia)
{
    trivia.bind(node_, ctx);
}
```

Listing 7: Automatic trivia binding in the builder constructor

4.2.3 Construction Primitives

The class provides several helper methods to simplify common tasks:

- `set` and `setBox`: Assign values to fields. The `setBox` version automatically wraps the value in a smart pointer to manage memory for child nodes.
- `maybe` and `maybeBox`: Check if a syntax element exists before processing it. This removes the need for repetitive null checks when handling optional grammar elements.
- `collect` and `collectFrom`: Convert lists of raw items from the parser into a vector of clean AST nodes in a single step.

For example: `makeGenericParam` uses `set` to populate mandatory fields like names and `maybe` for the optional default expression. Listing 8 demonstrates the implementation.

```cpp
// src/builder/translators/declarations/interface/generic.cpp
auto Translator::makeGenericParam(Interface_constant_declarationContext &ctx)
  -> ast::GenericParam
{
    return build<ast::GenericParam>(ctx)
      .set(&ast::GenericParam::names, extractNames(ctx.identifier_list()))
      .set(&ast::GenericParam::subtype, makeSubtypeIndication(*ctx.subtype_indication()))
      .maybe(&ast::GenericParam::default_expr,
             ctx.expression(), [&](auto &expr) { return makeExpr(expr); })
      .build();
}
```

Listing 8: Translating a GenericParam node

4.3 Trivia Binding

Formatters that rely on an AST often discard trivia, which refers to tokens like comments and whitespace that do not affect the program logic. Since these tokens are essential for human readability, the project implements a dedicated `TriviaBinder` to recover them. This component runs alongside the parser to attach hidden tokens to the correct tree nodes.

4.3.1 Context Extension

A major challenge is determining which node owns a comment. For example, a comment appearing after a semicolon typically refers to the statement ending at that semicolon. However, the parser considers the statement to end before the semicolon.

To solve this, the binder uses a `findContextEnd` method. This method looks ahead in the token stream. If it finds a separator like a semicolon or a comma immediately following the node, it extends the context range to include it.

```cpp
// src/builder/trivia/trivia_binder.cpp
auto TriviaBinder::findContextEnd(const ParserRuleContext &ctx) const -> std::size_t
{
    const auto next = ctx.getStop()->getTokenIndex() + 1;
    const auto tok = tokens_.get(next)->getText();

    // If the next token is a separator, include it in the range
    if (tok == ";" || tok == "," || tok == "else") {
        return next;
    }
    return ctx.getStop()->getTokenIndex();
}
```

Listing 9: Extending the node context to capture trailing separators

Listing 9 shows how the binder ensures that comments placed after a semicolon are correctly attached to the preceding statement rather than floating independently.

4.3.2 Ownership Management

Since AST nodes often nest within one another, multiple nodes effectively cover the same range of tokens. This creates a risk where a single comment could be attached to both a child node and its parent.

To prevent this duplication, the binder maintains a boolean vector called `used_` that tracks the status of every token in the file. When the `extractTrivia` method processes a range of tokens, it filters out any token that has already been claimed.

```cpp
// src/builder/trivia/trivia_binder.cpp
const auto filter = [this](auto *t) { return !isUsed(t); };

// Only process tokens that have not been attached to a previous node
for (auto *token : range | std::views::filter(filter)) {
    markAsUsed(token);
    // ... processing logic ...
}
```

Listing 10: Filtering used tokens to prevent duplication

As shown in Listing 10, marking tokens as used ensures that each comment appears exactly once in the final output.

4.3.3 Vertical Spacing

The final challenge is distinguishing between formatting newlines (which the tool should override) and semantic breaks (which the user inserted intentionally). The binder uses a heuristic based on the number of consecutive newlines.

```cpp
// src/builder/trivia/trivia_binder.cpp
if (isNewline(token)) {
    ++pending_newlines;
    continue;
}

// If 2 or more newlines are detected, create a Break node
if (pending_newlines >= 2) {
    result.emplace_back(ast::Break{ .blank_lines = pending_newlines - 1 });
}
```

Listing 11: Heuristic for preserving intentional vertical spacing

Listing 11 demonstrates this logic. Single newlines are ignored, allowing the pretty printer to enforce standard spacing. However, if the user places two or more newlines between constructs, the binder captures this as a `Break` object. This preserves the logical grouping of the code while still allowing the formatter to normalize the exact number of blank lines.

4.4 The Doc Class

The `Doc` class serves as the core data structure for the backend. It acts as a lightweight, immutable handle to the underlying document tree. To decouple the public API from the complex storage logic, the class uses the Pointer to Implementation (PImpl) pattern.

Listing 12 shows the interface. It exposes a concise DSL via operator overloading while delegating all storage and construction logic to the implementation layer.

```cpp
// src/emit/pretty_printer/doc.hpp
class Doc final
{
  public:
    // Primitives
    static auto text(std::string_view str) -> Doc;
    static auto line() -> Doc;

    // Algebraic Combinators (DSL)
    auto operator+(const Doc &other) const -> Doc; // Concatenation
    auto operator/(const Doc &other) const -> Doc; // Softline separation

    // High-Level Layout Constructs
    static auto group(const Doc &doc) -> Doc;      // Grouping for line breaking

  private:
    std::shared_ptr<const doc_impl::DocImpl> impl_;
};
```

Listing 12: The `Doc` class: A lightweight handle for the document algebra

Table 4.1 summarizes the operators available in the algebra.

Operator	Type	Semantics
`a + b`	Binary	Direct concatenation.
`a & b`	Binary	Space-separated concatenation (`a + ' ' + b`).
`a / b`	Binary	Softline-separated. Becomes a space if flat, newline if broken.
`a | b`	Binary	Hardline-separated. Always becomes a newline.
`a << b`	Binary	Nested softline. Equivalent to `a + (line + b).nest()`.
`a += b`	Assign	Append `b` to `a`.
`a &= b`	Assign	Append `b` with a space.
`a /= b`	Assign	Append `b` with a softline.
`a |= b`	Assign	Append `b` with a hardline.
`a <<= b`	Assign	Append `b` with a softline and increased indentation.

Table 4.1: Document Algebra Operators

4.4.1 Construction Optimizations

To ensure the document tree remains compact, the implementation employs “Smart Factories” that apply algebraic simplification rules eagerly during instantiation.

The `makeConcat` function in `doc_impl.cpp` intercepts every concatenation operation. Instead of naively creating new nodes, it detects and optimizes common patterns:

- **Identity Elimination:** Concatenating an `Empty` node with any document x simply returns x, preventing the tree from growing with useless null nodes ($\texttt{Empty} + x \to x$).
- **Text Fusion:** If two `Text` nodes are concatenated, they are merged into a single node immediately. This reduces the number of objects the renderer must traverse ($\texttt{Text}(a) + \texttt{Text}(b) \to \texttt{Text}(ab)$).
- **Hardline Fusion:** Consecutive hardlines are merged into a single `HardLines` node. This is particularly effective for vertical spacing, preventing deep recursion chains like `HardLine + HardLine + ...`.

Listing 13 demonstrates this optimization logic.

```cpp
// src/emit/pretty_printer/doc_impl.cpp
auto makeConcat(DocPtr left, DocPtr right) -> DocPtr
{
    // Rule 1: Identity Elimination
    if (!left || std::holds_alternative<Empty>(left->value)) return right;
    if (!right || std::holds_alternative<Empty>(right->value)) return left;

    // Rule 2: Text Fusion
    if (auto *l = std::get_if<Text>(&left->value)) {
        if (auto *r = std::get_if<Text>(&right->value)) {
            return makeText(l->content + r->content, -1);
        }
    }

    // Rule 3: Hardline Fusion (merges adjacent hardlines)
    // ...

    return std::make_shared<DocImpl>(Concat{std::move(left), std::move(right)});
}
```

Listing 13: Smart construction logic in `makeConcat`

4.4.2 Generic Tree Traversal

Recursive variant-based structures often come with the drawback of cumbersome traversal code. Standard `std::visit` approaches can lead to repetitive boilerplate when handling recursive nodes like `Concat` or `Union`.

To address this, the project implements a `DocWalker` utility. This class uses modern C++ meta-programming to abstract away the mechanics of recursion. As shown in Listing 14, it uses `if constexpr` to automatically detect the type of node and apply the correct recursion strategy.

```cpp
// src/emit/pretty_printer/walker.hpp
template<typename Node, typename Fn>
static auto mapChildren(const Node &node, Fn fn) -> Node
{
    using T = std::decay_t<Node>;

    if constexpr (std::is_same_v<T, Concat>) {
        return Concat{ .left = fn(node.left), .right = fn(node.right) };
    } else if constexpr (std::is_same_v<T, Union>) {
        return Union{ .flat = fn(node.flat), .broken = fn(node.broken) };
    } else if constexpr (IS_ANY_OF_V<T, Nest, Hang, Align>) {
        return T{ .doc = fn(node.doc) };
    } else {
        return node; // Leaf nodes are returned as-is
    }
}
```

Listing 14: Generic traversal using `if constexpr`

The class exposes three higher-order functions inspired by functional programming:

- `transform`: Maps a function over the tree bottom-up. This is used by the `flatten` function to convert `SoftLine` nodes into spaces when a group fits on a single line.
- `fold`: Reduces the tree to a single value, useful for calculating metrics like total height or width.
- `traverse`: Visits every node for side effects, which is used for debugging and validation.

This design abstracts away the error-prone mechanics of manual recursion. It enables developers to implement complex passes, such as alignment resolution, using concise and declarative code.

4.5 Visitor Pattern with C++23

Traditional visitors often rely on slow runtime polymorphism. This implementation uses C++23 "deducing this" to resolve the visitor type at compile time. Furthermore, the variadic `Args` parameter allows context to be passed down the traversal chain, avoiding the need for temporary state in member variables.

Listing 15 shows how the base class handles dispatch via `visit` while the derived class implements logic via the function call operator (`operator()`).

```cpp
// src/ast/visitor.hpp
template<typename ReturnType = void>
class VisitorBase {
  public:
    // 1. Variant Dispatch
    template<typename Self, typename... Ts, typename... Args>
    auto visit(this const Self &self, const std::variant<Ts...> &node, Args &&...args)
        -> ReturnType
    {
        return std::visit(
            [&](const auto &n) { return self.visit(n, std::forward<Args>(args)...); },
            node);
    }

    // 2. Concrete Node Dispatch (Middleware)
    template<typename Self, typename T, typename... Args>
        requires std::is_base_of_v<NodeBase, T>
    auto visit(this const Self &self, const T &node, Args &&...args) -> ReturnType
    {
        // Calls the derived class's operator()
        if constexpr (std::is_void_v<ReturnType>) {
            self(node, std::forward<Args>(args)...);
        } else {
            // Duck Typing: 'Self' must implement 'wrapResult'
            return self.wrapResult(node, self(node, std::forward<Args>(args)...));
        }
    }
};
```

Listing 15: VisitorBase implementation separating dispatch (visit) and logic (operator())

4.5.1 Automatic Trivia Injection

The second `visit` overload acts as middleware. It invokes the derived visitor and passes the output to `wrapResult`. This automatically injects comments and whitespace, allowing the `PrettyPrinter` 4.6 to return raw document nodes without manually handling trivia.

4.6 The Pretty Printer

The `PrettyPrinter` is the concrete implementation of the backend visitor. It inherits from `VisitorBase<Doc>` and is responsible for traversing the AST to produce the final document structure. By leveraging the document algebra defined in Section 4.4, the visitor focuses purely on structural composition, delegating layout complexity, such as line wrapping and indentation, to the underlying `Doc` primitives.

4.6.1 Trivia Reconstruction

A critical requirement for the formatter is the preservation of comments and vertical spacing. While the `TriviaBinder` attaches these elements to the AST nodes during the frontend phase, the `PrettyPrinter` is responsible for emitting them in the correct order.

This is achieved through the `withTrivia` middleware. As the visitor traverses the tree, the `wrapResult` hook intercepts the generated `Doc` for each node and augments it with the associated trivia. Listing 16 illustrates this reconstruction process:

```cpp
// src/emit/pretty_printer.cpp
auto PrettyPrinter::withTrivia(const ast::NodeBase &node, Doc core,
                               const bool suppress) -> Doc
{
    if (!node.hasTrivia()) {
        return core;
    }

    Doc result = Doc::empty(); // Initialize an empty document to accumulate parts

    // 1. Leading Trivia
    result += buildTrivia(node.getLeading(), suppress, Doc::empty(), formatTrivia);

    // 2. Core Doc
    result += core;

    // 3. Inline Comment
    if (auto comment = node.getInlineComment()) {
        result += Doc::text(std::string{ " " }.append(*comment)) + Doc::hardlines(0);
    }

    // 4. Trailing Trivia
    result += buildTrivia(node.getTrailing(), suppress, Doc::hardline(),
                          formatLastTrailing);
    return result;
}
```

Listing 16: Trivia Reconstruction: Wrapping the core document with comments

4.6.2 Functional Composition Helpers

To maintain concise and readable visitor logic, the implementation utilizes C++20 ranges and functional programming patterns. The `joinMap` helper, shown in Listing 17, abstracts the common pattern of iterating over a collection, transforming each element into a `Doc`, and joining them with a separator.

```
// src/emit/pretty_printer.hpp
template<std::ranges::input_range Range, typename Transform>
auto joinMap(Range &&items, const Doc &sep, Transform &&transform) const -> Doc
{
    return std::ranges::fold_left(
        std::forward<Range>(items), Doc::empty(),
        [&](const Doc &acc, auto &&item) {
            const Doc doc = std::invoke(transform, std::forward<decltype(item)>(item));
            return acc.isEmpty() ? doc : acc + sep + doc;
        });
}
```

Listing 17: Functional helper for joining collections

4.6.3 Example: Port Declarations

The formatting of port lists demonstrates the interaction between the visitor, the functional helpers, and the alignment algebra.

High-Level Layout

The `PortClause` visitor serves as the entry point for formatting the interface list. As shown in Listing 18, it establishes the high-level structure by wrapping the content in parentheses and an alignment scope. The construction process involves three key steps:

- **Iteration:** The `joinMap` helper iterates over the ports, separating them with softlines (`Doc::line()`).
- **Alignment:** The resulting document is wrapped in `Doc::align` to enforce columnar alignment of the inner elements.
- **Grouping:** Finally, `Doc::group` allows the list to be printed inline if it fits, or broken across multiple lines with automatic indentation if it exceeds the page width.

```cpp
// src/emit/pretty_printer/nodes/declarations/interface/port.cpp
auto PrettyPrinter::operator()(const ast::PortClause &node) const -> Doc
{
    if (std::ranges::empty(node.ports)) { return Doc::empty(); }

    const Doc opener = Doc::keyword("port") & Doc::text("(");
    const Doc closer = Doc::text(");");

    // Map ports to Docs, separated by softlines
    const Doc ports = joinMap(node.ports, Doc::line(), [&](const auto &port) {
        const bool is_last = (&port == &node.ports.back());
        return visit(port, is_last);
    });

    return Doc::group(Doc::bracket(opener, Doc::align(ports), closer));
}
```

Listing 18: Port Clause: Combining alignment, grouping, and indentation

Item Formatting

Delegated by the clause visitor, the individual `Port` visitor constructs the textual content of each line. As shown in Listing 19, it uses C++ ranges to efficiently join multiple signal names.

Crucially, it attaches `AlignmentLevel` tags to the identifier and mode nodes. These tags act as markers for the parent's alignment scope, ensuring that colons and types align vertically across the entire list as described in Section 4.9.

```cpp
// src/emit/pretty_printer/nodes/declarations/interface/port.cpp
auto PrettyPrinter::operator()(const ast::Port &node, const bool is_last) const -> Doc
{
    const std::string names = node.names
                            | std::views::join_with(std::string_view{ ", " })
                            | std::ranges::to<std::string>();

    Doc result = Doc::text(names, AlignmentLevel::NAME)
               & Doc::text(":")
               & Doc::keyword(node.mode, AlignmentLevel::MODE)
               & visit(node.subtype);

    if (node.default_expr) {
        result &= Doc::text(":=") & visit(node.default_expr.value());
    }

    return is_last ? result : result + Doc::text(";");
}
```

Listing 19: Formatting a single Port with alignment markers

4.7 Layout Resolution

The primary challenge of the algebraic approach is dynamically adapting the layout to the available page width. The formatter constructs an IR containing `Union` nodes (created via the `group` combinator), which offer two possible layouts: a single-line ("flat") version and a multi-line ("broken") version.

The layout resolution logic is embedded within the `Renderer::renderDoc` traversal. As the renderer encounters nodes, it tracks the current column position. When it reaches a `Union` node, it must make a binary decision: print the flat version if it fits within the remaining line width, or fall back to the broken version.

Listing 20 demonstrates this greedy decision process. If the renderer is already in `FLAT` mode (meaning a parent group has already decided to flatten), the child group implicitly accepts the flat layout.

```cpp
// src/emit/renderer.cpp
// Inside renderDoc visitor...
[&](const Union &node) -> void {
    const int remaining_width = config_.line_config.line_length - column_;

    // Decide: use flat or broken layout?
    if (mode == Mode::FLAT || fits(remaining_width, node.flat)) {
        // Fits on current line - use flat version
        renderDoc(indent, Mode::FLAT, node.flat);
    } else {
        // Doesn't fit - use broken version
        renderDoc(indent, Mode::BREAK, node.broken);
    }
}
```

Listing 20: The greedy solver logic within the main render loop

4.7.1 The `fits` Predicate

The core of the solver is the `fits` predicate, which determines if a document subtree can be rendered on a single line without overflowing. This is implemented via the `fitsImpl` helper, which simulates the rendering process in `FLAT` mode.

To ensure efficiency, `fitsImpl` employs aggressive pruning. It returns a negative value immediately upon failure, preventing unnecessary traversal of deep subtrees. The implementation details are shown in Listing 21.

```cpp
// src/emit/renderer.cpp
auto Renderer::fitsImpl(int width, const DocPtr &doc) -> int
{
    if (width < 0) return -1; // Pruning 1: Width exceeded
    if (!doc) return width;

    auto fits_visitor = common::Overload{
        // Text consumes width
        [&](const Text &node) -> int {
            return width - static_cast<int>(node.content.length());
        },

        // SoftLine becomes a space in flat mode
        [&](const SoftLine &) -> int { return width - 1; },

        // Pruning 2: HardLine makes flattening impossible
        [](const HardLine &) -> int { return -1; },

        // Concat threads the remaining width
        [&](const Concat &node) -> int {
            const int remaining = fitsImpl(width, node.left);
            return (remaining < 0) ? -1 : fitsImpl(remaining, node.right);
        },

        // Union only checks the flat variant
        [&](const Union &node) -> int { return fitsImpl(width, node.flat); },

        // ... recursive cases (Nest, Align, Hang) omitted ...
    };

    return std::visit(fits_visitor, doc->value);
}
```

Listing 21: The `fitsImpl` function with pruning optimizations

The function utilizes three specific optimizations:

- **Negative Width Check:** `if (width < 0)` aborts instantly.
- **Hardline Rejection:** `HardLine` nodes immediately return `-1`, as a group containing a mandatory newline can never be flattened.
- **Variant Selection:** For `Union` nodes, it only inspects the `flat` variant, ignoring the `broken` branch entirely.

These optimizations ensure that the `fits` check remains efficient, even for large documents with deeply nested structures.

4.7.2 Complexity Analysis

The specific design of the `fits` predicate dictates the algorithm's time complexity. While a naive backtracking approach could result in exponential time, the pruning steps create a bounded greedy solver. Since `fits` aborts as soon as the line width is exceeded, the lookahead is strictly bounded by the page width configuration (e.g., 80 or 120 characters).

Mathematical Model

The total time complexity $T(D)$ for a document D is defined as the sum of the standard visitation cost and the lookahead cost incurred at each `Union` decision point.

Let N denote the total number of nodes, and let $G \subseteq N$ denote the set of `Union` nodes. With W_{max} representing the configured line width and $\mathcal{L}(g)$ denoting the flattened length of a group g, the complexity can be expressed as:

$$T(D) = \underbrace{\sum_{n \in N} C_{visit}}_{\text{Emission}} + \underbrace{\sum_{g \in G} \min(\mathcal{L}(g), W_{max})}_{\text{Lookahead}}$$

Because the lookahead term $\min(\mathcal{L}(g), W_{max})$ is constant with respect to N (it never exceeds W_{max}), the second term does not grow quadratically. Even in the worst-case scenario where every node triggers a check, the cost per node remains bounded.

$$T(D) = O(N \cdot W_{max}) = O(N)$$

This confirms that the formatting engine has linear time complexity.

4.7.3 Performance Considerations

While the layout algorithm is linear, the algebraic approach incurs higher constant overhead than direct streaming formatters due to the construction of an intermediate object graph.

As seen in the code, the heavy use of `std::shared_ptr` (for the `Doc` tree) and `std::visit` (for dispatching node types) creates a fixed cost per node. However, empirical analysis (Section 4.10) demonstrates that this overhead is negligible for modern hardware, with the rendering stage typically consuming less than $2\,\%$ of the total execution time.

4.8 Algebra Extensions

While Wadler's document algebra [9] provides elegant primitives for line breaking and indentation, real-world VHDL formatting demands capabilities beyond the standard combinators. Professional VHDL code exhibits two layout patterns that cannot be expressed purely through `nest`, `group`, and `line`:

1. **Context-Sensitive Indentation:** Code blocks that align relative to a dynamic column position rather than a fixed offset (e.g., waveform lists).

2. **Tabular Alignment:** Vertically stacked declarations where tokens across multiple lines must align to common columns (e.g., port maps).

To address these requirements without compromising the linear-time complexity of the core layout algorithm (Section 4.7), two specialized combinators were introduced: `hang` for dynamic indentation and `align` for tabular layout.

4.8.1 Dynamic Indentation: The Hang Combinator

Standard indentation, achieved via the `nest` primitive, increases the indentation level by a fixed constant relative to the parent context. While sufficient for block structures with uniform nesting depth, this fails to capture VHDL's common practice of aligning continuation lines with the start of a construct.

Consider the following multi-line waveform assignment:

```vhdl
clk_gen <= '0',
           '1' after 10 ns,
           '0' after 20 ns;
```

Listing 22: Continuation lines aligned to the first waveform element

In this example, the column position (11) depends entirely on the length of the preceding text (`clk_gen <=`), making the indentation inherently context-dependent.

Implementation

The `hang` combinator captures the current column position at render time and uses it as the new indentation level. The implementation integrates seamlessly into the renderer through a single visitor case, as shown in Listing 23.

```cpp
// src/emit/renderer.cpp
void Renderer::renderDoc(int indent, Mode mode, const DocPtr &doc)
{
    auto render_visitor = common::Overload{
        // Hang: Capture current column as new indent level
        [&](const Hang &node) -> void {
            renderDoc(column_, mode, node.doc);
        },
        // ... other cases ...
    };
    std::visit(render_visitor, doc->value);
}
```

Listing 23: Hang implementation: Dynamic indentation based on current column

By passing `column_` (the current horizontal position) as the new indentation level, multi-line constructs remain visually cohesive regardless of the preceding content length. Since this operation requires no lookahead, the `hang` combinator maintains the $O(N)$ complexity of the baseline renderer.

4.8.2 Tabular Alignment: The Align Composite

Tabular alignment represents a fundamentally different challenge. While operations like `group` and `nest` make local layout decisions, alignment requires global information.

The Circular Dependency Problem

Consider the VHDL port clause shown in Listing 4. The colons must align to the same column across all declarations. However, this column position is determined by the longest identifier (e.g., `data_o`), which cannot be known until all rows have been examined.

This creates a circular dependency: to render a row, the column widths must be known, but to determine column widths, all rows must first be examined. To break this dependency without abandoning the efficiency of the single-pass renderer, the `Align` node introduces a localized scope for multi-pass resolution. The architectural implementation of this strategy is the subject of the following section.

4.9 Alignment Resolution

As motivated in Section 4.8, the `Align` composite requires global information about column widths that cannot be determined during standard layout resolution. This section details the specialized architecture used to implement tabular alignment.

4.9.1 Architectural Integration

The `Align` primitive resolves the circular dependency by scoping the alignment logic to specific subtrees. When the renderer encounters an `Align` node, it alters its control flow to execute a specialized two-pass algorithm:

1. **Pause:** Suspend normal single-pass rendering.
2. **Solve:** Execute the `AlignmentResolver` to calculate widths and insert padding.
3. **Resume:** Resume rendering with the transformed, padded subtree.

This design isolates the complexity of tabular layout. While the `Align` node pauses the stream, the specific algorithm ensures that the global performance remains linear by strictly bounding the scope of the double traversal.

4.9.2 Two-Pass Algorithm

The alignment process is split into two distinct passes. The entry point coordinates both phases:

```cpp
// src/emit/pretty_printer/algorithms/alignment_resolver.cpp
auto AlignmentResolver::resolve(const DocPtr &doc) -> DocPtr
{
    constexpr int MAX_LEVELS = 8;
    std::vector<int> widths{};
    widths.reserve(MAX_LEVELS);
    measure(doc, widths);      // Pass 1: Find max width per level
    return apply(doc, widths); // Pass 2: Insert padding
}
```

Listing 24: Two-Pass Alignment: Entry point

Pass 1: Measurement

The first pass traverses the tree, tracking the maximum width for each alignment level. A critical feature is the firewall that prevents recursion into nested alignment blocks:

```cpp
void AlignmentResolver::measure(const DocPtr &doc, std::vector<int> &widths)
{
    auto visitor = [&](const auto &node) {
        using T = std::decay_t<decltype(node)>;

        // Firewall: Don't recurse into nested Align blocks
        if constexpr (std::is_same_v<T, Align>) {
            return;
        }

        // Track maximum width for each alignment level
        if constexpr (IS_ANY_OF_V<T, Text, Keyword>) {
            if (node.level >= 0) {
                // Resize vector if needed...
                widths[node.level] = std::max(
                    widths[node.level],
                    static_cast<int>(node.content.length())
                );
            }
        }
        // ... Recurse into children ...
    };
    std::visit(visitor, doc->value);
}
```

Listing 25: Measurement Pass: Determining maximum column widths

Pass 2: Application

The second pass rebuilds the tree, inserting padding where necessary. It leverages structural sharing to return original nodes when no padding is required.

```cpp
auto AlignmentResolver::apply(const DocPtr &doc, std::span<const int> widths)
    -> DocPtr
{
    auto visitor = [&](const auto &node) {
        using T = std::decay_t<decltype(node)>;

        if constexpr (std::is_same_v<T, Align>) return doc; // Firewall

        if constexpr (IS_ANY_OF_V<T, Text, Keyword>) {
            // Check if padding is needed...
            const int padding = widths[node.level] - node.content.length();
            if (padding > 0) {
                // Create new Concat node with padding...
                return makeConcat(std::make_shared<DocImpl>(node),
                                  makeText(std::string(padding, ' ')));
            }
            return doc; // No padding needed - return original reference
        }
        // ... Recurse and map children ...
    };
    return std::visit(visitor, doc->value);
}
```

Listing 26: Application Pass: Inserting padding to achieve alignment

4.9.3 Complexity Analysis

The two-pass design allows for a rigorous complexity proof. Let N be the total number of nodes in the document, and $A \subseteq N$ be the subset of nodes within `Align` scopes.

The total rendering cost is expressed as:

$$T_{\text{total}} = \underbrace{O(N)}_{\text{Standard Rendering}} + \underbrace{O(2 \cdot |A|)}_{\text{Alignment Passes}}$$

The first term represents the standard single-pass emission of the document. The second term represents the extra work performed by `AlignmentResolver`, which visits every node in A exactly twice (once to measure, once to apply).

Since alignment is applied only to declarations, it is $|A| \ll N$ in practice. Even in the worst case where $|A| \approx N$, the total complexity simplifies to:

$$T_{\text{total}} = O(N + 2N) = O(N)$$

4.10 Performance Analysis

The preceding sections established the theoretical complexity of the formatting algorithms: $O(N)$ for layout resolution (Section 4.7), $O(N)$ for standard rendering, and $O(2 \cdot |A|)$ for alignment resolution (Section 4.9), where $|A| \ll N$ in practice. This section validates these theoretical guarantees through empirical benchmarking.

4.10.1 Benchmark Methodology

Performance measurements were collected using the `Catch2` micro-benchmarking framework. Each stage of the formatting pipeline was measured independently to identify bottlenecks and verify the linear scaling hypothesis.

The benchmark suite processes a representative VHDL design unit containing typical language constructs: entity declarations, architectures with signal assignments, port maps, and generic clauses. Each benchmark executes multiple iterations to ensure statistical significance, with the mean execution time and standard deviation reported.

4.10.2 Pipeline Breakdown

Table 4.2 shows the time taken for each stage for a typical design file (entity with port declarations, architecture with process, 40 LOC).

Stage	**Mean Time**	**Std Dev**	**% of Total**
1. Parsing (SLL Mode)	$60.86\,\mu s$	$4.10\,\mu s$	$32.1\,\%$
2. AST Translation	$47.31\,\mu s$	$7.34\,\mu s$	$24.9\,\%$
3. Doc Generation (Visitor)	$20.49\,\mu s$	$1.28\,\mu s$	$10.8\,\%$
4. Rendering to String	$2.57\,\mu s$	$0.24\,\mu s$	$1.4\,\%$
5. Verification	$58.32\,\mu s$	$3.94\,\mu s$	$30.8\,\%$
Total	$189.55\,\mu s$	–	$100\,\%$

Table 4.2: Formatting Pipeline Performance Breakdown (Micro-benchmark)

Key Observations

Parsing Dominates Execution Time Stage 1 (Parsing) accounts for $32.1\,\%$ of the total execution time using the hybrid strategy. As discussed in Section 4.1, utilizing the SLL prediction mode significantly reduces this overhead compared to pure LL(*) parsing.

However, due to the complexity of the VHDL grammar, parsing remains the primary computational cost.

BACKEND EFFICIENCY The combined Backend (Stages 3 and 4) accounts for only $12.2\,\%$ of the total time. This validates the design decisions in Sections 4.7 and 4.9:

- **Doc Generation (Stage 3):** The visitor-based emission completes in $20.49\,\mu s$, demonstrating that constructing the document algebra introduces minimal overhead compared to the parsing phase.
- **Rendering (Stage 4):** The layout resolution and alignment passes complete in just $2.57\,\mu s$. Despite the inclusion of the two-pass algorithm for tabular alignment, rendering remains the fastest stage, consuming only $1.4\,\%$ of the total time.

This empirically confirms that the complexity overhead of the `Align` extension ($O(2 \cdot |A|)$) is negligible in practice, as $|A|$ remains bounded by the number of declaration blocks in typical VHDL files.

VERIFICATION OVERHEAD Stage 5 (Verification) accounts for $30.8\,\%$ of the execution time. While high, this represents a deliberate safety-first design choice. The semantic token comparison ensures that formatting errors do not alter the logic, which is critical for a production tool.

4.10.3 Scalability Analysis

To verify the linear scaling hypothesis, the formatter was benchmarked against synthetically generated VHDL files ranging from 50 to 7000 lines of code. The results, shown in Figure 4.1, demonstrate that the normalized processing time (milliseconds per line of code) stabilizes after an initial startup period.

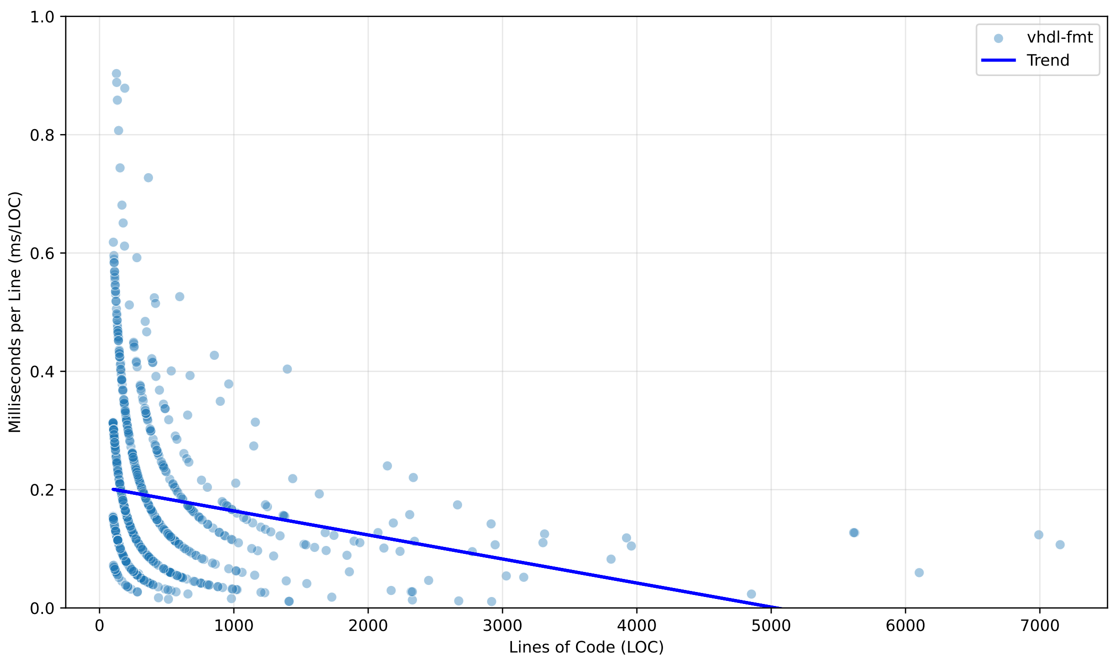

Figure 4.1: Normalized Performance: The linear scaling of the algebraic formatter.

Interpretation

Initial Overhead For very small files (< 100 LOC), the normalized cost is higher due to fixed overheads: ANTLR initialization, AST allocations, and the verification pass setup. These costs are rapidly amortized as the file size increases.

Linear Scaling Region Beyond approximately 200 LOC, the processing time per line stabilizes around $0.03\,\mathrm{ms}$/LOC. This confirms the theoretical $O(N)$ complexity, modeling the time $T(N)$ for a file of size N as:

$$T(N) = C_{\text{fixed}} + k \cdot N$$

where C_{fixed} represents startup costs and $k \approx 0.03\,\mathrm{ms}/LOC$ is the marginal cost per line.

Alignment Impact The stable slope indicates that the two-pass alignment algorithm does not degrade performance as the file size grows. Even for files with hundreds of signal declarations (each triggering the alignment logic), the impact remains constant relative to the standard parsing and emission costs.

4.10.4 Comparison with Requirements

The non-functional requirements (Section 2.1) established a strict performance budget:

- **Requirement:** Format typical VHDL files (< 100 LOC) in under $100\,\mathrm{ms}$.
- **Result:** At $0.03\,\mathrm{ms}$/LOC, a 1000-line file formats in approximately $30\,\mathrm{ms}$ (excluding I/O).

This result demonstrates that the algebraic approach, despite its higher memory overhead compared to direct streaming formatters, achieves the performance targets necessary for interactive editor integration (e.g., *format-on-save*).

4.10.5 Comparative Benchmarking

To contextualize the performance results, the formatter was benchmarked against *vhdl-style-guide* (VSG) [4], the currently most widely used open-source alternative. VSG is implemented in Python and prioritizes rule configurability over execution speed.

Figure 4.2 presents a normalized comparison of the two tools, plotting the execution time *per line of code* against file size.

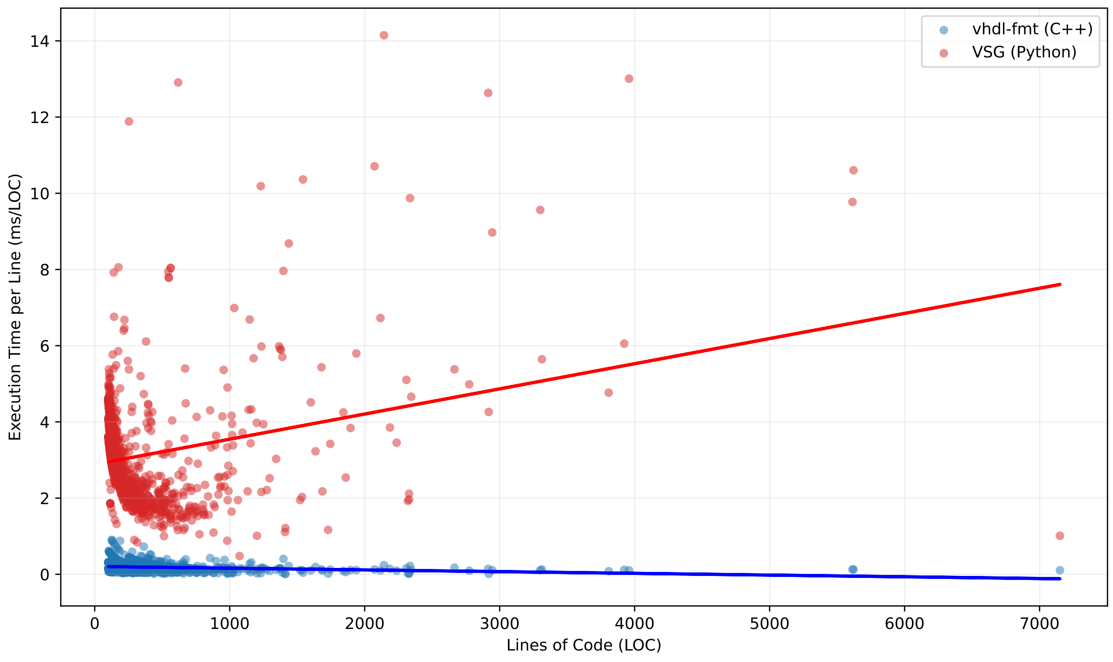

Figure 4.2: Normalized Performance Comparison: vhdl-fmt (C++) and VSG (Python).

The results reveal a fundamental difference in scalability:

- **VSG (Python):** Exhibits super-linear scaling, where the cost per line increases as the file grows larger. For a 5600-line file, VSG requires nearly $55\,\mathrm{s}$, resulting in a cost of approximately $10\,\mathrm{ms}$ per line.
- **vhdl-fmt (C++):** Maintains a constant marginal cost. The trend line is effectively flat, hovering near $0.03\,\mathrm{ms}$ per line regardless of file size. This linear complexity ($O(N)$) is a direct result of the single-pass algebraic design.

This orders-of-magnitude difference means the algebraic formatter is suitable for both interactive use (*format-on-save*) and CI pipelines. Its predictable, linear performance ensures that it remains fast regardless of file size, making it a reliable choice for any part of the development workflow.

4.10.6 Opportunities for Optimization

The empirical analysis reveals two primary areas for future optimization:

1. **Parser Performance:** The ANTLR-generated parser accounts for $32\,\%$ of execution time. While the hybrid parsing strategy provides significant improvement, further

gains could be achieved by optimizing the grammar or investigating a hand-written recursive descent parser.

2. **Verification Cost:** The semantic token comparison accounts for $31\,\%$ of execution time. This step could be made optional via a `--unsafe` flag for users who prioritize maximum speed over safety guarantees.

Overall, the performance analysis confirms that the algebraic formatter meets its design goals while providing a robust and safe foundation for VHDL development.

4.11 Verification Implementation

To guarantee that formatting never alters code semantics, the tool includes a verification layer implemented in the `ensureSafety` function, which runs after formatting but before writing to disk.

4.11.1 Token Stream Comparison

The verifier compares token streams rather than strings, avoiding false negatives from whitespace changes. Both the original and formatted token streams are filtered using C++20 ranges to extract only semantic tokens (identifiers, keywords, literals, symbols), stripping trivia.

The filtered streams are compared using `std::ranges::mismatch` with two strict rules:

- **Type Equality:** ANTLR token types must match exactly (e.g., `SIGNAL`, `IDENTIFIER`).
- **Content Equality:** Token text must match, with case-insensitive comparison for identifiers/keywords, since VHDL is case-insensitive, and exact comparison for literals.

```cpp
// src/builder/verifier.hpp
auto ensureSafety(CommonTokenStream &original, CommonTokenStream &formatted)
    -> std::expected<void, VerificationError>
{
    auto orig_view = original.getTokens() | std::views::filter(detail::IS_SEMANTIC);
    auto fmt_view = formatted.getTokens() | std::views::filter(detail::IS_SEMANTIC);

    // Predicate: Do these two tokens match?
    auto token_match = [](Token *t1, Token *t2) -> bool {
        return t1->getType() == t2->getType()
            && detail::EQUALS(t1->getText(), t2->getText());
    };

    // Find the first point of divergence
    auto [it_orig, it_fmt] = std::ranges::mismatch(orig_view, fmt_view, token_match);

    return (it_orig == orig_view.end() && it_fmt == fmt_view.end())
        ? std::expected<void, VerificationError>{}  // Success
        : std::unexpected(VerificationError{*it_orig, *it_fmt});
}
```

Listing 27: Semantic Verification using C++20 Ranges

If verification fails, the formatter aborts and logs the token mismatch, preventing semantic corruption.

4.11.2 Verification Scope and Limitations

The verification process validates semantic equivalence but explicitly excludes formatting-related changes:

- **Verified:** All semantic tokens (keywords, identifiers, literals, operators) must match in type, content, and order.
- **Not Verified:** Whitespace and vertical spacing are intentionally excluded from verification, as these are precisely what the formatter modifies. Comments, on the other hand, are planned for verification but are not yet implemented.

Since comment verification is not yet implemented, the current system cannot detect if comments are lost or misplaced during formatting. However, the critical safety guarantee remains intact: the semantic token stream comparison ensures that no executable logic is altered, preserving the functional correctness of the code regardless of the current limitations in comment verification.

4.12 Command-Line Interface

Invoking the formatter without any command-line arguments displays a usage overview describing the available options and their behavior.

```
Usage: vhdl_fmt [--help] [--version] [--write] [--check] [--location VAR] file.vhd

A VHDL formatter for beautifying and standardizing VHDL code.

Positional arguments:
  file.vhd        VHDL file or directory to format

Optional arguments:
  -h, --help      shows help message and exits
  -v, --version   prints version information and exits
  -w, --write     Overwrites the input file with formatted content
  -c, --check     Checks if the file is formatted correctly without modifying it
  -l, --location  Path to the configuration file
                  (e.g., /path/to/vhdl-fmt.yaml)

For more information, visit the project documentation.
https://github.com/niekdomi/vhdl-fmt
```

Listing 28: Command-line interface of vhdl_fmt

The CLI is implemented using the `argparse` library [10]. This library simplifies the declaration of command-line options and automatically generates a standardized help message, as shown in Listing 28.

The formatter supports the following options:

- `help`: Displays the help message and exits.
- `version`: Prints the current version of the formatter and exits. The version number is generated automatically during the build process using CMake. In future releases, this value will correspond to the release version.
- `write`: By default, the formatter outputs the formatted result to `stdout`. When the `--write` flag is provided, the input file is overwritten with the formatted content.
- `check`: Enables validation mode. The formatter compares the input file with the formatted output and exits with a non-zero status code if differences are detected. This mode is intended for integration into CI pipelines to enforce consistent formatting.
- `location`: Specifies the path to a YAML configuration file via the `--location` flag. If the flag is omitted, the formatter searches for a `vhdl-fmt.yaml` file in the current working directory. When no configuration file is found, default settings are applied.

4.13 Configuration System

The formatter supports a subset of configurable options to accommodate varying coding conventions. Configuration is provided exclusively through a YAML file rather than command-line flags. This design decision offers two advantages: it simplifies the command-line interface by reducing the number of available flags, and it encourages explicit documentation of formatting rules within project repositories, which facilitates team collaboration and integration into automated workflows.

While the formatter follows opinionated defaults, it provides flexibility to align with existing style guidelines. The available configuration options are shown in Listing 29.

```yaml
line_length: 100
indentation:
    size: 4
casing:
    keywords: "preserve"    # | "lower_case" | "UPPER_CASE"
    identifiers: "preserve" # | "lower_case" | "UPPER_CASE"
    constants: "preserve"   # | "lower_case" | "UPPER_CASE"
```

Listing 29: Example Configuration File

The current implementation supports line length and indentation size configuration. Additional options, including casing rules for keywords, identifiers, and constants, are planned for future releases.

5 Quality Assurance

Ensuring software quality requires more than a robust architecture. It demands a rigorous verification strategy, particularly when dealing with the complexities of C++ and the nuances of the VHDL standard. This chapter outlines the methodologies employed to guarantee that the tool meets its functional correctness, performance, and maintainability requirements.

5.1 Guidelines

To ensure long-term maintainability and correctness, the codebase follows modern C++ best practices and emphasizes safe and idiomatic language usage. The *C++ Core Guidelines* [11] serve as a foundational reference, as they provide well-established recommendations for writing robust and modern C++ code.

However, these guidelines intentionally focus on general language design principles and do not define project-specific conventions such as naming schemes or code formatting. Established style guides, including those by Google, LLVM, and Mozilla, impose fixed conventions that are not fully aligned with the design goals and modern C++ features used in this project. For this reason, a custom set of *Project Guidelines* was defined.

The defined conventions are automatically enforced using clang-format for code formatting and clang-tidy for naming conventions and additional static analysis checks.

5.2 Testing Strategy

The project employs a multi-layered testing strategy. It utilizes the `Catch2` framework to verify correctness at different levels of abstraction [12].

5.2.1 Unit Testing

Unit tests focus on individual components in isolation. This includes the domain logic within the AST and the formatting rules in the Emitter. Because the system is decoupled via the `Doc` intermediate representation, formatting logic can be verified by constructing AST nodes and asserting the expected `Doc` structure. This approach avoids the need for complex mocking frameworks since the tests rely solely on lightweight domain objects.

The test case shown in Listing 30 demonstrates the framework's simplicity. A VHDL snippet is parsed directly into an AST, and properties are verified using standard assertions without requiring a full formatter run.

```cpp
// tests/ast/nodes/expressions/test_binary.cpp
TEST_CASE("BinaryExpr: Addition operator", "[expressions][binary]")
{
    constexpr std::string_view VHDL_FILE = R"(
        entity E is end E;
        architecture A of E is
            signal x : integer := 10 + 20;
        begin
        end A;
    )";

    auto design = builder::buildFromString(VHDL_FILE);
    const auto *expr = getSignalInitExpr(design); // Helper to extract expr
    REQUIRE(expr != nullptr);

    const auto *binary = std::get_if<ast::BinaryExpr>(expr);
    REQUIRE(binary != nullptr);
    REQUIRE(binary->op == "+");
}
```

Listing 30: Unit Test Example for Binary Expressions

5.2.2 Integration Testing

Integration tests verify the end-to-end pipeline from file input to formatted text output. These tests ensure that the *Builder*, *Translator*, and *Emitter* components interact correctly. By processing complete VHDL files, these tests validate that the tool preserves semantic equivalence and handles complex nesting of VHDL constructs without corruption.

5.2.3 Benchmarking

To ensure the tool meets the non-functional performance requirement of formatting files in under $100\,\mathrm{ms}$, a dedicated benchmarking suite is implemented using `Catch2`'s micro-benchmarking features. These benchmarks measure the execution time of critical paths. This includes the parsing phase (SLL parsing vs. LL parsing) and the CST-to-AST translation. This allows for regression detection and performance profiling of the formatting engine against large inputs.

5.3 Static Analysis and Code Style

To maintain a high standard of code quality and consistency across the C++ codebase, automated static analysis tools are integrated into the build system and CI pipeline.

- **Clang-Format:** Automatically enforces a consistent coding style (e.g., indentation, brace placement, include ordering). This eliminates "nitpick" comments during code review and ensures the codebase remains uniform regardless of the contributor.
- **Clang-Tidy:** Provides deep static analysis to catch logical errors, performance bottlenecks, and modern C++ anti-patterns. It enforces best practices such as proper resource management and const-correctness which significantly reduces the likelihood of runtime errors.

5.4 Error Handling and Logging

5.4.1 Error Recovery Strategy

The formatter is designed to reject syntactically invalid input gracefully. When the parser encounters malformed VHDL code, the tool provides clear diagnostic messages indicating the nature and location of the error, rather than crashing or producing corrupted output.

5.4.2 Logging

Observability is provided via the `spdlog` library. A centralized logger singleton allows users to configure the verbosity level (e.g., `trace`, `info`, `error`) via command-line arguments. This flexibility ensures that developers can obtain detailed diagnostics during debugging without cluttering the standard output during normal CLI operation.

5.5 Development Environment

To achieve reproducibility and to avoid environment-specific issues, the project uses a Dev Container-based development environment. This approach provides a Docker-based setup that encapsulates all required build and runtime dependencies, including the compiler, CMake, Conan, and ANTLR. As a result, all developers and the CI system operate in an identical environment. This significantly minimizes platform-dependent debugging.

5.5.1 Build Infrastructure

The build infrastructure is designed to support reliable builds, automated testing, and CI. Dependency management is handled by Conan, which resolves and installs all required external libraries defined in `conanfile.txt`. After dependency resolution, Conan generates a `CMakeUserPresets.json` file containing the paths to the installed packages. This process is illustrated in Figure 5.1.

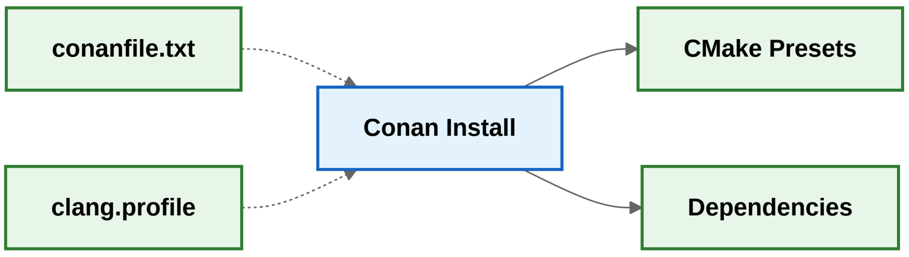

Figure 5.1: Conan-based dependency management

The project is built using CMake with Ninja as the build backend, enabling fast incremental builds. CMake consumes the generated presets file and executes code generation steps before compilation. In particular, the parser and lexer are generated from the grammar using the ANTLR tool. Although ANTLR is a Java-based tool, the required Java runtime is installed and managed by Conan, avoiding additional manual setup.

The overall build process is shown in Figure 5.2.

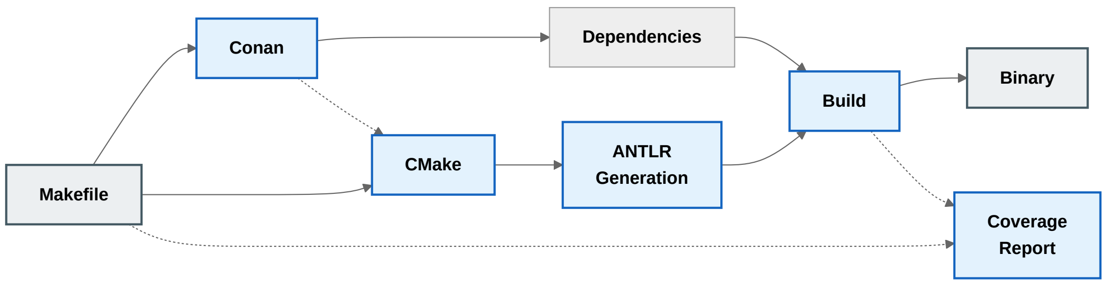

Figure 5.2: CMake-based build process

5.5.2 Continuous Integration and Quality Assurance

Continuous Integration is implemented using GitHub Actions. On each commit in a pull request, the CI pipeline performs a clean build and executes the full test suite. Code quality is enforced using clang-tidy, which checks for naming consistency and common programming errors. To reduce execution time, static analysis is executed separately for modified source and header files and performed in parallel.

Consistent code formatting is ensured through clang-format. Formatting violations result in a failed pipeline, preventing non-conforming code from being merged into the main branch.

In addition, all tests are executed with coverage tracking enabled. The generated coverage report provides reviewers a clear overview of how the proposed changes affect overall test coverage. The workflow is illustrated in Figure 5.3.

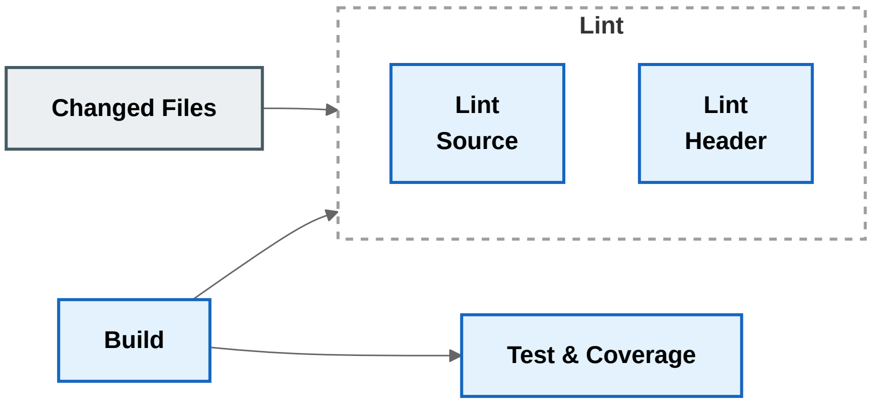

Figure 5.3: CI pipeline implemented with GitHub Actions

5.6 Release Process

The release process is intended to package and distribute prebuilt binaries for multiple platforms in a consistent and reproducible way. It is designed to be fully automated using CI pipelines. A release is triggered by pushing a version tag to the main branch. The pipeline then performs a release build and attaches the generated binaries to the GitHub release.

5.6.1 Cross-Compilation

To support multiple platforms such as Linux, Windows, and macOS, two general approaches are possible: cross-compilation from a single host system or native builds on each target platform.

Cross-compilation requires that a suitable toolchain for the target platform is available on the host system. For Windows, this is relatively straightforward, as `mingw-w64` is available via Linux package managers. For macOS, however, the situation is more complex. One possible solution for macOS cross-compilation is `osxcross` [13], which is not officially supported by Apple and requires extracting the macOS SDK from Xcode. This practice is often considered legally problematic, as it violates the Xcode license agreement [14].

The alternative, and more common approach, is to build the software natively on each target platform. Modern CI services such as GitHub Actions support matrix builds, allowing the same build steps to be executed on multiple operating systems. This approach enables generating binaries for all supported platforms and architectures, provided that suitable runners are available.

Native builds introduce their own challenges. On macOS, the default compiler is Apple Clang, which lags behind the latest Clang releases. Since this project relies on language features not yet supported by Apple Clang, a newer Clang version must be installed manually. On Windows, additional issues were encountered with CMake failing to locate the ANTLR executable.

As the formatter is still under development and not yet ready for production use, the implementation of a cross-platform release pipeline has been deferred.

6 Profiling

Section 4.10 established that the formatter achieves $O(N)$ complexity with the complete pipeline formatting typical files in under $200\,\mu s$. This chapter investigates two questions: What does typical VHDL code look like? Where is execution time spent?

6.1 VHDL Repository Analysis

To understand the characteristics of real-world VHDL code, the most prominent VHDL repositories on GitHub [15] were analyzed, covering over 6000 source files.

Figure 6.1 shows the distribution of lines of code across the analyzed files.

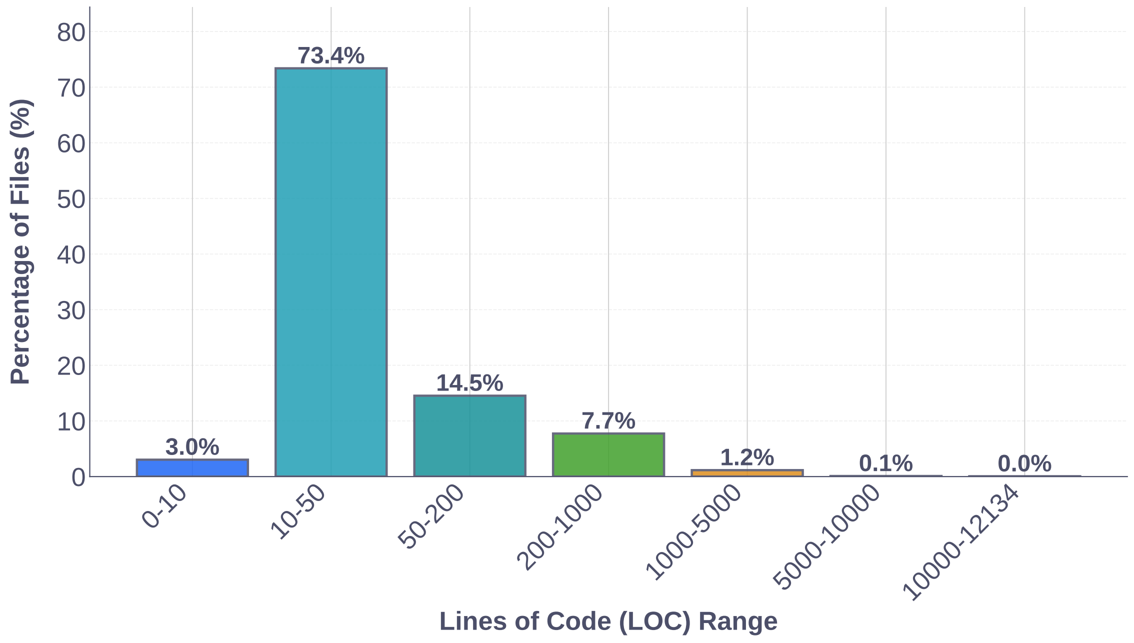

Figure 6.1: Distribution of LOC in prominent VHDL repositories on GitHub

Approximately $90\,\%$ of VHDL files contain fewer than 200 lines of code. This indicates that typical VHDL source files are comparatively small and structurally simple, with only a small fraction exceeding 1000 lines of code.

6.1.1 Implications for Optimization

Since $90\,\%$ of VHDL files are small and simple (< 200 LOC), optimization should target this common case rather than worst-case performance. This justifies the hybrid parsing strategy (Section 4.1): attempt fast SLL mode first (succeeds for simple files), falling back to LL(*) only when necessary (complex cases).

6.2 Parser Optimization

Profiling revealed that the ANTLR-generated parser dominated execution time. The pipeline was scrutinized using `Catch2`'s micro-benchmarking framework to analyze this bottleneck.

As established in Section 4.10 (Table 4.2), parsing in the optimized SLL parsing mode accounts for $32\,\%$ of total execution time ($60.53\,\mu s$). However, benchmarking reveals that with the default pure LL(*) prediction mode, this cost would skyrocket to $480.98\,\mu s$, consuming $78\,\%$ of the total budget.

6.2.1 Optimization Strategy

According to Tomassetti [16], ANTLR's prediction mode significantly impacts performance. ANTLR provides two modes:

- **LL(*):** Full adaptive prediction with arbitrary lookahead. Handles all ambiguous grammars. Measurements show this requires $480.98\,\mu s$ for the test file.
- **SLL:** Single-Lookahead LL prediction. Faster but fails on ambiguous constructs (function calls vs. array indexing). Measurements show this drops to $60.53\,\mu s$ for the same file.

A hybrid parsing strategy was implemented (Section 4.1): attempt SLL first, fall back to LL(*) on failure. Since the test file contains typical VHDL constructs, SLL succeeds without requiring fallback.

6.2.2 Performance Impact

For files that parse successfully with SLL (the common case), the improvement is $87\,\%$:

$$\text{SLL Improvement} = \frac{480.98 - 60.53}{480.98} \times 100\,\% = 87.4\,\%$$

Table 6.1 compares the strategies.

Strategy	**Parsing**	**Rest**	**Total**	**Parsing %**
Pure LL(*)	480.98 µs	135.16 µs	616.14 µs	$78.1\,\%$
Hybrid (SLL success)	60.53 µs	128.16 µs	188.69 µs	$32.1\,\%$
Reduction	-$87.4\,\%$	-$5.2\,\%$	-$69.4\,\%$	–

Table 6.1: Parsing Time Distribution: Pure LL(*) vs. Hybrid Strategy (common case)

Overall formatting time improves by $69\,\%$ (616.14 µs vs. 188.69 µs), with parsing shifting from $78\,\%$ to $32\,\%$ of total time. For files requiring LL(*) fallback, performance matches pure LL(*) mode plus negligible overhead from the failed SLL attempt.

Section 6.1 showed $90\,\%$ of VHDL files contain fewer than 200 lines with simple constructs. The hybrid strategy optimizes for this common case, achieving $87\,\%$ improvement for typical files while ensuring correctness for complex files via LL(*) fallback.

7 Results and Further Development

7.1 Achieved Results

The implemented formatter is able to process and format the majority of VHDL source files while preserving the underlying AST structure. As a result, the semantic meaning of the code remains unchanged after formatting. Some language constructs are not yet fully supported, and formatting for these cases is still incomplete.

The formatter operates exclusively on syntactically correct input. If parsing fails due to syntax errors, the formatting process is aborted and an error message is reported. This design decision prevents the formatter from producing unpredictable output on invalid source code.

For valid input, the formatting process is deterministic: identical input files always result in identical formatted output.

While the overall formatting quality meets the intended design goals, certain style details do not yet fully match the desired formatting and require further refinement.

7.2 Future Work

Although the formatter already supports a substantial subset of the VHDL language, it is not yet feature complete.

Support for the full VHDL-2008 standard remains an important goal. This includes handling of less frequently used language constructs and addressing remaining edge cases in both the grammar and the formatting logic.

The set of formatting options could be expanded to provide greater flexibility. Additional user-configurable rules would allow the formatter to better adapt to project-specific style guidelines.

Further refinement of the formatting logic is also required. In particular, complex and deeply nested constructs are not always formatted optimally and would benefit from improved layout rules.

In addition to language features, platform support should be extended. Providing prebuilt binaries for additional operating systems and architectures could improve popularity. Publishing the tool through package repositories, such as the AUR, would further reduce installation effort for users.

7.2.1 Sustainability and Community Adoption

The long-term sustainability of the project depends on its accessibility and maintainability. Community adoption can be encouraged by presenting the project to relevant developer communities, for example through discussion forums focused on FPGA and VHDL development.

The provided Dev Container setup further lowers the entry barrier for new contributors by enabling a consistent development environment without complex local configuration.

7.2.2 Railway Oriented Programming

The current error handling strategy within the `Translator` relies on implicit exception propagation. While functional, this approach obscures control flow and complicates the verification of system state during partial failures. A significant opportunity for architectural refinement lies in the adoption of Railway Oriented Programming.

This paradigm leverages the type system to model computations as a pipeline with two parallel tracks: a success path and a failure path. By refactoring the translation functions to return C++23 `std::expected` types, the system can utilize monadic combinators to chain operations. This structure ensures that errors are propagated deterministically, bypassing subsequent steps without the overhead or unpredictability of runtime exceptions. The result would be a translation pipeline that is more declarative, type safe, and rigorous in its handling of edge cases.

7.2.3 Verification Limitation

The current implementation of `IS_SEMANTIC` in the verification step focuses strictly on the `DEFAULT_TOKEN_CHANNEL`. Consequently, while the verifier guarantees that the code compiles and runs identically, it does not currently guard against the accidental loss of comments during the AST-to-Doc transformation. A proposed enhancement is to extend the verification filter to include the `COMMENT` channel, ensuring that the set of comments in the output matches the input, even if their layout has changed.

List of Figures

LIST OF TABLES

List of Listings

GLOSSARY

ANTLR	Parser generator that produces lexers and parsers from formal grammar specifications
API	Application Programming Interface; defined interface enabling interaction between software components
AST	Abstract Syntax Tree; hierarchical representation of program structure without syntactic trivia
AUR	Arch User Repository; community-maintained package repository for Arch Linux
benchmarking	Systematic measurement of performance characteristics such as runtime and memory usage
CI	Continuous Integration; automated building and testing of code changes
clang-format	Automatic C++ code formatter based on Clang
clang-tidy	Static analysis and linting tool built on the Clang frontend
CLI	Command-Line Interface; text-based program interaction model
CMake	Cross-platform build configuration system generating native build files
compiler	Program that translates source code into executable or intermediate representations
Conan	Package manager for C and C++ projects

CST	Concrete Syntax Tree; parse tree preserving all syntactic elements
DSL	Domain-Specific Language; language or API tailored to a specific problem domain
FPGA	Field-Programmable Gate Array; reconfigurable digital hardware device
grammar	Formal specification defining the valid syntax of a language
hybrid parsing	Parsing strategy combining fast SLL prediction with LL fallback
IR	Intermediate Representation; abstract form used for analysis and transformation
JSON	JavaScript Object Notation; lightweight text-based data format
language server	Process providing editor features via the Language Server Protocol
lexer	Component that converts character streams into tokens
linter	Static analysis tool detecting style and quality issues
LL parsing	Top-down parsing strategy using full lookahead
Ninja	Fast build system optimized for incremental builds
node	Single element within a syntax tree
parser	Component that constructs syntax trees from token streams
parser generator	Tool that generates parsers from grammar definitions
parsing	Process of analyzing tokens according to grammar rules
profiling	Runtime analysis for identifying performance bottlenecks

SLL parsing	ANTLR prediction mode using limited lookahead for performance
token	Atomic lexical unit produced by the lexer
tooling	Collection of development tools supporting software creation
trivia	Syntactically insignificant elements such as whitespace and comments
VHDL	VHSIC Hardware Description Language; IEEE-standard language for digital hardware design
YAML	Human-readable data serialization format commonly used for configuration

Bibliography

[1] Vhdl language server. Online. Accessed: 2025-09-23. [Online]. Available: https://github.com/VHDL-LS/rust_hdl

[2] Vhdlformatter. Online. Accessed: 2025-09-23. [Online]. Available: https://github.com/g2384/VHDLFormatter

[3] vhdlfmt. online. Accessed: 2025-09-23. [Online]. Available: https://vhdlfmt.com/

[4] J. Leary. vhdl-style-guide. online. Accessed: 2025-12-14. [Online]. Available: https://github.com/jeremiah-c-leary/vhdl-style-guide

[5] Antlr. Online. Accessed: 2025-11-27. [Online]. Available: https://www.antlr.org/

[6] Y. Hirose. cpp-peglib. Online. Accessed: 2025-11-27. [Online]. Available: https://github.com/yhirose/cpp-peglib

[7] Tree-sitter. Online. Accessed: 2025-11-27. [Online]. Available: https://tree-sitter.github.io/

[8] Vhdl grammar for antlr4. Online. Accessed: 2025-11-27. [Online]. Available: https://github.com/antlr/grammars-v4/tree/master/vhdl

[9] P. Wadler, "A prettier printer," *The fun of programming*, pp. 223–243, 2003.

[10] p ranav. argparse. Online. Accessed: 2025-12-10. [Online]. Available: https://github.com/p-ranav/argparse

[11] H. S. Bjarne Stroustrup. C++ core guidelines. online. Accessed: 2025-12-14. [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[12] P. Nash. Catch2. online. Accessed: 2025-12-19. [Online]. Available: https://catch2.org/

[13] tpoechtrager. osxcross. Online. Accessed: 2025-12-12. [Online]. Available: https://github.com/tpoechtrager/osxcross

[14] Apple. Xcode and apple sdks agreement. Online. Accessed: 2025-12-12. [Online]. Available: https://www.apple.com/legal/sla/docs/xcode.pdf

[15] Most prominent vhdl repositories on github. Online. Accessed: 2025-11-09. [Online]. Available: https://github.com/search?q=lang%3AVHDL+is%3Apublic+stars%3A%3E700&type=repositories&s=stars&o=desc

[16] F. Tomassetti. Improving the performance of an ANTLR parser. Online. Accessed: 2025-11-04. [Online]. Available: https://tomassetti.me/improving-the-performance-of-an-antlr-parser/