VOLVO

CAPLint: A Static Analysis Tool for CAPL in Automotive Software Development

Automated Code Quality Enforcement for HIL Test Scripts

Bachelor's thesis in Computer Engineering

AMIR SHOJAY
SHARIQ MUSHARAF

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Bachelor's thesis 2026

CAPLint: A Static Analysis Tool for CAPL in Automotive Software Development

Automated Code Quality Enforcement for HIL Test Scripts

AMIR SHOJAY
SHARIQ MUSHARAF

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

CAPLint: A Static Analysis Tool for CAPL in Automotive Software Development
Automated Code Quality Enforcement for Hardware-in-the-Loop Test Scripts

AMIR SHOJAY
SHARIQ MUSHARAF

Supervisor: Sakib Sistek, Department of Computer Science and Engineering
Examiner: Jean-Philippe Bernardy, Department of Computer Science and Engineering
Industry Supervisor: Johan Friberg, Volvo Cars

Bachelor's thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Abstract

CAPL (Communication Access Programming Language) is a domain-specific language widely used in the automotive industry for Hardware-in-the-Loop (HIL) testing of electronic control units. Despite its critical role in vehicle software validation, CAPL has limited or no widely adopted static analysis tooling to enforce coding standards or detect common programming mistakes. This gap forces development teams to rely on manual code review, a process that could be time-consuming, error-prone, and difficult to scale.

This thesis presents CAPLint, a static analysis tool designed specifically for CAPL. The tool was developed in collaboration with the HIL development team at Volvo Cars in Gothenburg, Sweden, and informed by a formative survey of engineers on their CAPL debugging practices. CAPLint addresses this problem through a two-tier analysis architecture: a fast text-based tier that operates on raw source lines using pattern matching, and a structured tier that performs analysis on an abstract syntax tree generated by a custom-built LALR parser. The parser was developed from scratch due to the absence of a publicly available CAPL grammar specification, and achieves a 100% parse success rate on a large production corpus of CAPL files. CAPLint implements 39 lint rules across fifteen categories, including naming conventions, formatting, documentation, semantic checks for unused and undeclared variables, and cross-file include analysis. The tool supports hierarchical configuration, inline suppression comments, and integrates into existing command-line workflows and CI/CD pipelines. The system was developed using an Agile methodology over a 17-week period and evaluated through corpus validation and developer feedback. The results show that automated static analysis can be applied to a domain-specific language with limited public documentation, and that CAPLint can reduce the amount of repetitive checking engineers do during code review.

Keywords: static analysis, linting, CAPL, domain-specific language, automotive, HIL testing, parser, code quality.

Acknowledgements

This bachelor's thesis was carried out between January and May 2026 at the Department of Computer Science and Engineering, Chalmers University of Technology, in collaboration with the Hardware-in-the-Loop (HIL) development team at Volvo Cars in Torslanda.

We thank our academic supervisor, Sakib Sistek, for steady guidance throughout the project, for sharp questioning of our design decisions, and for keeping our scope honest. We thank our examiner, Jean-Philippe Bernardy, for valuable input on the academic framing of the work.

We owe a particular debt of gratitude to our industry supervisor, Johan Friberg of Volvo Cars, for his continuous support, valuable feedback, and for providing ongoing access to engineers, source code, and internal coding guidelines throughout the project. We are also deeply grateful to Alireza Arjomand, Hiring Manager, who opened the door to the HIL development team, believed in our proposal, and selected us among all candidates. Without their trust and the support of the wider team, this thesis would not have been possible.

Finally, we would also like to thank the engineers at Volvo Cars who participated in our survey and provided insights into the challenges of manual code review in automotive software development. We are particularly grateful to Sarvsav Sharma and Ali Kiarostami for the time they took to teach us how things actually work in practice.

Amir Shojay and Shariq Musharaf, Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

API	Application Programming Interface
AST	Abstract Syntax Tree
CAN	Controller Area Network
CAPL	Communication Access Programming Language
CI/CD	Continuous Integration / Continuous Deployment
CLI	Command-Line Interface
DSL	Domain-Specific Language
ECU	Electronic Control Unit
HIL	Hardware-in-the-Loop
LALR	Look-Ahead Left-to-Right
PR	Pull Request
TDD	Test-Driven Development
TOML	Tom's Obvious Minimal Language

Nomenclature

Below is the nomenclature of key terms that have been used throughout this thesis.

General Terms

Diagnostic	A warning or error message produced by a lint rule when a violation is detected in source code
False positive	A diagnostic reported by the linter that does not correspond to an actual violation
Fixture	A sample CAPL source file used in testing, annotated with expected diagnostics
Lint / Linter	A static analysis tool that checks source code for stylistic errors, potential bugs, and guideline violations without executing the code
Parse tree	The concrete syntax tree produced by the parser, representing the full grammatical structure of the source code
Rule	A single check implemented in the linter, identified by a category prefix and number (e.g., FMT001)

CAPLint-Specific Terms

L0 rule	A text-based rule that operates on masked source lines without requiring parsing
L1 rule	A tree-based rule that operates on the abstract syntax tree or symbol table
Masked line	A source line where string literals and comments have been replaced with whitespace, preserving column positions
Suppression	An inline source comment that instructs the linter to ignore specific rules on a given line or file
Tier	One of the two analysis levels in CAPLint: Level 0 (text-based) or Level 1 (tree-based)

Contents

List of Figures

List of Tables

1
Introduction

This chapter introduces the context and motivation for this thesis, defines the problem being addressed, states the research questions, and outlines the scope and structure of the report.

1.1 Background

Modern vehicles are increasingly defined by their software: a typical passenger car contains dozens of electronic control units (ECUs) that manage everything from engine performance and braking to infotainment and driver assistance [2], and as this software grows in complexity so does the need for accurate and exhaustive testing.

Hardware-in-the-Loop (HIL) testing is a standard verification technique in the automotive industry, in which real ECUs are connected to a simulated vehicle environment to validate their behavior [8]. At Volvo Cars, HIL test engineers use CAPL (Communication Access Programming Language), a domain-specific language developed by Vector Informatik, to write test scripts that simulate CAN bus communication, trigger events, and verify ECU responses within the CANoe simulation environment [11]. CAPL is syntactically C-like but event-driven: rather than a sequential `main()` flow, programs are organized as handlers that respond to messages, signals, timers, and system variables on the bus. Combined with its tight coupling to the proprietary CANoe platform, this leaves CAPL without the tooling ecosystem that developers of mainstream languages take for granted. A more detailed treatment of the CAN bus, the CAPL execution model, and HIL testing is given in Chapter 2.

In particular, CAPL has limited or no widely adopted and dedicated static analysis tool or ecosystem. While languages such as Python, JavaScript, and C benefit from mature linters like Pylint [9], ESLint [5], and Clang-Tidy [3] that automatically enforce naming and formatting conventions and flag style, structural, and maintainability issues, CAPL developers must rely on manual processes. At Volvo Cars, the HIL development team maintains internal coding guidelines documented on an internal page, but enforcement is not automated: which can lead to engineers having to spend time on reviewing the code manually.

A formative survey of engineers in the HIL development team and adjacent teams at Volvo Cars (Appendix A cross-references the rules; some survey data is reported in Section 4.9) showed that most respondents spend a meaningful amount of time each week debugging and reviewing CAPL code for guideline violations, many of

which fall within the scope of automated static analysis.

1.2 Problem Statement

CAPL is a domain-specific language with no publicly available grammar specification, no formal language standard, and no widely adopted static analysis tooling. The absence of a linter creates several problems for development teams:

- **Manual review overhead:** Engineers must manually check code for stylistic issues and common mistakes during code review, diverting time from higher-value activities such as reviewing logic and test coverage.
- **Inconsistent enforcement:** Without automated checks, guideline adherence depends on reviewer diligence and could potentially vary across the team.
- **Late detection:** Issues that could be caught at authoring time are instead discovered during review or, in the worst case, risk never being caught at all.
- **Scalability:** As codebases grow and teams expand, manual review becomes increasingly difficult to sustain.

The core challenge is not merely implementing lint rules, but building the foundational infrastructure, i.e. a parser and abstract syntax tree representation, for a language that lacks public documentation of its grammar. Any static analysis tool for CAPL must first solve the parsing problem before it can perform meaningful analysis.

1.3 Purpose and Research Questions

The purpose of this thesis is to design, implement, and evaluate a static analysis tool for CAPL that can automatically detect coding guideline violations and common programming mistakes, thereby reducing the burden of manual code review in automotive HIL development.

To guide the work, three research questions are formulated:

RQ1: *How can a static analysis tool be designed and implemented for a domain-specific language that lacks a publicly available grammar specification and existing tooling support?*

This question addresses the technical challenge of building a linter from scratch for CAPL, encompassing grammar development, parser construction, AST design, and rule architecture. It explores the design decisions and trade-offs involved in creating analysis infrastructure for a language that provides no formal grammar specification, and its syntax must be reconstructed from real code.

RQ1 is framed as a software-engineering goal rather than a general research question. There is an established body of work on grammar inference from corpora, parser generation, and incremental language tooling, and a full survey of that literature is outside the scope of this project. The aim here is concrete and bounded: to produce, within the time available for a bachelor thesis, a working static analysis tool for one specific dialect of CAPL as it appears in the Volvo Cars production codebase. The discussion of

RQ1 therefore reports the design choices and trade-offs that made that goal achievable in this context, not a general method that is claimed to work for arbitrary DSLs.

RQ2: *In what ways can automated static analysis reduce the time spent on manual code review of CAPL scripts in an automotive HIL testing environment?*
This question addresses the practical impact of the tool. It examines which categories of issue engineers currently identify manually that CAPLint can automate, and how the implemented rules redistribute review effort, rather than measuring time saved in a controlled before/after study. A controlled productivity measurement is identified as future work in Chapter 6.

RQ3: *What are the structural limits of static analysis for an event-driven proprietary DSL?*
This question addresses what static analysis can and cannot do for a language like CAPL. It looks at the structural features of the language and its toolchain (such as external symbol environments from `.dbc` CAN-database files, opaque preprocessor directives, and event-driven entry points) that no purely text-based or AST-based analyzer can fully resolve on its own. The findings generalize beyond CAPL to other proprietary DSLs with similar properties.

1.4 Scope and Limitations

The scope of this thesis encompasses the design, implementation, and evaluation of CAPLint as a command-line tool. The following boundaries apply:

- **Language coverage:** The parser targets the subset of CAPL used in production at Volvo Cars, validated against a large production corpus of CAPL files. Rarely used or undocumented constructs not present in the corpus may not be supported.
- **Rule set:** The implemented rules are based on guidelines collected from the Volvo Cars HIL team. The rule set is not exhaustive; it covers a representative subset of those conventions rather than every possible check.
- **Integration:** CAPLint is delivered as a CLI tool. IDE integration (e.g., as a VS Code extension) and automated fix suggestions are outside the scope of this thesis.
- **Evaluation:** The evaluation is based on corpus validation and a formative developer survey. A controlled before/after study of productivity impact was not feasible within the project window and is identified as future work.
- **Generalizability:** While the design principles may be applicable to other domain-specific languages, the implementation is specific to CAPL.

1.5 Ethical Considerations

This thesis was conducted in collaboration with Volvo Cars, which raises considerations regarding proprietary information. The CAPL source files used for corpus validation are proprietary to Volvo Cars and are not included in or distributed with

this thesis. The grammar and lint rules were developed independently based on the publicly observable structure of the CAPL language and general coding best practices; no proprietary documentation from Vector Informatik was used in their construction.
The developer survey was conducted with informed consent. Participation was voluntary, and responses were anonymized. No personally identifiable information is reported in this thesis.
The tool itself is designed to assist developers, not replace them. CAPLint provides suggestions and warnings; it does not modify source code automatically or make decisions about code correctness. The final judgment on whether to address a diagnostic remains with the developer.

1.6 Report Structure

The remainder of this report is organized as follows:

Chapter 2 provides the theoretical background on static analysis, parsing, and automotive software development needed to understand the design of CAPLint.

Chapter 3 describes the methodology, development process, and key design decisions.

Chapter 4 presents the implemented system, its capabilities, and the evaluation results.

Chapter 5 discusses the results in relation to the research questions, design trade-offs, and threats to validity.

Chapter 6 summarizes the contributions and suggests directions for future work.

2

Theory

This chapter presents the theoretical foundations underlying the design of CAPLint. It covers static analysis, parsing and compiler front-end theory, the CAPL language and its automotive context, and code review practices.

2.1 Static Analysis

Static analysis is the examination of source code without executing it [1]. Unlike dynamic analysis, which observes program behavior at runtime, static analysis reasons about all possible executions by inspecting the code's structure and content. This makes it well-suited for detecting certain classes of errors early in the development process, before code reaches testing or production.

Static analysis can be performed at several levels:

- **Pattern matching:** The simplest form, where textual patterns (often regular expressions) are matched against source lines. This approach requires no parsing and can detect formatting issues, banned API calls, and simple stylistic violations. Its limitation is that it operates without understanding the code's structure, leading to false positives when patterns match inside strings or comments.
- **Syntactic analysis:** Analysis performed on a parse tree or abstract syntax tree (AST), enabling checks that depend on code structure. Examples include verifying naming conventions for functions versus variables, detecting unreachable code after return statements, and enforcing documentation requirements.
- **Semantic analysis:** A deeper form of analysis that builds symbol tables, resolves variable scopes, and tracks data flow. This enables detection of undeclared variables, unused declarations, and type mismatches.
- **Abstract interpretation and model checking:** Advanced techniques that compute properties over all possible program executions. These are used in safety-critical domains but are computationally expensive and typically applied to smaller, well-defined languages [4].

CAPLint employs the first three levels of this spectrum: pattern matching for fast text-based rules, syntactic analysis for structure-dependent rules, and semantic analysis for scope-related checks.

2.1.1 Linting Tools

A linter is a specific category of static analysis tool focused on code quality rather than correctness proofs. The term originates from the `lint` utility for C, developed by Stephen Johnson at Bell Labs in 1978 [6]. Modern linters have evolved to cover stylistic conventions, potential bugs, and best practices, and are integral to professional software development workflows.
Table 2.1 summarizes prominent linters for mainstream languages. These tools share common design patterns that informed the architecture of CAPLint: rule-based analysis, configurable severity levels, and inline suppression mechanisms.

Table 2.1: Prominent linters for mainstream programming languages.

Language	Tool	Key Characteristics
C	Lint, Clang-Tidy	Pattern-based and AST-based checks; deep integration with compiler infrastructure
Python	Pylint, Flake8	Convention enforcement, error detection, complexity metrics; plugin architectures
JavaScript	ESLint	Highly configurable rule system; auto-fix capabilities; parser-pluggable design
Java	Checkstyle, SpotBugs	Style checking and bytecode-level bug detection
Go	`go vet`, golangci-lint	Built into the language toolchain; enforces idiomatic Go patterns

A common architectural pattern across these tools is the separation of parsing from analysis: a front-end produces a structured representation of the code, and a set of independent rules traverse that representation to produce diagnostics. This separation enables modularity, which means new rules can be added without modifying the parser, and it is the pattern adopted by CAPLint.

2.2 Parsing and Compiler Front-Ends

Parsing is the process of transforming a sequence of characters (source code) into a structured representation that preserves the grammatical relationships between language constructs. This section covers the key concepts used in CAPLint's parser.

2.2.1 Formal Grammars

A context-free grammar (CFG) defines the syntax of a programming language through a set of production rules. Each rule specifies how a non-terminal symbol can be expanded into a sequence of terminal and non-terminal symbols. For example, a simplified grammar for an if-statement might be:

```
if_stmt : "if" "(" expression ")" block
        | "if" "(" expression ")" block "else" block
```

The grammar serves as the formal specification of a language's syntax and is the input to parser generators, which produce parsers automatically from grammar definitions.

2.2.2 LALR Parsing

LALR (Look-Ahead LR) parsing is a bottom-up parsing technique that reads input left-to-right and constructs a rightmost derivation in reverse. LALR parsers are efficient (linear time in the input length), deterministic, and capable of handling most practical programming language grammars. They operate by maintaining a stack and a state machine: at each step, the parser either shifts (pushes the next token onto the stack) or reduces (replaces a sequence of stack elements with a nonterminal according to a grammar rule).
LALR parsers are widely used in practice. Tools such as Yacc, Bison, and Lark generate LALR parsers from grammar specifications. CAPLint uses the Lark parsing library for Python, which provides LALR parsing with a contextual lexer, i.e. a lexer that adjusts its tokenization rules based on the current parser state, enabling it to handle context-sensitive tokenization without complicating the grammar.

2.2.3 Abstract Syntax Trees

An abstract syntax tree (AST) is a simplified representation of a program's structure that omits syntactic details (such as parentheses, semicolons, and keywords) irrelevant to analysis. While a parse tree (also called a concrete syntax tree) preserves every grammatical detail, an AST retains only the semantically meaningful structure.
For example, the CAPL expression `x = a + b * c;` would produce an AST with an assignment node at the root, an identifier `x` on the left, and a binary addition on the right, where the right operand of the addition is itself a binary multiplication, reflecting operator precedence without requiring parenthesis nodes.
The AST is the primary data structure consumed by analysis rules. A well-designed AST simplifies rule implementation by providing typed, structured access to program elements. CAPLint's AST uses Python frozen dataclasses [10] to represent nodes, ensuring immutability and type safety.

2.2.4 Semantic Analysis

Semantic analysis goes beyond syntax to reason about the meaning of programs. The most common form is scope analysis, which builds a symbol table mapping identifiers to their declarations and tracks where each identifier is used.
A symbol table enables two important checks:

- **Undeclared variable detection:** An identifier is used but has no corresponding declaration in any enclosing scope.
- **Unused variable detection:** A variable is declared but never referenced in the code.

Scope analysis must account for nested scopes (local variables inside functions, blocks within loops), parameter declarations, and global variables. In CAPL, scope

analysis is further complicated by the `includes` mechanism, where `.cin` fragment files contribute declarations to the including `.can` file's namespace.
Figure 2.1 illustrates how a symbol table is constructed for a simple CAPL function.

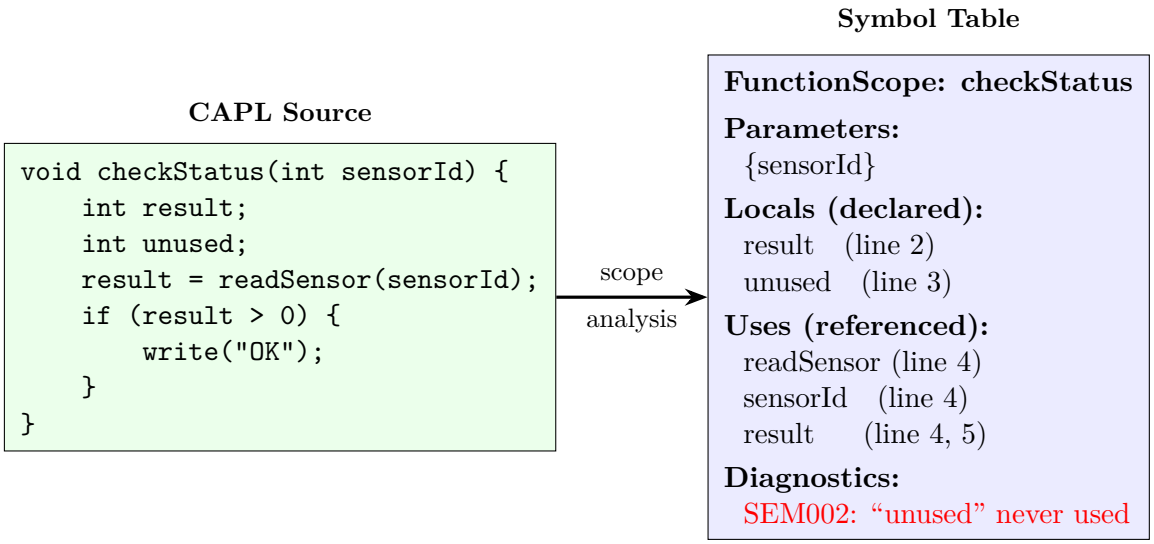

Figure 2.1: Scope analysis builds a symbol table for each function, tracking parameter names, local declarations, and identifier uses. Comparing declared names against used names reveals unused variables (SEM002).

2.3 CAPL and the Automotive Context

2.3.1 The CAN Bus

The Controller Area Network (CAN) is a serial communication protocol developed by Bosch in the 1980s for in-vehicle networking. CAN enables ECUs to communicate without a central host computer, using a priority-based message arbitration scheme. It has become the dominant bus system in automotive electronics and is standardized as ISO 11898.
In a CAN network, communication is message-oriented: ECUs broadcast messages identified by a CAN ID, and any ECU on the bus can listen for messages of interest. This publish-subscribe model influences the design of CAPL, whose event handlers respond to messages, signals, and system variables on the bus.

2.3.2 CAPL Language Overview

CAPL is a domain-specific language developed by Vector Informatik for use within their CANoe and CANalyzer simulation and analysis tools [11]. While syntactically similar to C, CAPL has several distinctive characteristics:

- **Event-driven execution model:** Instead of a sequential `main()` function, CAPL programs consist of event handlers triggered by bus events. Common event types include `on message` (CAN message received), `on timer` (timer

expiry), `on key` (keyboard input), `on sysvar` (system variable change), and `on start` (simulation start).

- **Global variables block:** Variables declared in a top-level `variables {}` block are accessible to all event handlers and functions in the file.
- **Built-in communication functions:** CAPL provides functions for CAN bus interaction such as `output()` (send a message), `setTimer()`, and `sysSetVariableInt()`.
- **Include mechanism:** CAPL files (`.can`) can include fragment files (`.cin`) that contribute declarations to the including file's namespace. This is similar to C's `#include` but operates at the CAPL source level.
- **No public grammar specification:** Unlike standardized languages, CAPL's syntax is defined only by the CANoe implementation. There is no formal grammar specification, language standard, or open-source parser available.

Listing 2.1 shows a simple CAPL program that demonstrates the event-driven model.

```
variables {
    int messageCount = 0;
    msTimer pollTimer;
}

on start {
    setTimer(pollTimer, 1000);
    write("Simulation started");
}

on timer pollTimer {
    messageCount++;
    write("Messages received: %d", messageCount);
    setTimer(pollTimer, 1000);
}

on message EngineStatus {
    messageCount++;
    if (this.RPM > 6000) {
        write("Warning: High RPM detected");
    }
}
```

Listing 2.1: A simple CAPL program demonstrating event handlers.

2.3.3 Hardware-in-the-Loop Testing

Hardware-in-the-Loop (HIL) testing is a verification technique where a real ECU is connected to a test system that simulates the rest of the vehicle [8]. The HIL system generates realistic sensor signals and bus messages, allowing the ECU to be tested in conditions that closely approximate the real vehicle environment without requiring a physical prototype.

In the automotive V-model development process, HIL testing occupies the integration and system testing levels. CAPL scripts are central to HIL testing: they simu-

late communication partners, inject fault conditions, and implement test sequences that verify ECU behavior against requirements.
At Volvo Cars, the HIL development team maintains large CAPL codebases, which is the corpus used in this thesis and spans large in-production test projects. As these codebases grow, maintaining code quality becomes increasingly important for test reliability and maintainability.

2.4 Code Review

Code review is the systematic examination of source code by developers other than the original author. It is one of the most effective software quality assurance practices and is standard in industrial software development.
In practice, code review serves multiple purposes:

- Detecting bugs and logic errors
- Enforcing coding standards and conventions
- Sharing knowledge across the team
- Improving code maintainability

However, the effectiveness of code review depends on reviewer effort and attention. When reviewers must spend significant time on mechanical checks, verifying naming conventions, checking formatting, looking for common anti-patterns, less cognitive capacity remains for the more valuable task of reviewing logic and design.
This is precisely the motivation for linters: by automating the detection of guideline violations and common mistakes, linters free reviewers to focus on aspects of code quality that require human judgment. The goal of CAPLint is to shift mechanical checks from manual review to automated analysis, improving both the efficiency and consistency of the code review process.

2.5 Agile Software Development

Agile software development is an iterative approach to software engineering that emphasizes incremental delivery, continuous feedback, and adaptability to changing requirements. In contrast to traditional waterfall models, Agile methodologies organize work into short iterations (sprints), each producing a working increment of the software.
Key Agile practices relevant to this thesis include:

- **Iterative development:** Functionality is built incrementally, with each iteration adding new capabilities and refining existing ones based on feedback.
- **Test-driven development (TDD):** Tests are written before the implementation code, ensuring that each feature is developed against a clear specification.
- **Continuous integration:** Changes are integrated frequently, with automated tests providing rapid feedback on regressions.

Agile is particularly well-suited to tool development projects where requirements emerge through use. In the case of CAPLint, the lint rules were identified and

prioritized iteratively through ongoing consultation with the development team at Volvo Cars, rather than specified exhaustively upfront.

3

Methods

This chapter describes the methodology used to develop CAPLint, the development process, and the key design decisions that shaped the tool's architecture.

3.1 Development Methodology

CAPLint was developed over 17 weeks (January 19 to May 15, 2026) using an Agile methodology with iterative sprints. The project was carried out on-site at Volvo Cars in Torslanda, Gothenburg, providing direct access to stakeholders and the production CAPL codebase.

Both authors worked collaboratively on all aspects of the project, including grammar development, parser implementation, rule design, testing, and evaluation. Task allocation was managed through daily coordination rather than formal division, consistent with Agile pair programming practices.

The development was organized into the following phases:

1. **Requirements gathering (weeks 1–2):** Interviews with the industry supervisor and HIL team members to understand the existing code review process, identify common guideline violations, and collect the coding guidelines documented on the team's internal notes.
2. **Grammar and parser development (weeks 2–6):** Incremental construction of the CAPL grammar and parser, validated continuously against production files from the corpus. This was the most technically challenging phase, as no grammar specification existed.
3. **Rule implementation (weeks 4–14):** Iterative implementation of lint rules, prioritized by frequency of violation as reported by the team. Text-based (L0) rules were implemented first due to their lower complexity, followed by tree-based (L1) rules requiring the parser.
4. **Semantic analysis (weeks 10–14):** Development of the scope analyzer, include resolver, and builtins registry to enable cross-file and scope-aware analysis.
5. **Evaluation and refinement (weeks 14–17):** Corpus-wide validation, developer survey, and refinement of rules based on false positive analysis.

These phases overlapped significantly: grammar improvements were driven by parse failures encountered during rule testing, and new rules were often inspired by patterns discovered during corpus validation. Figure 3.1 visualizes the timeline and overlap of these phases.

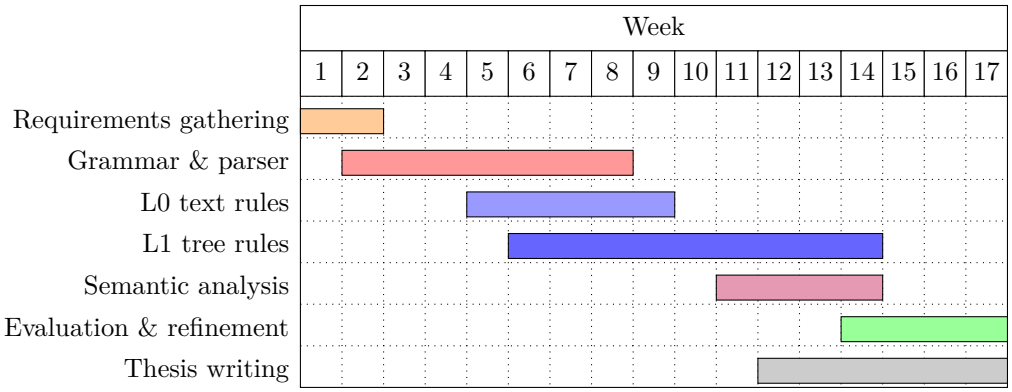

Figure 3.1: Development timeline showing the 17-week project phases. Phases overlapped as grammar improvements were driven by rule testing, and new rules were inspired by corpus validation.

3.1.1 Test-Driven Development

CAPLint was developed using test-driven development (TDD). For each lint rule, the development cycle followed this sequence:

1. **Fixture creation:** Write a CAPL fixture file containing both violating and non-violating code, annotated with `// EXPECT: RULE@LINE` comments indicating where diagnostics should appear.
2. **Test implementation:** Write a test that runs the linter on the fixture and asserts that exactly the expected diagnostics are produced.
3. **Rule implementation:** Implement the rule to make the test pass.
4. **Corpus validation:** Run the rule against the full production corpus to check for false positives and unexpected behavior.

This approach ensured that every rule was developed against a concrete specification (the fixture file) and validated against real-world code before being considered complete.

3.2 Design Decisions

Several key design decisions shaped the architecture of CAPLint. This section presents each decision, its rationale, and the alternatives considered.

3.2.1 Alternatives Considered at the Project Level

Before settling on the approach described in the rest of this section, three broader alternatives were evaluated and rejected.

Regex-only linter. A purely text-based linter, with no parser, would have been quicker to build and would have avoided the grammar-development effort entirely. It was rejected because many of the rules engineers asked for (naming by declaration kind, scope-based undeclared and unused variable detection, switch-statement structure, doc-comment placement) depend on knowing what a token is, not just

how it looks. A regex-only design would have either omitted those rules or accepted a high false positive rate on them, undermining trust in the tool.

Reimplementing CANoe semantics. A semantically complete analyzer would also interpret `.dbc` CAN-database files, follow Windows-style `#include` paths, and model the preprocessor and the event-handler scheduling rules. This would close most of the false positive clusters reported in Section 4.8 but was clearly out of scope for a 17-week project. It is identified as future work in Chapter 6 rather than attempted here.

Grammar inference instead of a hand-written grammar. An automatically inferred grammar would in principle remove the manual iteration cycle. As discussed further in Section 5.2, grammar inference for full programming languages is an active research area with limited practical tooling, and the manual approach gave better control over ambiguity resolution and error recovery on the production corpus. The chosen approach (hand-written grammar plus a two-tier architecture that degrades gracefully when parsing fails) accepts a higher up-front cost in exchange for predictable behavior and rules that can be reasoned about individually.

These three alternatives frame the choices presented below: the two-tier architecture (Section 3.2.2) and the hand-written grammar (Section 3.2.3) are the concrete outcome of preferring a parser-backed design with bounded scope over either a regex-only shortcut or a much larger semantic reimplementation.

3.2.2 Two-Tier Analysis Architecture

The most significant architectural decision was the adoption of a two-tier analysis pipeline:

Level 0 (L0): Text rules: Operate on raw source lines using regular expressions and pattern matching. Before matching, each line is "masked": string literals and comments are replaced with whitespace, preserving column positions. This prevents false matches inside strings or comments while maintaining positional accuracy for diagnostic reporting.

Level 1 (L1): Tree rules: Operate on the abstract syntax tree produced by the parser. L1 rules are further subdivided:

- **L1-tree:** Rules that traverse the raw Lark parse tree.
- **L1-ast:** Rules that traverse the typed domain AST (Python dataclasses).
- **L1-semantic:** Rules that use the symbol table built by the scope analyzer.

The rationale for this separation is *progressive enrichment*: rules should use the simplest analysis tier that solves the problem. Text rules are fast and always run while tree rules only run when parsing succeeds. This design means that even if a file contains a syntax error that prevents parsing, the text-based rules still produce useful diagnostics.

The alternative, i.e. requiring all rules to operate on the AST, would have made simple formatting checks unnecessarily dependent on successful parsing and would have complicated rule implementation for checks that are naturally expressed as text patterns.

Figure 3.2 provides a side-by-side comparison of the two tiers.

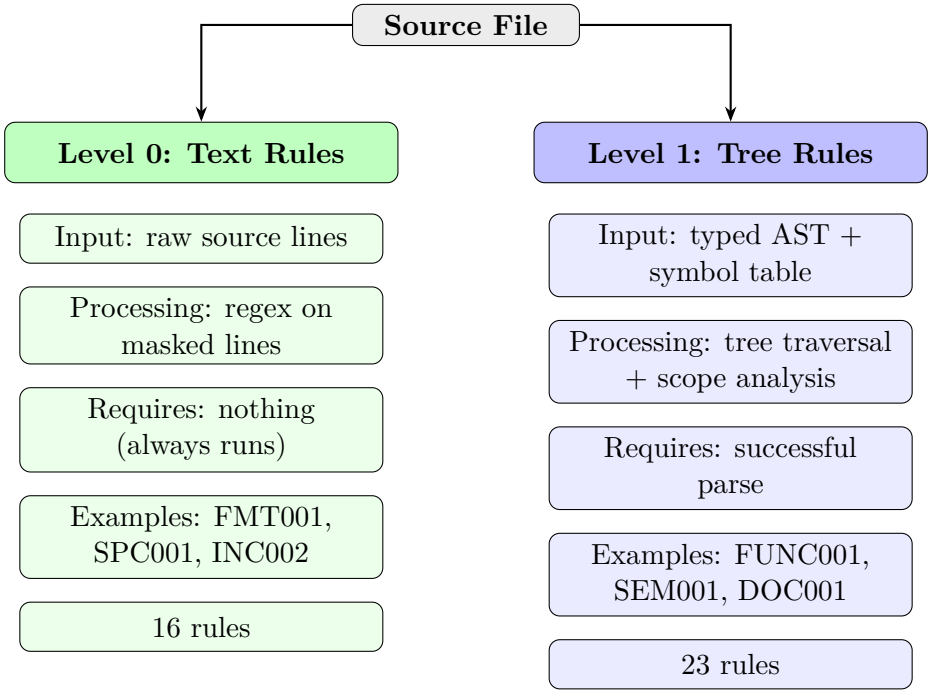

Figure 3.2: Comparison of the two analysis tiers in CAPLint. Level 0 rules are simpler but always applicable; Level 1 rules are more powerful but require successful parsing.

3.2.3 Hand-Written Grammar

The CAPL grammar was developed by hand, without access to a formal language specification. The process was empirical:

1. Start with a minimal grammar covering the most common constructs (functions, event handlers, basic statements).
2. Attempt to parse files from the production corpus.
3. When a parse failure occurs, examine the failing construct, determine its grammatical structure from context, and extend the grammar.
4. Repeat until all files in the corpus parse successfully.

This approach required several weeks of iterative refinement and resulted in a 686-line Lark grammar that achieves a 100% parse success rate on the full production corpus. Figure 3.3 illustrates this iterative process.

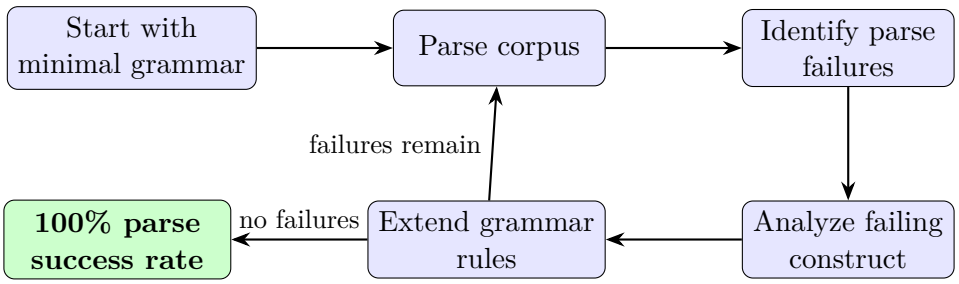

Figure 3.3: The iterative grammar development cycle. The grammar was refined until all corpus files parsed successfully.

The grammar uses Lark's LALR parser with a contextual lexer [7]. The contextual lexer is important for CAPL because certain tokens (such as `message` and `timer`) serve as both keywords in event handler declarations and as type names in variable declarations. A contextual lexer resolves these ambiguities by adjusting its tokenization rules based on the parser's current state.

3.2.4 Immutable AST Nodes

AST nodes are implemented as Python frozen dataclasses (Listing 3.1). Freezing ensures that nodes cannot be accidentally mutated during analysis, preventing a class of bugs where one rule's analysis inadvertently affects another.

```
@dataclass(frozen=True)
class FunctionDecl:
    name: str
    return_type: str
    params: tuple[VarDecl, ...]
    body: tuple[Statement, ...]
    pos: SourcePos
```

Listing 3.1: Example AST node definition using a frozen dataclass.

An optional production-mode invariant (activated by setting the environment variable `CAPLINT_ENV=production`) verifies at runtime that no Lark parse tree objects leak into the AST, ensuring a clean separation between the parser's concrete output and the analysis-facing data structures.

3.2.5 Masked Line Analysis

Text-based rules operate on "masked" lines where string literals and comments are replaced with whitespace. This is necessary because many text rules use pattern matching that would produce false positives if it matched content inside strings or comments. For example, a rule checking for tabs would incorrectly flag a string containing a literal tab character.
The masking process preserves column positions: each replaced character is substituted with a space, so that column numbers reported in diagnostics correspond to positions in the original source. This is achieved through a two-pass algorithm that separately handles block comments (which may span multiple lines) and string literals.

3.2.6 Configuration and Suppression

CAPLint supports three layers of configuration, applied in order of increasing specificity:

1. **File-based configuration:** A `caplint.toml` or `pyproject.toml` file, located by searching upward from the linted file. Supports `select` (whitelist) and `ignore` (blacklist) arrays of rule IDs.
2. **CLI overrides:** The `-select` and `-ignore` flags override file-based configuration.

3. **Inline suppression:** Source comments in the form `// caplint: disable=RULE1,RULE2` suppress specific rules on a single line, and `// caplint: disable-file=RULE1` suppresses rules for the entire file.

This hierarchical approach mirrors the configuration model of ESLint and Pylint, allowing teams to define project-wide defaults while permitting file-level and line-level exceptions where justified.

3.3 Technology Stack

CAPLint is implemented in Python 3.12, chosen for its strong ecosystem of parsing libraries, rapid development speed, and accessibility to the target audience (engineers who may extend the tool with custom rules). Table 3.1 lists the key dependencies.

Table 3.1: Key dependencies of CAPLint.

Library	Version	Purpose
Lark	1.1.9+	LALR parser generator with contextual lexer
Rich	14.3.2+	Terminal output formatting and colored diagnostics
tomli	1.2.0+	TOML configuration file parsing (Python < 3.11)
pytest	8.0.0+	Test framework

3.4 Evaluation Method

The evaluation of CAPLint was conducted along two dimensions:

3.4.1 Corpus Validation

The full production corpus, drawn from large in-production test projects at Volvo Cars, was used for validation. Validation focused on:

- **Parse success rate:** Whether the parser can successfully parse all files without errors.
- **False positive analysis:** Manual review of diagnostics produced on corpus files to identify rules that generate an unacceptable rate of false positives. Rules exceeding a threshold were either refined or removed (see Section 4.3.3).

3.4.2 Developer Survey

A formative, anonymous Microsoft Forms survey was distributed in week 9 of the project to the HIL development team and adjacent teams that interact with CAPL code regularly. A group of engineers responded. The survey collected data on:

- The respondent's primary relation to CAPL (write, review, both, indirect) and years of experience
- Time spent per week debugging CAPL issues
- Where linter-solvable issues are mostly discovered (review, runtime, both)

- The three categories of issues the respondent would most like a linter to catch
- Free-text descriptions of common bad practices and bugs the respondent wished a linter had caught

The survey was designed to check that we were targeting the right rule categories, not to measure productivity impact in a controlled way. The sample size is small, the results are treated qualitatively and used to complement the corpus validation. Selected aggregate responses most relevant to CAPLint's design and evaluation are reported in Section 4.9.

4

Results

This chapter presents the implemented system, its architecture and capabilities, the rule catalog, and the results of the evaluation.

4.1 System Overview

CAPLint is a command-line static analysis tool for CAPL that processes source files through a multi-stage pipeline. Figure 4.1 illustrates the high-level architecture.

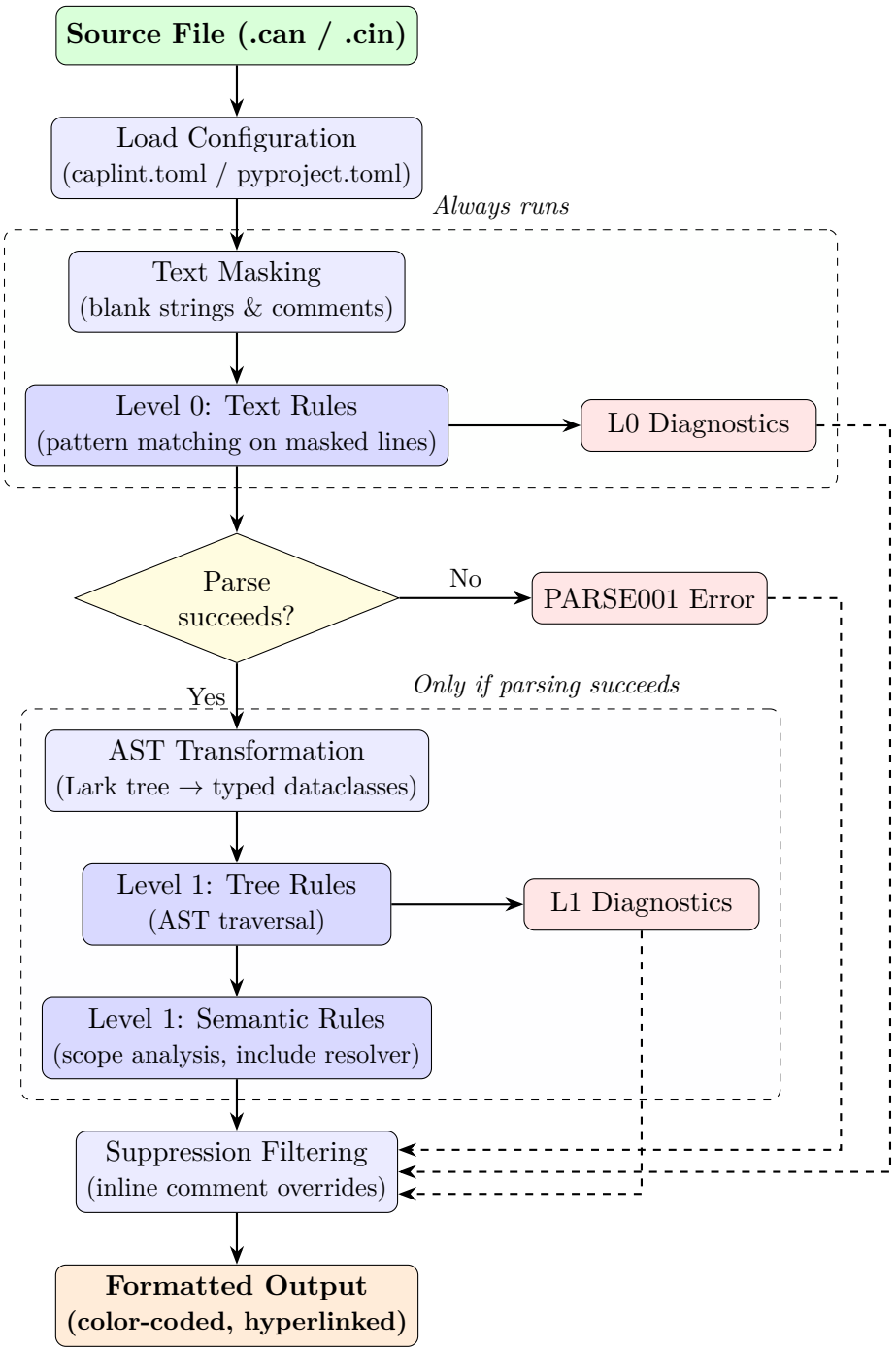

Figure 4.1: The CAPLint analysis pipeline showing the two-tier architecture. Level 0 text rules always run; Level 1 tree and semantic rules run only when parsing succeeds. All diagnostics pass through suppression filtering before output.

The pipeline processes each file in the following stages:

1. **File discovery:** The CLI recursively discovers `.can` and `.cin` files in the specified path.
2. **Configuration loading:** The tool searches upward from each file for a `caplint.toml` or `pyproject.toml` configuration file to determine which rules are active.
3. **Text masking:** Source lines are masked (strings and comments replaced with

whitespace) for safe pattern matching.
4. **L0 rule execution:** All enabled text-based rules are applied to the masked lines. This stage always runs, regardless of whether parsing succeeds.
5. **Parsing:** The Lark LALR parser attempts to parse the source file. If parsing fails, a `PARSE001` diagnostic is emitted and tree-based rules are skipped.
6. **AST transformation:** The raw Lark parse tree is transformed into a typed AST using frozen Python dataclasses.
7. **L1 rule execution:** Tree-based rules traverse the AST to detect structural and naming violations.
8. **Semantic analysis:** The scope analyzer builds symbol tables, and semantic rules check for undeclared and unused variables.
9. **Suppression filtering:** Diagnostics suppressed by inline comments are removed.
10. **Output:** Remaining diagnostics are formatted and displayed with file paths, line numbers, and clickable hyperlinks.

4.2 Grammar and Parser

The CAPL grammar developed for CAPLint consists of 686 lines of Lark grammar rules. It covers the following language constructs:

- **Top-level structures:** includes blocks, global variables blocks, function declarations, and event handler declarations.
- **Type system:** Built-in types (`int`, `long`, `byte`, `float`, `double`, `char`, `word`, `dword`, `msTimer`, `timer`, `signal`, `envvar`, `sysvar`, `diagRequest`, and `diagResponse`), `struct` and `enum` declarations, message types, and array types.
- **Event handlers:** `on message`, `on signal`, `on signal_update`, `on sysvar`, `on sysvar_update`, `on sysVar_change`, `on timer`, `on key`, `on preStart`, `on start`, `on stopMeasurement`, `on ethernet`, `on PDU`, and `on FRFrame`.
- **Statements:** `if/else`, `switch/case`, `for`, `while`, `do-while`, assignments with all compound operators, `return`, `break`, `continue`.
- **Expressions:** Full C-like operator precedence including ternary, bitwise, logical, comparison, arithmetic, and unary operators. Function calls, member access, array indexing, and type casts.
- **Preprocessor directives:** `#include`, `#define`, `#ifdef`/`#ifndef`/`#endif`, treated as opaque tokens.

4.2.1 Parse Success Rate

The grammar was validated against the full production corpus at Volvo Cars. It parsed every file successfully: a 100% parse success rate, with zero PARSE001 failures.

4.2.2 Corpus Composition

Not all of the corpus is subject to linting. A large portion consists of vendor-supplied libraries and auto-generated code (for example, output from the CANoe

code generator and Vector library modules). These files follow their own conventions, which differ from the ones CAPLint enforces, so linting them would produce many un-actionable diagnostics. They are therefore excluded via glob patterns in the project configuration file, leaving an in-scope subset of hand-written test code. The exclusion list is conservative: it removes only directories that carry an unambiguous vendor or auto-generation marker, and any team adopting CAPLint should review and extend it to match its own repository layout.

4.2.3 AST Transformation

The parser produces a raw Lark parse tree, which is then transformed into a typed domain AST using frozen Python dataclasses. Figure 4.2 illustrates this transformation for a simple CAPL expression.

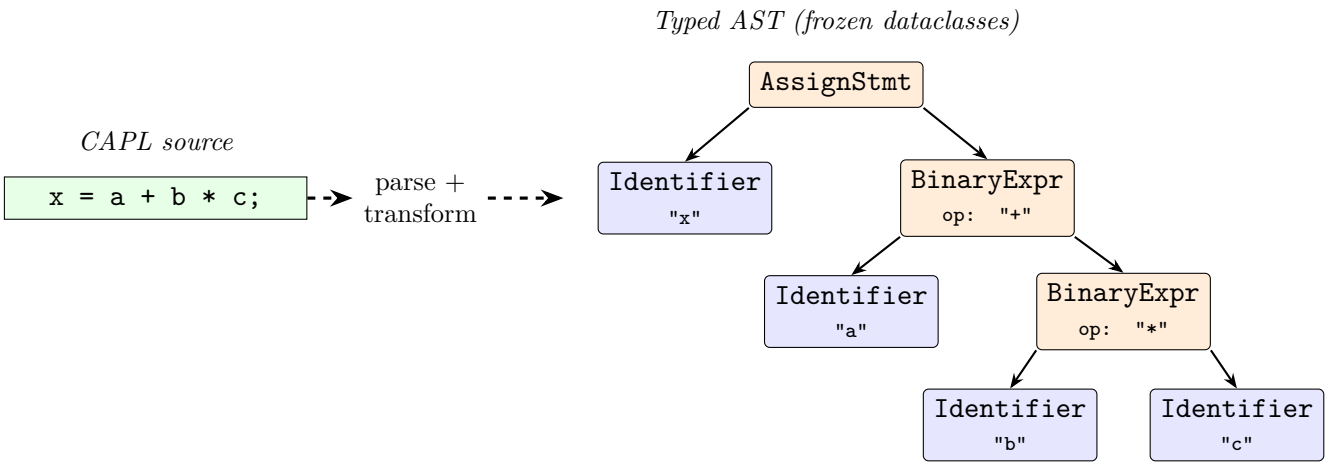

Figure 4.2: Transformation from CAPL source code to a typed abstract syntax tree. The AST preserves operator precedence (multiplication binds tighter than addition) without requiring parenthesis nodes.

4.3 Rule Catalog

CAPLint implements 39 lint rules organized into fifteen categories. Tables 4.1 and 4.2 provide the complete catalog.

4.3.1 Level 0: Text Rules

Table 4.1: Level 0 (text-based) rules implemented in CAPLint.

Rule ID	Category	Description
FMT001	Formatting	No tab characters; use spaces for indentation
FMT003	Formatting	No trailing whitespace
FMT004	Formatting	Avoid multiple consecutive blank lines inside blocks
FMT005	Formatting	File must end with a newline
SPC001	Spacing	No space before opening parenthesis in function calls
SPC002	Spacing	No spaces immediately inside parentheses
SPC003	Spacing	Exactly one space after commas
INC002	Includes	Includes must be in an `includes {}` block at the top of file
INC003	Includes	No duplicate include directives
INC005	Includes	Only `.cin` files can be included
LAY001	Layout	Two blank lines required between top-level blocks
COM001	Comments	Prefer line comments (`//`) over block comments (`/* */`)
FILE001	Files	File names must use `snake_case`
FILE002	Files	Folder names must use `snake_case`
API001	API	Avoid the `Wait()` function (blocks event processing)
SWI001	Switches	Case body must start on a new line

4.3.2 Level 1: Tree Rules

Table 4.2: Level 1 (tree-based) rules implemented in CAPLint.

Rule ID	Category	Description
FUNC001	Functions	Exported function names must use `camelCase`
FUNC002	Functions	Non-exported function names must use `PascalCase`
FUNC003	Functions	Export keyword must match naming: `camelCase` requires `export`, `PascalCase` forbids it
FUNC004	Functions	`if`/`else` body must use block form (braces)
FUNC005	Functions	Avoid output parameters; prefer return values
FUNC007	Functions	Function body must not exceed 40 lines
FUNC008	Functions	Status-returning function must return 0 (0 = OK convention)
VAR001	Variables	Variable names must use `camelCase`
VAR003	Variables	Avoid global variables where possible
DOC001	Document.	Functions must have doc comments
DOC002	Document.	Function doc comments must use `/** ... */` format
DOC004	Document.	Docstrings should include an example section

Rule ID	Category	Description
ENU001	Enums	Enum type names must use `PascalCase`
ENU002	Enums	Enum members must use `UPPER_SNAKE_CASE`
STR001	Structs	Struct names must use `PascalCase`
STR002	Structs	Struct members must use `snake_case`
SWI002	Switches	Switch statements must include a `default` case
SWI003	Switches	Each case must end with `break` or `return`
TEST001	Testcases	Testcase functions must not contain `return` statements
SEM001	Semantic	No undeclared variables (skipped for `.cin` files)
SEM002	Semantic	No unused local variables
SEM003	Semantic	No unused global variables (`.can` files only; `.cin` globals are exported)
INC004	Includes	Only include files whose exports are actually used

4.3.3 Removed Rules

Seven rules were implemented during development but subsequently removed based on corpus validation:

- **FMT006** (missing semicolons): Produced 487 false positives and zero true positives on the corpus. The parser already reports missing semicolons as parse errors, making this rule redundant.
- **FMT002** (4-space indentation): Superseded by FMT001 (no tabs) combined with editor configuration.
- **LAY002** (single blank line inside blocks): Superseded by FMT004 (max consecutive blank lines).
- **DOC003, DOC005** (docstring style details): Overly prescriptive heuristics that produced more noise than useful diagnostics.
- **INC001** (includes only in `.can` files): Removed because `.cin`-to-`.cin` chaining is a legitimate pattern.
- **FUNC006** (allow output parameter for string returns): Superseded by FUNC005, which excludes `char[]` parameters entirely. The original intent of FUNC006 was to permit `char[]` as an output parameter for string-return patterns; once FUNC005 was designed to exclude `char[]` from the start, FUNC006 became unnecessary.

The removal of these rules demonstrates the value of corpus validation: rules that seemed reasonable in theory produced unacceptable false positive rates when applied to real-world code.

4.4 Naming Convention Summary

A central contribution of CAPLint is the codification of naming conventions that were previously documented internally. Table 4.3 summarizes the enforced conventions.

Table 4.3: Naming conventions enforced by CAPLint.

Construct	Convention
Exported functions	`camelCase`
Private functions	`PascalCase`
Variables and parameters	`camelCase`
Enum type names	`PascalCase`
Enum members	`UPPER_SNAKE_CASE`
Struct names	`PascalCase`
Struct members	`snake_case`
File names	`snake_case`
Folder names	`snake_case`

4.5 Cross-File Analysis

CAPLint's include resolver enables cross-file analysis, specifically the INC004 rule that detects unused includes. The resolver works by:

1. Scanning included `.cin` files with a fast character-level scanner (avoiding the overhead of full LALR parsing) to extract exported names (functions, variables, types).

2. Building a mapping from each include directive to the set of names it contributes.

3. Checking whether any of those names are actually used in the including file.

4. Reporting includes whose exported names are never referenced.

Figure 4.3 illustrates this process.

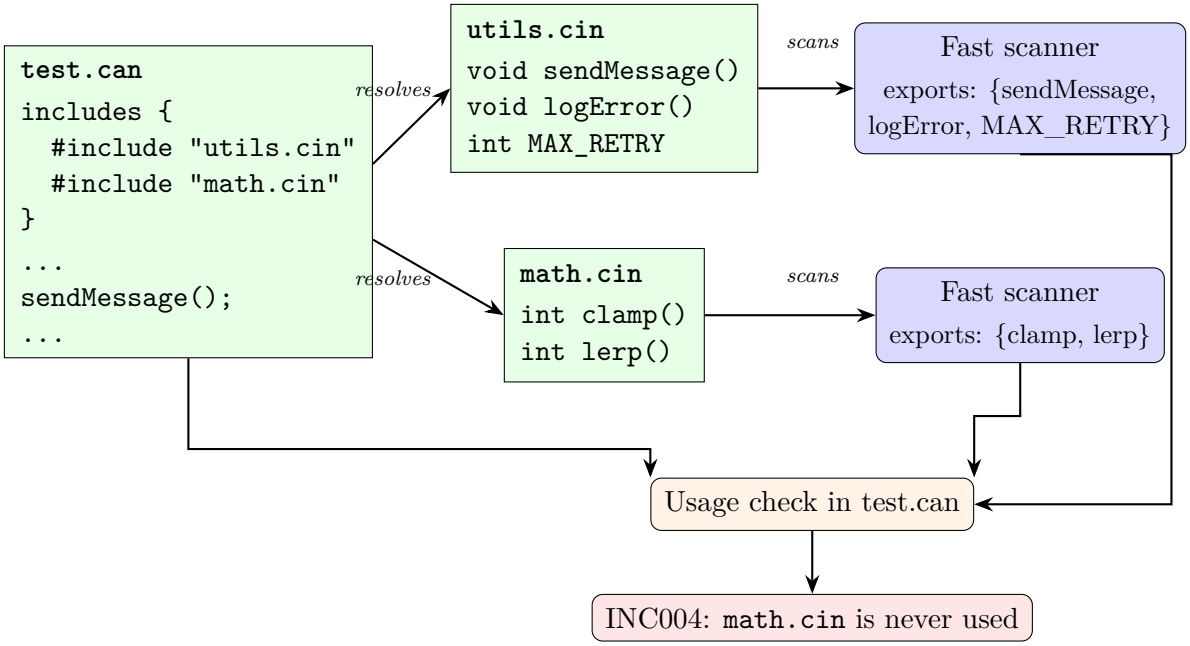

Figure 4.3: The include resolver process. Each included `.cin` file is scanned for exported names (column 3), which are checked against actual usage in the source file. Includes with no used exports trigger INC004.

This approach is deliberately conservative: if the resolver cannot determine what a file exports (e.g., if it contains only preprocessor directives), the include is assumed to be needed. This design avoids false positives at the cost of potential false negatives.

4.6 Diagnostic Output

CAPLint produces diagnostics in a compact, developer-friendly format. Each diagnostic includes the file path, line and column numbers, rule ID, and a descriptive message. The output uses color coding (via the Rich library) to distinguish severity levels and generates clickable OSC-8 hyperlinks compatible with modern terminals (iTerm2, VS Code integrated terminal), allowing developers to jump directly to the flagged line.
An example output for a file with several violations:

```
$ python -m caplint test_engine.can

Found 4 diagnostics

test_engine.can
10:1   warning  DOC001  Missing doc comment immediately above function.
10:24  info     SPC001  Unexpected space before '(' after 'checkEngine'.
11:7   warning  SEM002  'tempVal' is declared but never used.
12:2   warning  FMT005  File must end with a newline.

Linted 1 file(s) in 0.36s
```

Figure 4.4: Example diagnostic output produced by CAPLint.

4.7 Testing Results

The test suite consists of 397 test functions across 26 test modules, supported by 176 fixture files. Table 4.4 summarizes the test coverage.

Table 4.4: Test suite summary.

Metric	Value
Test modules	26
Test functions	397
Fixture files	176
Corpus files tested	Full corpus
Parse success rate	100%

Each lint rule has at least one dedicated fixture containing both violating and non-violating code. The fixture annotation format `// EXPECT: RULE@LINE` enables automated verification that the correct diagnostics are produced at the correct locations.

4.8 Corpus Validation and False Positive Analysis

Running CAPLint with all rules active on the in-scope subset surfaced a substantial number of mechanically detectable inconsistencies and improvement opportunities across the corpus, confirming that the rule set engages with real code rather than firing only on synthetic inputs. On a developer workstation running Python 3.12, CAPLint processes files at roughly tens of milliseconds each, making it suitable for both interactive use and CI/CD pipelines.
Iterative development against the corpus revealed several bugs and design issues that were fixed during the project. Notable examples are: SEM001 missed enum member names because the scope builder did not iterate `EnumDecl.members` (fixed); TEST001 fired on implicit-void functions (fixed by requiring the `TESTCASE_KW` token); INC001 wrongly flagged `.cin`-to-`.cin` chained includes (rule removed); and INC004 flagged event-handler-only includes (fixed by detecting event handlers in scanned files). This shows why testing rules on real code was necessary: some rules that looked reasonable in isolation did not work well in practice.

4.8.1 Remaining Known Limitations

How false positives were identified. The false positive figures reported in this subsection (and in particular the SEM001 precision estimate below) are the result of manual tagging performed by the authors of this thesis. There was no external annotation team and no pre-existing labelled benchmark for CAPL. The procedure was as follows. CAPLint was run with all rules active on the in-scope subset of the corpus. For the three rules where false positives clustered (INC004, SEM001, SEM003) we read each diagnostic together with the surrounding source code and, where needed, the file's include graph, and classified it as either a true positive

(the rule's premise actually holds in the code) or a false positive (the diagnostic is a consequence of information that is unavailable to a CAPL-only analyzer, such as identifiers defined in `.dbc` files, declarations behind absolute-path includes, or preprocessor-defined symbols). For SEM001, where the volume of warnings was too large to label every diagnostic individually, the classification proceeded cluster by cluster: each cluster was identified from a recurring pattern in the source (such as "unresolved identifier comes from an absolute-path include") and the share of warnings belonging to that cluster was estimated. The same authors also tagged the rules during development, so the labelling is not blind; the limitations of this self-labelling are revisited in Section 5.4, and the future-work item in Chapter 6 proposes a follow-up external labelling study that would convert these estimates into properly measured precision and recall figures per rule.
After iterative refinement, three recurring sources of false positives remain. None of the three is a bug in rule code; each is a consequence of a structural feature of CAPL or its toolchain that no text-based analyzer can resolve on its own. Table 4.5 summarizes these sources and their impact on the in-scope corpus.

Table 4.5: Known limitations and remaining false positives on the in-scope corpus.

Rule	Relative size	Limitation
INC004	largest	A shared test-helper library's boilerplate includes contribute names only via preprocessor `#define` constants, which CAPLint's fast scanner does not interpret. The cluster is concentrated in a small set of files and a single configuration entry removes it.
SEM001	moderate	Absolute-path includes (`C:\...`) are unresolvable; exported names from those files are invisible to the scope analyzer.
SEM001	small	CAN message and signal names defined in external `.dbc` CAN-database files are exposed by CANoe as CAPL identifiers but are not visible to a CAPL-only analyzer.
SEM003	a few	CANoe reads and writes global variables via operator panels and Symbol Explorer; those external accesses are invisible to static analysis.

For SEM001, the effective precision on the in-scope corpus is approximately 52%: roughly half of its warnings come from the two structural clusters above, and the remainder are genuine identifier-resolution failures (typos, references to renamed variables, references to functions moved between files). Adding configurable include search paths or a `known_names` list (Chapter 6) would push precision sharply higher without sacrificing recall.
SEM003 was deliberately separated from SEM002 so that teams can suppress it independently (via `--ignore SEM003` or `// caplint: disable-file=SEM003`) without losing unused local variable detection.

4.9 Developer Survey Results

An anonymous Microsoft Forms survey was distributed mid-project to the HIL development team and adjacent teams at Volvo Cars. A small group of engineers responded. The survey was designed as a formative check that the rule categories CAPLint targets are the ones engineers actually care about; it is not a statistical study, and the small sample size is acknowledged throughout. Two findings stand out.

Where linter-solvable issues are discovered. Of the substantive responses to Q5 ("where are linter-solvable issues mostly discovered"), 40% said mostly during code review, 25% said mostly during runtime, and 35% said about equal. In other words, issues that a linter could catch earlier are currently found later, either during human review or during runtime testing on the HIL rig.

What engineers want a linter to catch. Q6 asked respondents to choose the three categories of issue they would most like a linter to catch. Message and signal handling issues were selected most often, closely followed by formatting and style consistency and then unreachable or dead code. Uninitialized variables and timer misuse came next, while naming convention violations and unused variables or includes were selected least often. No respondent chose the open "other" option.

These priorities align with what CAPLint delivers in two of the three top categories, only partially in the third, and not yet at all in the rest. Two categories are well covered: *formatting and style consistency* is the largest group by far (eleven rules across FMT, SPC, LAY, COM and FILE, producing the largest share of diagnostics on the corpus), and *naming convention violations* comes next (six rules across VAR, FUNC, ENU and STR, producing the next-largest share). *Unreachable or dead code* is only partially covered: three structural rules (SWI002 for missing `default`, SWI003 for missing `break`/`return` in case arms, and FUNC008 for status-return conventions) catch only a very small number of diagnostics, two orders of magnitude fewer than the other two categories. Reach-based dead-code detection (unreachable statements after `return`, unreachable branches) would require control-flow analysis and is out of scope for this prototype. Three further categories ranked by engineers (uninitialized variables, timer misuse, and message and signal handling) are not yet covered by CAPLint at all. The partial and uncovered categories are identified as primary future-work items in Chapter 6.

These results should be interpreted with caution due to the small sample size. They are used qualitatively, to validate the prioritization of rule categories, and not as a basis for population-level claims about productivity impact.

4.10 Codebase Metrics

Table 4.6 summarizes the size and composition of the CAPLint codebase.

Table 4.6: CAPLint codebase metrics.

Component	**Lines of Code**
Grammar (Lark)	686
Parser and transformer	1,513
AST node definitions	510
Analysis (scope, includes, builtins)	1,009
L0 text rules	∼1,310
L1 tree rules	∼1,720
CLI, config, diagnostics	∼590
Total source code	∼7,340

5

Discussion

This chapter discusses the results in relation to the research questions, analyzes the design trade-offs made during development, identifies threats to validity, and reflects on the broader implications of the work.

5.1 Addressing the Research Questions

5.1.1 RQ1: How can a static analysis tool be designed and implemented for a domain-specific language that lacks a publicly available grammar specification and existing tooling support?

As stated in Section 1.3, RQ1 is treated as a software-engineering goal for one specific dialect of CAPL within the time available for a bachelor thesis, not as a general method claim about DSL tooling. The discussion below reports what worked in that bounded setting.

The development of CAPLint shows that it is feasible to build a comprehensive static analysis tool for a domain-specific language that has no formal grammar specification, provided that a sufficiently large corpus of real-world source code is available for grammar development and validation.

The key enabler was the empirical grammar development process described in Section 3.2.3. Rather than working from a specification, the grammar was induced from the corpus through an iterative cycle of parsing, failure analysis, and grammar extension. This approach is inherently bottom-up: it discovers the language as used in practice, rather than as defined in theory. The resulting grammar covers exactly the constructs present in the production codebase, which is both a strength (no effort wasted on unused constructs) and a limitation (constructs not present in the corpus are not supported).

The two-tier architecture (Section 3.2.2) proved essential to making the tool practical. By separating text-based rules from tree-based rules, CAPLint delivers value even on files that contain parse errors. This graceful degradation is important in an automotive context where test scripts may be in various stages of development.

Several design principles from mainstream linters (ESLint, Pylint) transferred well to the CAPL domain:

- Rule modularity (each rule is an independent unit with its own tests).
- Hierarchical configuration (project-level defaults with file-level overrides).

- Inline suppression (allowing developers to acknowledge and document exceptions).

However, CAPL's event-driven execution model required domain-specific adaptations. For example, the distinction between exported and private functions (enforced by FUNC001 and FUNC002) reflects a CAPL convention without a direct analogue in C-style linters. Similarly, the `.cin` fragment file handling required special treatment in both the parser and the semantic analyzer.

Short answer to RQ1. For this corpus and this timeframe, the workable recipe is an empirical grammar developed against the corpus, a two-tier architecture that still produces useful diagnostics when parsing fails, and the established linter design patterns of modular rules, hierarchical configuration, and inline suppression. The grammar covers the production codebase; the rest is open in the sense that CAPL constructs not present in the corpus may not be supported.

5.1.2 RQ2: In what ways can automated static analysis reduce the time spent on manual code review of CAPL scripts in an automotive HIL testing environment?

The survey results (Section 4.9) indicate that debugging CAPL issues consumes a meaningful share of respondents' weekly effort, and that many of these issues are of a kind that automated static analysis could plausibly detect (Q5). CAPLint's 39 rules cover the major categories of violations engineers reported as common: naming conventions, formatting, documentation, and semantic issues such as unused variables.

The corpus validation demonstrates that these rules can be applied at scale: CAPLint produces many diagnostics across the in-scope corpus in well under a minute. The rule categories engineers ranked highest in Q6 (formatting, naming, dead code) account for roughly two-thirds of all diagnostics, with the layout rule alone responsible for the single largest share of mechanically resolvable findings. The seven rules that produced unacceptable false-positive rates or redundancy were identified and removed during development (Section 4.3.3); the remaining rules produce actionable diagnostics on real-world code.

A controlled before/after study of review time was not conducted (see Section 5.4). The CI pilot itself was completed in the final iteration of the project, but it has not yet been running long enough for a meaningful number of pull requests to flow through it. A precise figure of the form "CAPLint saves N hours per engineer per week" would require running the pilot for a longer period and comparing review time on pull requests before and after CAPLint became active. This measurement is the most important follow-up and is identified as primary future work in Chapter 6.

A more cautious conclusion is that the survey results, the corpus measurement, and the mapping between survey priorities and implemented rule coverage all suggest that CAPLint could reduce the number of mechanical issues reviewers need to handle manually, even before more demanding rules (data-flow-based uninitialized-variable detection, message-and-signal-handling semantics) are added.

When reviewers no longer need to check naming conventions and formatting manually, they can redirect that attention to verifying test logic, assessing coverage, and

identifying design issues that require human judgment.

Short answer to RQ2. CAPLint reduces manual review effort in three concrete ways: by automating high-volume mechanical findings (formatting, spacing, layout, file and folder naming), by enforcing naming conventions for functions, variables, enums and structs that are otherwise checked by hand, and by surfacing scope-level issues (unused locals, unused globals in `.can` files, unused includes) that previously had to be noticed during review or runtime testing on the rig. A precise hours-per-engineer-per-week figure is not claimed; the CI pilot needed to measure that is identified as the primary follow-up in Chapter 6.

5.1.3 RQ3: What are the structural limits of static analysis for an event-driven proprietary DSL?

Three classes of structural limit emerged during the project. None of them is a bug in rule code; each comes from a structural property of CAPL or its toolchain that no purely text-based or AST-based analyzer can fully resolve on its own. The detailed evidence is in Section 5.2 and Section 5.4; the answer to RQ3 consolidates them here.

External symbol environments. CAPL receives part of its symbol environment from outside the source. `.dbc` CAN-database files expose message and signal names as if they were ordinary CAPL identifiers, and some files use directives that the resolver cannot follow. The result is the SEM001 cluster reported in Section 4.8: roughly 50% of all SEM001 hits are not bugs in user code but information that CAPLint cannot currently see. More generally, when a language depends on external files for names and symbols, a static analyzer must either read those files too or accept some false positives.

Opacity of preprocessor exports. CAPL's preprocessor directives (`#define`, `#undef`) are treated as opaque tokens by the grammar and do not enter the parsed AST. A `.cin` file whose only exports are preprocessor constants is therefore invisible to the include scanner used by INC004. Any DSL whose surface syntax distinguishes a preprocessor layer from a parsed layer forces the analyzer to choose between missed violations and a much more complex implementation that interprets the preprocessor.

Event-driven entry points. A C program has one entry point and a well-posed notion of reachability. A CAPL program has many entry points (one per event handler), most of them short. Reachability analysis must therefore treat every event handler as live, which weakens any conclusions the analyzer can draw about which functions or branches are unused. CAPLint's rules that touch reachability (SWI002, SWI003, and indirectly FUNC007) are deliberately syntactic rather than reach-based, because a reach-based version would require control-flow analysis beyond the scope of this prototype.

The three limits are not independent. An analyzer that ingests `.dbc` files still faces the preprocessor limit; one that interprets the preprocessor still faces the entry-point limit. The general finding is that static analysis for an event-driven proprietary DSL with external symbol environments has a richer structure of limits than analysis for a single-file, single-entry-point, fully-resolved language. Honest reporting of those

limits, with quantified per-cluster impact and per-cluster mitigations, is part of what makes such a tool trustworthy.

Short answer to RQ3. The structural limits encountered in this work fall into three classes: external symbol environments (notably `.dbc` CAN-database identifiers and absolute-path includes), preprocessor opacity (`#define`-only contributions that never reach the AST), and the event-driven entry-point structure that weakens reachability-based reasoning. Each limit has a documented impact on the corpus and a proposed mitigation; together they delimit what a purely text-and-AST analyzer can claim for a DSL of this kind.

5.2 Design Trade-Offs

Several trade-offs were made during development that merit discussion:

Precision vs. recall in semantic analysis. The scope analyzer (SEM001, SEM002, SEM003) deliberately errs on the side of precision (minimizing false positives) at the expense of recall (potentially missing some true violations). For example, `.cin` files are entirely excluded from undeclared variable checks (SEM001) because their scope is determined by the including file, which may not be available during analysis. Similarly, SEM003 (unused globals) only runs on `.can` files, since `.cin` globals are exported symbols by design.

Corpus validation revealed that SEM001's effective precision is approximately 52%. The remaining false positives fall into two structural causes: roughly 30% of all SEM001 warnings stem from absolute-path includes that the resolver cannot open, and roughly 18% from CAN message and signal names injected by CANoe from `.dbc` database files. Both categories represent architectural limitations of purely static analysis rather than bugs in the rule logic. Configurable include search paths (to resolve absolute paths) and a `known_names` list (for database-sourced identifiers) are identified as future improvements that would raise precision without sacrificing recall.

SEM003 faces a different challenge: CANoe reads and writes CAPL global variables via operator panels and Symbol Explorer, making these external accesses invisible to static analysis. SEM003 was deliberately split from SEM002 so that teams can suppress it independently without losing unused local variable detection.

These design choices reflect a practical priority: in a code review context, false positives erode trust in the tool more quickly than missed violations.

Fast include resolver vs. full parsing. The include resolver uses a character-level scanner rather than the full LALR parser to extract exported names from included files. This is faster, which matters when processing projects with many includes, but cannot detect exports that are defined through preprocessor macros. The resulting false negatives (unreported unused includes) were deemed acceptable because INC004 is a code cleanliness rule, not a correctness check.

Naming convention conflicts. CAPLint enforces a specific set of naming conventions (e.g., `camelCase` for variables, `UPPER_SNAKE_CASE` for enum members). However, Vector's own code and some developer teams use a prefix-based convention (`gMyVar` for globals, `cMyConst` for constants, `cETH3_...` for enum members). These conventions are mutually exclusive, and CAPLint's rules will flag prefix-convention

code. The per-file suppression mechanism (`// caplint: disable-file=VAR001`) provides a workaround, but a future version could support configurable convention patterns.

Documentation style. DOC001 requires a doc comment immediately above each function, and DOC002 requires that this comment use the `/** ... */` documentation-block style. Ordinary line comments and non-documentation block comments are therefore treated as missing documentation rather than accepted doc comments. A future refinement could make the accepted documentation style configurable for teams with different conventions.

Hand-written grammar vs. grammar inference. An alternative approach would have been to use grammar inference techniques to automatically derive a grammar from the corpus. This was considered but rejected: grammar inference for programming languages remains an active research area with limited practical tooling, and the manual approach provided better control over ambiguity resolution and error recovery.

5.3 Comparison with Mainstream Linters

CAPLint's architecture shares fundamental patterns with established linters but differs in its constraints and operating environment.

Table 5.1: Comparison of CAPLint with mainstream linters.

Aspect	CAPLint	ESLint	Pylint
Grammar source	Hand-written	ECMAScript	CPython parser
Parser	Custom (LALR)	Espree / custom	Built-in
Rule count	39	200+	300+
Auto-fix	No	Yes	Limited
IDE integration	No (CLI only)	Yes	Yes
Plugin system	No	Yes	Yes
Cross-file analysis	Include resolver	Module resolution	Import analysis

The comparison highlights both the maturity gap and the fundamental similarity in approach. CAPLint's rule count is smaller because it targets a narrower domain with fewer conventions, not because the architecture cannot support more rules. The absence of auto-fix and IDE integration reflects scope constraints of a 17-week thesis project rather than architectural limitations.

5.4 Threats to Validity

5.4.1 Internal Validity

Grammar completeness. The grammar was developed against a single organization's codebase. CAPL constructs not used at Volvo Cars may not be supported,

potentially causing parse failures on code from other organizations. The 100% parse rate applies only to the evaluated corpus.
Rule correctness. While rules are tested with fixtures and validated against the corpus, the absence of a formal CAPL specification means that some rules may encode conventions specific to the Volvo Cars team rather than universal CAPL best practices.

5.4.2 External Validity

Survey sample size. The small number of respondents to the formative anonymous survey is insufficient for statistically significant conclusions, and there is no random sampling frame. The survey results are reported as indicative and qualitative rather than definitive
Single-organization evaluation. The tool was developed and evaluated in the context of a single team of an automotive company. The coding guidelines, codebase characteristics, and review practices may differ at other organizations using CAPL.
No controlled productivity study. The claim that CAPLint reduces review time is based on the survey data and the tool's capabilities, not on a controlled experiment measuring actual time savings before and after adoption.

5.4.3 Construct Validity

Lines of code as a metric. Codebase size (Table 4.6) is reported in lines of code, which is a crude metric that does not reflect complexity or effort. It is included for context, not as an evaluation measure.
False positive rate varies by rule. The overall false positive rate is not uniform across rules. Formatting and naming rules (FMT, SPC, VAR, FUNC) have near-zero false positive rates on the developer corpus, while semantic rules have higher rates due to environmental limitations. SEM001 achieves approximately 52% precision, primarily because absolute-path includes and database-sourced names are invisible to the linter. INC004 retains a cluster of warnings from a shared test-helper library's boilerplate includes that may or may not be genuinely unused. Reporting an aggregate false positive rate would obscure this variation.
Vendor and auto-generated code exclusion. The decision to exclude the vendor and auto-generated files from linting is pragmatically necessary but means that CAPLint's effectiveness is evaluated only on the in-scope subset. If the exclusion patterns are misconfigured, violations in developer code could be silently skipped, or auto-generated code could produce noise.

5.5 Ethical and Societal Considerations

CAPLint is a tool designed to improve software quality in automotive testing. Reliable automotive software is a matter of public safety: defects in ECU software can affect vehicle behavior. By improving the quality and consistency of CAPL test scripts, CAPLint indirectly contributes to more thorough testing and, by extension, safer vehicles.

The tool does not replace human judgment in code review. It automates mechanical checks, freeing engineers to focus on the aspects of review that require expertise and contextual understanding. This shift represents an augmentation of human capability, not a substitution.
From a sustainability perspective, reducing unnecessary manual work improves the efficiency of the development process. This is consistent with the broader trend toward automation of repetitive tasks in software engineering, which can improve both productivity and job satisfaction by allowing engineers to spend more time on intellectually engaging work.
The tool processes source code locally and does not transmit data to external services. No privacy concerns arise from its use.

5.6 Software Sustainability

This section discusses sustainability in the software-engineering sense (long-term maintainability and extensibility of the tool), rather than environmental sustainability. The project has been designed with long-term sustainability in mind. The modular rule architecture allows new rules to be added without modifying existing components. The hierarchical configuration system enables teams to adapt the tool to their specific conventions. The comprehensive test suite (397 tests, 176 fixtures) provides a safety net for future development.
The choice of Python as the implementation language lowers the barrier for adoption and extension by the target user community, as Python is widely known among software engineers in the automotive industry. The small number of dependencies (four runtime libraries) reduces the maintenance burden and the risk of supply-chain issues.
Beyond the tool itself, automating mechanical review checks has a small but concrete broader effect: catching guideline violations at authoring time rather than during a rig run shortens the feedback loop, which in turn reduces the number of HIL test cycles needed to reach a clean review state. Fewer rig cycles means less compute, less time on the bench, and less engineer attention spent on issues that did not need a human in the loop.

6
Conclusion

This chapter summarizes the contributions of this thesis and suggests directions for future work.

6.1 Summary of Contributions

This thesis presented CAPLint, a static analysis tool for CAPL, that previously had no dedicated linting support. The main contributions are:

1. **A CAPL grammar and parser.** A 686-line Lark LALR grammar was developed from scratch through analysis of production code. The parser achieves a 100% success rate on a large real-world corpus of CAPL files from Volvo Cars, showing that a practical grammar can be built from a corpus even without a formal language specification.
2. **A two-tier analysis architecture.** CAPLint's design separates fast text-based rules (Level 0) from structure-dependent tree-based rules (Level 1). This architecture ensures that useful diagnostics are produced even when files cannot be parsed, and that rules use the simplest analysis tier sufficient for the check.
3. **A comprehensive rule set.** 39 lint rules across fifteen categories translate informal CAPL coding conventions into a machine-checkable specification. Seven additional rules were implemented but removed after corpus validation revealed unacceptable false positive rates or redundancy, showing the value of testing rules on real code.
4. **Cross-file analysis.** The include resolver enables detection of unused includes by scanning included files for exported names without requiring full parsing, balancing analysis depth with performance.
5. **Practical impact evidence.** A formative survey of engineers and 17 weeks of on-site collaboration at Volvo Cars indicate that CAPL debugging consumes a meaningful share of weekly engineering effort, and that a substantial portion of these issues fall within the scope of automated static analysis. CAPLint automates several high-volume, mechanically checkable categories of review findings, indicating potential to reduce repetitive review work.

6.2 Answering the Research Questions

RQ1: *How can a static analysis tool be designed and implemented for a domain-specific language that lacks a publicly available grammar specification and existing*

tooling support?
The answer lies in an empirical, corpus-driven approach. By iteratively developing the grammar against a large body of real-world code, validating continuously, and adopting a two-tier architecture that separates text-based from tree-based analysis, a practical linter can be built even without a formal language specification. Key enablers include the availability of a production corpus, access to domain experts, and the adoption of established linter design patterns (modular rules, hierarchical configuration, inline suppression) from mainstream languages.
RQ2: *In what ways can automated static analysis reduce the time spent on manual code review of CAPL scripts in an automotive HIL testing environment?*
CAPLint's 39 rules cover several guideline categories engineers reported checking manually: naming conventions, formatting, documentation, and semantic issues such as unused locals, unused `.can`-level globals and unused includes. The reduction is qualitative rather than measured in hours: a controlled before/after study was not conducted, but the survey, the corpus measurement (lots of improvement opportunities on the in-scope corpus), and the mapping between survey priorities and implemented rule coverage all indicate that CAPLint automates the categories of finding that engineers currently identify by hand. A precise time-saving figure requires running the CI pilot for long enough to accumulate before/after pull-request data and is identified as primary future work.
RQ3: *What are the structural limits of static analysis for an event-driven proprietary DSL?*
Three classes of structural limit were identified and quantified on the in-scope corpus: external symbol environments (`.dbc` CAN-database files and absolute-path includes that the resolver cannot follow); preprocessor opacity (`#define`-only includes are invisible to the post-parse AST); and the event-driven entry-point structure that weakens reachability and dead-code analysis. Each limit has a documented impact on the corpus (Section 4.8) and a proposed mitigation. These limits are not unique to CAPL; similar DSLs would face similar problems. Reporting them clearly helps make the evaluation trustworthy.

6.3 Future Work

Several directions for future development are identified:

- **IDE integration:** Developing a VS Code extension or Language Server Protocol (LSP) implementation would provide real-time feedback as developers write code, catching violations at authoring time rather than review time.
- **Auto-fix capabilities:** Many rules (particularly formatting and naming convention rules) have deterministic fixes that could be applied automatically, further reducing developer effort.
- **Controlled review-time measurement on the CI pilot.** A CI pilot was wired into a Volvo Cars build pipeline in the final iteration of the project, but it has not yet processed enough pull requests for a meaningful measurement. A structured before/after comparison of pull-request review time is the highest-value follow-up and the measurement that would justify a precise answer to RQ2.

- **Deeper CI/CD integration:** Beyond running CAPLint in the build, deeper integration (e.g., posting diagnostics as PR comments, blocking merges on violations) would strengthen automated enforcement.
- **External false-positive labelling study:** Several survey respondents pre-committed to helping label diagnostics. A small follow-up labelling study would compute true-positive and false-positive rates per rule and give CAPLint an empirical signal-to-noise figure.
- **Additional semantic rules:** The scope analysis infrastructure supports rules beyond unused/undeclared variable detection, such as detecting unreachable code after return statements, identifying functions with excessive complexity, and type-aware analysis.
- **Configurable naming conventions:** Supporting multiple naming convention patterns (e.g., allowing `gMyVar` prefix convention alongside `camelCase`) would reduce false positives for teams that follow Vector's naming conventions rather than the conventions currently enforced by CAPLint.
- **Plugin system:** A plugin architecture would allow teams to implement organization-specific rules without modifying the core codebase, improving extensibility and adoption across different automotive teams.
- **Cross-organization validation:** Validating CAPLint against CAPL codebases from other automotive companies would test the generalizability of the grammar and rule set beyond the Volvo Cars context.

Bibliography

[1] Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E. and Winter, C. (2015) 'Tricorder: Building a Program Analysis Ecosystem', in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pp. 598–608. Available at: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43322.pdf (Accessed: 8 May 2026).

[2] Charette, R.N. (2009) 'This Car Runs on Code', *IEEE Spectrum*. Available at: https://spectrum.ieee.org/this-car-runs-on-code (Accessed: 15 May 2026).

[3] LLVM Project (n.d.) *Clang-Tidy: Extra Clang Tools Documentation*. Available at: https://clang.llvm.org/extra/clang-tidy/ (Accessed: 11 May 2026).

[4] Cousot, P. and Cousot, R. (1977) 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints', *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 238–252. Available at: https://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml (Accessed: 15 May 2026).

[5] ESLint (n.d.) *ESLint: Pluggable JavaScript Linter*. Available at: https://eslint.org/ (Accessed: 5 May 2026).

[6] Johnson, S. C. (1978) *Lint, a C Program Checker*. Available at: https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf (Accessed: 10 May 2026).

[7] Shinan, E. (2017) *Lark: a Parsing Toolkit for Python*. Available at: https://github.com/lark-parser/lark (Accessed: 15 May 2026).

[8] National Instruments (n.d.) *Hardware-in-the-Loop Testing for Transportation*. Available at: https://www.ni.com/en/solutions/transportation/hardware-in-the-loop.html (Accessed: 14 May 2026).

[9] Pylint contributors (n.d.) *Pylint: Static Code Analyser for Python*. Available at: https://pylint.readthedocs.io/ (Accessed: 15 May 2026).

[10] Python Software Foundation (2024) *dataclasses: Data Classes*. Python 3 Documentation. Available at: https://docs.python.org/3/library/dataclasses.html (Accessed: 1 May 2026).

[11] Vector Informatik GmbH (n.d.) *CAPL: Programming Language for CANoe and CANalyzer*. Available at: https://www.vector.com/int/en/know-how/capl/ (Accessed: 13 May 2026).

A

Complete Rule Reference

This appendix provides the complete reference of all lint rules implemented in CAPLint, including their rule IDs, categories, analysis tier, and descriptions.

Level 0: Text-Based Rules

ID	Category	Description
FMT001	Formatting	No tab characters; use spaces for indentation
FMT003	Formatting	No trailing whitespace at end of lines
FMT004	Formatting	Avoid multiple consecutive blank lines inside blocks
FMT005	Formatting	File must end with a newline character
SPC001	Spacing	No space before opening parenthesis in function calls
SPC002	Spacing	No spaces immediately inside parentheses
SPC003	Spacing	Exactly one space after commas
INC002	Includes	Includes must be in an `includes {}` block at the top of file
INC003	Includes	No duplicate include directives
INC005	Includes	Only `.cin` files can be included
LAY001	Layout	Two blank lines required between top-level blocks
COM001	Comments	Prefer line comments (`//`) over block comments (`/* */`)
FILE001	Files	File names must use `snake_case`
FILE002	Files	Folder names must use `snake_case`
API001	API	Avoid the `Wait()` function (blocks event processing)
SWI001	Switches	Case body must start on a new line

Level 1: Tree-Based Rules

ID	Category	Description
FUNC001	Functions	Exported function names must use `camelCase`
FUNC002	Functions	Non-exported function names must use `PascalCase`
FUNC003	Functions	Export keyword must match naming: `camelCase` requires `export`, `PascalCase` forbids it
FUNC004	Functions	`if/else` body must use block form (braces)
FUNC005	Functions	Avoid output parameters; prefer return values
FUNC007	Functions	Function body must not exceed 40 lines
FUNC008	Functions	Status-returning function must return 0 (0 = OK convention)
VAR001	Variables	Variable names must use `camelCase`
VAR003	Variables	Avoid global variables where possible
DOC001	Document.	Functions must have doc comments
DOC002	Document.	Function doc comments must use `/** ... */` format
DOC004	Document.	Docstrings should include an example section
ENU001	Enums	Enum type names must use `PascalCase`
ENU002	Enums	Enum member names must use `UPPER_SNAKE_CASE`
STR001	Structs	Struct names must use `PascalCase`
STR002	Structs	Struct member names must use `snake_case`
SWI002	Switches	Switch statements must include a `default` case
SWI003	Switches	Each `case` must end with `break` or `return`
TEST001	Testcases	Testcase functions must not contain `return` statements
SEM001	Semantic	No use of undeclared variables (skipped for `.cin` files)
SEM002	Semantic	No unused local variables
SEM003	Semantic	No unused global variables (`.can` files only; `.cin` globals are exported)
INC004	Includes	Only include files whose exports are actually used

Removed Rules

The following rules were implemented during development but removed after corpus validation revealed unacceptable false positive rates or redundancy with other rules:

ID	Reason for Removal
FMT002	4-space indentation: superseded by FMT001 combined with editor configuration
FMT006	Missing semicolons: 487 false positives, zero true positives on corpus; redundant with parse errors
LAY002	Single blank line inside blocks: superseded by FMT004
DOC003	Docstring style: overly prescriptive heuristic
DOC005	Docstring section format: overly prescriptive heuristic
INC001	Includes only in `.can` files: removed because `.cin`-to-`.cin` chaining is legitimate
FUNC006	Allow output parameter for string returns: superseded by FUNC005, which excludes `char[]` parameters entirely

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY