TU WIEN Informatics

Dokumentationsgenerator für die Grammatik Sprache Coco/R

Mithilfe von Doxygen als Basis

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

David He
Matrikelnummer 12215880

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof.i.R. Dipl.-Ing. Dr.techn. Andreas Krall

Dokumentationsgenerator für die Grammatik Sprache Coco/R © 2026 David He ist lizenziert unter der Creative Commons Namensnennung 4.0 International Lizenz (CC BY 4.0). Eine Kopie dieser Lizenz finden Sie unter https://creativecommons.org/licenses/by/4.0/

Wien, 23. Jänner 2026

David He

Andreas Krall

TU WIEN Informatics

Documentation Generator for the Grammar Language Coco/R

Using Doxygen as the Foundation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

David He
Registration Number 12215880

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.-Prof.i.R. Dipl.-Ing. Dr.techn. Andreas Krall

Documentation Generator for the Grammar Language Coco/R © 2026 by David He is licensed under CC BY 4.0. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/

Vienna, January 23, 2026

David He	Andreas Krall

Erklärung zur Verfassung der Arbeit

David He

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 23. Jänner 2026

David He

Danksagung

Ich möchte mich an dieser Stelle besonders bei meinem Betreuer, Professor Andreas Krall, bedanken, der mich im Laufe des Projektes stehts unterstützt hat und sich immer die Zeit genommen hat, Unklarheiten sowie auftretende Probleme zu identifizieren und mir diese klar kommuniziert hat. Durch seine kontinuierliche Betreuung und sein zeitnahes Feedback, konnte das Projekt ohne Komplikationen erfolgreich abgeschlossen werden. Darüber hinaus möchte ich mich bei Benedikt Lukas Huber und Johannes Zottele bedanken, die bei technischen Fragestellungen und Problemen stets rasch und hilfsbereit agiert haben, wodurch das Projekt ohne Verzögerung umgesetzt werden konnte.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Andreas Krall, for his continuous support throughout the course of this project. He always took the time to clarify open questions and point out complications and errors that arose during the planning and implementation of this project. His timely feedback and continuous guidance were instrumental in completing this project successfully without major setbacks. I would also like to express my gratitude to Benedikt Lukas Huber and Johannes Zottele for their quick support when technical difficulties and questions arose during the project, which significantly reduced the amounts of delays throughout the implementation.

Kurzfassung

Dokumentation ist ein wesentlicher Bestandteil moderner Softwareentwicklung, da sie das Verständnis, die Wartbarkeit und die Weiterentwicklung von Softwareprojekten und Softwareanwendungen erhöht. Die Erstellung und Wartug solcher Dokumentationen ist jedoch oft mit enormen Zeitaufwand und vielen inhaltlichen Fehler verbunden, insbesondere bei großen und komplexen Softwareanwendungen. Dies gilt in besonderem Maße ebenfalls für formale Grammatiken, welche in Compiler-Generatoren eingesetzt werden, da mit zunehmenden Umfang die Grammatik rasch unübersichtlich, schwer navigierbar und schlecht lesbar wird.

Im Rahmen dieser Arbeit wurde ein Dokumentationsgenerator entwickelt, der die bestehende Open-Source-Software Doxygen erweitert und speziell auf die Grammatik des Compiler-Generators Coco/R ausgerichtet ist. Der Generator analysiert Coco/R-Grammatiken, extrahiert deren strukturelle Elemente, sowie die zugehörigen Kommentare, gruppiert und formatiert diese und erzeugt daraus eine konsistente, lesbare und gut navigierbare Dokumentation. Die Ausgabe der Dokumentation erfolgt im nativen Doxygen-Verfahren und umfasst sowohl eine strukturierte LaTeX-/PDF-Dokumentation als auch eine interaktive HTML-Dokumentation. Der entwickelte Ansatz erleichtert somit die Erstellung und Wartbarkeit von Dokumentation für Coco/R-Grammatiken.

Abstract

Documentation is an essential component of modern software development, as it significantly increases the comprehensibility, maintainability and further development of software projects and software in general. However, the creation and maintenance of such documentation is often time-consuming and error-prone, particularly in the case of big complex software systems. This general problem also applies to formal grammars used in compiler generators, which tend to become difficult to comprehend, navigate and hard to read as their size and complexity increase.

In the course of this thesis, a documentation generator was developed that extends the existing Open-Source-Software Doxygen and is specifically tailored to grammars of the Coco/R compiler generator. The generator analyzes Coco/R grammars, extracts their structural elements along with associated comments, which will be grouped together and formatted appropriately to produce a consistent, readable and well-navigable documentation. The output is generated using Doxygen's native mechanisms, which includes both a structured LaTeX/PDF documentation and an interactive HTML documentation. The proposed approach simplifies the creation and maintenance of documentation for Coco/R grammars.

Contents

CHAPTER 1

Introduction

Software documentation is one of the most important non-functional requirements of any software project. The reasons for this is evident: good documentation improves the understanding of the codebase, enhances maintainability, and simplifies the usage of the software for reuse or further development. Consequently, numerous software tools have been developed over the years to facilitate the creation of such documentation with minimal effort. These tools often parse the source code and comments that are formatted and placed according to predefined rules in order to generate a structured and consistent documentation. By automating this process, they significantly reduce the manual workload while ensuring uniform documentation across the whole project.

While such documentation tools are well established for mainstream programming languages, comparable solutions for grammar languages, which define the structure of input streams, remain largely absent. For this reason, this thesis aims to address this gap by developing a document generator for the grammar language Coco/R based on Doxygen. The proposed extension enables Coco/R grammars to include Doxygen-style comments, which are subsequently processed by the Doxygen parser to generate a structured documentation. In addition, the generated documentation can serve as a comprehensive reference manual for the language specified by the grammar.

CHAPTER 2

Related Work

Automating documentation is not a new topic, yet it is regularly revisited due to its ongoing relevance. Creating and maintaining a documentation for a software is both time-intensive and often error-prone. Therefore, over the past decade, numerous studies have proposed different methods for automatically summarizing code. These approaches include extracting keywords from the source code and applying vector space models to determine their individual importance for the summary [Hai+10], as well as generating natural language summaries of methods [MM14].

In recent years, with the rapid advancement of artificial intelligence technologies like ChatGPT, Gemini and DeepSeek, there has been growing interest in leveraging AI for automatic code documentation. A few examples include using a GPT-3 based model pretrained on both natural language and programming language [KU23], as well as using a retrieval based model that relies on large collections of predefined methods and retrieves them using similarity-based ranking techniques [Zho+23].

Despite continuous research efforts, the underlying problem remains unsolved due to the diversity of programming languages and variations in developers programming styles. This is even more pronounced for grammar languages, which receive comparatively little attention. Consequently, solutions for automatic grammar documentation are nearly non-existent for the same reasons that complicate automatic code documentation.

CHAPTER 3

Core Concepts

Before addressing the technical implementation of the proposed document generator, it is essential to introduce the fundamental concepts upon which this work is based. This chapter provides an overview of automated documentation generation, introduces Doxygen as the underlying documentation framework, and outlines the core ideas behind Coco/R grammars as the target of the documentation process. These concepts form the foundation for the design and implementation decisions discussed in the subsequent chapters.

3.1 Document Generation

3.1.1 Traditional Code Documentation

Code documentation, in its simplest form, consists of descriptive comments embedded within certain parts of a codebase. Depending on their placement, different interpretations and purposes are associated with these comments. The most common categories are document comments, block comments, and line comments. Document comments are typically used to describe structural entities of the source code like classes, functions, attributes or global fields. In contrast, block comments and line comments are primarily used to explain the logic and functionality of the target code. While block comments are placed adjacent to a section of code, which can be placed before or after the code, line comments are directly associated with a line of code and are typically placed at the end of that line. [FWG07; Zha+24]

Considering the different possible comment placements and levels of descriptive details, creating and maintaining accurate and consistent documentation manually is highly challenging. Nevertheless, most software development projects require documentation. Although creating and maintaining documentation demands considerable time and effort, the benefits in terms of improved maintainability, enhanced knowledge transfer, and

long-term sustainability outweigh the associated costs. In order to reuse and further develop a software effectively, comprehensive documentation is essential. Developers are often required to read substantial amounts of code to understand its structure and behavior. Without good documentation the time required to comprehend a codebase increases significantly, especially with a large and complex codebase. [FWG07]

Furthermore, documentation of poor quality, that are characterized by redundancy, insufficient or inadequate descriptions and language ambiguities, may even negatively impact the developers understanding of the code. [Agh+20]

For these reasons, many researchers over the years revisit this topic to analyze the existing methodologies to identify their shortcomings [RBG22; Zha+24] and propose improvements. Examples include examining documentation practices from a practitioners perspective [Agh+20] and prioritizing documentation efforts based on dependency structures in order to lessen the overall effort required [Liu+21].

3.1.2 Modern Automatic Code Documentation

To reduce the repetitive workload and substantial human effort required for documentation, numerous automatic code documentation tools, such as JavaDoc, Doxygen and Pydoc, have been developed. These tools extract structured comments embedded in the source code and generate a comprehensive documentation for the entire codebase. Compared to fully manual documentation, this approach represents a significant improvement. However, there are several limitations that remain, as the embedded code comments themselves are still written manually. Consequently, with the rise of deep learning and generative artificial intelligence, automatic code documentation has once again gained renewed attention. The objective is to leverage modern deep learning techniques to automatically generate meaningful code summaries. Nevertheless, automatic source code summarization remains an open research challenge for the future. [Zha+24]

3.2 Doxygen

3.2.1 Basic Concept

Many programming languages support embedded documentation mechanisms that allow developers to add structured comments to entities of the code such as functions, variables and classes. Such documentation is highly beneficial for understanding the functionality and intent behind individual code segments. However, these documentations fragments are typically distributed throughout the whole codebase without a higher-level structure, making it difficult to obtain a comprehensive overview of the project. Effective documentation should not only describe individual elements but also provide contextual information, including architectural decisions, abstractions, inheritance structures and design choices. Developers should be able to navigate the documentation efficiently and locate related information without excessive effort.

To extract the relevant information from the source code and generate a structured documentation, a separate tool is required. One widely used open source tool for this purpose is Doxygen [Hee]. It was initially developed for the programming language C++, but it has evolved significantly over the years and now also supports a variety of programming languages, including Python and VHDL.

Doxygen analyzes the provided source code, identifies syntactic entities such as classes, functions, and variables, and associates them with adjacent documentation comments. These comments must follow a specific structure defined by Doxygen. An example is shown in Listing 3.1. After organizing every extracted element, Doxygen generates a structured documentation of the source code in HTML- and/or LaTeX-format. Additionally, it provides a configuration file, which can be adapted for a given software project. It allows for a high degree of customization for the documentation process and output.

A major advantage of Doxygen is its ability to resolve global references automatically. This enables cross-linking between documented entities, which in turn allows the user to navigate it with ease. Such functionality is particularly beneficial for grammar languages, as it allows references on the right-hand side of a production to link directly to the corresponding non-terminal definition. Even though Doxygen does not natively support any source code processing regarding grammar languages, its modular design permits the integration of additional *language parsers*. For this reason, Doxygen was selected as the foundation of this thesis, which will be discussed more extensively in the following chapter.

Listing 3.1: Doxygen Comment

```
/*!
 * @brief This is a brief documentation to provide a quick overview
 *
 * A detailed description for more indepth context
 *
 * @param a: some documentation for the parameter
 * @return documentation regarding the return of the function
 */
int example(char* a) {
    return 0;
}
```

3.2.2 Flex

Basics

Internally, Doxygen is primarily implemented in C++ and uses fast lexical analyzer generators (Flex) for its scanners. In this context, a scanner performs lexical analysis on an input stream by identifying patterns and converting the input into a sequence of tokens, which are then processed by subsequent parsing stages.

Flex uses regular expressions to define token patterns. When a pattern matches a sequence of input characters, the corresponding C action code is executed and a token is returned. The matching process follows the maximum munch principle, meaning that the longest possible match is preferred. In case multiple patterns match the same longest sequence, the rule that was defined first in the specification is selected.

While executing action codes, Flex provides access to the matched input string via the global variable *yytext* and *yyleng*. These variables allow the developer to process the recognized lexeme within the associated action.

Listing 3.2 illustrates a minimal Flex example. In this program, a sentence such as "I love eating 20 Chicken Nuggets." is analyzed sequentially. Numerical values (e.g. "20") produce a token *TOK_NUMBER*, while words (e.g. "I", "love", "eating", "Chicken", "Nuggets") produce *TOK_STRING*. Whitespace and line breaks are ignored, and unknown characters trigger an error message.

Listing 3.2: Simple Flex Example

```
%{
#include <stdio.h>
%}

%%
[0-9]+          { return TOK_NUMBER; }
[a-zA-Z]+       { return TOK_STRING; }
[ \t\n]+        ;
.               { printf("UNKNOWN: %s\n", yytext); }
%%
int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Less Common Flex Constructs

Although the fundamental principles of Flex are straightforward, it provides several advanced features that are particularly useful in more complex scanning scenarios or are only useful in very niche situations.

The most important of these features is the concept of *start conditions*, commonly also referred to as states. States are defined at the beginning of a Flex specification and allow rules to be activated only under specific conditions. A rule can be restricted to a state by prefixing it with <STATE>. This mechanism enables more precise control over lexical analysis and simplifies the handling of context-sensitive constructs.

Listing 3.3 demonstrates the usefulness of states when analyzing function definitions. When parsing a function such as *add*, the scanner must decide whether to generate tokens for parameters *a* and *b* while preserving the overall function structure and ignoring inline comments of each parameter. Such situations are significantly easier to handle using flex states.

Listing 3.3: Example function

```
1
2  int add(int a, /**<  A number */
3          int b, /**<  A second number */
4          ) {
5      return a + b;
6  }
```

A more elaborate example is shown in Listing 3.4, which implements a simple calculator supporting addition and subtraction with comment handling. The scanner initially starts in the *Calculation* state. In this state, arithmetic tokens are generated, while whitespaces and inline comments are ignored. When the sequence "/*" is encountered, the scanner switches to a dedicated comment state to handle the comment block. In this state all inputs are discarded until the closing sequence "*/" is detected, which also includes the start end closing sequence, at which point the scanner switches back to the *Calculation* state.

The <*> state is a special state that applies to all start conditions. Patterns and actions defined under this state are active across all states, which is useful for handling fallback cases. It is recommended to use it at the end due to the principle of maximum munch.

Listing 3.4: Flex as a State Machine

```
1  %{
2  #include <stdio.h>
3  #include "example.tab.h"
4  #define YY_DECL int yylex()
5  %}
6
7  %x Calculation
8  %x Comment
9
10 %%
11
12 <Calculation>{
13     "//".*\n         ;
14     [ \t\n\r]+       ;
15     "/*"             { BEGIN(Comment); }
16     [0-9]+           {
17                        yylval.number = atoi(yytext);
18                        return TOK_NUMBER;
19                      }
```

```
20      "+"             { return TOK_PLUS; }
21      "-"             { return TOK_SUBTRACT; }
22      "="             { return TOK_EQUALS; }
23  }
24
25  <Comment>{
26      .               ;
27      "*/"            { BEGIN(Calculation); }
28  }
29
30  <*>.                ;
31
32  %%
33  int yywrap(void) {
34      return 1;
35  }
36
37  int main(void) {
38      BEGIN(Calculation);
39      yylex();
40      return 0;
41  }
```

Besides the states in Flex, the following advanced constructs were used:

1. **Lookahead ("foo/bar")** - The slash operator within the regular expression acts as trailing context. In the given example the pattern foo/bar matches "foo" only if it is immediately followed by "bar", without consuming "bar". This construct is rarely needed, but can be very useful in specific lexical disambiguation scenarios.

2. **REJECT;** - Forces Flex to skip the chosen match and consider the next best alternative. Since this mechanism significantly reduces performance, it is rarely used in practice.

3. **%option <opt>** - The option directive allows configuration of the scanner at specification time. Common options include "reentrant", "prefix", "extra-type" and "debug". These options are particularly relevant in large and complex projects, when integrating scanners into complex systems such as Doxygen.

All scanners in Doxygen rely heavily on the state machine construct provided by Flex, as it offers fine-grained control over the the lexical analysis. For consistency and future maintainability, the Cocor/R scanner developed in this thesis follows the same design principles.

3.3 Coco/R

Coco/R is a compiler generator developed as a research project by Hanspeter Mössenböck, Markus Löberbauer, Albrecht Wöß at the University of Linz, Austria [Han]. It generates a scanner as well as a recursive descent parser from an attributed grammar describing a source language. A detailed explanation of its internal generation process is beyond the scope of this thesis. For the purposes of this project, the relevant aspect is the structural organization of a Coco/R grammar specification.

Listing 3.5: Coco/R Overview

```
1 Cocol =
2  [Imports]
3  "COMPILER" ident
4  [GlobalFieldsAndMethods]
5  ScannerSpecification
6  ParserSpecification
7  "END" ident '.'
8  .
```

Listing 3.5 outlines the overall structure of a Coco/R grammar, which is also referred to as *Cocol*. The grammar definitions in this section are described using Extended Backus–Naur Form (EBNF), meaning that before the keyword "COMPILER" an optional imports section may appear, containing language specific import directives from C#, C++ or Java. The identifier following the keyword "COMPILER" specifies the grammar name. After this declaration optional global fields and methods may be defined in the target language of the generated parser. The grammar body then consists of a *ScannerSpecification* and a *ParserSpecification* with the keyword "END" followed by the exact same identifier as the grammar name and a dot marking the end of the grammar definition.

Listing 3.6: ScannerSpecification Overview

```
1 ScannerSpecification =
2  ["IGNORECASE"]
3  ["CHARACTERS" {SetDecl}]
4  ["TOKENS" {TokenDecl}]
5  ["PRAGMAS" {PragmaDecl}]
6  {CommentDecl}
7  {WhiteSpaceDecl}.
```

The structure of the ScannerSpecification is shown in Listing 3.6. It consists of several optional sections defining lexical elements of the language. These include character sets, tokens, pragmas, comment structures and whitespace declarations. While ignorecase and comment and whitespace declarations describe lexical conventions character, token and pragma define symbols that may be referenced later on within the *ParserSpecificaiton*.

To illustrate each of these components, the following subsections provide formal definitions as well as simple examples.

1. **Characters** - This section allows the definition of characters sets. Sets may be constructed from basic sets that are combined using set union (+) or set difference (-) operations (Listing 3.7). In Coco/R, the predefined keyword *ANY* represents the set of all valid Unicode characters supported by the implementation. In modern versions with UTF-8 support, this corresponds to the full Unicode code point range. Character sets may reference previously defined sets.

Listing 3.7: Character Declaration Structure

```
SetDecl = ident '=' Set '.'.
Set = BasicSet {('+'|'-') BasicSet}.
BasicSet = string | ident | char [".." char] | "ANY".
```

Listing 3.8: Character Declaration Example

```
digit       = "0123456789".
hexDigit    = digit + "ABCDEF".
noDigit     = ANY - digit.
```

2. **Tokens** - The main section of the scanner specification, in which the lexical tokens of the language are declared. Tokens can be categorized into *Literals* and *Token Classes*, where *Literals* represents fixed strings such as keywords(*while*,*return*) or symbols/operators (=, >=) and *Token Classes* describe patterns using EBNF expressions. Notably, token declarations do not have to be LL(1). The formal structure is given in Listing 3.9 while examples are shown in Listing 3.10. If multiple token declarations match the same input (e.g. "while" and ident), literal tokens must be declared after the token class declarations.

   The keyword *CONTEXT* introduces context-dependant token. This allows the scanner to differentiate between inputs like "1.23" and "1..2". After scanning "1." the scanner may defer the decision between a floating-point literal and the range operator. If a second dot follows, the context is pushed back into the input stream and a number token will be returned.

   If all token expressions are omitted, Coco/R assumes a handwritten scanner, and no scanner code is generated.

Listing 3.9: Token Declaration Structure

```
TokenDecl = Symbol ['=' TokenExpr '.'].
TokenExpr = TokenTerm {'|' TokenTerm}.
TokenTerm = TokenFactor {TokenFactor} ["CONTEXT" '(' TokenExpr ')'].
TokenFactor = Symbol
 | '(' TokenExpr ')'
 | '[' TokenExpr ']'
 | '{' TokenExpr '}'.
Symbol = ident | string | char.
```

Listing 3.10: Token Declaration Example

```
ident       = letter {letter | digit | '_'}.
number      = digit {digit}
                | digit {digit} CONTEXT ("..").
float       = digit {digit} '.'
                {digit} ['E' ['+'|'-'] digit {digit}].
while       = "while".
geq         = ">=".
```

3. **Pragmas** - This section introduces special tokens that may appear anywhere in the input stream. Since handling these tokens explicitly in the grammar would be cumbersome, they are declared separately. Pragmas may optionally include a semantic action which consists of target-language code, that is executed immediately upon recognition.

Listing 3.11: Pragma Declaration Structure

```
PragmaDecl = TokenDecl [SemAction].
SemAction = "(." ArbitraryStatements ".)".
```

Listing 3.12: Pragma Declaration Example

```
option = '$' {letter}. (.   foreach (char ch in la.val)
                            if (ch == 'A') ...
                            else if (ch == 'B') ...
                            ... .)
```

4. **Comments** - This section specifies comment delimiters. These specifications are fairly straight forward as can be observed in Listing 3.14. The optional keyword *NESTED* allows recursive comment structures.

Listing 3.13: Comment Declaration Structure

```
CommentDecl = "COMMENTS" "FROM" TokenExpr "TO" TokenExpr ["NESTED
    "].
```

Listing 3.14: Comment Declaration Example

```
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO eol
```

5. **Whitespace** - This section defines additional characters to be treated as whitespace by the scanner. Blanks are ignored by default and every further characters that should be ignored must be declared explicitly in this section. Listing 3.16 illustrates an example that explicitly states that tabulators and line breaks should be ignored by the scanner.

Listing 3.15: Whitespace Declaration Structure

```
1 WhiteSpaceDecl = "IGNORE" Set.
```

Listing 3.16: Whitespace Declaration Example

```
1 IGNORE '\t' + '\r' + '\n'
```

Besides the ScannerSpecification, the ParserSpecification forms the main part of the grammar. It consists of a sequence of productions defining the syntactic structure of the language.

Listing 3.17: ParserSpecification Structure

```
1  ParserSpecification = "PRODUCTIONS" {Production}.
2  Production          = ident [FormalAttributes] [LocalDecl] '='
       Expression '.'.
3  Expression          = Term {'|' Term}.
4  Term                = [[Resolver] Factor {Factor}].
5  Factor              = ["WEAK"] Symbol [ActualAttributes]
6                      | '(' Expression ')'
7                      | '[' Expression ']'
8                      | '{' Expression '}'
9                      | "ANY"
10                     | "SYNC"
11                     | SemAction.
12 Symbol              = ident | string | char.
13 SemAction           = "(." ArbitraryStatements ".)".
14 LocalDecl           = SemAction.
15 FormalAttributes    = '<' ArbitraryText '>'.
16 ActualAttributes    = '<' ArbitraryText '>'.
17 Resolver            = "IF" '(' {ANY} ')'.
```

Listing 3.18: Example Productions

```
1 String =
2     letter      (. int len = 1; .)
3     { letter    (. len++; .)
4     }           (. Console.WriteLine("String length = " + n); .)
5     .
```

A production consists of a left-hand side and a right-hand side that is separated by an '=' sign. The left-hand side declares the non-terminal symbol as well as optional formal attributes, which are surrounded in pointed brackets, and optional local declarations. The right-hand side defines the structure of the non-terminal using EBNF expressions. The formal definition of the structure can be seen in Listing 3.17 and Listing 3.18 illustrates the calculation of a strings length.

Productions correspond to parsing methods in the generated recursive descent parser. This means that non-terminals on the right-hand side can be interpreted as method calls. Consequently, attributes function similar to method parameters and may represent input or output values (e.g., out or ref depending on the target language). Since attribute brackets may themselves contain generic type parameters such as List<T>, Coco/R provides the alternative delimiters <. and .> to avoid ambiguity.

With this overview of the Coco/R grammar structure, it becomes clear which syntactic elements must be recognized by the scanner that will be introduced into the Doxygen framework. The extension must identify and retain the semantic structure of grammar components of Scanner- and ParserSpecification to enable meaningful documentation generation.

CHAPTER 4

Implementation

After introducing the foundational concepts of document generators, Doxygen, and the target language Coco/R, this chapter focuses on the implementation of the Coco/R extension within Doxygen.

For clarity, the term *grammar production* will be used throughout the following sections as an umbrella term encompassing all grammar definitions, including *Characters*, *Tokens*, *Pragmas* or *Productions*, unless stated otherwise.

4.1 Extending Doxygen

To add support for Coco/R grammars in Doxygen, it is necessary to consider Doxygen's internal architecture as well as the structural properties of the Coco/R input language. Doxygen processes input files in multiple stages, including configuration handling, language detection, lexical scanning, parsing, and documentation generation. Nevertheless, from an architectural perspective, there are only two primary entry points into the toolchain.

The *Config Parser* is the first entry point. It processes the configuration file commonly referred to as the *Doxyfile*, and extracts parameters that control and customize the documentation generation process. Each project may define their own configuration file or in some cases multiple configuration files may be used, where separate configurations for HTML and LaTeX-Output are defined. For the integration of Coco/R, only minor adjustments were required at this stage. Specifically, the registration of the *.atg* file extension and an additional language-specific configuration flag was introduced. This flag enables filtering of grammar specific constructs that should not appear in the generated documentation and allows customization that is isolated from other supported languages to preserve their functionalities.

The second entry point is the processing of source files. Depending on the file extension, Doxygen selects the appropriate scanner and parser for the *Language Parsing* phase. To support Coco/R, a dedicated language module was implemented and registered within Doxygen's language mapping mechanism. After associating the *.atg* extension with the new custom Coco/R language module, a custom scanner was implemented in the file *cocorscanner.l*. This scanner is responsible for recognizing and extracting the structural grammar elements described in Section 3.3 and mapping them to Doxygen's internal abstract syntax tree (AST).

Since grammar languages differ conceptually from traditional programming languages, an additional semantic layer had to be introduced. New entity types were created to represent grammar productions and related constructs. These definitions are subsequently processed during layout generation and documentation parsing. The details of this integration are discussed more extensively in the following sections.

4.2 Language Parsing - Coco/R

4.2.1 Layout

A key difference between grammar specification languages such as Coco/R and generic programming languages lies in their structural organization. In programming languages such as Java, declarations of functions, classes and variables can typically appear in a rather arbitrarily order, provided that visibility and scope rules are respected. In contrast, Coco/R enforces a strict structural layout.

Character sets, tokens, pragmas, comments and whitespace declarations, and productions must be declared within their respective sections. For example, after entering the *PRAGMAS* section, it is no longer possible to declare character sets or tokens. Similarly, pragmas are confined to the *PRAGMAS* section. This strict ordering requires the scanner to process the grammar sequentially and to maintain knowledge of the current section. Each encountered grammar production must be categorized according to its enclosing section to ensure correct semantic interpretation.

At the same time, this rigid layout simplifies the parsing process. Once a section header has been recognized, only the constructs valid within that section need to be considered. For instance, after the *TOKENS* section, only the pragma section, comment and whitespace declarations, or the production section may follow. All other constructs can be safely ignored. This deterministic structure significantly reduces ambiguity during lexical analysis and categorization.

4.2.2 Categorization and references

By default, Doxygen does not support grammar languages. This means that there are no dedicated entity types in Doxygen's internal AST to represent grammar productions. Without such entity types, grammar constructs could only be approximated as existing

programming-language elements (e.g., functions or variables), which would be semantically incorrect. Furthermore, Doxygen's internal cross-referencing mechanism relies on well-defined entity categories. Simply rendering grammar productions as plain text would prevent proper linking between non-terminals and their occurrences.

To address this limitation, new entity types were introduced to represent the structural components of Coco/R grammar, including character sets, tokens, pragmas, declarations, and productions. In addition to defining these entity types, corresponding processing logic had to be implemented to facilitate these new entity types into Doxygen's indexing, layout generation, and documentation pipeline. Once these entity types were introduced, the scanner implementation became comparatively straightforward. Since Coco/R grammars are strictly divided into sections, the scanner only needs to maintain the current section context and create a new entity of the appropriate type whenever a grammar production is encountered.

Lastly, there was another enhancement introduced in the context of this project, which is the custom Doxygen command `\refFrom`. This command allows a grammar production to reference all other productions in which it appears on the right-hand side. In other words, it provides reverse references for non-terminals. This feature is particularly valuable for grammar languages, as it enables readers to navigate dependencies between productions efficiently. Listing 4.1 shows a small example, which produces the output illustrated in Figure 4.1.

Listing 4.1: Example referencing

```
1 CHARACTERS
2   /** #letter is used by \refFrom. */
3   letter = 'a' .. 'z' + 'A' .. 'Z'.
4   digit  = '0' .. '9'.
5
6 TOKENS
7   identifier = letter { letter | digit | "_" }.
8   number     = digit { digit }.
```

4.2.3 Bloating

Coco/R grammars can become dense and difficult to read, particularly when making extensive use of semantic actions, global declarations, and attributes. While these constructs are essential for understanding the behavior of the generated parser, they are not always relevant when the goal is to understand the pure grammar structure. In Listing 3.18 semantic actions perform tasks mostly unrelated to the syntactic structure itself. From a documentation perspective focused on grammar comprehension, such embedded code introduces significant visual noise and clutter. Therefore, semantic actions are omitted in the generated documentation. This decision improves readability by emphasizing the structural aspects of the grammar rather than the whole scope of its implementation details.

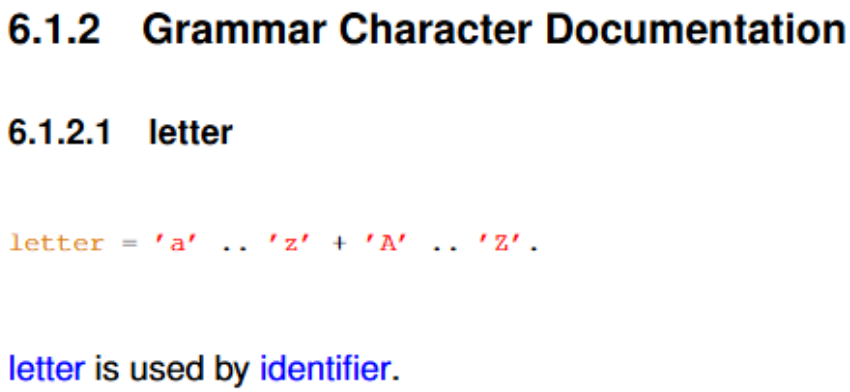


Figure 4.1: Output of the example in Listing 4.1

Additionally, Coco/R requires the use of the alternative delimiters `<.` and `.>` in certain cases to distinguish attribute brackets from generic type parameters. Although this notation is necessary within the grammar source to avoid ambiguity, it is irrelevant to syntactic structure as well and only reduces the readability in the generated documentation. Therefore, in documentation, the dot markers in attribute brackets are removed entirely. While this slightly deviates from the original source syntax, it preserves semantic meaning and significantly improves clarity.

4.3 Text Highlighting

In addition to the *Language Parser*, Doxygen allows the implementation of a separate parser, the so called *Source Parser*. While the *Language Parser* is responsible for structural analysis and entity extraction, the *Source Parser* is used for syntax highlighting within rendered code blocks. During the scanning process of the *Language Parser* code snippets, after the previously described structural bloat is removed, are wrapped inside the `\code{.atg}` tag. This ensures that Doxygen processes these snippets as source code written in the `.atg` language.

Doxygen already includes several built-in *Source Parser*, such as *code.l*, which provide syntax highlighting for supported programming languages. However, these parsers are not compatible with Coco/R-specific keywords and grammar constructs. As a result, grammar sections would either remain unhighlighted or be highlighted incorrectly. To address this limitation, a custom *Source Parser* named *cocorcode.l* was implemented and associated with the `.atg` extension used inside the `\code` tag. This parser identifies Coco/R-specific keywords and uses Doxygen's internal interface functions to emit formatting instructions, such as color and style tags, into the generated output. These tags are then interpreted by the HTML- and LaTeX *OutputGenerators* to produce the desired result of syntax highlighted grammar definitions.

The introduction of a dedicated *Source Parser* enables consistent and accurate syntax

highlighting for Coco/R grammars, thereby significantly improving readability and overall user experience.

CHAPTER 5

Evaluation

After completing the implementation, the next step was to evaluate the performance impact of the Coco/R extension within Doxygen. All performance measurements were conducted on a Kali Linux virtual machine configured with 4 GB of memory and two processors with two cores each (four virtual cores in total). The underlying host CPU was an *AMD Ryzen 5 5600X* with a base clock frequency of 3.7 GHz and 32 MB of L3 cache. For PDF generation, the LaTeX distribution *pdfTeX 3.141592653-2.6-1.40.28 (TeX Live 2025/Debian)* was used.

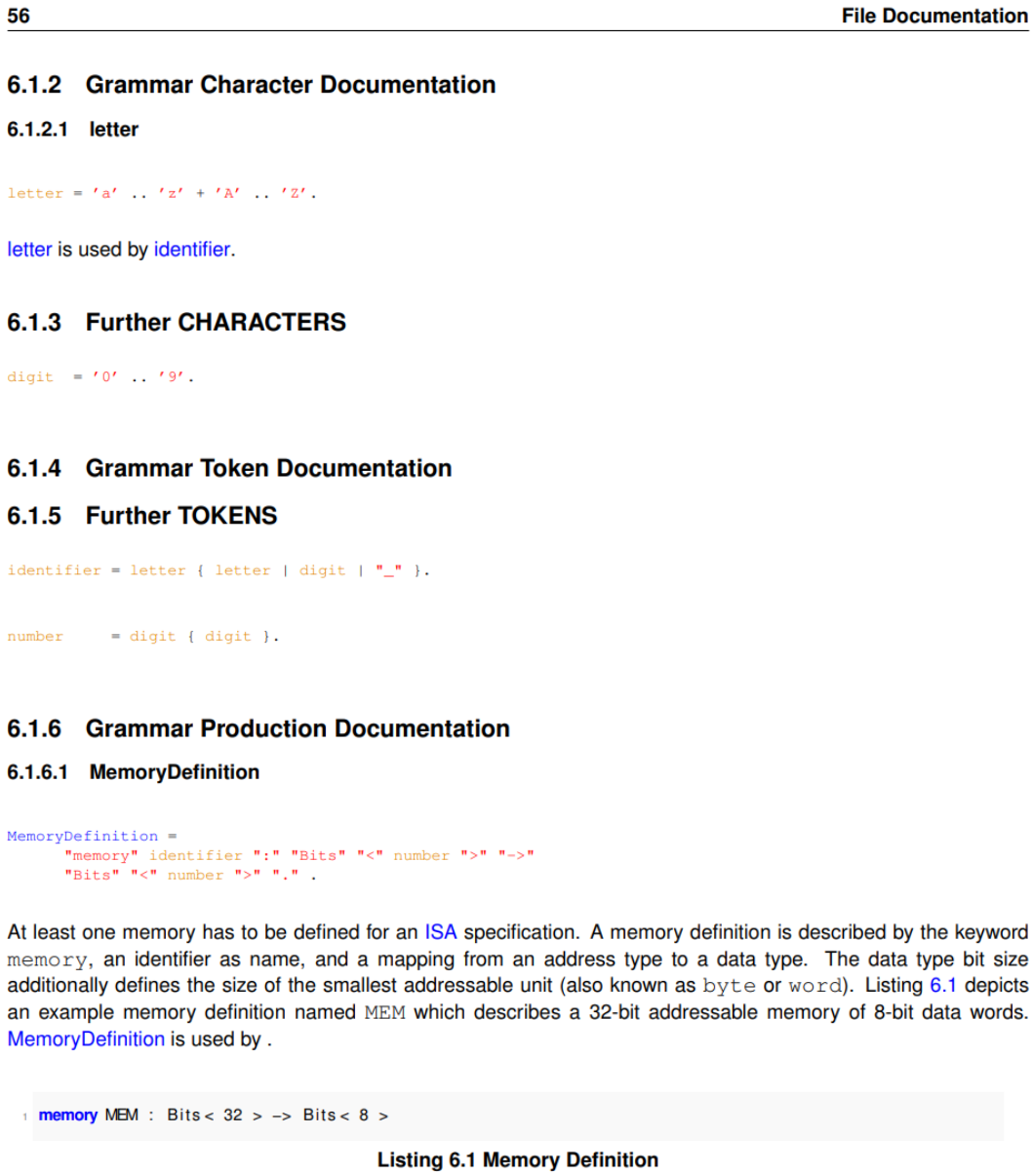
Listing 2 shows a simplified example of a Coco/R grammar used as a baseline test case. As a more realistic scenario, a full grammar provided by the OpenVADL GitHub repository [Opeb] was used. In addition to the grammar file itself, supplementary project files from the same repository [Opea] were included in both tests to simulate a practical usage scenario.

To illustrate the output generated by the extension, Figure 5.1 displays the documentation produced from the simplified grammar shown in Listing 2. The output demonstrates the structural categorization of grammar production as well as correct syntax highlighting and reference resolution within the generated documentation.

Within sections such as *CHARACTERS* and *TOKENS*, the generated documentation is further divided into two subsections: ** Documentation* and *Further **. The former contains grammar productions that include Doxygen's native brief and/or detailed descriptions, while the latter groups all remaining grammar productions of the respective section. This design keeps the documentation concise and compact without introducing separate headers for every undocumented production.

To access the documentation generated from the full OpenVADL grammar, please refer to the OpenVADL project page at `https://openvadl.github.io/openvadl`. The generated documentation can be found under the *Files* section. At the time of writing,

the grammar file is listed as *vadl.ATG*. The exact filename may change in future revisions of the project.



56 File Documentation

6.1.2 Grammar Character Documentation

6.1.2.1 letter

```
letter = 'a' .. 'z' + 'A' .. 'Z'.
```

letter is used by identifier.

6.1.3 Further CHARACTERS

```
digit  = '0' .. '9'.
```

6.1.4 Grammar Token Documentation

6.1.5 Further TOKENS

```
identifier = letter { letter | digit | "_" }.

number     = digit { digit }.
```

6.1.6 Grammar Production Documentation

6.1.6.1 MemoryDefinition

```
MemoryDefinition =
      "memory" identifier ":" "Bits" "<" number ">" "->"
      "Bits" "<" number ">" "." .
```

At least one memory has to be defined for an ISA specification. A memory definition is described by the keyword `memory`, an identifier as name, and a mapping from an address type to a data type. The data type bit size additionally defines the size of the smallest addressable unit (also known as `byte` or `word`). Listing 6.1 depicts an example memory definition named `MEM` which describes a 32-bit addressable memory of 8-bit data words. MemoryDefinition is used by .

```
1 memory MEM : Bits< 32 > -> Bits< 8 >
```

Listing 6.1 Memory Definition

Figure 5.1: A snippet of the PDF generated from the simple grammar example displayed in Listing 2

The performance evaluation focuses on the time required to generate:

1. HTML documentation

2. Latex documentation
3. PDF documentation
4. Both HTML and PDF simultaneously

Execution time was measured using the built-in Bash utility *time*. The following metrics were recorded:

1. **Real** - Real time required for the document generation in seconds
2. **User** - CPU time spent in user space (Doxygen and related processes) in seconds
3. **Sys** - CPU time spent in kernel space (file access, process creation, system calls) in seconds
4. **CPU** - Overall CPU utilization

Each test case was executed five times, and the reported values represent the arithmetic mean.

The results are summarized in Table 5.1.

Test Case	Real (s)	User (s)	Sys (s)	CPU (%)
Simple grammar (LATEX)	0.05	0.04	0.01	94
Simple grammar (PDF)	12.91	12.78	0.12	99
Simple grammar (HTML)	1.35	1.19	0.16	99
Simple grammar (BOTH)	14.17	13.87	0.29	99
Full grammar (LATEX)	0.10	0.08	0.01	96
Full grammar (PDF)	15.76	15.62	0.13	99
Full grammar (HTML)	1.41	1.21	0.19	99
Full grammar (BOTH)	16.97	16.64	0.32	99

Table 5.1: Results of the performance test

The results show that although the full grammar contains approximately 24.59 times more content, the total time required for generating both HTML ad PDF output increases by ony 19.76%. This indicates that document generation does not scale linearly with grammar size. Instead, a substantial portion of the runtime appears to stem from fixed overhead, particularly during the PDF generation, as the time difference between Latex and PDF generation differs greatly. Latex and HTML both require negligible time, while majority of the runtime is consumed by the PDF output, which mostly stems from the Latex compilation process. Additionally, when generating both outputs simultaneously, the runtime is approximately the sum of the individual generation times, which indicates sequential rather than parallel execution. Lastly, the CPU utilization is consistently

around 99%, with Latex generation being the only exception. This indicates that the workload is heavily reliant on the CPU rather than the I/O operations and the processor remains almost fully utilized during the whole document generation process.

Overall, the results demonstrate that the Coco/R extension introduces no measurable performance degradation. The dominant factor in the total runtime lies in the external Latex compilation rather than the grammar parsing introduced in this work.

CHAPTER 6

Conclusion

Documentation generators have become an essential component of modern software engineering, as software systems continue to increase in size and complexity. To remain effective, such tools must evolve and adapt accordingly. This thesis extends the open-source tool Doxygen by introducing support for the grammar language Coco/R. By implementing a dedicated *Language Parser* and *Source Parser*, Doxygen is now capable of processing *.atg* files, analyzing their syntactical structure, extracting structural grammar entities, and generating structured documentation that preserves the logical structure of the grammar. The extension introduces Coco/R specific categorization, reverse referencing via the custom *\refFrom* command, and tailored syntax highlighting. Together, these features enable Coco/R grammars to be documented with the same level of clarity and navigability as conventional programming languages without introducing noticeable overhead as the performance evaluations show.

Overall, this work contributes to improving support for grammar languages in documentation systems and lays the groundwork for further extensions in this area.

Usage and Configuration

In order to use the Coco/R extension for Doxygen, please refer to the OpenVADL GitHub page [Opec]. Two installation methods are available:

1. Clone the OpenVADL Doxygen repository and follow the build instructions provided in the *README.md* file for your operating system.
2. Use the provided Docker image instead of building from source. Setup instructions can be found either in the OpenVADL Doxygen repository or under the *Packages* section of the repository.

After completing one of the installation methods, a `doxygen` executable should be available in your environment. A configuration template can be generated using:

```
doxygen -g <config-name>
```

For optimal results when documenting Coco/R grammars, it is recommended to adjust the configuration parameters as shown in Listing 1.

1. **COCOR_REDUCE_FILE_DOC** - Is the language-specific configuration flag, which is used for filtering of grammar specific constructs that should not appear in the generated documentation
2. **USAGELIST** - Is the flag used for toggling the *\refFrom* command
3. **SHOW_USED_FILES** & **LATEX_HIDE_INDICES** - Both flags are used in tandem to limit the output of the *Files* section to only **.atg* files without including supplementary project files that were used for the generation

All other settings may be modified according to project-specific requirements.

Listing 1: Recommended configuration options

```
1 COCOR_REDUCE_FILE_DOC  = YES
2 USAGELIST              = YES
3 SHOW_USED_FILES        = NO
4 LATEX_HIDE_INDICES     = YES
```

Simple Coco/R Grammar Example

Listing 2: Simple grammar example

```
$package=vadl.ast
import java.util.ArrayDeque;

COMPILER Example

SymbolTable macroTable = new SymbolTable();
Map<String, Identifier> macroOverrides = new HashMap<>();
  /**
  * This list is for our custom generated errors that are richer than
      the one COCO/R produces.
  */
  List<Diagnostic> diagnostics = new ArrayList<>();

  SourceLocation lastTokenLoc() {
    return locationFromToken(this, t);
  }

CHARACTERS
  /** #letter is used by \refFrom. */
  letter = 'a' .. 'z' + 'A' .. 'Z'.
  digit  = '0' .. '9'.

TOKENS
  identifier = letter { letter | digit | "_" }.
  number     = digit { digit }.

COMMENTS FROM "//" TO '\n'
COMMENTS FROM "/*" TO "*/" NESTED

IGNORE '\t' + '\r' + '\n'
IGNORE 'A' .. 'Z' + '\r' + '\n' + ANY

PRODUCTIONS
```

```
33
34  /**
35
36  \lbl{langref_memory}
37
38  At least one memory has to be defined for an \ac{ISA} specification
       . A memory definition is described by the keyword `memory`, an
       identifier as name, and a mapping from an address type to a data
        type. The data type bit size additionally defines the size of
       the smallest addressable unit (also known as `byte` or `word`).
       Listing \r{memory_definition} depicts an example memory
       definition named `MEM` which describes a 32-bit addressable
       memory of 8-bit data words.
39  #MemoryDefinition is used by \refFrom.
40
41  \listing{memory_definition, Memory Definition}
42
43  ~~~{.vadl}
44
45  memory MEM : Bits< 32 > -> Bits< 8 >
46
47  ~~~
48
49  \endlisting
50
51
52  The variable `MEM` is a relation type, mapping an address to data.
       An indexed memory access `MEM(index)` can be used as a read
       access on the right side of an expression and as a write access
       on the left side of an expression. Furthermore, in order to
       access multiple memory elements \ac{VADL} provides a vector call
        syntax which allows to define the amount of memory elements to
       access. Listing \r{memory_access} depicts an example instruction
        `LW` which loads a word (32 bits) from the memory `MEM` by
       indexing the memory with the computed address `addr` and
       specifying to use 4 memory cells which are 8 bit wide through
       the syntax `MEM<4>(addr)`.
53
54
55  \listing{memory_access, Memory Access}
56
57  ~~~{.vadl}
58
59  instruction LW : I_TYPE = {
60    let addr = X(rs1) + ImmediateI in {
61      X(rd) := MEM<4>( addr )
62    }
63  }
64
```

```
  ~~~

  \endlisting


  When accessing memory with a vector of more than one element the
     endianess of the access is important. With a big endian access
     the most significant part of a register is mapped to the lowest
     element of the vector. With a little endian access the least
     significant part of a register is mapped to the lowest element
     of the vector. Processors either have a fixed endianess, the
     endianess can be defined at the processor startup or reset or
     the endianness can be changed dynamically depending on the value
      of a configuration register. \ac{VADL} supports both static and
      dynamic endianess via an annotation to memory. The annotation `
     littleEndian` defines a little endian memory access. The
     annotation `bigEndian` defines a little endian memory access.
     The default vector access is little endian. To dynamically
     define the endianess the endianess annotation is selected by a
     conditional expression. In the listing \r{endianess_annotation}
     the endianess field `bigendian` of the status register `SR` sets
      the big endian access mode.


  \listing{endianess_annotation, Endianess Annotation}

  ~~~{.vadl}

  [if SR.bigendian then bigEndian else littleEndian]

  memory MEM : Bits< 32 > -> Bits< 8 >

  ~~~

  \endlisting

*/
  MemoryDefinition =
      "memory" identifier ":" "Bits" "<" number ">" "->"
      "Bits" "<" number ">" "." .

  /** #Arule is used by \refFrom. */
  Arule = "A" "Rule" | number | ANY.

  /** #vadl is used by \refFrom. */
  vadl = "a" | Arule .
END Example.
```

Overview of Generative AI Tools Used

AI-assisted tools, mainly ChatGPT, were used for grammar correction, improving articulation, removing repeated content, and improving the clarity and readability of individual text passages. The technical content, conceptual design, and implementation of this thesis were developed independently.

Listings

List of Figures

List of Tables

Bibliography

[Agh+20] Emad Aghajani et al. "Software documentation: the practitioners' perspective". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 590–601. ISBN: 9781450371216. DOI: 10.1145/3377811.3380405.

[FWG07] Beat Fluri, Michael Wursch, and Harald C. Gall. "Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes". In: *14th Working Conference on Reverse Engineering (WCRE 2007)*. 2007, pp. 70–79. DOI: 10.1109/WCRE.2007.21.

[Hai+10] Sonia Haiduc et al. "On the Use of Automated Text Summarization Techniques for Summarizing Source Code". In: *2010 17th Working Conference on Reverse Engineering*. 2010, pp. 35–44. DOI: 10.1109/WCRE.2010.13.

[Han] Albrecht Wöß Hanspeter Mössenböck Markus Löberbauer. *Coco/R*. https://ssw.jku.at/Research/Projects/Coco/. Accessed: 2026-02-19.

[Hee] Dimitri van Heesch. *Doxygen*. https://www.doxygen.nl/index.html. Accessed: 2026-02-15.

[KU23] Junaed Younus Khan and Gias Uddin. "Automatic Code Documentation Generation Using GPT-3". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3559548.

[Liu+21] Shiran Liu et al. "Prioritizing code documentation effort: Can we do it simpler but better?" In: *Information and Software Technology* 140 (2021), p. 106686. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106686.

[MM14] Paul W. McBurney and Collin McMillan. "Automatic documentation generation via source code summarization of method context". In: *Proceedings of the 22nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 279–290. ISBN: 9781450328791. DOI: 10.1145/2597008.2597149.

[Opea] OpenVADL. *Additional files for performance testing.* https://github.com/OpenVADL/openvadl/tree/master/docs. Accessed: 2026-02-19.

[Opeb] OpenVADL. *Full example of a Coco/R grammar.* https://github.com/OpenVADL/openvadl/blob/master/vadl/main/vadl/ast/vadl.ATG. Accessed: 2026-02-19.

[Opec] OpenVADL. *The OpenVADL offical GitHub page.* https://github.com/OpenVADL. Accessed: 2026-02-19.

[RBG22] Sawan Rai, Ramesh Chandra Belwal, and Atul Gupta. "A Review on Source Code Documentation". In: *ACM Trans. Intell. Syst. Technol.* 13.5 (June 2022). ISSN: 2157-6904. DOI: 10.1145/3519312.

[Zha+24] Xuejun Zhang et al. "A review of automatic source code summarization". In: *Empirical Software Engineering* 29.6 (2024), p. 162. DOI: 10.1007/s10664-024-10553-6.

[Zho+23] Ziyi Zhou et al. "Towards Retrieval-Based Neural Code Summarization: A Meta-Learning Approach". In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 3008–3031. DOI: 10.1109/TSE.2023.3238161.