SOFDCAR CONSORTIUM

WHITEPAPER

„How to use dependency graph modeling from ECU source code to represent automotive vehicle architectures”

Published by: **P3 digital services GmbH**

Authors: **D. Bartuseck, M. Omar, S. Wolf, L. Marks, F. van Meggelen**

Issue date: **December 4th, 2023**

Funded by the European Union
NextGenerationEU

Supported by:
Federal Ministry for Economic Affairs and Climate Action
on the basis of a decision by the German Bundestag

Contents

1. Introduction

1.1 About SofDCar project

The Software-Defined Car project [SofDCar] is part of a publicly sponsored research program. It is sponsored by the German Ministry of Business Affairs and Climate Control (BMWK) and focuses on the challenges of future electric/electronics and software architecture in vehicles.

1.2 Scoping

This white paper aims to provide a comprehensive overview of dependency graph modeling, highlighting its basic concepts, methods generally used in software engineering and its specific relevance for software systems in the automotive industry. We intend to explore various features of automotive software that make dependency graph modeling not only applicable but also highly beneficial in the current technological landscape.

However, it is important to clarify the boundaries of this work. While we address the general principles of dependency graph modeling and its various methods, the paper does not extend to a detailed technical implementation of these models in software development. Even though we discuss the characteristics of automotive software, the paper does only cover an overview of automotive software engineering and does not cover the hardware aspects of electronic control units (ECUs).

Furthermore, several existing methods for modeling dependency graphs are evaluated, with a focus on their applicability and efficiency in the automotive industry. This includes a discussion of the strengths, weaknesses, and situational preferences of each method. However, this evaluation focuses on a theoretical and conceptual level and does not provide in-depth empirical data or case-specific programming details.

Finally, an outlook is provided, identifying potential success factors and areas for improvement in the area of dependency graph modeling in automotive software systems. This outlook is intended to guide future research and development efforts, but without predicting specific technological advances or market trends. Our goal is to present a balanced and comprehensive view based on current knowledge and practices, as well as insights we have gained in the course of our work in the SofDCar project.

1.3 Motivation for graph modelling of ECU source code to represent automotive vehicle architectures

As explained in more detail in chapter 1.4 the software engineering in the automotive industry is highly decentralized in the general collaboration of an OEM with dozens of suppliers. To provide not yet existing transparency for all involved parties (including authorities) we want to research the application of dependency graph modelling on automotive source code to represent a vehicle's software architecture as it is deployed in different versions of a car model and changed over its lifetime via software updates. This innovation is high interest to the automotive industry, as the change of software updates comes with changes in fulfilling compliance requirements, such as national regulations and technical standards required in possible liability lawsuits.

Therefore, we view dependency graph modelling as a valid candidate to further investigate this technology for use cases in which the software architecture of a vehicle must be represented and

updated in order to maintain an overview of all implemented requirements over lifetime, as well as providing enough abstraction to protect each involved party's intellectual property.

At its core, dependency graph modeling is a method used to visually represent and analyze the dependencies between various components within a software system. This modeling approach is instrumental in simplifying complex software structures, making it easier to understand and manage interdependencies, especially in large and intricate systems. The utility of dependency graph modeling lies in its ability to provide clear insights into how different software modules interact, the potential impact of changes, and the identification of critical components that could affect the system's stability and performance. By mapping out these relationships, it becomes feasible to predict and mitigate issues related to software integration, scalability, and maintenance.

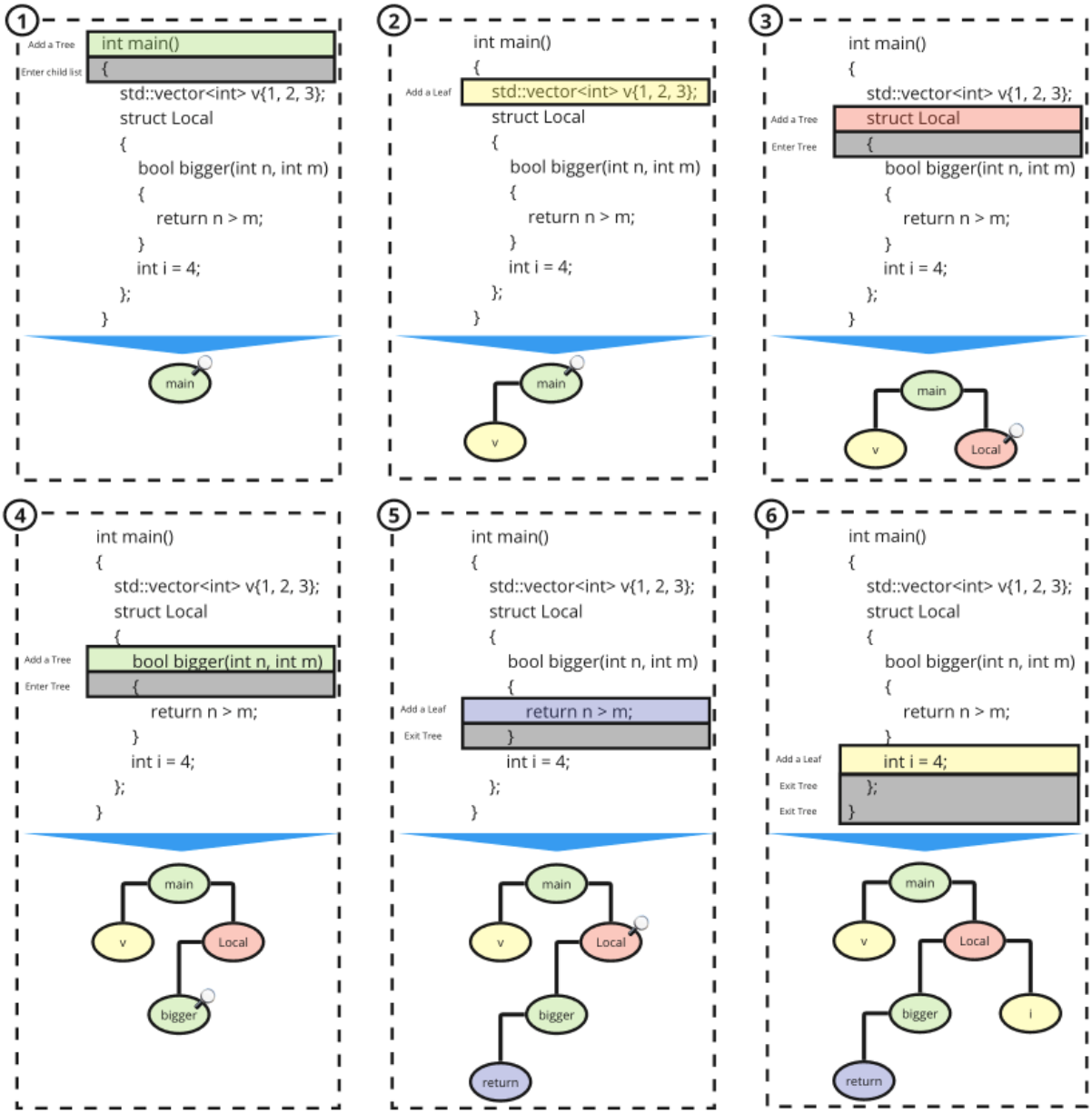

Figure 1: Generic dependency graph modelling approach to build a graph/tree from source code

The general approach of constructing a dependency graph from source code can be methodically broken down into a series of incremental steps, as illustrated in Figure 1.

Initially, the main function is identified as the root of the graph, symbolizing the starting point of the software system. In the subsequent phase, dependencies, such as functions and data structures within 'main', are recognized and linked to it. This includes the vector 'v' and a structure 'Local', as established in the second step. Progressing to the third step, the 'Local' structure is delineated as a scoped dependency, connected to 'main', indicating its localized relevance within the 'main' function. The fourth step expands the graph to include the 'bigger' function, which is integrated based on its interaction with 'main' or related dependencies. Delving further, the fifth step marks the return statement within the 'bigger' function as a terminal node, illustrating the data flow and control endpoints within the function. Finally, the graph is completed by appending literals and variables, such as 'int i = 4;', as leaf nodes in the sixth step. This comprehensive representation captures the nuanced web of interdependencies found within the source code, allowing the dependency graph to serve as an analytical and illustrative tool for understanding the software's architecture.

One of the primary advantages of employing dependency graph modeling in vehicle source code is the identification of critical paths. These paths are vital conduits through which essential data and control signals flow, particularly within safety-critical systems such as braking, steering, and airbag deployment. Recognizing these paths ensures that engineers can prioritize the stability and integrity of the most crucial system interactions during both development and troubleshooting phases. Furthermore, dependency graphs greatly facilitate the debugging and troubleshooting process. By mapping out the dependencies and their connections, engineers can more easily identify potential points of failure and address software bugs effectively. The visualization of dependencies allows for a systematic approach to pinpointing and resolving issues, which can otherwise be a time-consuming and error-prone task in the absence of such clear mappings. Beyond their technical utility, dependency graphs serve a vital communicative function. They act as a common visual language among cross-functional teams, including software developers, system architects, and quality assurance professionals. This shared language enables clearer, more productive discussions about system architecture and aids in bridging the gap between technical and non-technical stakeholders, ensuring that all team members have a coherent understanding of the system's structure and functionality.

In addition, regulatory compliance is a significant concern in the automotive industry, where safety and performance standards are stringent. Dependency graph modeling supports compliance efforts by making it transparent how different components and their interactions adhere to required standards. By illustrating the flow of control and data across various modules, dependency graphs can be used to demonstrate that the system has been architected with regulatory requirements in mind. The benefits of dependency graph modeling in the automotive domain underscore its value as a strategic asset. By enhancing visual understanding, identifying critical system paths, simplifying debugging, aiding communication, and supporting regulatory compliance, dependency graphs not only streamline the development process but also contribute to the creation of safer, more reliable vehicles. As vehicle systems continue to grow in complexity with the advent of advanced driver-assistance systems and autonomous driving technologies, dependency graph modeling will undoubtedly remain an essential element of automotive software engineering.

A deeper dive into the specifics of dependency graph modeling, including its methodologies, tools, and techniques, will be thoroughly explored in a later chapter.

1.4 Background Information on ECU development and automotive software engineering

The evolution of automotive Electronic Control Units (ECUs) and the accompanying software engineering is a testament to the advancements in vehicular technology and complexity. ECUs, serving as the decentralized brains of modern vehicles, are specialized embedded systems dedicated to managing a multitude of functions ranging from engine control, vehicle dynamics and safety systems to online connectivity and infotainment. The orchestration of these functions is underpinned by a layered software architecture that ensures efficient operation and scalability. The uppermost application layer comprises high-level applications like infotainment systems and advanced driver-assistance features, whereas the middleware layer acts as a conduit for data exchange between software components, relying on communication protocols and operating systems. The hardware layer interfaces directly with the vehicle's physical components, such as sensors and actuators.

These embedded systems are characterized by their ability to operate in real-time environments, relying on real-time operating systems (RTOS) for deterministic execution of control algorithms, crucial for safety-critical applications like ABS and airbag deployment. Modern vehicles also feature over-the-air (OTA) update capabilities, allowing for software enhancements and feature additions without physical interventions. Furthermore, with the surge in vehicular connectivity, cybersecurity has become paramount, prompting the integration of robust security measures within the software architecture to safeguard against cyber threats.

Moreover, the ECUs' functionality extends to sensor fusion, where data from cameras, radar, lidar, and ultrasonic sensors is synthesized to provide a holistic view of the vehicle's surroundings, essential for advanced driver-assistance systems (ADAS) and autonomous driving. The software architecture encapsulates components for diagnostics and telematics, which facilitate fault detection and enable communication with external services.

ECUs also communicate internally via networks such as Controller Area Network (CAN), Local Interconnect Network (LIN), and FlexRay, ensuring harmonious operation across various systems. Each ECU is tailored for specific functions, exemplified by units dedicated to engine control, transmission management, braking systems, and climate control, among others. The interaction between ECUs and their specific functions, undergirded by complex software involving intricate algorithms, is central to the vehicle's performance. The architecture not only prioritizes functionality but also emphasizes safety and reliability, incorporating redundancy and fail-safe mechanisms.

As automotive technology further evolves with applications like autonomous driving, the complexity of ECUs continue to expand, necessitating the adaptation and enhancement of ECUs to manage emerging technologies. The cumulative development of these systems reflects the dynamic nature of automotive software engineering, underscoring the need for continuous innovation and adaptation to maintain the harmony between performance, safety, and technological advancement in the automotive industry.

2. State of the Art

Dependency graph modeling is a multifaceted discipline that leverages a variety of methods and tools to address complex challenges in software engineering. The state of the art reflects a mature but ever-evolving landscape, where the focus remains on providing deeper insights into software structure and behavior, while striving to manage the inherent complexity of modern software systems [Dep_Graph_1].

Dependency graph modeling has become an indispensable tool in the domain of software engineering and information science, offering multifaceted benefits that streamline the development and maintenance of complex automotive systems. By providing a visual representation of the software architecture, dependency graphs furnish engineers with an intuitive understanding of the intricate web of component interactions. This visual clarity is particularly beneficial when grappling with the complexities of modern vehicles, which are often equipped with numerous interdependent systems and sub-systems.

2.1 Overview of the dependency graph modelling methodology

2.1.1 General Overview of the dependency graph modelling

In software engineering, dependency graph modeling stands as a commonly used methodology for comprehending and managing the complex interrelations and interdependencies inherent in software systems. The foundational elements of dependency graph modeling encompass nodes and edges, representing software entities (such as classes, functions, or variables) and their interdependencies, respectively, as is shown in the visualization of Figure 1. This graphical representation is instrumental in describing the structure of software and the potential impact of changes within the system.

The current state of the art in dependency graph technologies is marked by a spectrum of sophisticated tools and platforms that cater to varying needs within the field, for which examples are given in 2.2. Among the leading technologies are static analysis tools, which parse source code and extract dependency relationships without executing the program. Dynamic analysis tools, on the other hand, observe the execution of a program to capture runtime dependencies, offering insights that static methods cannot. Integrated Development Environments (IDEs) frequently incorporate these tools to provide real-time dependency analysis, facilitating immediate feedback to developers. Version control systems also play a crucial role, as they can track changes in dependencies over time and across different versions of the software.

In terms of challenges, the primary issues revolve around the scalability of dependency graphs, the accuracy of the depicted relationships, and the meaningful representation of these relationships in a way that is useful to developers. Large-scale systems can result in graphs that are dense and unwieldy, complicating the task of identifying key components or critical paths. To address this, filtering techniques and abstraction layers are employed to simplify graphs without sacrificing essential information. Clustering related nodes and focusing on particular types of dependencies, like control flow or data flow, can also help manage complexity [Dep_Graph_2].

2.1.2 Specific Overview of the dependency graph modelling for automotive source code

The process of dependency graph modeling in the context of automotive vehicle source code as it was initiates with a meticulous identification of the major software components within the vehicle's architecture, such as Electronic Control Units (ECUs), sensors, actuators, and various software modules.

Once the components are cataloged, the next critical step is to define the dependencies that exist among them. This involves a detailed analysis of the software code and specifications, identifying the connections that manifest in the form of function calls, data flows, or communication links. Understanding these dependencies is crucial as they illustrate the interplay between different ECUs and other software entities, providing insights into the software ecosystem of the vehicle.

The subsequent phase involves the creation of the dependency graph, employing advanced graph modeling tools. These tools facilitate the generation of a visual representation of the dependency network where nodes symbolize the software entities, and edges represent the dependencies between them. This graphical representation can be further enhanced through layered representation, organizing the dependencies across different levels of the software architecture to mirror the application, middleware, and hardware layers of the vehicle’s systems.

To augment the clarity and utility of the dependency graph, color coding and annotations can be incorporated. This enables a more intuitive differentiation between various types of dependencies, such as those related to communications versus function calls or data dependencies. Including communication channels explicitly in the graph is also vital, as it can visually depict protocols like CAN, LIN, or FlexRay, clarifying the data flows between components and how they collaborate within the network.

Addressing the aspect of fault tolerance and redundancy, the graph should encapsulate information that showcases the vehicle's fault-tolerant architectures, if present. This includes illustrating duplicated components and alternative paths that can be activated in the event of a failure, ensuring continuous system operation. Such representations are particularly important in automotive systems where safety and reliability are paramount.

For the dependency graph to remain relevant, it must be maintained with regular updates as the software evolves. This ensures that it reflects the current state of the vehicle's software architecture accurately, thereby maintaining its integrity as a reliable document for both current utility and historical reference.

Integration with system requirements is another important dimension, linking the dependencies in the graph to specific functional and non-functional requirements. This integration ensures that the software components fulfill their intended roles and adhere to the predetermined specifications, providing a clear trace from requirements to implementation.

Finally, the dependency graph should not exist in isolation but be shared with all relevant stakeholders, including software developers, system architects, and testers. Its utility as a collaborative tool is unparalleled, facilitating discussions and serving as a key piece of documentation that supports the collective understanding and continuous improvement of the system architecture. Thus, the dependency graph becomes not just a technical artifact but also a communication medium, bridging the gap between various domains of expertise within the automotive software development lifecycle.

2.2 Current technologies, tools and use cases for dependency graph modelling

Dependency graphs come in various forms, each tailored to specific types of analysis. Control flow graphs, for instance, are used to represent the order in which individual statements, instructions, or function calls are executed within a program. Data flow graphs track the flow of data through the system and are crucial in identifying potential data-related issues. Object graphs and class diagrams are employed in object-oriented programming to illustrate relationships between objects and classes, respectively.

Several tools have emerged as standard bearers in the industry due to their robustness and utility. The existing solutions to choose from contain a variety of open-source and commercial tools for visualizing and analysing graph's dependencies from different programming languages. A selected range of tools that will be analyzed in the further course of this work is presented in 3.1.

The application of dependency graph modeling spans several areas, including software maintenance, where it aids in impact analysis, refactoring, and regression testing. In the context of build systems, dependency graphs are crucial in determining the order of compilation and linking. They are also applied in the domain of software security, to analyze the potential impact of vulnerabilities and to trace the propagation of tainted data through a system.

2.3 Components and steps in an exemplary dependency graph modeling of automotive vehicle source code

The construction of a dependency graph from Electronic Control Unit (ECU) source code is a methodical process that translates the complex interrelations of software components into a coherent visual framework. This multistep approach commences with a comprehensive analysis of the ECU source code using sophisticated source code analysis tools, such as Clang for C/C++ or equivalent tools for other programming languages. These tools are adept at parsing the source code to extract pivotal information about the structure, functions, and dependencies within the codebase. This first step is showcased in phase 1 of Figure 2.

Following the initial analysis, dependency identification is conducted, focusing on function calls, data flow, and communication links within the source code. This step is crucial for revealing the intricate web of interactions, represented by include statements, function calls, global variables, and other mechanisms that indicate relationships between different components, which can be seen in phase 2 of Figure 2. Subsequently, a detailed data structure is created to represent the graph, where nodes correspond to software components, like functions, modules, or ECUs, and edges represent the dependencies between these nodes. Attributes are defined for both nodes and edges, capturing additional information such as the type of component or dependency, whether it's a function call or data flow (visible as phase 3 in [SofDCar).

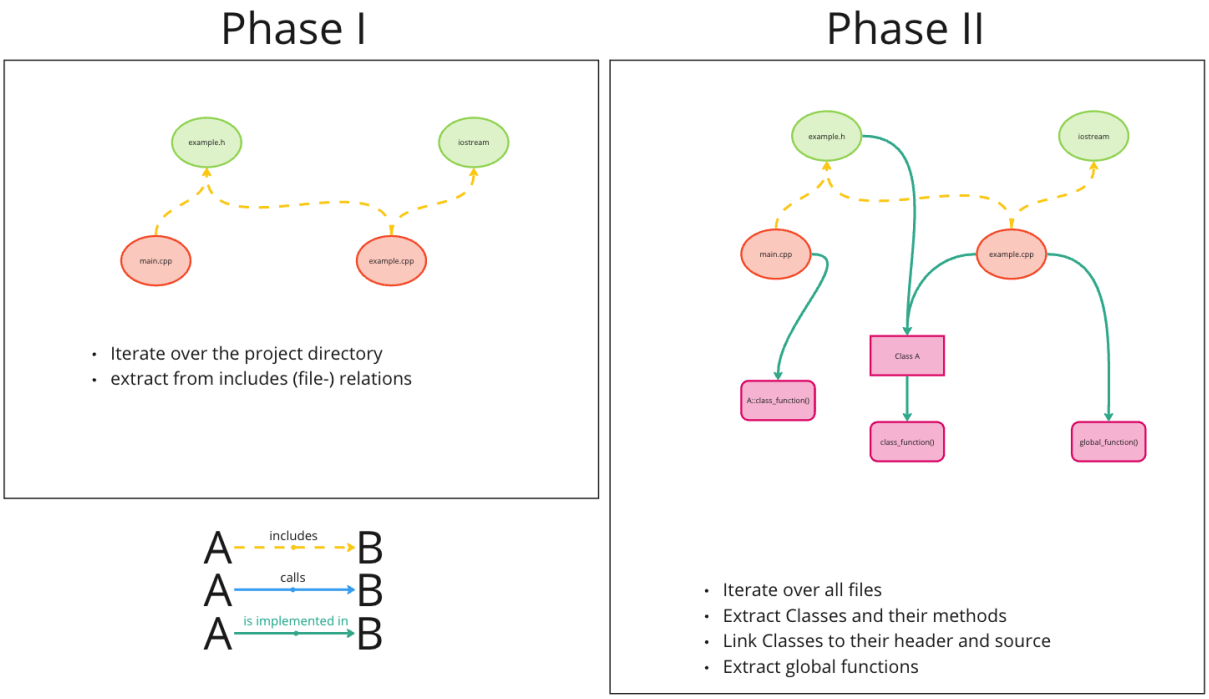

Figure 2: Phases 1 and 2 of examplary dependency graph modelling to showcase the iterative growth in graph modelling

The graph is then constructed and populated with these nodes and edges based on the identified dependencies, with each function or module represented as a node within the graph (visible as phase 4 in [SofDCar). To enhance understanding, the graph can be organized into layers to represent different levels of abstraction or architectural layers, providing a clearer visualization of the software architecture.

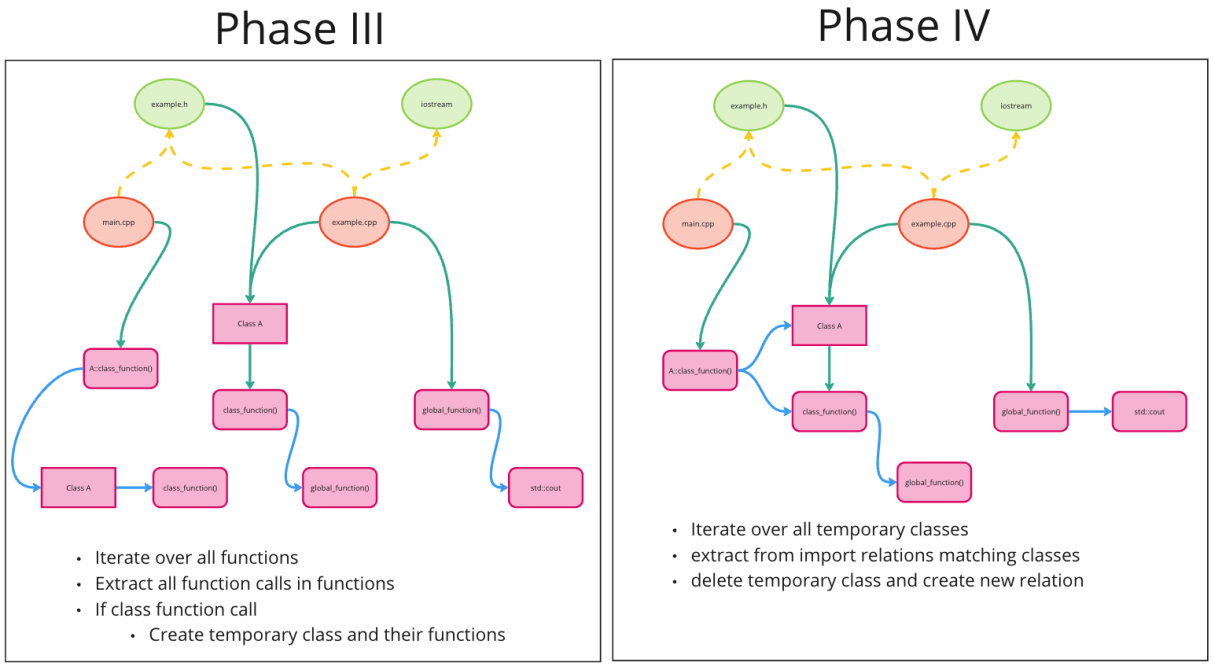

Figure 3: Phases 3 and 4 of examplary dependency graph modelling to showcase the iterative growth in graph modelling

Further refining the graph involves the incorporation of color coding, annotations, or other enhancements to provide additional context and highlight critical paths or types of dependencies.

These visual cues are instrumental in quickly conveying the nature of the dependencies to those analyzing the graph.

Once the graph is populated and enhanced, the model can be refined by focussing on local call relations and iterating over all unrelated functions, which may not be part of any class but still contribute to the program's behavior, which is displayed as phase 5 in Figure 4. To finalize the modelling process phase 6, shown in Figure 4, involves a cleanup process where unnecessary relations are removed, and a review is conducted for any functions without matches. This phase also includes the handling of standard library calls and namespacing, which may require a return to Phase 3 for a more thorough analysis. The final representation is converted into a standard graph format, such as [DOT] (from Graphviz) or [GML] (Graph Modeling Language), to facilitate compatibility with various graph visualization tools. These tools, such as [Graphviz], [yEd], or [Gephi], provide interactive interfaces for users to navigate, analyze, and extract meaningful insights from the dependency graph.

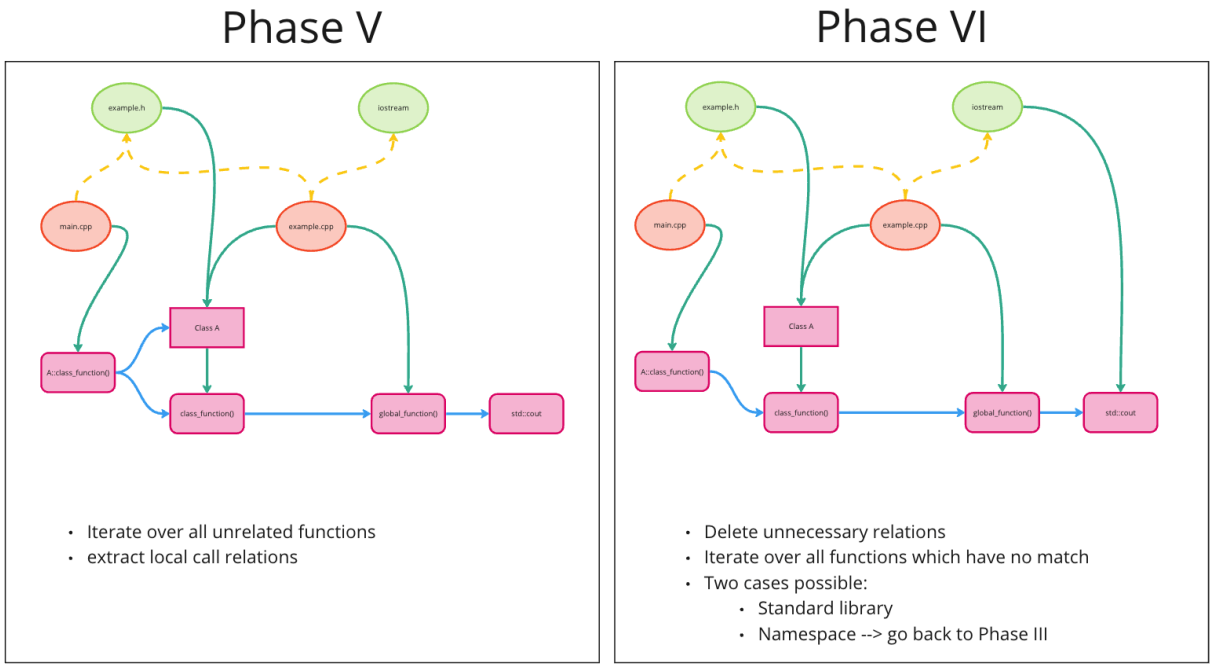

Figure 4: Phases 5 and 6 of examplary dependency graph modelling to showcase the iterative growth in graph modelling

Through this process, the dependency graph evolves into a dynamic, interactive model of the ECU software architecture. It serves as a crucial instrument for developers, system architects, and testers, aiding in tasks ranging from code maintenance to the implementation of new features, and ensuring that all stakeholders have a shared understanding of the system's structure and functionality.

3. Dependency Graph Modelling Methods

This chapter introduces a comparative analysis of various methodologies utilized in crafting visual representations that delineate the intricate web of dependencies and relationships between software components within a vehicle.

By comparing selected methods of dependency graph modeling, this chapter seeks to evaluate their key differences and applicability in the automotive industry. This evaluation is grounded both in the research and testing of these methods within our work for the SofDCar project. The objective is to enhance the understanding which modeling technique can expose paths to optimization, reveal potential areas of risk for changes in software updates, and support a general robust design of vehicle software systems.

In the forthcoming sections, we will provide an overview for each of the selected methods, followed by introducing the evaluation criteria and subsequently a comparison of the selected methods based on these criteria.

3.1 Selected methods

3.1.1 Regular Expressions

[Regular Expressions], or regex, are sophisticated patterns that are utilized for matching sequences of characters within strings. They serve as a powerful and flexible tool, enabling the detection, extraction, and manipulation of specific text patterns from a larger corpus of code. The versatility of regular expressions lies in their ability to articulate complex search patterns and perform intricate text manipulations with precision. Regular expressions are the go-to method when the search patterns are intricate and the text parsing requirements are highly nuanced. This method commonly used in the analysis and comprehension of the interdependencies within source code by translating raw code into actionable data structures that reveal the underlying architecture of software systems.

3.1.2 String Splitting

[String Splitting] is a technique often used to dissect a string at specified delimiter characters, thus segmenting the text into an array of substrings or tokens. This method is particularly useful for simpler parsing tasks where the structure of the data is well-defined and less complex. While string splitting is generally more limited in scope compared to the robust capabilities of regular expressions, its simplicity and ease of use make it an attractive option for basic tokenization and substring extraction tasks. String splitting is preferred when the parsing involves clear, delineated sections of text that can be easily separated by known delimiters. This method commonly used in the analysis and comprehension of the interdependencies within source code by translating raw code into actionable data structures that reveal the underlying architecture of software systems.

3.1.3 Clang (libClang)

[Clang], a well-regarded frontend for the [LLVM project], is not only a compiler for C, C++, and Objective-C but is also celebrated for its exceptional language compatibility and rapid compilation times. It boasts an architectural design that is inherently modular, which allows it to integrate seamlessly into a variety of development environments, an attribute that is immensely beneficial for complex software analysis.

Accompanying Clang, [libclang] stands as its core interface, offering access to the intricacies of Clang's abstract syntax tree (AST). This interface extends a suite of APIs that permit a language-agnostic approach to accessing the structure and contents of source code, thereby enabling a broad spectrum of development tools. Through libclang, developers are empowered to construct advanced tools that can facilitate source-to-source transformation, carry out static analysis, drive refactoring efforts, and offer comprehensive Integrated Development Environment (IDE) support.

These two methods combined form a robust foundation for constructing dependency graphs, providing a detailed view of the interdependencies in the source code which is crucial for accurate software analysis. With the power of Clang and libclang, dependency graph modeling transcends beyond mere representation, evolving into an insightful exercise in understanding and improving the architecture of complex software systems.

3.1.4 PyCParser

[Pycparser] is a comprehensive tool for dependency graph analysis, specifically tailored for C source code. It's a lightweight yet powerful Python library that parses C code and constructs an Abstract Syntax Tree (AST), enabling a detailed examination of the code's structure. This library is particularly advantageous for developers aiming to perform C code analysis, source-to-source transformation, or code generation. The strength of pycparser lies in its simplicity and effectiveness in breaking down C code into its fundamental components, making it an indispensable asset for developers engaged in advanced code manipulation and analysis tasks.

3.1.5 gcc Python Plugin

The [gcc-python-plugin] is a remarkable extension for the GNU Compiler Collection (GCC) that harnesses the flexibility of Python to write GCC plugins. It acts as a conduit between the Python language and the internals of GCC, offering developers an innovative way to extend the capabilities of the GCC. With this plugin, tasks such as code analysis, optimization, and transformation can be approached through Python, providing a high level of accessibility and customization. It opens up a world of possibilities for automating and enhancing the GCC's functionality, making it a valuable tool for developers seeking to innovate within the compilation process.

3.1.6 pygccxml

[Pygccxml] stands out as a pivotal tool for parsing C++ source code. It offers a streamlined Python package that generates an XML-based representation of the AST. This functionality is of particular importance for developers who are involved in C++ code analysis, code generation, or interacting with C++ APIs. By providing a clear and structured XML representation of code components, pygccxml

simplifies the otherwise complex task of navigating C++ source code, making it easier to analyze and manipulate the codebase programmatically.

3.1.7 Doxygen

[Doxygen] is a widely recognized documentation generation tool that has become essential in modern software development. It automatically generates comprehensive documentation from annotated source code, significantly simplifying the understanding and navigation of complex codebases. By parsing source files with great attention to detail, Doxygen creates documentation that not only helps in maintaining and understanding the code but also in visualizing the relationships between various code entities, which is crucial for dependency graph modeling.

3.1.8 Tree-sitter

[Tree-sitter] is an open-source parsing system designed to be a fundamental tool in the creation and manipulation of ASTs. Known for its performance, robustness, and versatility, Tree-sitter is extensively used in code editors and programming language analysis tools. Developed and maintained by GitHub, it supports a wide range of programming language grammars and provides incremental parsing, which allows for real-time feedback in development environments. Tree-sitter's ability to handle complex language patterns and its speed make it a preferred choice for applications that require rapid parsing and a deep understanding of code.

3.2 Comparison of dependency graph methods

3.2.1 Comparison criteria for evaluating dependency graph methods

The criteria employed in the comparative analysis of dependency graph modeling methodologies, as presented in this whitepaper, have been selected based on empirical evidence gathered through our testing and application in a variety of scenarios applicable to the automotive industry. These factors, constituting our comparison criteria, are reflective of the practical challenges and requirements faced by professionals in the field when applying these approaches to real-world automotive software architectures.

Direct Extraction:

> Direct extraction as a criterion is pivotal because it ensures the immediacy and purity of data retrieval from the source code. This capability is essential for the precise capture of structural and semantic information, which is foundational for any subsequent analysis or transformation. It not only streamlines the parsing process but also mitigates the risk of errors that could be introduced by intermediary processing stages.

Easy Implementation:

> The ease of implementation of a dependency graph method is a critical criterion because it directly affects the speed and efficiency with which a tool can be deployed. Methods that are

simple to implement can significantly reduce the learning curve and resource allocation, allowing teams to focus on analysis rather than grappling with complex setup procedures.

Code Interpretation:

The capability of a method to interpret code is crucial as it determines the accuracy of the analysis. Accurate code interpretation allows for a deeper understanding of the code's behavior and logic, enabling precise modifications and informed decision-making. This accuracy is particularly important when the analysis results drive critical development and refactoring decisions.

Cross-referencing:

Cross-referencing capabilities are essential because they enable the identification of interconnections and dependencies within the codebase. This aspect of a method enhances the comprehensibility of complex software architectures by allowing developers to trace the impact of changes across various components.

Internal Expressions:

The clarity with which a method represents internal expressions impacts its ability to generate a detailed and useful structural representation of the code. This clarity is essential for creating accurate ASTs, which are the backbone of in-depth code analysis and understanding.

Compiler Independent:

Compiler independence is a key criterion because it ensures that the method's applicability is not constrained by the choice of compiler. This broadens the method's utility across various environments and future-proofs it against evolving compiler technologies.

Flat Learning Curve:

A flat learning curve is beneficial as it allows developers to quickly master the method, leading to faster integration and utilization within a project. This accelerates the overall development cycle and enhances productivity.

Low RAM Demand:

Low RAM demand is an important consideration, especially when working with large and complex codebases. Methods that are efficient in memory usage can scale better and accommodate the simultaneous parsing of multiple projects without performance degradation.

Stability:

Stability is a critical criterion for ensuring the reliability of the parsing method over time. A stable method provides consistent results, which is crucial for maintaining confidence in the tools and systems built upon it.

Extensive Language Support:

Extensive language support is vital for a method's applicability to diverse codebases. In a multilingual software environment, the ability to parse and analyze different programming languages is essential for comprehensive analysis.

AST Extraction:

The extraction of an AST is a cornerstone for advanced code analysis and transformation. Methods that can extract and manipulate ASTs allow for sophisticated operations like refactoring and can serve as the basis for a variety of code analysis tools.

Code Analyzer:

A robust code analyzer is key to understanding the intricacies of a codebase. It enables the detection of potential issues, aids in optimization, and provides valuable insights into code structure, making it an indispensable tool for developers.

Development Time:

Minimizing development time for parsing strategies is essential for rapid tool development and deployment. Faster development leads to quicker iterations and improvements in the analysis and understanding of code.

Low Maintenance:

A parsing approach that necessitates minimal maintenance is highly desirable as it ensures sustained functionality over time with less resource investment, translating into a more efficient and cost-effective solution for long-term use.

3.2.2 Comparative analysis of dependency graph methods

In the pursuit of an unbiased assessment within this whitepaper, we have refrained from applying any weighting to the various criteria used for comparing dependency graph modeling methodologies. The intention behind this approach is to avoid unintentional prioritizing certain aspects that may favor specific use cases over others. Instead, our comparative analysis is structured to deliver a balanced and holistic overview, systematically describing whether each selected method fulfills the established criteria. This approach ensures that our evaluation remains neutral, allowing readers to consider the applicability of each method to their unique scenarios without preconceived emphasis on specific characteristics.

3.2.2.1 Regular Expressions

Regular expressions (regex) have been a mainstay in text processing and pattern matching due to their versatility and efficiency, which is why they score highly in several criteria in our comparison of dependency graph methods. Their ability to perform direct extraction is excellent because they can match complex patterns within a body of text with precision, making them a robust tool for data retrieval. Their implementation is relatively straightforward for those familiar with their syntax, thus earning a good score for easy implementation.

When it comes to code interpretation, regex can effectively identify and extract syntactic patterns, which is why they receive a good score in this domain. However, their utility diminishes in cross-referencing and interpreting internal expressions, where they are not sufficient. Regex lacks the contextual understanding needed to link related code elements and may struggle with the nested or recursive structures common in these expressions.

Regex's compiler independence is a strong advantage, as they are supported across various platforms and environments without dependency on specific compilers, hence scoring good in this criterion. They also have a flat learning curve for those with a basic understanding of pattern matching, and their text-based nature ensures low RAM demand, leading to good scores in both respects.

Stability, however, is rated as not sufficient due to the potential for complex regex patterns to become unwieldy and error-prone, especially when regex becomes more complex or when used by less experienced practitioners. While regex boasts extensive language support due to their general applicability across programming languages, they are not inherently suited as a comprehensive code analyzer, lacking the semantic understanding necessary for in-depth code analysis.

The ability of regex to assist in AST extraction is good when patterns are well-defined and consistent; however, this can vary greatly with the complexity of the source code. When considering development time and maintenance, regex methods can become time-consuming and maintenance-heavy as patterns become more complex and as the codebase evolves, thus scoring not sufficient in these areas. Our research and testing have highlighted these specific scores in comparison to other methods evaluated, reflecting regex's strengths and limitations within the scope of dependency graph modeling.

3.2.2.2 String Splitting

The method of string splitting in dependency graph modeling offers a mixed range of effectiveness across various criteria based on our research and testing. For direct extraction, string splitting scores well because it can quickly and effectively segregate a text based on predefined delimiters, making it suitable for straightforward pattern identification and data extraction.

However, the method does not score sufficiently in ease of implementation, primarily because string splitting alone often requires additional logic to handle complex parsing tasks effectively. For code interpretation, string splitting performs well when the structure of the code is consistent, and the delimiters are clearly defined.

Cross-referencing is rated as neutral because while string splitting can identify sections of code, it does not inherently provide the means to link related elements without supplemental processing. It scores not sufficiently for internal expressions due to its basic nature, which lacks the sophistication to parse nested or complex internal structures within the code accurately.

String splitting is compiler independent and has a flat learning curve—both of which score good—because it is a fundamental method that does not rely on compiler-specific features and is relatively easy to understand and apply for basic parsing tasks. It also demands low RAM, given its simplicity, which ensures good performance even when dealing with large datasets.

In terms of stability, string splitting is rated not sufficient because it can be brittle; changes in the code structure can easily break the parsing logic. While it has extensive language support, as it can be applied to any text-based code, its abilities as a code analyzer are not sufficient due to the lack of depth in the analysis it can provide.

Regarding AST extraction, the method scores good, given that the structure to be extracted is simple and well-defined. However, in terms of development time and maintenance, string splitting is not sufficient, as maintaining parsing logic can become complex and time-consuming as the codebase evolves or if the parsing requirements become more intricate.

3.2.2.3 Clang (libClang)

Clang, as a dependency graph method, presents nuanced performance across an array of evaluation criteria. When it comes to direct extraction, Clang is rated as not sufficient. Despite its powerful parsing capabilities, Clang does not inherently extract dependencies without a considerable amount of setup and configuration. Similarly, its implementation is not straightforward, requiring in-depth knowledge of Clang's internals, which accounts for the not sufficient rating in ease of implementation.

In contrast, Clang excels in code interpretation, cross-referencing, and internal expression analysis. It scores good in these areas due to its sophisticated understanding of C/C++ syntax and semantics, which enables it to parse complex code structures and cross-reference elements within a codebase accurately. This also extends to the parsing of internal expressions, where Clang can reliably interpret and represent nested and intricate code constructs.

However, Clang's compiler dependency affects its versatility, hence the not sufficient rating for compiler independence. This is because Clang is tightly coupled with LLVM's infrastructure. The learning curve for Clang is steep due to its comprehensive nature and the depth of features it offers, leading to a not sufficient score in this criterion.

Clang's memory consumption is substantial when processing large codebases, which leads to a not sufficient rating for low RAM demand. As for stability, Clang receives a neutral rating. While it is a robust tool, changes in the LLVM infrastructure or Clang itself can introduce variability in its behavior over time.

Extensive language support is rated as not sufficient since Clang is primarily focused on C, C++, and Objective-C, with limited support for other languages. Clang shines in AST extraction and as a code analyzer, providing good utility in these criteria due to its ability to generate detailed ASTs and analyze code with precision.

The scoring of good for development time and maintenance may seem counterintuitive given Clang's complexity, but these ratings reflect Clang's effectiveness once properly implemented. It can facilitate rapid development through its automation capabilities and once set up, requires relatively low ongoing maintenance, which is favorable for long-term projects. Our research and testing have provided these specific scores in comparison to other methods evaluated, underscoring Clang's strengths in detailed code analysis and its challenges with ease of use and resource demands.

3.2.2.4 PyCParser

PyCParser stands as a specialized tool in the domain of dependency graph modeling, particularly for C language source code analysis. Our research and testing indicate that while PyCParser performs adequately in several areas, it has limitations in others when compared to other methods evaluated.

For direct extraction, PyCParser scores as not sufficient. It often requires additional layers of processing to extract meaningful data directly from the source code, which may not be as straightforward as some other methods. Its implementation also receives a not sufficient rating because, while PyCParser is a powerful tool, it demands a certain level of understanding of both C and Python, which can complicate its integration into existing systems.

Where PyCParser shines is in code interpretation, cross-referencing within the code, and analyzing internal expressions, for which it scores good. Its ability to parse C code and generate an AST facilitates a deep understanding of code semantics and structure.

However, PyCParser is not entirely compiler-independent, scoring not sufficient in this criterion, as it relies on specific formats and conventions found in GCC-style C code. The learning curve for effectively leveraging PyCParser is also not flat, particularly for those not well-versed in C or Python's nuances, thus earning a not sufficient score.

In terms of RAM demand, PyCParser can be resource-intensive, especially when dealing with large codebases, leading to a not sufficient score for low RAM demand. The stability of PyCParser is rated as neutral; while it generally produces consistent results, updates or changes in the library can occasionally affect its behavior.

PyCParser's support for languages is focused on C, resulting in a not sufficient rating for extensive language support. However, it performs well in AST extraction and as a code analyzer—areas where its capabilities are well-aligned with the requirements, thus scoring good. Its effectiveness in these aspects supports a good score in developing time, as it allows for the rapid creation of analysis tools once the initial learning curve is surmounted.

Finally, the maintenance of PyCParser is considered neutral. It does not require excessive upkeep, but nor is it entirely free from maintenance needs, especially when adapting to new codebases or C language standards.

3.2.2.5 gcc Python Plugin

The gcc-python-plugin, as a method for dependency graph modeling, exhibits a distinct set of capabilities and constraints. Based on our extensive research and testing, which compared a variety of methods, the plugin's scores in specific criteria have been established as follows.

In terms of direct extraction, the gcc-python-plugin is rated as not sufficient. While it allows for the extension of the GCC compiler's functionalities using Python, it does not inherently provide direct dependency graph extraction without significant user-defined logic.

Its ease of implementation is also scored as not sufficient, reflecting the specialized knowledge required to effectively utilize GCC internals and the Python interface. The method, however, achieves a good score in code interpretation, given its capability to access and manipulate the compiler's understanding of code structure and semantics.

For cross-referencing and interpreting internal expressions, the gcc-python-plugin also scores good. It can leverage GCC's robust analysis to link code elements and handle complex language constructs, enhancing the understanding of interdependencies within the code.

Compiler independence is rated as not sufficient because the tool is inherently tied to the GCC compiler. The learning curve is not flat due to the intricacy of GCC's internal structures and the need to bridge them with Python, which requires a deeper understanding of both ecosystems.

The plugin does not score well on low RAM demand, as it operates within the context of GCC, known for its comprehensive but memory-intensive processing. Stability receives a neutral rating, acknowledging the general reliability of GCC while considering the potential variability introduced by custom plugins.

Extensive language support is rated as not sufficient because, although GCC supports multiple languages, the plugin's functionality may vary across different language frontends. The method's performance in AST extraction and as a code analyzer is deemed good, benefiting from the depth of analysis provided by GCC's compilation process.

Development time is rated as good, suggesting that once the initial setup is complete, the plugin can facilitate rapid tool development. Maintenance is considered neutral, reflecting a balance between the ongoing upkeep required for complex GCC extensions and the robust support provided by the GCC community.

3.2.2.6 pygccxml

Pygccxml is a specialized tool within the realm of dependency graph methods, and it exhibits a performance profile that reflects the complexity of parsing C++ source code into XML-based representations. Our research and comparative analysis of various methods has led to the following assessments for pygccxml.

For direct extraction, pygccxml does not score sufficiently, as it requires a considerable setup to parse and convert C++ code into an XML AST. It is not a direct extraction tool but rather a parser that needs the GCC-XML output as an intermediary, which can introduce complexities.

The implementation of pygccxml is also not straightforward, rating as not sufficient. It necessitates a solid understanding of both the C++ code structure and XML schema to effectively utilize its capabilities, which can present a steep learning curve to new users.

In terms of code interpretation and cross-referencing, pygccxml achieves a good score. It effectively interprets the structure and semantics of C++ code, and its XML output can be used to cross-reference elements within the codebase. Internal expression handling also receives a good score due to the detailed nature of the AST generated, allowing for nuanced representation of the code's logic.

The tool is not compiler independent, scoring not sufficiently in this category because it relies on the specific output format generated by GCC-XML, tying its functionality to the GCC compiler's representation of the code.

The learning curve for pygccxml is not flat, attributed to the complex nature of C++ code parsing and XML handling, thus scoring not sufficient. The RAM demand for running pygccxml is also not low, particularly when dealing with large C++ codebases, resulting in a not sufficient score for memory efficiency.

Stability is rated as neutral, acknowledging that while the tool generally provides reliable output, it can be affected by changes in the underlying GCC-XML or the specifics of the C++ code it parses.

Extensive language support is not sufficient since pygccxml is tailored for C++ and may not be suitable for other programming languages without significant adaptations.

However, pygccxml scores good in AST extraction due to its ability to produce a detailed and navigable XML representation of the AST, which is highly beneficial for in-depth analysis. It also scores good as a code analyzer, leveraging the comprehensive nature of the generated AST to provide insights into the C++ code.

Development time with pygccxml is rated as good, suggesting that despite the initial complexity, the tool can be efficient for ongoing development after mastery. Maintenance is scored as neutral,

reflecting the balanced effort required to keep the tool operational against evolving codebases and potential updates to the C++ language or the GCC-XML.

3.2.2.7 Doxygen

Doxygen is a documentation generation tool, widely used for its ability to create software documentation from annotated source code. Our evaluation of Doxygen as a dependency graph method reveals a diverse set of outcomes across several criteria.

For direct extraction, Doxygen receives a neutral score. It is capable of extracting comments and documentation directly from the source code but is not designed for extracting complex dependency graphs without additional processing or tooling.

The implementation of Doxygen is straightforward, meriting a good score. Its ease of setup and configuration allows developers to quickly generate documentation with minimal effort. However, in terms of code interpretation, Doxygen is not sufficient. While it effectively captures comments and some structural elements of code, it is not intended for deep semantic analysis.

Cross-referencing is one of Doxygen's strengths, as it can effectively link documentation across various code elements, resulting in a good score. Similarly, it handles internal expressions well by documenting the interfaces and hierarchies within the code, assuming they are properly annotated.

Doxygen is compiler independent, which earns a good rating. It does not rely on any particular compiler to generate documentation, making it versatile across different programming environments. Its learning curve is relatively flat, especially for those familiar with comment documentation practices, and this user-friendliness contributes to a good score.

In terms of RAM demand, Doxygen is rated not sufficient, as generating documentation for large codebases can be memory-intensive. Stability, however, is a strong point for Doxygen, and it scores good, thanks to its mature and well-maintained codebase.

Doxygen boasts extensive language support, covering a wide range of programming languages, which warrants a good score. Its ability to extract an AST is not sufficient, as its primary function is documentation, not detailed code analysis. As a code analyzer, it also scores not sufficient, as it does not provide the in-depth analysis typically required for dependency graph modeling.

The good score for development time reflects Doxygen's facilitation of quick documentation setup and generation. Finally, maintenance of Doxygen is relatively straightforward, leading to a good score in this area, as it generally requires little effort to update or adapt to new code, provided the codebase is well-commented.

3.2.2.8 Tree-sitter

Tree-sitter, an open-source parsing system, has been evaluated across a range of criteria for its effectiveness as a dependency graph method. Our research and comparative analysis with other methods has led to the following conclusions.

Tree-sitter scores good for direct extraction because of its ability to parse source code quickly and generate syntax trees in real-time, making it highly efficient at extracting structural information from the code directly.

Its implementation is considered good, as Tree-sitter provides a high-level API that simplifies the integration process. This ease of use extends to the method's learning curve, which is also rated as good due to comprehensive documentation and an active community that supports new users.

When it comes to code interpretation, Tree-sitter performs admirably, offering good precision in understanding the grammatical structure of multiple programming languages. However, for cross-referencing, Tree-sitter is scored as not sufficient. While it excels at parsing, the linking of related code elements across different files or modules is not inherently within its primary functionality.

Tree-sitter handles internal expressions well by providing detailed parse trees that capture the nuances of code semantics, earning a good rating in this area. The tool is compiler independent, which also receives a good score, as it is designed to work independently of any specific compiler's AST format.

Regarding RAM demand, Tree-sitter is highly optimized for performance, which includes memory efficiency, thus scoring good. Its stability is equally rated as good; the project is actively maintained and has a robust design, which ensures consistent parsing across versions.

Tree-sitter boasts extensive language support, with parsers available for many programming languages, contributing to a good rating. In terms of AST extraction, it excels by creating detailed and editable syntax trees, essential for advanced code analysis, which also scores good.

For the role of a code analyzer, Tree-sitter provides a solid foundation by enabling syntax-aware code navigation and analysis, thus receiving a good score. The good ratings for development time and low maintenance underscore Tree-sitter's overall efficiency and ease of use, making it an attractive option for developers seeking to incorporate syntax-aware tooling into their workflows.

4. Conclusions & Outlook

4.1 Evaluation of current findings

The comprehensive evaluation is visually summarized in Figure 5, where a green field for a method is equal to stating a high degree of fulfillment for the specific criteria and evaluated as a +1. Subsequently a red field is describing a low degree of fulfillment for the criteria (-1) and a grey field is neutral and not contributing to the score.

In This comparison of various, selected dependency graph methods it has become evident that no single method offers a complete "out of the box" solution for automotive use cases, specifically in the compliance management regarded in our work of the SofDCar project. Each evaluated method, while possessing unique strengths, invariably falls short in at least one key aspect crucial for the stringent demands of the automotive industry. This shortcoming is particularly pronounced in the context of compliance management, where the requirements extend beyond mere code analysis to encompass regulatory adherence, traceability, and comprehensive documentation.

Our unweighted comparison across a spectrum of criteria reveals that none of the methods singularly fulfills all the needs for these specific automotive applications. Consequently, the most prudent path forward is not reliance on a single method but the development of a bespoke solution that synergistically combines the strengths of two distinct methods: Doxygen and Tree-sitter.

Doxygen's proficiency in generating detailed documentation and its capability for cross-referencing make it invaluable for maintaining compliance records and understanding the interrelations within complex automotive software systems. Tree-sitter, on the other hand, excels in its parsing efficiency, language support, and code analysis capabilities. This makes it exceptionally suited for understanding the nuanced syntax and structure of diverse codebases, a critical requirement in compliance management.

By integrating the documentation and cross-referencing capabilities of Doxygen with the syntactic precision and versatility of Tree-sitter, the resulting tailored solution could effectively bridge the gaps identified in individual methods. This hybrid approach would leverage Doxygen's ability to generate compliance-friendly documentation and Tree-sitter's astute code analysis to create a more holistic, efficient, and compliant workflow for automotive software development and management.

Figure 5: Overview-matrix summarizing the comparison of the dependency graph methods, where red equals -1 and green +1, grey fields neither add nor subtract to the score

4.2 Future areas of application

In the automotive industry, the relevance and potential of dependency graph modeling methods are markedly amplified when considering the existing and upcoming regulations and standards that govern software updates and changes. These regulatory frameworks necessitate a meticulous approach to tracking, evaluating, and managing software modifications, an area where dependency graph methods can play a pivotal role. The growing emphasis on software integrity, safety, and compliance in automotive systems, particularly with the advent of connected and autonomous vehicles, positions these methods as indispensable tools in (semi-)automated compliance management.

The ability of dependency graph methods to map out and visualize complex interdependencies within software architecture provides an unparalleled advantage. This visualization is crucial for understanding the impact of software changes, whether they are minor updates or significant overhauls. It allows for a systematic assessment of how alterations in one part of the system might affect other components, a process essential for maintaining compliance with stringent automotive standards.

Moreover, these methods facilitate the automation of compliance checks, streamlining the verification process to ensure that any software modification aligns with the relevant regulatory requirements. This automation not only enhances efficiency but also reduces the likelihood of human error, a significant factor in compliance management.

As the industry continues to grapple with the challenges of software-centric automotive systems, the adoption of dependency graph modeling methods in compliance management workflows is not just beneficial but increasingly becoming a necessity. Their capacity to adapt to the dynamic nature of automotive software, coupled with their ability to provide clear, actionable insights, makes them

invaluable assets in navigating the complex regulatory environment. Consequently, these methods are poised to find extensive application in the industry, underscoring their importance in the current and future landscape of automotive software development and maintenance.

4.3 Limitations, Success factors and required improvements

In evaluating the dependency graph modeling methods for automotive use cases, it's crucial to acknowledge their limitations, success factors, and areas requiring improvement.

A notable limitation lies in their applicability to non-standardized code. These methods often struggle to efficiently parse and analyze code that deviates from standard conventions, necessitating manual intervention to tailor solutions for specific, non-standardized scenarios. This requirement for manual customization can significantly hamper the scalability and efficiency of these methods in diverse coding environments.

However, the success factors of these methods are equally compelling. They offer unparalleled transparency in the visualization of code and code changes, which is instrumental in the automotive industry's collaborative ecosystem involving OEMs, suppliers, and regulatory authorities. This transparency, coupled with the protection of intellectual property, facilitates a harmonious balance between openness and confidentiality, essential for trust and cooperation across different stakeholders.

In terms of required improvements, scalability emerges as a primary concern. The methods must evolve to handle a broader spectrum of use cases, particularly in testing and regulation compliance checking. This includes enhancing their capacity to manage larger, more complex codebases and diversified coding standards, thereby reducing reliance on manual customization. Another critical area for improvement is the enhanced handling of non-standardized code. By developing more sophisticated and adaptable parsing algorithms, these methods can become more universally applicable, catering to the varied and evolving coding practices prevalent in the automotive sector.

Overall, while dependency graph modeling methods hold significant promise in automotive software management, their evolution must be aligned with the industry's dynamic coding practices and regulatory landscape. This alignment will ensure that they remain not just relevant but also indispensable tools in the future of automotive software development and compliance management.

III References

Name	Content
[SofDCar]	https://sofdcar.de/language/en/consortium_en/
[Dep_Graph_1]	https://en.wikipedia.org/wiki/Dependency_graph
[Dep_Graph_2]	https://docs.github.com/de/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph
[DOT]	https://graphviz.org/doc/info/lang.html
[GML]	https://docs.yworks.com/yfiles/doc/developers-guide/gml.html
[Graphviz]	https://graphviz.org/
[yEd]	https://www.yworks.com/products/yed
[Gephi]	https://gephi.org/
[Regular Expressions]	https://docs.python.org/3/library/re.html
[String Splitting]	https://www.geeksforgeeks.org/python-string-split/
[Clang]	https://clang.llvm.org/docs/index.html
[LLVM project]	https://llvm.org/docs/
[libclang]	https://github.com/sighingnow/libclang https://pypi.org/project/libclang/
[Pycparser]	https://github.com/eliben/pycparser
[gcc-python-plugin]	https://gcc-python-plugin.readthedocs.io/en/latest/
[Pygccxml]	https://pygccxml.readthedocs.io/en/develop/
[Doxygen]	https://www.doxygen.nl/
[Tree-sitter]	https://tree-sitter.github.io/tree-sitter/

IV Figure list