SOFDCAR CONSORTIUM

WHITEPAPER

Software Update Management System

„ Modeling Change Propagation in Software Systems through Dependency Graphs."

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

Published by:	**P3 digital services GmbH**
Authors:	**L. Marks, D. Bartuseck, T. Müller**
Issue date:	**December, 19th 2024**

Abstract

The Software Update Management System addresses the challenges of multi-language dependency analysis and secure communication in software update processes. By generating detailed dependency graphs, modelling change propagation, and providing a secure, interactive web-based platform, the system enhances transparency and collaboration while safeguarding proprietary information.

It incorporates a universal compiler built with Rust, supporting diverse programming languages, and a custom graph viewer optimized for exploring software dependencies. Evaluated on the ECU software of the Flex-Car research project across Java, Python, C++ and Go, the system operates without relying on a compiler and can handle invalid or incomplete code, offering greater flexibility compared to Clang and superior accuracy and functionality compared to Doxygen.

Key strengths include its customization potential and robust multi-language support, though scalability and visualization challenges were noted for larger codebases. Future directions focus on optimizing graph visualization with dynamic zooming and layout algorithms, expanding language integration, and improving computational efficiency.

This work establishes a theoretical foundation for secure and scalable dependency analysis, with applications across industries requiring complex software management. During the SofDCar consortium a prototypical system for demonstration and validation of the research findings has been created.

Table of Contents

1. Introduction

Modern vehicles are increasingly reliant on electronic control units (ECUs) for functionality ranging from basic operations to advanced driver assistance systems. As the complexity of software in ECUs grows, the management of updates has become a critical challenge. Original Equipment Manufacturers (OEMs) currently depend on product concept catalogues to communicate software updates to suppliers. However, these catalogues often do not capture the detailed impact of changes in interconnected ECUs. This lack of granularity can result in inefficiencies, miscommunication, and risks to the reliability of downstream systems.

A significant gap exists in the current processes used to manage software updates in ECUs. Effective communication between OEMs and suppliers is essential, but it must balance the need for transparency with the imperative to safeguard proprietary information. Suppliers require precise details on the impacts of software changes on their systems, but the exchange of such details is hindered by the risk of exposing sensitive source code. This challenge is further complicated by the need for update mechanisms to operate seamlessly across multiplatform environments and by the heightened security risks associated with sharing low-level implementation details.

To address these issues, this work presents a Software Update Management System (also referred to as “the system”) designed to improve transparency and precision in software update communications while maintaining robust security. The system is supposed to enable detailed tracking of software changes at the source code level and employ a dependency graph-based model to represent how the changes propagate across different ECUs. By integrating these features, the system is supposed to facilitate accurate and effective communication of update impacts between OEMs and suppliers without requiring the exchange of proprietary source code, specifications or other accompanying information.

The primary contribution of this research paper is to evaluate and model different alternatives development for a secure, web-based application that bridges the gap between the need for detailed update information and the protection of intellectual property and data security. The system is designed to enhance transparency in software update communication by generating dependency graphs that illustrate impacts of changes across interconnected, complex ECUs.

Moreover, it ensures that sensitive source code is not exposed during the process, preserving security and IP-related requirements. Moreover, a web application architecture could ensure multi-platform compatibility, making the future solution accessible and practical for diverse environments and different industries. The described approach also has the potential to streamline update management workflows and improve reliability of software systems in modern vehicles.

2. Fundamentals

Software Update Mechanisms in ECUs

Electronic Control Units (ECUs) serve as the backbone of modern vehicle functionality, encompassing tasks ranging from basic engine control to complex systems such as adaptive cruise control. Software updates for ECUs are essential for maintaining system performance, addressing security vulnerabilities, and introducing new features. Traditionally, these updates are distributed using over-the-air (OTA) mechanisms or through service centers. However, the intricate dependencies among ECUs complicate this process. Updates to one ECU may inadvertently impact others, potentially causing system-wide disruptions.

Change Propagation in Software Systems

Change propagation is a critical aspect of software engineering, especially in systems with high levels of interdependency, such as ECUs. A single modification in the source code of one component can have cascading effects on related components. Dependency graphs are an established method for modeling such relationships.

A dependency graph $G = (V, E)$ is a directed graph where:

- V represents the set of vertices (nodes), each corresponding to a software component, such as an ECU or a module within an ECU.
- $E \subseteq V \times V$ represents the set of directed edges, where an edge $(u, v) \in E$ indicates that the component v depends on component u.

Each edge can also be annotated with weights $w(u, v)$ to denote the degree of dependency or the impact of a change in u on v. For instance, a higher weight may represent a stronger or more critical dependency.

To enhance the expressiveness of the model, each node $v \in V$ and each edge $e \in E$ is associated with an n-dimensional vector $\boldsymbol{a}_v$ and $\boldsymbol{a}_e$, respectively. These vectors encode key attributes relevant to software updates and dependencies:

- Nodes ($\boldsymbol{a}_v$): Attributes include properties such as the name of the software component, version information, data type, and a time-stamp indicating when the node was last modified.
- Edges ($\boldsymbol{a}_e$): Attributes include the type of connection (e.g., functional dependency, data exchange), the nature of the data being exchanged, time-stamps for when the relationship was last updated and other metadata.

These attribute vectors enable the system to perform fine-grained analyses, such as identifying specific kinds of dependencies or filtering nodes and edges based on their characteristics. The inclusion of this additional information supports more precise impact analysis and facilitates better communication between stakeholders.

Common operations on dependency graphs include:

- Transitive closure: Determines the full set of components affected by a change in any given component.
- Cycle detection: Identifies circular dependencies, which are problematic in update propagation.
- Impact analysis: Quantifies the potential spread of a change by analyzing the paths originating from a given node.

By using dependency graphs, developers can systematically evaluate the consequences of changes and devise strategies to mitigate unintended effects.

Product Concept Catalogues and Their Role

In the automotive industry, OEMs rely on product concept catalogues to manage the communication of software updates with suppliers. These catalogues describe the functional and operational aspects of ECUs and serve as a reference for assessing the impact of changes. While they provide a high-level view, product concept catalogues often lack the granularity needed to capture the intricate effects of software updates on interdependent ECUs. This limitation underscores the need for a more precise and detailed approach to managing software update impacts.

Importance of Source Code Analysis

Source code analysis plays a vital role in understanding the behavior and dependencies of software components. By examining the source code, developers can identify relationships between components, determine the impact of changes and predict potential issues. However, in collaborative environments involving multiple stakeholders, sharing raw source code is often impractical due to intellectual property concerns and security risks. This challenge necessitates alternative methods for extracting and communicating dependency information without exposing the underlying source code.

Web-Based Application for Multi-Platform Support

In today’s increasingly connected world, software systems must support a wide range of platforms and devices. A web-based application is an ideal choice for managing software updates in ECUs, as it ensures compatibility across different operating systems and devices. By leveraging web technologies, the proposed prototypical system provides a user-friendly and universally accessible interface, facilitating seamless interaction between OEMs and suppliers.

3. Related Work

Tools and Insights for Dependency Analysis in Software Systems

Dependency analysis is a critical aspect of software engineering, with various tools and frameworks developed to manage and visualize dependencies. These tools have been instrumental in understanding and optimizing software structures, yet they often fall short in addressing the needs of multi-language systems or complex interdependencies.

For language-specific analysis, tools such as Snakefood and pydeps generate dependency graphs for Python codebases, offering insights into module imports. Similarly, JDepend and Classycle provide dependency analysis and design quality metrics for Java, while tools like Include-What-You-Use and Doxygen assist in managing header file dependencies in C and C++ projects. Doxygen, in particular, generates class hierarchies, function call graphs, and collaboration diagrams, providing a structured view of software systems. However, its dependency visualizations are static and lack granularity, such as connection types or time-stamped changes, making it unsuitable for dynamic or multi-language environments. Tools like CodeTriage DepGraph cater to Ruby projects but are confined to that specific ecosystem.

Advanced tools such as Structure101, CodeScene, SonarQube, and SourceTrail extend beyond single languages by offering code complexity analysis and dependency visualization. Despite their capabilities, these tools often struggle with integrating dependencies across multiple programming languages or require extensive customization to handle diverse systems effectively.

Academic research complements these tools by addressing their limitations and proposing methodologies for managing dependencies in multi-language systems. Callo Arias et al. (2011)[1] conducted a systematic review of dependency analysis solutions, emphasizing the challenges posed by complex interconnections and the need for explicit dependency representation. Abdellatif et al. (2020)[2] introduced techniques for analyzing inter-language dependencies, focusing on both static and historical approaches to unify dependency data across heterogeneous systems. Similarly, Shatnawi et al. (2019)[3] proposed foundational requirements for static analysis in multi-language environments, highlighting the difficulty of integrating analysis across diverse languages.

[1] https://doi.org/10.1007/s10664-011-9158-8

[2] https://doi.org/10.1109/QRS51102.2020.00070

[3] https://arxiv.org/abs/1906.00815

More theoretical perspectives, such as those offered by Blanco et al. (2022)[4], define multi-language semantics and explore the challenges of static analysis in heterogeneous systems. Their work provides a formal foundation for future tools capable of bridging the gap between languages. Finally, real-world studies, such as the catalog of unintended software dependencies in Multi-Lingual Systems at ASML by Groot et. al. (2024)[5], shed light on practical challenges like unexpected interactions between components written in different languages and offer actionable recommendations for mitigating these issues.

Limitations of Existing Tools and Approaches

Despite the variety of tools and methodologies, several key challenges persist. Most existing tools are language-specific, making them unsuitable for modern software systems that often incorporate multiple languages. Even advanced tools and frameworks lack the capacity to integrate dependency data seamlessly across diverse programming languages. Additionally, current solutions often provide high-level overviews of dependencies without capturing detailed attributes such as connection types, data formats, or time-stamped changes, which are critical for understanding the impact of software updates. Scalability is another significant issue, as tools designed for single-language projects often struggle to handle the complexity of multi-language, interconnected systems.

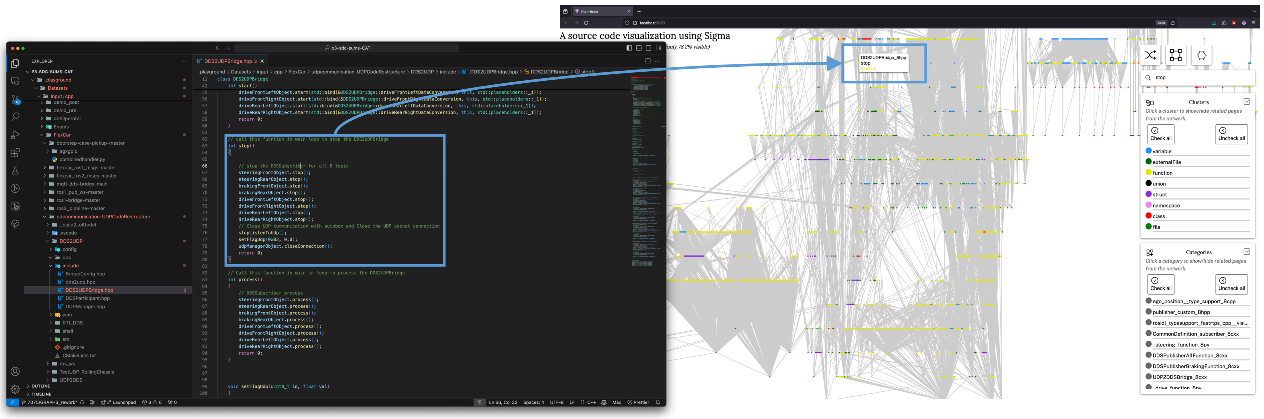
Approach

To address these limitations, we researched on a prototypical Software Update Management System designed for multi-language support and detailed dependency analysis. By merging dependency graphs generated from source code written in different languages, the prototypical approach provides a comprehensive view of the system. Furthermore, system captures detailed attributes of dependencies, such as connection types, data formats and timestamps, ensuring precise and actionable insights into change propagation. The approach bridges the gap left by existing tools, enabling seamless communication between OEMs and suppliers while maintaining security, IP constraints as well as transparency.

[4] https://doi.org/10.1007/s10703-022-00405-8

[5] https://doi.org/10.1145/3639477.3639725

4. Methodology



```
class DDS2UDPBridge
    int start()
        driveFrontLeftObject.start(std::bind(&DDS2UDPBridge::driveFrontLeftDataConversion, this, std::placeholders::_1));
        driveFrontRightObject.start(std::bind(&DDS2UDPBridge::driveFrontRightDataConversion, this, std::placeholders::_1));
        driveRearLeftObject.start(std::bind(&DDS2UDPBridge::driveRearLeftDataConversion, this, std::placeholders::_1));
        driveRearRightObject.start(std::bind(&DDS2UDPBridge::driveRearRightDataConversion, this, std::placeholders::_1));
        return 0;
    }

    // call this fucntion in main loop to stop the DDS2UDPBridge
    int stop()
    {


        // stop the DDSSubscriber for all 8 topic
        steeringFrontObject.stop();
        steeringRearObject.stop();
        brakingFrontObject.stop();
        brakingRearObject.stop();
        driveFrontLeftObject.stop();
        driveFrontRightObject.stop();
        driveRearLeftObject.stop();
        driveRearRightObject.stop();
        // Close UDP communication with autobox and Close the UDP socket connection
        stopListenToUdp();
        setFlagUdp(0x03, 0.0);
        udpManagerObject.closeConnection();
        return 0;
    }
    // Call this function in main in loop to process the DDS2UDPBridge
    int process()
    {
        // DDSSubscriber process
        steeringFrontObject.process();
        steeringRearObject.process();
        brakingFrontObject.process();
        brakingRearObject.process();
        driveFrontLeftObject.process();
        driveFrontRightObject.process();
        driveRearLeftObject.process();
        driveRearRightObject.process();
        return 0;
    }



    void setFlagUdp(uint8_t id, float val)
    {
```

p3-sdc-sums-CAT
DDS2UDPBridge.hpp
.playground > Datasets > input > cpp > FlexCar > udpcommunication-UDPCodeRestructure > DDS2UDP > include > DDS2UDPBridge.hpp > DDS2UDPBridge
Ln 66, Col 32
Spaces: 4
UTF-8
LF
A source code visualization using Sigma
Clusters
Check all
Uncheck all
variable
externalFile
function
union
struct
namespace
class
file
Categories

Figure 1: Transformation of source Code into dependency graphs (prototype application)

System Overview

The modeled Software Update Management System is designed to investigate and address challenges in multi-language dependency analysis and secure communication of software updates. By generating dependency graphs directly from source code (see Figure 1), modeling change propagation, and providing a secure web-based interface, the system bridges the gap between software insights and the need to protect intellectual property. The key components of the demonstrator include a multi-language compiler, techniques for dependency graph generation, algorithms for change propagation and a custom-built graph viewer optimized for software exploration.

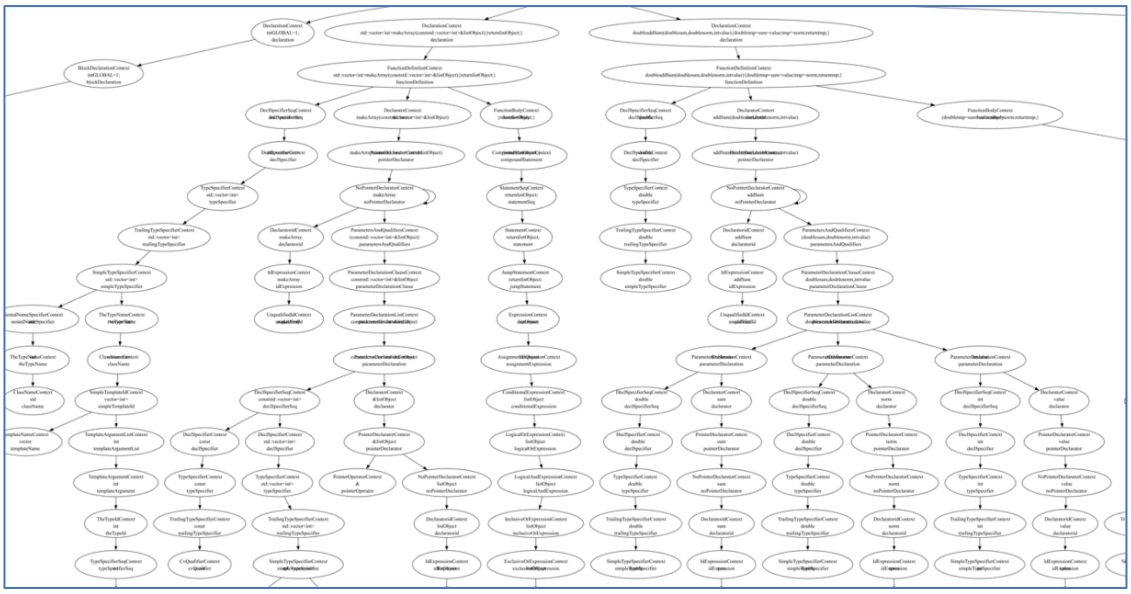


Figure 2: Exemplary CST of source Code (cropped)

Generation of Dependency Graphs

The system parses source code from various programming languages and generates Concrete Syntax Trees (CSTs) (see Figure 2). This parsing process analyzes all source files to ensure complete coverage. From these CSTs, dependency graphs are created (see Figure 3), with each node and edge enriched by an n-dimensional vector that includes attributes such as memory allocation details, timestamps and data types.

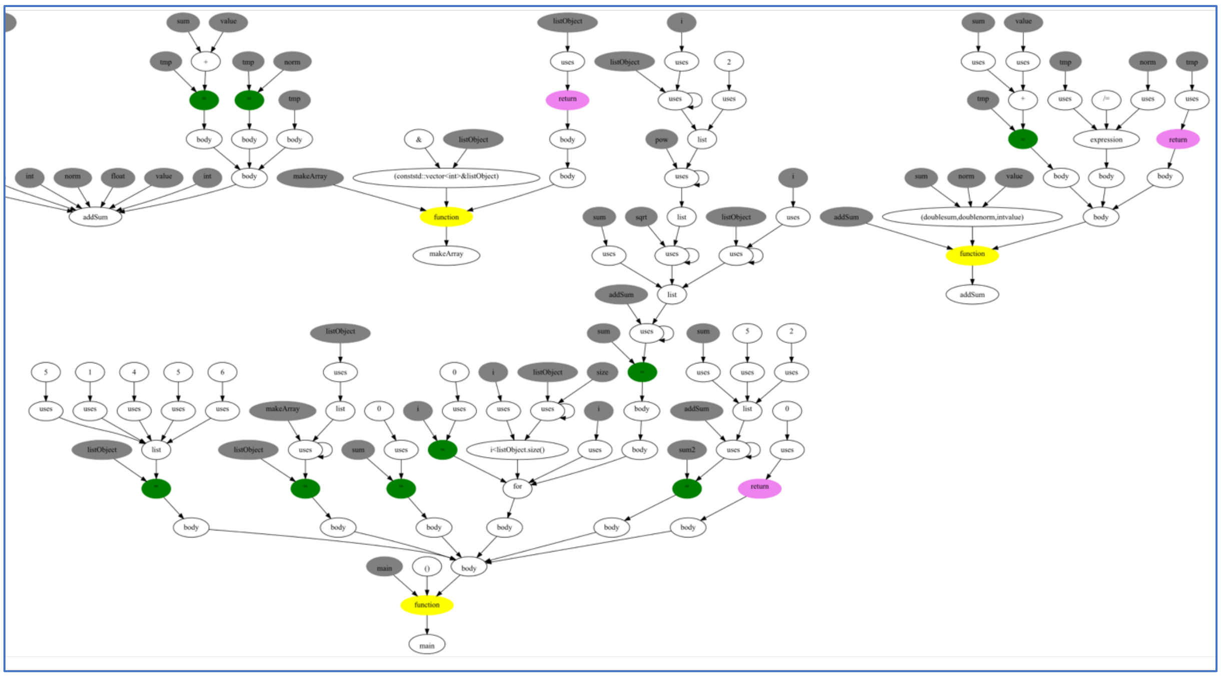


Figure 3: First step : CST extraction (copped picture)

To handle multi-language systems, the demonstration system serves as a universal compiler by defining language-specific grammars. Dependency graphs are abstracted to prevent inversion, achieved by merging edges or converting groups of nodes into single edges (see Figure 4). Sensitive details are hidden in “metadata”, ensuring the graphs are useful for analysis without exposing proprietary code.

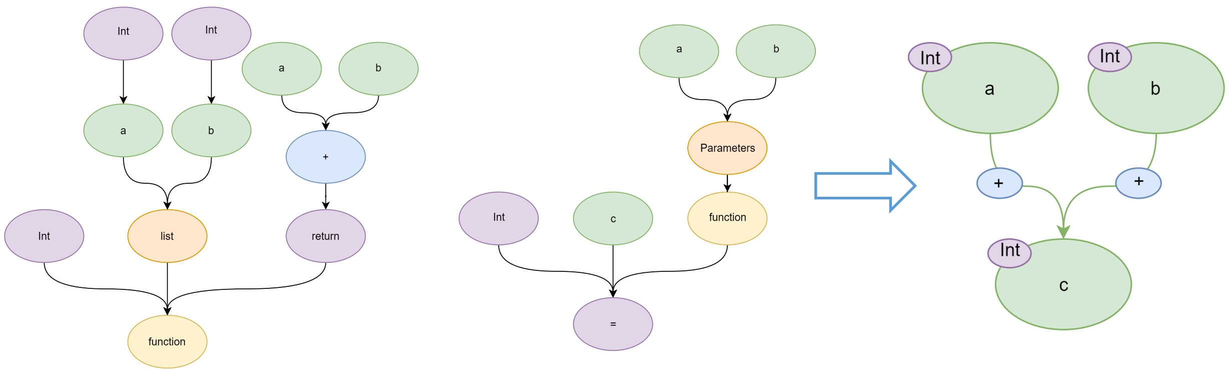


Figure 4: Simplified abstraction process (3-step approach)

Web Application Architecture

The demonstration system is implemented as a web-based application for accessibility across platforms. The backend manages server-side operations and APIs. The frontend provides an interface for users to interact with dependency graphs.

A custom graph viewer powers near real-time visualization capabilities which are crucial for a potential practical use of the viewer by software and test engineers (see Figure 5). The use of Rust ensures sufficient performance for real-time rendering of complex dependency graphs and an efficient exploration and modification of the graphs.

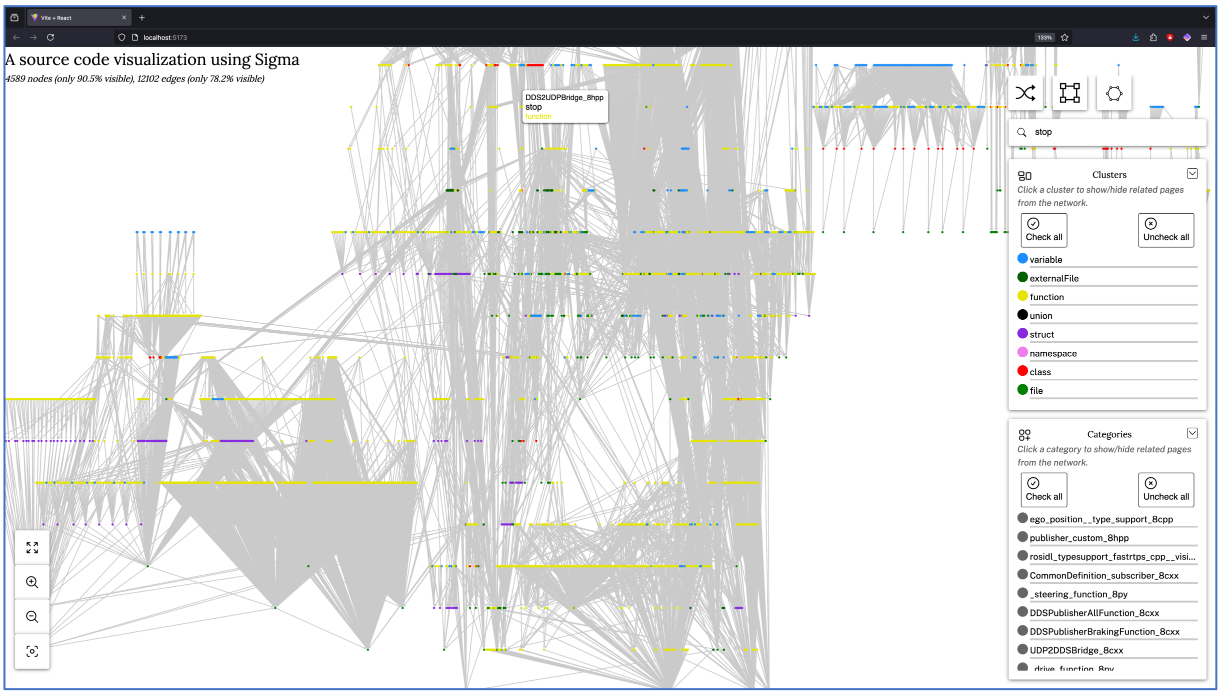

Vite + React
localhost:5173
133%
A source code visualization using Sigma
4589 nodes (only 90.5% visible), 12102 edges (only 78.2% visible)
DDS2UDPBridge_8hpp
stop
function
stop
Clusters
Click a cluster to show/hide related pages from the network.
Check all
Uncheck all
variable
externalFile
function
union
struct
namespace
class
file
Categories
Click a category to show/hide related pages from the network.
Check all
Uncheck all
ego_position__type_support_8cpp
publisher_custom_8hpp
rosidl_typesupport_fastrtps_cpp__visi...
CommonDefinition_subscriber_8cxx
_steering_function_8py
DDSPublisherAllFunction_8cxx
DDSPublisherBrakingFunction_8cxx
UDP2DDSBridge_8cxx
_drive_function_8py

Figure 5: Visualization of tool prototype / graph viewer

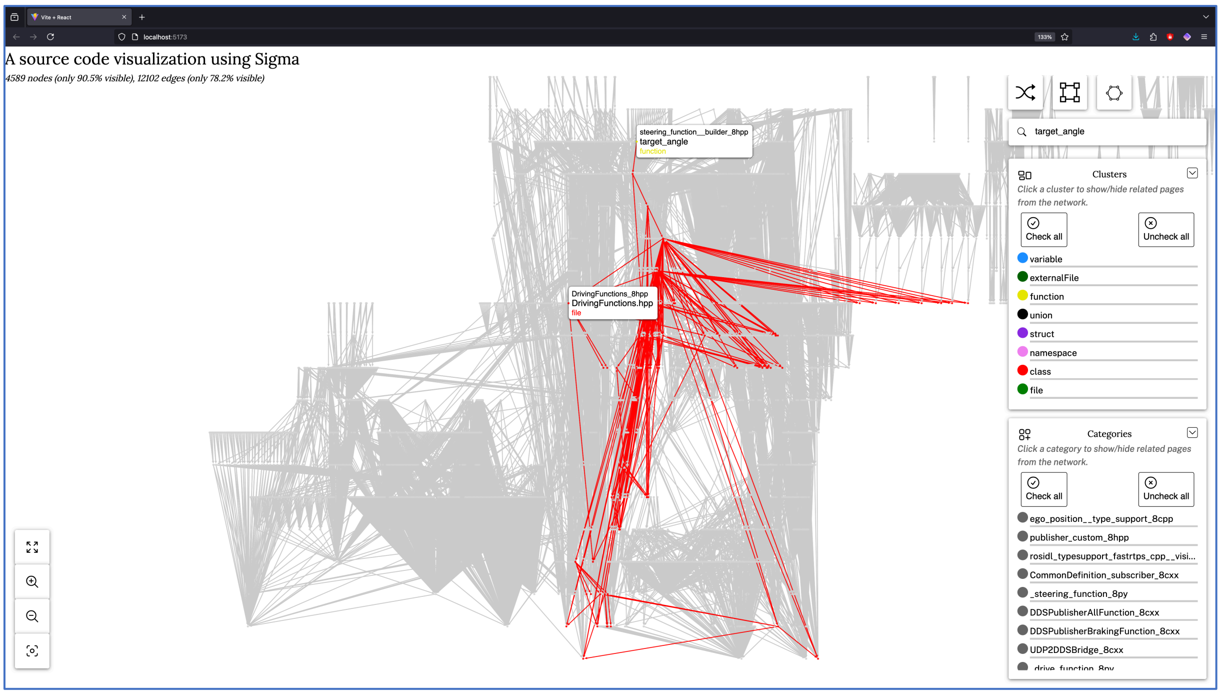
Modeling Change Propagation

The system can use combinations of pathfinding algorithms, graph traversal techniques, and methods from binary addition and Set Theory to model change propagation (see Figure 6).

These techniques enable the demonstration system to:

- Identify **"components" affected** by a change (graph traversal)
- Quantify the **cumulative impact** of changes (binary addition)
- Manage **overlapping dependencies** and maintain consistency (set theory operations)

The algorithms handle edge cases such as cyclic dependencies or disconnected components, providing accurate modeling of real-world software systems. These methods offer detailed insights into the effects of software updates, aiding in in-depth analysis, decision-making and planning.



A source code visualization using Sigma

4589 nodes (only 90.5% visible), 12102 edges (only 78.2% visible)

Vite + React
localhost:5173
133%
steering_function__builder_8hpp
target_angle
function
DrivingFunctions_8hpp
DrivingFunctions.hpp
file
target_angle
Clusters
Click a cluster to show/hide related pages from the network.
Check all
Uncheck all
variable
externalFile
function
union
struct
namespace
class
file
Categories
Click a category to show/hide related pages from the network.
Check all
Uncheck all
ego_position__type_support_8cpp
publisher_custom_8hpp
rosidl_typesupport_fastrtps_cpp__visi...
CommonDefinition_subscriber_8cxx
_steering_function_8py
DDSPublisherAllFunction_8cxx
DDSPublisherBrakingFunction_8cxx
UDP2DDSBridge_8cxx

Figure 6: Visualized Change Propagation (example project)

Security and Data Privacy

The system architecture is capable of confidentiality for source code by abstracting dependency graphs into non-invertible representation. Techniques such as edge merging, node-to-edge conversion and metadata obfuscation obscure the original structure of the source code. Inner vector components of the graphs are encrypted, with decryption keys accessible only to the custom graph viewer.

The dual-layered security approach of the demonstration system balances transparency and confidentiality, protecting intellectual property while enabling effective and efficient collaboration between different parties.

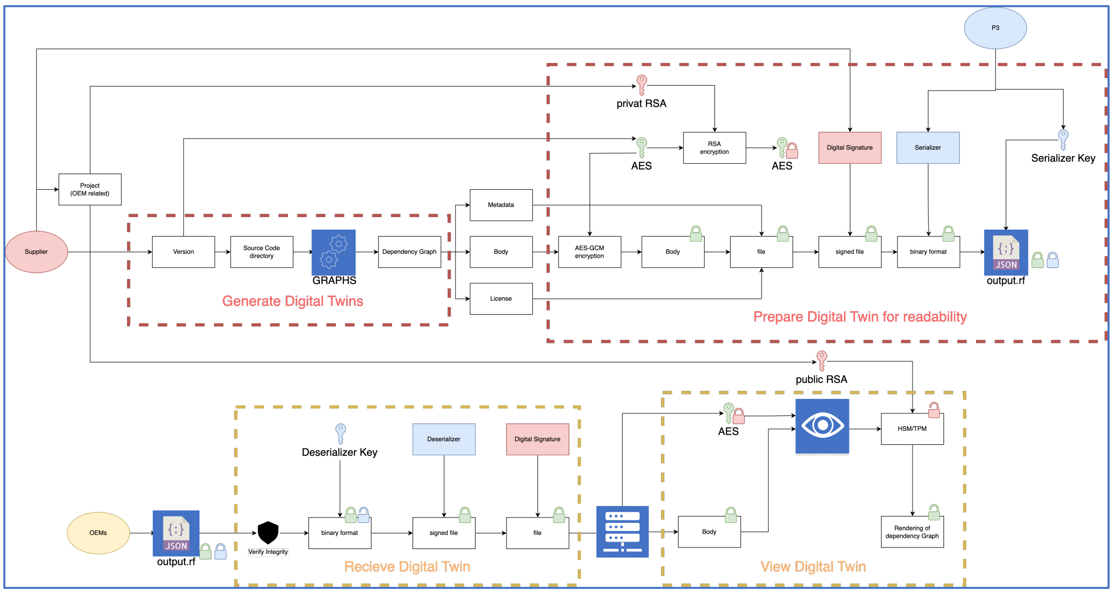


Figure 7: Multi-Layered System architecture for Secure Digital Twins

The overall concept for the system allows secure data sharing through the internet by implementing advanced encryption and verification mechanisms (see Figure 7). Before sharing, digital twins are encrypted by various encryption methods to safeguard their integrity and confidentiality. The encryption secures the metadata and decryption keys as well the main data payload, ensuring a holistic and robust data protection approach. The digital twin is also signed digitally, providing cryptographic proof of its origin and integrity of data and source.

To enhance usability, the system converts data into a different file format for structured transmissions. This enables seamless integration with existing systems while maintaining strict security and readability protocols. By combining encryption, digital signatures, and secure data formats, the system establishes a trusted pipeline for sharing sensitive data over the internet.

Researched Features and Innovation

For the demonstration system innovative languages (e.g Rust) have been evaluated in order to improve computational efficiency and scalability for future use in real ECU network environments to enable fast parsing of diverse codebases and real-time visualization of large dependency graphs / networks. This enabled the system already during research phase to perform even well with complex multi-language codebases.

By integrating various success factors and the capabilities described above into a cohesive framework, future systems can provide scalable and valuable solutions for dependency analysis and software update management. The evaluation of the different methodologies and approaches resulted in a demonstrating system, that needs to be developed further to become a tool for OEMs and their suppliers in software driven industries.

5. Results

Evaluation Scenarios

The proposed system was evaluated on multiple programming languages (Java, Python, C++, Go) to demonstrate its multi-language capabilities. Testing was conducted on ECU software of the Flex-Car research project, which provided a complex and realistic use case for multi-language systems. The evaluation criteria and test cases included generation of dependency graphs, modeling of change propagation and visualization of relevant outputs using the custom prototype graph viewer.

Evaluation Metrics

The system's performance was assessed using two primary metrics:

1. Memory Efficiency: The ability of the system to process and visualize large codebases without excessive memory consumption.
2. Accuracy: The precision of the generated dependency graphs and change propagation models, measured against expected outputs and manual verification.

Comparative Analysis

The system was compared against two established tools (Doxygen, Clang), which are widely used for dependency analysis and code visualization. The comparison revealed several strengths of the new system approach:

- The prototypical compiler outperformed Doxygen and Clang in handling diverse programming languages while maintaining higher accuracy in dependency graphs.
- The system's ability to abstract and encrypt sensitive details in the graphs provided a significant advantage in terms of security.

However, certain challenges were noted that need to be addressed by future development:

- Performance Decline with Larger Codebases: As the number of source code components was increased during research, visualization became cluttered and less intuitive – thus limiting its usability.
- Decreasing Computational Performance: The prototype system (demonstrator) exhibited a drop in processing speed as the codebase size grew.

Visualizations

The prototype system was used to evaluate alternative visualization options by showcasing its capabilities and compare alternative GUI formats.

Dependency graphs can provide an enhanced view on relationships between various software components, capture key attributes (such as memory allocation, data flow, timestamps) for changes. These graphs were designed to enable developers and other stakeholders to understand the intricate dependencies in the software, offering valuable insights into how individual components interacted.

Change propagation visualizations is supposed to add further dimensions by illustrating how modifications in one part of the code could or actually do affect others. These “views” support the dynamic nature of software systems helping to predict and manage potential ripple effects of updates.

Additionally, the custom graph viewer demonstrated its interactivity and flexibility by allowing users to explore these visualizations in a tailored manner. Features such as zooming, filtering, and metadata access were seamlessly integrated, providing users with the tools necessary to focus on specific areas of interest within the dependency graphs. The visualizations, while powerful, revealed challenges in handling larger codebases, as the increasing complexity led to cluttered representations that became less intuitive. These insights underscored the need for continued optimization to enhance clarity and usability for larger projects.

Observations and Insights

While the system demonstrated robust multi-language support and superior accuracy compared to existing tools, several key observations emerged:

- Scalability Challenges: Visual clutter becomes a significant issue for larger projects, indicating needs for additional features (e.g. graph simplifications)

- Strengthening the Compiler: the prototypical compiler consistently outperformed other established “solutions” in multi-language support and processing capabilities, still reaffirming and challenging of current architecture footprint & code design needs to be further investigated

Previous research findings demonstrated that the demonstrator system excels in its ability to provide accurate and secure dependency analysis for multi-language software systems. A key strength is the high degree of customization within the compiler, allowing new languages to be integrated with “relative ease”. The accuracy of the generated dependency graphs and change propagation models also stands out as a major advantage compared to existing tools.

Despite strengths of the demonstration system, the system concept still faces challenges in two areas: computational performance and graph visualization.

As the size and complexity of codebases increase, the system’s performance declines, and the visualizations become cluttered and harder to interpret - these limitations of the demonstrator indicate the focus of further refinements to ensure usability and scalability in complex real-life projects.

Limitations

The primary limitations identified during the SofDCar consortium include:

- High Expertise Requirements: Integrating a new language into the universal compiler requires advanced knowledge and can be time-intensive.
- Visualization Challenges: The current graph viewer struggles to maintain clarity with larger and more complex codebases, resulting in reduced intuitiveness.
- Performance Bottlenecks: Computation time increases significantly with larger projects, affecting the efficiency of dependency analysis and visualization.

Future Works

To address the above limitations and enhance the system's capabilities, several improvements are proposed and will be focus of further next steps:

1. Pre-computing of Node Positions (as focus topic for graph viewer enhancement – target: more "intuitive" and more performant user experience)

2. Dynamic Zooming Techniques as feature for complexity reduction

3. Language Integration (easy-to-use module for fast integration of new languages)

4. Expanding Language Support (increasing range of supported languages to cover domain-specific systems and emerging programming languages)

5. Overall Performance Optimization

Broader Implications

The system architecture and overall concept have the potential to significantly improve collaboration between OEMs and suppliers by provision of a secure and transparent platform for software update management.

Beyond the automotive domain, the methodology could also be adapted for use in other practices and industries (healthcare, finance), where complex, multi-language systems require robust dependency analysis and change management.

By addressing current limitations and pursuing the proposed future directions, the system can become a versatile and indispensable approach for effective software development in the future.

6. Conclusion

Our research addressed the challenges of multi-language dependency analysis and secure communication in software update management. By developing a demonstrator system that generates accurate and secure dependency graphs, models change propagation and visualizes complex software relationships, it provides a promising basis for a reliable software system for future update management processes.

Key research artefacts included a universal compiler and the custom graph viewer. The integration of advanced algorithms and abstraction techniques ensures a sufficient level of protection of the data and overall systems while maintaining high analytical precision. The system demonstrated superior accuracy and flexibility compared to existing tools and its ability to customize the compiler function for new languages highlights its adaptability.

Performance and visualization challenges were identified, particularly with larger codebases, emphasizing the need for ongoing optimization.

Based on the prototypical solution far-reaching implications for industries reliant on complex, multi-language systems could be derived.

7. List of figures

P3 group

P3 is an independent and international consulting company that offers consulting and engineering services, as well as software development for numerous customers. Since its founding in 1996 in Aachen, Germany, P3 always found new branches and has over 1900 employees in 26 locations in close vicinity of its customers.

Vision - We advise our clients strategically in the areas of technology strategy, business process optimization and organizational development. P3 develops new business models, opens up future revenue sources for the clients and accompanies them in building up competencies.

Perspective - We carry out complex projects and optimize business processes in customer organizations and their global supplier network. P3 supports and empowers customer organizations to create robust structures to operate sustainably and grow in the future.

Insight - We develop & test innovative IT solutions for specific business requirements. From individual solutions to cloud-based IoT services. P3 delivers end-to-end solutions in the area of security consulting and ensures a seamless service and product rollout.