Automated Data Dependency Visualization for Embedded Systems
Programmed in C

JOSIP PANTOVIC

Degree project in Information Technology
Second level, 30.0 HEC
Stockholm, Sweden
Automated Data Dependency Visualization for Embedded Systems Programmed in C

Master Thesis

Last revised: 2014-01-26

Josip Pantovic
Email: josephpanx@gmail.com

Examiner:
Professor Mark T. Smith (KTH)

Supervisor:
Mattias Nyberg (SCANIA)
Acknowledgements

- Professor Mark T. Smith, KTH, Thesis Examiner, for the exhaustive help with defining and directing the thesis as well as shaping this report.

- Mattias Nyberg, Scania REPA Driver Assistance Systems, Thesis Supervisor, for the help with performing the thesis work with his experience with previous related work, design of Scania systems, company needs and organization.

- Professor Christian Schulte, KTH, for the help with better comprehending the scope of the problem area of program parsing and compilation.
# Table of Contents

Abstract ........................................................................................................................................ v

1 Introduction .............................................................................................................................. 1

1.1 Report Description .............................................................................................................. 2

2 Problem .................................................................................................................................. 3

2.1 Problem Description .......................................................................................................... 3

2.1.1 Scania State of Affairs ................................................................................................. 3

2.2 Hypothesis / Problem Statement ...................................................................................... 5

3 Literature Review .................................................................................................................. 7

3.1 Static Analysis ................................................................................................................... 7

3.2 Code Parsing, Abstract Syntax Tree and C Language ....................................................... 8

3.3 Dataflow vs. Data Dependency Analysis ......................................................................... 9

3.4 Visualization ..................................................................................................................... 10

4 Analysis .................................................................................................................................. 12

4.1 Analysis of Data from User Interviews .......................................................................... 12

4.2 Analysis of Related Existing Tools ................................................................................. 14

4.3 Delimiting the Scope of Implementation ....................................................................... 16

5 Implementation ..................................................................................................................... 18

5.1 Previous Work .................................................................................................................. 18

5.2 Toolchain Overview ........................................................................................................ 19

5.3 Stage I: Code Processing ................................................................................................. 20

5.3.1 Discussion .................................................................................................................... 23

5.4 Intermediate Form ........................................................................................................... 23

5.4.1 Discussion ..................................................................................................................... 27
Abstract

This thesis deals with analysis and visualization of data dependencies in embedded system software. The goal was to create a comprehensive, practical analysis tool that fits the needs at Scania well, while being as general and reusable as possible. Given the complexity of problem area, the goal was to position the outcome of this project as the basis for further development, making future extensibility an important consideration.

This type of analysis has previously been performed either manually, or with help of problem-specific and system-specific tools developed by specific users for their own needs. Here, a wider user spectrum was interviewed to collect and analyze their needs in order to delimit the scope of the project and provide fundamentals of a solution that would benefit most users to the largest degree.

Application developed throughout this thesis is of a strongly layered structure, to provide most opportunity for future reuse and extension. In the first layer, XML representation of abstract syntax trees is obtained through source code analysis. This part relies on the work performed as a part of the thesis by Martin Pruscha [23], modifying and extending the tools developed for C code parsing and analysis there.

Intermediate part of the toolchain transforms this into a generalized XML representation of desired data relationships, based on detection of patterns in abstract syntax tree that correspond to code structures that signify global variables and functions within C modules, as well as their mutual dependencies, such as function calls or reading or writing to variables. While actual implementation deals only with inter-functional analysis, issues pertaining to problems necessitating intra-functional analysis of data flow (deeper pointer variable analysis, function pointers, flow of data through non-global variables) are discussed.

In the final stage, based on user preferences selected via a GUI, this format is converted to GraphML format which can be graphically represented in yEd application.

Finally, results and performance were analyzed to provide guidelines to future work. The key conclusion is that while the current tool is applicable for visualizing smaller sub-systems, successful rendering of a large, complex system in detail requires a significant amount of manual post-processing. With that in mind, future projects should consider development of own or application of different existing tools and formats for graphical representation.
1 Introduction

As embedded systems grow in complexity and sophistication, the scope of their possible applications is expanding, establishing them as ubiquitous in a wide range of industries. Automotive industry is a prime example of this. Vehicles produced by Scania today contain a multitude of embedded systems related to many of vehicle’s subsystems, managing a variety of essential and auxiliary aspects of vehicle utilization, from steering to control of exhaust emissions. Each of the embedded systems developed at Scania features a complex software architecture, implemented in C programming language following industry standards like MISRA [2], as well as internally developed standards and recommendations.

Degree of complexity of the systems developed at Scania raises the demand for producing graphical representations of the systems. Graphical system visualizations can provide designers with better system overview which, in turn, may provide benefits like facilitating modular development, reducing potential for errors, or helping introduce new developers to the system.

Graphical system representations are currently being used at Scania, but are mostly manually created which is a complex and time-demanding process. This, in turn, limits the scope of their possible applications; for example, in this framework it is impossible for developers to get a real-time visual representation of how currently developed parts of a system affect the architecture as a whole.

In light of this, development of automated tools for creation of visual representations appears necessary. While there are many existing tools with the same general purpose, a more domain-specific approach focused at particular systems at hand and based on existing practices and manually generated visualizations, with utilization potential as the main principle applied in generation of those representations, could provide a wider range of benefits and better meet the particular company demands at Scania. On the other hand, no detailed formalization of such visualizations has been defined. Keywords like “data flow” and “data dependency” are applied to describe visualized relationships, but those terms are globally commonly interpreted in many different ways, resulting in great variance in actual approach and output of different existing tools aimed at detecting and/or presenting such relationships. It is necessary to focus on clarifying and, throughout implementation, addressing the meaning of applied terms as interpreted in practice within the company.

Another aspect of complexity encountered is the complex structure of a large company itself. At Scania, a multitude of mutually dependent departments deal with product design, development, production, testing, and other aspects of product life. Each of them operates within paradigms in which the hardware and software systems and their components are represented in different ways, with varying levels of abstraction and emphasis on different features. Within a heterogeneous environment like this, maintenance of communication and understanding between departments is not a trivial task. Graphical representation aimed at
being as intuitively comprehensible as possible to observers from an array of applicable departments can be a step towards the goal of having a common language between them.

### 1.1 Report Description

*Chapter 2 – Problem* deals with detailed description of outcomes from this thesis work as initially defined. State of the art in related areas at Scania is described, and a working hypothesis / problem statement is derived.

*Chapter 3 – Literature Review* is aimed to bridge the gap between the practical Scania paradigm and desired outcomes and related theoretical concepts and research. Control flow, data flow, data dependency concepts are analyzed; theoretical considerations related to graphical visualization and graphs in general are investigated.

*Chapter 4 – Analysis* presents results of analysis of various preparatory researches performed as groundwork for future implementation. Existing tools that provide outputs related to thesis requirements are analyzed, to discover if any of them can be directly applicable, avoid redundancy and unnecessary work, and motivate internal development of tools showing the potential for advantages over existing external tools. In-company interviews have been performed and the collected data is analyzed to obtain a good idea on company structure and guidelines for development of a visualization tool. For latter purpose, specifications related to code development and system review are analyzed as well. Finally, based on the analysis, goal and scope of the implementation performed as a part of this thesis are defined in detail.

*Chapter 5 – Implementation* describes the toolchain for code analysis and visualization developed as a part of this thesis. Previous work, existing tools used and techniques are described. Toolchain components are analyzed in detail, as well as the output formats obtained through all stages. Reasons for design choices are discussed throughout.

*Chapter 6 – Conclusions* discusses the quality of obtained results, problems encountered during implementation and possibilities and recommendations for future work.
2 Problem

2.1 Problem Description

2.1.1 Scania State of Affairs

Given the purpose of embedded systems created at Scania, which is control of a vehicle, great emphasis has to be put on system safety. Various internal company standards, as well as global industry standards deal with this and are applied in system design and development. One of the planned company goals is meeting the requirements of ISO 26262 functional safety standard, which is of special interest to this topic because this standard requires the use of unambiguous graphical representations in modeling and coding guidelines [1].

Majority of Scania software for embedded systems is written in C programming language, constrained by industry standards and internal rules and guidelines. MISRA C standard is used, providing rules to be applied with the primary purpose of defining a safer subset of C in which possibilities for unpredictable and implementation dependent behavior are reduced [2].

Developed software systems, as well as the corresponding source code, are organized in a modular structure common to non-trivial C software. Systems are organized into layers (commonly, high and low level application layers and platform layer), which, in turn, consist of managers, that is sets of modules joined by functionality. In source code, layers and managers are usually represented by corresponding folder structure organization of source code, and modules correspond to actual C source code files.

This structure is used as a basis for top-down system design. System specifications determine the functions of subsystems and allowed ways of interaction between them. During development, however, discrepancies from this model can arise, not only through hypothetical accidental errors and imposed time constraints for development, but also through intentional choices made in development process that contribute to code optimization and performance.

Manual design of graphical representations takes considerable time which is taken into account in project planning. They are created when they are needed for a concrete purpose, and may be reused, but the possibility that they are not fully up-to-date limits their use.

Automated system visualization was previously applied by individual developers, with a limited, system-specific domain and tailored for personal use only. A more universally applicable, general approach is required, to meet the needs of diverse types of users.

Finally, software analysis and graphical representation is a problem that has been performed as a part of previous Scania projects, many of them parts of previous thesis works. These were usually specialized for a particular system or purpose, with limited scope of application. Artifacts developed in this way were not often reused or developed further. Given the complexity of the problem, the new approach is aimed at thinking in long term – incremental development moving towards a general solution that could be applied in actual development.
process. This thesis continues on the work started in a previous thesis work by Martin Pruscha [23], utilizing developed artifacts as infrastructure for the source code analysis aspect of the work.

2.1.1.1 EEC3

EEC3 - Exhaust Emission Control system, used for hydraulic control and diagnostics of exhaust emission subsystems [3], was chosen as the exemplar of embedded systems developed in Scania. While there are larger, more complex systems developed at Scania, EEC3 is complex enough to show the key aspects of system organization applied in development of most systems.

EEC3 consists of eight layers, each subdivided into managers which consist of modules. Layers deal with operations on different levels of abstraction. Application layer utilizes functionalities of low level layers such as file handler and system manager layers, which, in turn, utilize functionalities in common platform layer. Apart from this main hierarchy, utilities layer provides common utility functions, while real time database layer is used for signal information exchange between modules in different layers [3].

One of the envisioned goals in development of a graphical representation of a system was to be able to reverse-engineer a top-level diagram which clearly distinguishes separate layers and communication which happens between them through source code analysis. In source code terms, the layers represent topmost packages, i.e. folders directly under the folder designated for EEC3 source code. Managers are sub-packages of those packages. Both layer and manager folders contain C source code files corresponding to modules.

While some of the communication patterns can be obtained from source code in an intuitive manner, that is, from direct communication between source code structures from different layers, this is not always the case. For example, communication performed through the real time database (RTDB) layer is directly represented by access to RTDB functions and variables by modules from other layers in source code, which, in many cases, is well known by the users and unnecessary to emphasize, the main concern being to determine which end layers communicate with each other. Additionally, source code related to the common platform (COMP) layer is developed and compiled separately from EEC3 code. EEC3 source code contains only header source code files for this layer.

Manually generated graphical representations generated by employees well acquainted with the system specification and current version of source code take a long time to create and are hard to update. They provide system-specific information, on a level of abstraction higher than relationships of C code constructs, involving signals, services, and specific messages. They also allow for subdivisions of the system based on the previous knowledge of the system, its purpose and mode of operation as a whole, all of which is not easily distinguishable by direct analysis of source code structures.

Preferred aim of an automated representation is to provide a more general solution, applicable to a wide array of systems and comprehensible to users from a wide scope of backgrounds.
Manually generated diagrams can provide insight towards what is considered a useful graphical representation and what level of complexity of representations is desired in current practice.

2.2 Hypothesis / Problem Statement

There is an expressed interest in usage of graphical system representations within Scania. It is necessary to explore and quantify the degree of that interest in relation to different stages of system design, development and review, as well as the actual scope of applicability of such representations and methods for their efficient integration into aforementioned processes.

Given that currently used manual representations rely exclusively on static code analysis, it is possible to develop useful automated visualizations based the same principle. While dynamic analysis that includes observation of program execution states could provide more data, it would significantly increase the problem complexity and time for development of a useful tool.

Automated generation of system visualization is much faster than manual approach. It is necessary to analyze the potential for new scope of applications for such visualizations, especially within system development process, that the significant increase in speed obtained by switching to an automated process could provide.

In order to obtain a compiled, executable system, compilers obtain unique meaning of source code through parsing and compilation process. Therefore, it is possible to obtain this unique meaning for the purpose of accurate system visualization.

However, source code parsing is a complex task. It is to be expected that full potential of parser-based source code analysis for the purposes of system visualization cannot be covered in its entirety within a single thesis time-frame. Similarly, same source code structures could be visualized in many different ways, each possibly useful for different stages of system design, development and review, many of which could become obvious only after extensive utilization of already developed views. Therefore, to maximize the potential for its application, the developed tool should be implemented in a way that facilitates its extension and modification.

Utilization of parser-based techniques for source code analysis allows robust analysis that could be applicable to any system developed in the same programming language (same parser grammar). On the other hand, more system-specific approach that would limit the scope of applicability of a developed tool could, in turn, yield more meaningful data related to that particular system. The relationship of these opposing principles should be carefully explored to obtain a solution providing a good balance between the quality of obtained data and robustness of scope.

While previous similar works were aimed at focusing on a small segment of a system and a limited range of applications, the approach taken here was to implement a solution that would represent a starting point towards a robust, generalized suite applicable for a wide range of
purposes. This solution provides a range of intermediate artifacts fully reusable in future development, such as abstract syntax tree models of source code and the intermediate format containing easily accessible information applicable to a wide range of purposes. Finally, developed graphical representation achieves the primary aim of demonstrating the potential of this approach. Refinement and optimization of this representation, however, was not the key focus. Instead, third party rendering tools were utilized to the best of their capabilities for this purpose.
3 Literature Review

In order to be able to formulate a clear approach to solving the problem we are faced with, it is necessary to explore the related existing theoretical concepts. It is necessary to unambiguously define the terminology which will be used for defining the problem and solution, given that many concepts related to this topic (e.g. dataflow) are, in practice, interpreted in many different ways.

3.1 Static Analysis

Two complementary approaches to program analysis are static program analysis and dynamic program analysis or dynamic testing [4]. Dynamic analysis involves execution of a program, tracking of values in operative memory, execution time, input/output causation etc. It provides information on empirical program behavior. Static analysis refers to analysis of static artifacts of a program, from source code, through intermediate to executable files (machine code).

Experienced developers and system designers can employ both static and dynamic analysis in manual system analysis, by not only observing the code, but utilizing previous knowledge from testing and specifications. For automated analysis of program structure, it is necessary to carefully examine the intricacies and limitations of these approaches in order to determine the most efficient way to reach the best possible solution. Previous in-company work on the problem at hand implicitly hints towards focusing on static analysis, but it is desirable to observe differences between two approaches to justify this and better understand its meaning.

Dynamic analysis provides access to data which is beyond the scope of static analysis, providing precise information on actual program behavior for a given test case [5]. It is theoretically conceivable that the program structure could be obtained through purely dynamic approach: by tracking a sufficient number of a sufficiently varied sets of test cases while observing the instruction pointer of the program and all data affected by each step. However, for a complex system, this is impossible to achieve in practice. To obtain reliable results, a test case concept would need to cover full context of the system on which a program is run and all input and intermediate values would need to be known or deterministically discoverable. For a given test case, dynamic aspects of solving this can be described as progression from observational program analysis, involving a single program execution, through inductive program analysis, observing multiple executions, to experimental program analysis where program is actively observed and manipulated during multiple executions, where each approach relies and expands on the previous one [6]. All of these, however, may also rely on deductive program analysis, that is, static analysis performed independently of program execution [6], which becomes necessary for a complex program: it is not feasible to generate test cases for all possible values of all possible variables used. Therefore, in practice, static analysis becomes necessary for efficient implementation of dynamic analysis, making static analysis mandatory for in-depth analysis.
The scope of data covered by static analysis is, in principle, smaller than that of dynamic analysis, making static analysis inherently less powerful in terms of precision of obtained information – it can only describe what it can analyze. On the other hand, it is possible to perform static analysis in a way that provides reliable data within a well-defined context [5] – that which it can analyze. It can provide a more manageable, simpler collection of information than dynamic analysis, information which can be directly linked to how the analysis is defined and what kinds of structures are covered by the analysis.

Static analysis may include dynamic aspects, such as variations in program behavior depending on an abstract program-specific state [5] (e.g. providing sets of data separated by whether a variable used in a condition fulfills the condition), which is still considered static analysis because the program is not explicitly run, but only parts of it simulated within the very process of analysis. However, by expanding on this by conceiving incremental addition of new layers of such evaluations to a static analyzer up to simulating the whole program, it is possible to argue that static and dynamic analysis are, in fact, merely two approaches to the same underlying process [5].

Within the context of analysis of data-flow between program components of an embedded system, aimed at being performed in parallel with development and readily available to a wide scope of users, an approach employing pure static analysis as a starting point, which allows hypothetical expansion by dynamic aspects appears as the most appropriate approach.

### 3.2 Code Parsing, Abstract Syntax Tree and C Language

Static analysis can be performed on various artifacts from the source code to the runnable program. In order to obtain the program structure in terms of instances of high-level programming language constructs such as names of variables, functions or modules, analysis of the source code itself is the most intuitive choice. The most common steps of source code analysis will be examined here.

Source code analysis is the initial part of the very compilation process which translates it into a runnable program. Key phases of compilation process which are of interest to us are lexical analysis, parser analysis and semantic analysis [7]. These steps enable the compiler to understand the code, mapping the text of the source code to the structures such as variables or functions based on language definition. Just like they are necessary in the compilation process for the purpose of constructing a runnable program, they are required in the process of constructing a graphical representation of a system.

Lexical analysis maps source code to predefined language tokens in the *lexical grammar*. It utilizes regular expressions to map strings (arrays of symbols) or sets of strings to language tokens, such as predefined keywords, constants and identifiers. At the end of this step, we still do not know much about the program (e.g. at this point both C function and variable names are identified as ‘identifiers’ and there is no difference between their definitions and uses), but it is required for the next step.
Parser analysis involves mapping the array of tokens to list of allowed reductions in a predefined language grammar, as well as execution of reductions defined by parser grammar. Reductions define the allowed relationships of tokens (enabling detection of illegal sequences of tokens), while also linking sequences of tokens to their meaning. Here, we begin to distinguish between concepts such as conditional statements, loops, or, indeed, variables and functions and their definitions and uses, needed both in the process of generating the runnable program and the process of generating a visualization which will display those variables and functions and their relationships. By this, arrays of tokens are gradually mapped to an abstract syntax tree – a concrete tree structure linking concrete source code to syntax of the language which is used in later stages of compilation [7].

Parser grammar used to define a formal language is defined by a set of terminal symbols (symbols allowed in a language), a set of non-terminal symbols (different from terminal symbols), a starting non-terminal symbol, and production rules which, in general, link sets of such symbols. A specific subset of grammars, context-free grammars, where all production rules are defined as derivations of a single non-terminal symbol to other symbols is of primary interest in practice [7]. There are various parsing techniques which vary in performance and domain applicability, which can be grouped towards how the abstract syntax tree is generated: top-down or bottom-up. Examples of top-down approaches are recursive descent parser which is intuitively linked to production rules of a grammar, or LL(k) parsers focused on predictive reductions based on k look-ahead tokens retrieved linearly with reduced domain of grammars to which they can be applied practically or at all. Bottom-up parsers preform reductions incrementally, from terminals up to more and more compound non-terminals. LR(k) parsers are bottom-up parsers applicable for deterministic context-free grammars, which use a stack to store symbols until a reduction is distinctly identified [7]. Parser generators applied in previous work which will be used in this thesis rely on LALR(1) parser with one token of look-ahead, which is a simplified version of LR(1) parser [8].

Semantic analysis is applied to link nodes of a concrete syntax tree to concrete meaning [7]. This is where the data obtained is prepared for further processing: for a compiler - towards an executable program; for us – towards a graphical representation.

### 3.3 Dataflow vs. Data Dependency Analysis

The concept of dataflow/data dependency relevant to the requirements of this thesis work observes data transfers between higher level concepts like signals (roughly corresponding to C variables), modules (C files), managers and layers (C packages).

On the other hand, dataflow analysis in general refers to analysis of a programs control flow graph obtained through control flow analysis in order obtain information on flow of information through variables, for the purpose of program optimization, dealing with concepts of register allocation, dead code elimination, constant computation during compilation, etc. through analyses like reaching definitions (which definitions of variables influence which
uses) or liveness analysis (discovering program paths where a variable value is actively used) [7].

From compiler perspective, these analyses are performed in advanced stages of compilation process. Control flow graphs are graphs with basic blocks as nodes, units of code transformed to a form closer to one that can be easily translated to machine code [7]. This is understandable, as this form is closer to what actually happens in a running program. High level language concepts like functions or modules exist primarily to make the structure of a system more human-readable and manageable in development process. However, for visualization purposes, we do not wish to lose track of these concepts, so this is where the analysis we need starts to diverge from the analysis performed in the standard compilation process. We would not wish to “over-process” the code and then step back to the form more informative for our purpose.

A top-down approach, applying methods similar to dataflow analysis on the code while maintaining a higher level of code abstraction, is more intuitive and easier to implement, even though we may lose insight into details which can only be obtained in the control flow graph form of code. This is not dataflow analysis in the true sense, and a more general term such as “data dependency analysis” would be preferable. However, the application of term “dataflow” for the target dependencies that need to be discovered is still justified, given that the higher level “data dependencies” could be linked to “dataflow” as observed in dataflow analysis.

3.4 Visualization

Graphical software representation (visualization) has to be designed in a way that would provide greatest usability for target users. Manually generated software visualizations that are currently used at Scania are to be a guideline for the design of visualization that will be the output of the automated process. Presently, while a common general pattern for visualizations in use exists, no strict standard is adhered to, as it is preferred to utilize the flexibility that manual design makes possible, by allowing ad-hoc customization of a visualization with respect to the purpose for which it is required. An automated process designed to be a general, multi-purpose solution applicable to a multitude of systems cannot provide this in itself; at best, it could be designed using tools that allow manual post-processing of representations. For an automatically generated visualization, it is necessary to reflect on established techniques and principles of software visualization and the inherent limitations therein.

Firstly, while other approaches are theoretically conceivable, an approach based on graphs, sets of nodes/vertices and edges/connections associated to one or two of them [9] fits the underlying concept of system components connected by dataflow best.

The problem of automated generation of useful, human-readable graphs is very complex. [10] provides a comprehensive bibliography related to problem of graph drawing algorithms. Various types of graphs can be generated (trees, directed graphs, etc.) depending on the problem at hand. Algorithms for drawing various types of graphs are usually based on heuristic approaches [10], achieving optimizations in readability in subsets of cases pertaining
to properties on which they focus. Some of the key aesthetic issues considered in graph drawing are: achieving display symmetry, avoiding edge crossings, avoiding bends in edges, achieving uniform edge lengths and achieving uniform distribution of vertices [10].

Utilization of 3D graphical representations can alleviate some of the problems inherent in drawing complex graphs with many vertices and edges, providing an extra dimension for distribution of vertices and making it easier to avoid edge crossings, with the cost of making the generated visualization harder to manage and navigate through with the existing technology [11]. On top of this, this limits the scope of software applicable for representation and may be more complex to implement.

Dataflow diagram (DFD) representation is one graph model in use for representing data associated with dataflow. Its nodes can be processes or data stores, while edges which can exist between processes or between a process and a data store represent data-flow [12].

![Dataflow Diagram Components (DFD)](image)

Figure 3-1 - Dataflow Diagram Components (DFD) - representations of a process, a data store and an instance of data-flow, respectively [12]

This representation directly addresses the core ideas that we want to represent in the most simple, straightforward way. It appears easy to translate source code structures like functions and variables to processes and data stores, and by extending this by representation of package structure of a system, no more complex considerations would be required to achieve our purpose.

Unified Modeling Language, the widespread standard for design, implementation, analysis and modeling of software [13], provides a comprehensive, well-defined paradigm for development of a wide range of software models. It can be used to visualize existing code both through manual and automated analysis. While adherence to it guarantees a level of unambiguous formalism easily recognizable by a variety of users acquainted with it, and it would be desired in the long-term development of increasingly complex visualizations, development of visualizations with strict compliance with it in mind is not deemed necessary here. If necessary, simple DFD-based representation can be translated to UML-compatible form, for example by using Activity Diagram activities and objects [13].
4 Analysis

In order to choose the approach, methods and tangible targets for the implementation intended to solve the described problem, preparatory analysis work was performed. Related existing tools have been investigated to gain insight into the advantages and disadvantages of previously undertaken approaches to the subject, review the quality of output they provide and discover whether they could be applied directly to obtain sufficiently good results, while weighing the cost/benefit ratio of their direct application.

In parallel to this, in-company interviews have been performed with a range of interested parties, i.e. employees who could provide insights into relevant stages of software design and development process, as well as current users of graphical representations of systems and potential future users of an automated tool for this purpose. From this, insights on current practices and desired outcomes were compiled. Based on conclusions from interviews, analysis of related tools and analysis of applied coding standards and guidelines, scope and end goals of implementation performed in this thesis were determined and demonstrated on the exemplar EEC3 system.

4.1 Analysis of Data from User Interviews

Interviews were performed in parallel with early stages of implementation done as a part of this thesis, on employees which could provide insight into key details of the problem area, as well as those interested in using the developed or recommended tools. The goal of the interviews was to collect answers from different perspectives to questions that could roughly be grouped in topics:

- job description - categorization of person’s perspective towards software development process
- details on their concrete, day-to-day relationship to development process in general, role and relevance of system specifications from their perspective
- details on their current use, if any, of graphical representations of data, as well as their concrete needs and abstract desires that a future automated implementation could satisfy
- demonstration of approach taken in concrete implementation as a part of this thesis at that point, followed by interviewee’s assessment of its potential, advantages and flaws, based on which revisions of the implementation were performed.

For the full list of questions used as guidelines for semi-structured interviews, see appendix A1.

Apart from better defining the goal of implementation and collecting advice on how to get there, important aspects of the interviews were: to get to know the company structure itself, see in what way and how much the different departments communicate with each other, discover how the developed or recommended tools could be fitted into the company processes
to provide most benefits, along with determining what can and what cannot be changed in those processes, in order to provide a set of feasible recommendations for the future.

Interviews were performed with seven employees of varying backgrounds, some of which could provide multiple perspectives on the subject, having personally taken part in multiple roles in relation to the development process. The compiled conclusions can, in turn, be grouped into three key perspectives, labeled here as: System Architect Perspective, Developer Perspective, and Company Perspective.

System Architect Perspective covers the feedback from users dealing with system design and overview. An emphasis was put on the problem that manual system visualization is time-consuming (an example of two weeks for two people to cover a small part of one of the embedded system’s software was given). On the other hand, the minimum requirement for a potential automatic visualization in order for it to be used was defined as it having to be at least as good as manually created visualization in providing the necessary information. A need for different levels of detail was identified: while a system overview with uniform module-level detail was applicable for some uses, other views would require an option to focus on some parts of the system in more detail than the other parts, e.g. creating a view that would analyze a module down to the variable level and their links to modules from another layer and, even further, only managers from a third layer. Furthermore, abstraction of components of layers (namely, RTDB in EEC3) into communication between other layers is often desired. From this perspective, automated visualizations would take place of manual visualizations in processes such as: providing an aid in understanding the system when presented to new people, generating reports, providing feedback to developers, providing a design aid, as well as review of system compliance to ISO 26262 standard.

Developer Perspective covers the feedback from developers. One of the key insights into developers’ view of the problem area can be summed up as “the less coding restrictions, the better”. While coding restrictions (from MISRA C to internal guidelines) facilitate analysis and overview, they sometimes restrict the choices available in development to a degree that can significantly influence system performance. With this in mind, a policy of minimum reliance to such restrictions in implementation done as a part of this thesis was taken, given that such an approach, although harder, could provide a more robust solution applicable to a larger scope of systems. The key application of a graphical visualization tool for developers was determined to be signal propagation tracking, i.e. tracking of propagation of a single value through modules or variables, which plays a role in error correction, debugging and performance analysis. For all other purposes, consensus among the developers is that direct code analysis is good enough, being likely faster than any other methods given that they have to be already well acquainted with the system as a whole to do their work efficiently in practice.

Company Perspective denotes aggregate feedback collected from developers, system architects, as well as employees from other roles. The idea of lack of “common language” between different departments was explored in practice. Multiple departments have almost no
direct contact with software, dealing with hardware aspects of embedded systems, or managing elements seen as “truck components”, yet can, sometimes, be affected by issues resulting from software design. Graphical representations of software can be designed in a way that would make it easier for people with different backgrounds to understand the aspects of the system relevant to them, and if automated and thus possible to rapidly generate knowing that they correspond the actual current state of the system, they could be deployed to all interested parties and tested as tools for providing the “common language” between them. On the other hand, a consensus among all interviewees was a preference for an internally developed tool. At Scania, it was stated, the goal is to develop internally, as opposed to using 3rd party tools, given that this makes changes to developed software easier and faster, which is deemed beneficial in the long run.

Table 4-1 Summary of conclusions from the interviews

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Key Notes</th>
</tr>
</thead>
</table>
| **System Architects** | Manual visualization time-consuming  
To be useful, automatic visualization must be at least as good as manual  
Potential uses of visualization: understanding, reporting, feedback to developers, design aid, ISO26262 review and reporting |
| **Developers** | The less coding restrictions the better  
Main use: signal propagation tracking |
| **Company** | No “common language” between different departments  
At Scania goal is to develop internally  
Goal to fully address this problem though a series of connected projects: emphasis on extensibility and ease of modification |

4.2 Analysis of Related Existing Tools

There are many tools, such as QA-C[14] used at Scania, that focus, primarily, on code analysis for verification of coding standards and guidelines and generation of text reports used in debugging and performance analysis, which cannot be used to directly obtain the type of graphical representations that could fit the goal of this thesis.

There are tools which can be applied for visualization of related concepts. For example, open source tool for generation of C call graph data cgraph [15] can be used in combination with Cflow2vcg [16] to generate visualizations of C call graphs. An example of a proprietary tool
which can be used for software architecture visualization on top of in-depth source code analysis is CodeSonar [17].

yEd [18] is the freeware tool already commonly used within the company for manually designing graphical representations, which makes it a good choice for providing final representations. Another example of a graph rendering tool dot, a part of Graphviz package used to generate graphs in formats such as SVG [19].

However, the only application that was considered as a contender for full implementation of the analysis required by this thesis is Understand, a proprietary application by Scientific Toolworks, Inc, for source code analysis and metrics [20]. It provides options for verification of coding standards and generation of system visualizations.

The approach taken by this tool involves preprocessing of code in order to build a detailed database storing information compiled from a given codebase of a software system. After this database is built, a variety of analyses can be performed on it, rapidly providing a variety of outputs. Exploration of core functionalities of the program has not yielded results that would adequately address the identified requirements (see Table 4-1) – no inbuilt visualization scheme fits the purpose. On the other hand, Perl and Python API plugins for the application are also available, providing a comprehensive and versatile set of commands and scripts that allow customization of both code analysis, based on the data stored in compiled database, and generation of graphical representations. Through analysis of these command sets, it has been determined that it is highly likely that it would be possible to develop a set of scripts capable of fulfilling the goal of this thesis.

However, this approach was not taken due to the following factors:

- The software is proprietary – an internally developed solution using free and/or proprietary software already in use results in immediate cost reduction

- It is third party software – as determined through in-company interviews, internally developed software is preferred, providing more efficient management in terms of support and development of new features

- Time constraints – it would still take time to study and apply the available API in order to obtain the required results; while a higher level of abstraction of software analysis information would be available, it was not deemed a sufficiently convincing reason against continuing with extension of previous in-company work on the software analysis part instead

- It constrains possible future development – potential future development on top of an implementation that would use this software would be have to use this tool as well, as opposed to an approach intentionally aimed at providing simple artifacts easy to integrate even into fundamentally different approaches to the subject.
4.3 Delimiting the Scope of Implementation

Based on conclusions from initial problem description, current state of affairs, conclusions from literature review and analyses performed, it is necessary to define a scope of the implementation feasible in this thesis work.

Source code analysis and generation of good graphical representations are very complex problems. The approach taken here is one pertaining to the long-term, incremental development aimed at developing not just a proof-of-concept solution, but one that could be used as a building block towards a powerful, practically applicable suite. As a result, an emphasis should be put on making the tool extensible and easy to modify. On the other hand, it is, still, necessary to provide a tangible output as well, in order to enable further investigation of how the automated graphical representation can be used in work processes, and a benchmark for further projects concerned with the topic. Finally, while likely easy to achieve, it should be noted that automated visualization should be implemented in a way that would grant significant speed gains over manual visualization.

Existing theoretical dataflow analysis concepts do not fully address the company requirements, which address authentic needs of designers, developers and the company as a whole. Furthermore, there is no formal internal description of related concepts that do correspond to requirements or existing manual visualization methods. Dataflow/data dependency concepts need to be precisely and accurately defined to reflect the requirements, allow generation of required output in visualization, and take into consideration the approach towards future extensibility.

Static analysis will be performed from incrementally, prioritizing coverage of different types of code structures by frequency of occurrence in source code of real systems and complexity of analysis that needs to be performed to fit it. Complex issues requiring more detailed semantic analysis, like pointer analysis, data type analysis, or analyses of intra-functional dataflow like liveness analysis are considered of least priority. On the other hand, provisions for future implementations of coverage of such concepts should be taken into consideration.

While a robust solution, as universally applicable to different systems and structures allowed by C grammar is preferred when possible without a significant increase in complexity of analysis, if necessary, solutions will be simplified by assumption of adherence to MISRA coding standard.

Form of graphical representations must be defined in such a way that would conform to rules for generating useful, intuitive and clear representations. However, given the complexity of this problem and the limited time-frame for development, as a part of this thesis existing automated mechanisms for graph generation and allocation of graph components will be applied, by generating output readable by yEd [18] application which provides graphical representations of textual graph definitions as well as allocation mechanisms.
Process of data dependency analysis, should, in itself, be observed as separate from a particular process of generation of visualizations. The modular approach to development of those processes should be emphasized, given that this would allow for further separate inclusion of either of them in future projects, increasing the potential of reuse of developed artifacts.
5 Implementation

5.1 Previous Work

Deliverables developed within this thesis work represent a direct continuation of work started by previous thesis by Martin Pruscha [23]. Initially, it was envisioned with the same overall goal in mind as presented in this thesis – developing an automated tool for obtaining graphical representations using source code as the only input. However, upon thorough analysis and real-time implementation which gave better insight into the breadth of scope of the topic of source code analysis, and the amount of work needed in order to obtain tangible output that could actually be applied by real users to a real system, it evolved into an infrastructure for a project of a larger scale.

C++ application was developed, using Flex/Bison lexer/parser for C source code analysis. Initially, the approach was to utilize the Bison version of ANSI C Yacc grammar [21], which provided access to parsed code structures as they appear in syntax tree (e.g. variable definitions, expressions, function bodies, or atomic lexer structures like operators, constant numbers, variable identifiers…), instead of analyzing the code as text. Most of the code structures were not of interest, and Scania code is well-ordered and adheres to standards that improve code readability, so developing a specialized parser from scratch dealing only with most relevant parts of text was considered, but using the existing parser with full ANSI C grammar was considered a more general and robust solution.

Bison grammar can be expanded with actions at a given point in abstract syntax tree. Initially, the approach was to make the jump from identified code structures to their representation as quickly as possible, by adding complex actions to Bison grammar in appropriate places. Primarily due to the fact that many actions relevant for generation of a visualization are not independent, this approach was abandoned because the grammar code would have quickly become too complex to manage. Next step was the addition of generation of a tree-structured object storing information on a minimum of relevant abstract syntax tree structures, upon which the actions related to generation of a visualization would be performed.

In the next stage, when the project was split into two thesis works to be developed partly in parallel, the focus of the described application under development as a part of the first thesis was gradually shifted towards providing an abstraction of source code structures in a form that could be efficiently processed by a separate application on the higher end of the toolchain in order to obtain graphical representations.

Intermediate form XML format (discussed in chapter 5.4) was developed, with the aim of being both readily obtainable from source code/abstract syntax tree analysis, and as close as possible to structures necessary for graphical representation.

Finally, it was established that potential for efficient abstract syntax tree analysis was far greater via XML analysis tools applied in the “front-end”, C# part of the toolchain developed
by this thesis, so the focus of the back-end C++ application was shifted on obtaining the abstract syntax tree in XML form. As a part of this thesis, the approach of obtaining only the minimum required abstract syntax tree was abandoned, considering that obtainment of a full abstract syntax tree did not have a significant impact on its analysis, while it provided greater space for future improvements and modifications without changing the back-end application. As a part of this thesis, besides minor extensions to existing Bison/Flex grammar (handling of “//”-style comments, “the lexer hack” – recognition of user-defined types as actual types), back-end application was modified, therefore, to generate abstract syntax trees by adding tree nodes for all parser reductions and export the created abstract syntax tree objects to XML representations of the abstract syntax trees.

5.2 Toolchain Overview

Figure 5-1 shows the full working toolchain developed throughout this thesis. The processing performed by the toolchain is separated into three stages:

- Code Processing
- Abstract Syntax Tree (AST) Processing
- Intermediate Form (ImF) Processing
The goal of stage separation was to enable development of specialized, simplified modules focusing on least number of necessary steps in the process necessary, linked with the other stages only by the intermediate input/output files. Thus, Code Processing starts from source code and generates XML representations of abstract syntax trees (AST XML files), which are used as input for AST Processing, resulting in Intermediate Form XML files used as input for the final stage of ImF Processing, resulting in actual graphical representations.

Such separation of functionality allows parallel, out-of-sequence development of independent parts of the toolchain, simplifies the modules, and is better from the perspective of future extensibility and modification.

5.3 Stage I: Code Processing

Code Processing stage is performed by the version of back-end source code analysis application previously developed in thesis by Martin Prucha [23], modified for use in this
toolchain as described in 5.1. This back-end application is represented by the block *Code Analyzer* within *Code Processing* stage of the toolchain.

*Code Analyzer* takes as input the path to topmost folder of C source code package/project to be analyzed, as well as the path to folder containing *.i* files, intermediate files generated during code compilation. These files are required because they represent the code in a preprocessed and linked form, with special preprocessor directives and macros (constants and functions defined via `#define`) integrated into the code and their definitions removed, as well as all definitions from files referenced via `#include` directives added, making it possible for the parser based on Bison/Flex ANSI C grammar to accurately, completely and meaningfully interpret the code of a file currently under processing during abstract syntax tree generation. AST generated as an object tree is then exported to an XML file. Source code folder information, that is, C module package information, is preserved through the naming scheme of output XML files, containing paths of source code files relative to the folder provided as topmost source code folder.

Using *.i* files provides integration of *.c* and *.h* files of the same name, resulting in an abstract syntax tree per full C module, which makes the most sense for further system analysis and representation. On the other hand, some information, like linking the abstract syntax tree structures to lines of code in original *.c* and *.h* source code files is lost this way.

For example, we can start from a source code segment like this:

```c
void SampleFunction(void) {
    const SampleTYPE SampleLocalVariable = SampleCalledFunction(SampleGlobalVariable);
}
```

No preprocessor macros are used here, so the segment will look the same in the *.i* file. Some parts of the AST in XML format corresponding to the code above are shown in Figure 5-2.
Figure 5-2 Sample segments of AST XML

Names of XML tags in AST XML files are actual grammar terminal and nonterminal symbols, except in case of terminal symbols which use non-letter symbols, where the aliases beginning with "_" are used (like "_LPAREN" in the example above), to make the resulting XML file a valid XML file readable by an XML parser (otherwise the XML structure would be invalidated - most obviously, symbols like ">" or "<" would cause problems).

V attribute stores the actual text of the token upon which the parser reduction was performed.

The XML tree hierarchy shows the actual hierarchy of the AST of the analyzed .i file.

XML tag names, their relative hierarchies and, in certain cases, content of the V attributes are used in the next stage to unambiguously identify sets of instances of different code structures (e.g. variable and function definitions, variable use, function calls, etc.), and content of V attributes is used to obtain information on actual elements involved in a concrete instance (e.g. name of the variable defined in an instance of variable definition).
5.3.1 Discussion

In the previous example, several lines of relatively simple code result in a much larger, seemingly more complex representation as an AST in XML format. Source code seems much more human-readable. Even the indentation shown in the example above to emphasize the hierarchical relationships within is not used in the actual files, in part, this save disk space, but the key reason is that, nesting depth rapidly grows with the size of the source file, which would, for all but shortest, most trivial source files, result in output files filled mostly by indentation characters. This is because even a sequence of independent statements results in an increase in depth because of the way the AST is constructed:

```plaintext
statement_list :
    statement
  | statement_list statement
;
```

A sequence of statements will be represented as a tree where each statement_list node (except the last which is reduced directly from a single statement) has as children one last statement and another statement_list with all preceding statements.

However, this format actually reduces the complexity of automated code analysis. Position within the AST uniquely identifies the actual nature of a segment of instructions, in the same way in which it is later mapped to executable code by the compiler. A more complex code segment can contain hierarchies of code blocks within loops or conditional statements, strings, comments, nested function calls, etc. Implementation of an automatic analysis that would take all of this into account and be applicable to more than just code most heavily constrained by coding rules, would result in repeating most of the work in a similar way in which it is already performed by the parser during AST generation. Compared to this, implementation of accurate, automated pattern recognition within AST, applicable to a wider range of compilable code, is a simpler task.

Finally, given the described process of obtaining AST files requires successful compilation of valid source code files, it should be noted that further steps of analysis presuppose valid, compilable source code files.

5.4 Intermediate Form

Before the description of the second stage of the toolchain, AST processing, the desired output of that stage must be discussed. This output is the Intermediate Form, an XML-based representation of data structures and relationships required for desired graphical representation. The aim in development of this format was to keep it intuitively close to the actual code structures necessary for detection in order to obtain the data needed for visualization, yet make it intuitively linked to the structures in the desired visualization, to the point of making it feasible for the end user to directly read its contents to obtain the needed information even without generating visualization, when necessary.
Specification of the intermediate form is the pivotal point for the toolchain, where the actual desired graphical representation has to be taken into account. While conceptually maintaining the notion of the intermediate form, it would be possible to develop any kind of source code analysis and representation by creating different versions of the intermediate form, making parts of this toolchain as well as the general principles behind it reusable for a vastly different types of analyses of source code.

From this point, Intermediate Form (ImF) will be used to indicate the version of intermediate form developed as a part of this thesis, focusing on detection of data dependencies, corresponding to the concept of module-level dataflow as interpreted in the requirements for this thesis.

Developed ImF is consists of XML files, each containing the data corresponding to a single module (a group of uniquely named source code files excluding the file extension, that is, the .i file associated with the set of .h and .c files). File name of each file consists of the dot-delimited full path to the module within the source code folder structure, ending with the module name (same filename as for AST XML files).

Figure 5-3 illustrates the schema of the contents of ImF files. The root node of a module XML file is Module, with metadata attributes related to XML namespace conventions, as well as dot-delimited full path of the module Path, and module name BaseName.

Attributes URL and Description present in root node as well as some of the sub-nodes store extra information used showcase the possibilities of graphical representation tools used in the ImF Processing stage, containing only the .c source code file paths.
The root node can contain Variable and Function sub-nodes.

Variable nodes contain data on global variables defined in the corresponding module. They contain attributes Name (name of the variable) and Type (data type of variable).

Variables can contain Field sub-nodes, storing the information on access to variable fields, when variable is of a custom record type (struct type). Record types with many fields, which, themselves, can also be of record types, are commonly used in Scania code, and to be able to produce useful output, it is necessary to take the field names into account and display them in the visualization. Field nodes are added upon encounter of access to a global variable field in the code. Attributes of field nodes are Name (name of field variable that is a member of a record) and Indirection, storing information on indirection operators applied to access the field, to differentiate between cases when the container variable is a record, or a pointer to a
record, or, in theory, a pointer to a pointer to a record etc. For example, a field access to a field of global variable \texttt{a}, like \texttt{a.b}, would add a \textit{Field} sub-node with \texttt{Name} = ”\texttt{b}” to the \textit{Variable} node corresponding to definition of \texttt{a}. In this case, \textit{Indirection} attribute would be empty, while in case of encountering \texttt{a->b} or \texttt{(*a) .b}, the difference would be that the \textit{Indirection} attribute would be “\texttt{*}”.

\textit{Field} nodes can have \textit{Field} nodes as sub-nodes when the field variable is of record type as well and access to its field is encountered (e.g. encountering \texttt{a.b.c} would add a field for \texttt{b} under \texttt{a}, and a field for \texttt{c} under \texttt{b}).

\textit{Function} nodes contain data on functions defined in the corresponding module. They contain attributes \texttt{Name} (name of the function) and \texttt{Type} (data type returned by the function).

\textit{Function} nodes can contain sub-nodes of type \textit{Relationship}, which store the data on the actual dependencies that we want to observe. The attributes of \textit{Relationship} nodes are:

\begin{enumerate}
  \item \textit{Target}. Dot-delimited full path to the variable or function that parent function is related to through this relationship (packages, module and name of the target function or variable).
  \item \textit{Category}. Name of the category of this relationship (VAR_WRITE, VAR_READ, RETURN, ARGUMENT), describing the nature of the relationship.
  \item \textit{Indirection}. Stores information on indirection operators involved in an access to the target.
  \item \textit{TargetField}. When field access is involved in access to the target (target variable or function return being of record type), full path to the accessed field (which may be a field of target record, or a field of a field of a target record etc.) is stored here.
  \item \textit{Content}. Aimed at storing additional content for a Relationship. Implemented to store names of variables passed as function arguments in case of Relationship of category ARGUMENT.
\end{enumerate}

\textit{Variable} nodes can contain also sub-nodes of type \textit{Relationship}. The only category possible for these \textit{Relationship} nodes is \texttt{VAR_INIT}.

Table 5-1 describes the categories of relationships we observe.
Table 5-1 Relationship Categories

<table>
<thead>
<tr>
<th>Category Name</th>
<th>Description</th>
<th>Code Example</th>
<th>Dataflow Direction</th>
<th>Description of Code Example</th>
</tr>
</thead>
</table>
| VAR_WRITE     | A value is assigned to a global variable via assignment operators (=, +=, -=, etc.) | void foo() {
    ...
    gv = 1;
    ... 
} | foo → gv | Function foo updates global variable gv |
| VAR_READ      | A global variable’s value is read (all occurrences except when it is directly assigned to via ‘=’) | void foo() {
    ...
    int a = gv;
    ... 
} | foo ← gv | Function foo uses variable gv |
| RETURN        | Function is invoked and its return value is actually used in some way by caller function | void foo() {
    ...
    int a = bar();
    ... 
} | foo ← bar | Function foo calls function bar and uses its return value |
| ARGUMENT      | An argument (one or more) is passed to a function | void foo() {
    ...
    bar(1);
    ... 
} | foo → bar (←)? | Function foo calls function bar while passing 1 as argument |
| VAR_INIT      | Global variable initialized via another global variable (in declaration outside function) | int gv1=gv2; | gv1 ← gv2 | Variable gv1 initialized by value of gv2 |

5.4.1 Discussion

ImF specification reflects the module-level dataflow model derived from the analysis of manually created visualizations corresponding to user needs and interviews with employees involved in their creation and use.

Its atomic elements are functions and global variables. This allows providing extended information on the relationships between modules (functions and global variables involved), and makes it possible to partially address the need for discovering data propagation through specific global variables.

The actual “flow of data” between these elements is represented by relationships. The key requirement was to discover the direction of flow. The four relationship categories were defined to address this as well as possible. While VAR_READ and RETURN denote that the caller function (and therefore, its module) definitely receives data from the target variable or
function (that is, their module), VAR_WRITE and ARGUMENT indicate that the caller is definitely sending information.

However, this does not exclude the possibility that a code structure corresponding to any of the relationship categories might not open up a channel for communication in the opposite direction, *if pointers and local variables are involved*, either through use of indirection operators or variables of pointer type. This is especially important to emphasize in case of relationships of ARGUMENT category, given that the type of communication where passing the address of a variable as an output argument to an invoked function (“`f(a)`”) is a well-known and commonly used coding pattern in C.

It is theoretically possible to solve this through static analysis. This could be done by implementing intra-functional analysis that would consider dataflow with respect to local variables (defined within a function), along with a more detailed analysis of pointer operators and pointer types. Given the complexity of these, however, intra-functional analysis was considered completely out of scope of this implementation, and pointer use was only partially addressed, by directly preserving information on explicitly used indirection operators.

Another consequence of these limitations can be seen in case of tracing data propagation between global variables. All relationships except those of VAR_INIT link the analyzed function and the target function or global variable. Therefore, only VAR_INIT relationships between global variables are direct, while others involve connecting functions. Two global variables being linked to the same function may imply, but does not necessarily mean that there is true dependency between the variables themselves. Existence of indirect connections through functions will then only provide information on which global variables are *possibly* connected. On the other hand, if there are no detectable connections between global variables, this will mean that they are not connected in any other way except possibly by obscured channels created by local variable and pointer use.

Important, commonly used C structures that are not treated in any special way by this ImF are arrays. Global variables representing arrays will be treated as any other global variable, and global variables and functions contained within expressions in the array indexes in code will be treated as in any other place in the code.

Finally, a more complex C structure, however often used in Scania code, is the *function pointer*. Function pointers are declared like pointer variables, followed by parentheses (empty, or containing a list of argument types, if function is supposed to have arguments). The actual eligible function invoked by a function pointer is assigned to it in a function body, and the function is then invoked via a function pointer. Accurate and meaningful analysis of this type of code structure would require a degree of intra-functional analysis (linking invocations via function pointers to actual functions assigned to them in a function body), which, as stated before, was considered out of scope of this implementation.
5.5 Stage II: AST Processing

This chapter describes the process of generation of Intermediate Form (ImF) XML files from Abstract Syntax Tree (AST) files. As illustrated by Figure 5-1, the toolchain module where this functionality is implemented is a part of XML Analysis and Visualization C# application, functionally independent of other parts of the application.

AST XML files store the information on source code structures organized as an abstract syntax tree represented in XML form. The goal is to accurately and unambiguously obtain the Intermediate Form structures described in 5.4.

The first step in obtaining these structures and accurately interpreting them is to discover a pattern or set of patterns for each structure type that can be used to uniquely select all the segments of an AST XML file that uniquely identify all and only occurrences of a given structure type (e.g. all and only global variable definitions within a module that will be used to generate Variable nodes in the corresponding ImF XML file).

AST XML files store information on grammar nonterminal and terminal symbols and their relative positions within the abstract syntax tree. Given that these are used to interpret the meaning of code segments during parsing and compilation, it is theoretically possible to use the same to describe the patterns we want.

5.5.1 XML Pattern Recognition and XPath Query Language

While information on nonterminal and terminal symbols, stored as AST XML tag names, is readily accessible by reading the AST XML tags, discovering their mutual relationships is a more complex task, bringing into consideration the parent, child, ancestor, sibling, etc. tags of a current tag.

For example, let us assume that we have already identified a segment of AST XML corresponding to the body of a known function. Within it, we want to discover all variables read by that function. We know that some of the variable reads can, in source code terms, be described as “where a variable appears on the right side of the equal sign”. By direct observation of source code segments corresponding to that description, and the AST data generated from those segments, we can recognize patterns in the AST data that differentiate those segments from the rest of AST data, and rephrase the query to look for that pattern instead. The query analogous to the question is this example could be phrased like this: “any identifier node that has as its parent node a postfix_expression node with an ancestor node assignment_expression that has as its parent node another assignment_expression node and as preceding sibling node a node signifying an equal sign operator”.

This would select only the parts of the observed AST XML segment that resemble this illustration:
From there, we would be able to read the value $V$ attribute of the $identifier$ node and obtain the actual name of the variable, enabling us to link it to the known function.

This kind of queries can be implemented using $XPath$ $Query$ $Language$ [24], used for retrieving data structured in XML format. $XPath$ deals precisely with selection of data based on node contents and their hierarchical relationships. $XPath$ provides a very powerful, intuitive syntax, and the XML-node based phrasings of the queries can easily be translated to $XPath$ syntax. An $XPath$ query that would correspond to the query described in the above example would look like this:

```
//identifier[parent::postfix_expression
  [ancestor::*
    [parent::assignment_expression
     and
     preceding-sibling::assignment_operator[@V='=']]]
```

While getting from discovered patterns to corresponding $XPath$ queries is easy, discovering the accurate patterns that would provide us with full coverage of a type of code structure, satisfying the perquisite of providing us with all and only occurrences of each type of structure in mind, is a much more complex problem. The query from the example only covers reading variables that appear to the right of the equal sign, but they could also be passed to functions, used in conditional statements, passed to functions, or be read when operators like “+=” are applied, where both reading and writing of a variable value occurs. On the other hand, it is not obvious whether the described query would select only variables. Names of invoked functions are also stored in $identifier$ nodes, and a function call can occur on the right side of the equal sign.

Discovery of complete, accurate queries needed for generation of ImF data, started from describing the specifics of a desired type of code structure in source code terms as completely and unambiguously as possible. Then, by applying the knowledge of C grammar used to generate the AST, as well as observation of a small set of code examples, the description was translated to an $XPath$ query. Finally, by iterative analysis of performance on increasingly large sets of source code used in Scania as well as custom-designed examples aimed at covering a wide range of variations in source code, queries were improved until they satisfied
the requirement of covering the perquisite of retrieving all and only structures of a desired type.

The actual query from the final implementation that corresponds the example we described here, used to retrieve all and only structures that are considered in creation of Relationship nodes of VAR_READ category, looks like this:

```xml
//identifier[ancestor::unary_expression[parent::cast_expression] and not(ancestor::unary_expression[parent::assignment_expression]) and not(preceding-sibling::_DOT) and not(preceding-sibling::PTR_OP)]
| //identifier[ancestor::unary_expression[parent::assignment_expression[child::assignment_operator[@V='=')]]] and not(preceding-sibling::_DOT) and not(preceding-sibling::PTR_OP)]
```

The “|” sign signifies a union of two complex queries. Both queries include parts denoting exclusion of detection of fields of variables (and not(preceding-sibling::_DOT) and not(preceding-sibling::PTR_OP)), which, in order to be properly assessed, need to be treated in a different way in a separate sub-query. The first of the two queries covers uses of variables appearing anywhere except on the left side of assignment operators, while the second covers uses on the left side of assignment operators which need to be treated as reading a variable as well as writing to it; that is, when the operator in question is a compound one (a statement like “a+=b” is equivalent to “a=a+b” – old value of a needs to be retrieved). Conversely, nested assignments are not treated as reading a variable that is being assigned to on the right side of another assignment operator (e.g. in case of “a=b=c” old value of b is not actually used, so only read of c and writing to a and b should be detected).

This example provides a general illustration on how other queries in the implementation have been formed. While their level of complexity may vary, the principles applied in their formation are similar.

5.5.2 AST Processing Scheme: Divergence from Query-Action Model; Two-Pass Structure

For the purpose of extensibility, the ideal model of AST Processing, if possible, would be one that would adhere to a linear query-action model, in which a query used to obtain a type of code structure from AST would be assigned an action that creates the appropriate ImF structure for each of the results retrieved from the query. While this is possible in theory, analyzing query results in such a case, in order to obtain the necessary data, would be very complex in some cases.

The example from the previous section shows how to get to the AST XML node containing the name of the target global variable in case of a relationship of VAR_READ category. In
In order to write it to ImF XML, however, we also need to know where to write it, i.e. which function actually reads the variable. We could get there “in reverse”, moving upwards from the obtained AST XML node to a parent node containing the container function name, but getting there would require another complex query.

Consequently, an approach focusing on keeping the XPath queries simple and AST analyzer intuitive with respect to what is being analyzed was taken. In the VAR_READ example, this is manifested by organizing the analysis in such a way that for each query discovering subsets of AST XML representing function definitions with function bodies, a query used to discover names of read variables is performed on that subset only.

However, yet more information is needed to obtain data for the ImF XML structures we want. To create a data model that will store information on global variables and functions and the connections between them across the whole analyzed system with its modular structure in mind:

- It is necessary to distinguish between global and local variables
- We should focus on global variables and functions actually defined within that system
- To accurately link two elements, at the time of link creation we need to possess the full path information for each of them (their container modules and packages).

To deal with these problems, analysis of AST XML files and creation of corresponding ImF XML files is performed in two passes:

- In the first pass, all AST XML files are analyzed, one by one, for global variable and function definitions only. All variables declared outside functions are treated as global variables. Function and global variable names are buffered along with full path information, given that, for these structures, this is simply obtained from the name of the AST XML file currently analyzed. ImF files without Relationship or Field nodes are created as well.

- In the second pass, all AST XML files are analyzed from beginning. For each file:
  
  o Subsets of code related to function bodies are discovered by queries that re-discover the name of the function to which the body is related, and then queried for possible relationships. For each relationship discovered, a Relationship node is created under the Function node of the analyzed function. Buffers created in first pass are examined to obtain the full path of the target variable or function, and written to Target attribute of the Relationship node. If access to a field of a global variable of a structured type is discovered, the buffered path information is used to reach the Variable node in the appropriate module corresponding to that variable, and a Field node is assigned to it.

  o Similarly, subsets of code related to variable definitions are discovered and analyzed for relationships of VAR_INIT category, to create the corresponding Relationship nodes of VAR_INIT category under the Variable node related to
the variable being defined. Buffers are consulted in a similar fashion to obtain the target path of the variable used in initialization, and find where to create Field nodes in case of field access.

5.6 Stage III: ImF Processing and Visualization

As illustrated in Figure 5-1, this stage deals with generation of the actual visualization, based on the ImF XML files generated by the previous stage. In this stage, ImF XML data is translated to GraphML format, which can be displayed as a graphical representation in yEd application. View customizations are also applied here, based on the options chosen in the developed graphic user interface (GUI).

5.6.1 GUI

Figure 5-4 shows the developed Windows Forms GUI. The complete toolchain is controlled from this form. Folders of input files (source code and .i files) and intermediate files are set on the left. Button Generate AST invokes the separate C++ application that performs Stage I – Code Processing. Button Generate ImF invokes the module of C# application that performs Stage II – AST Processing.

When ImF XML files are generated, they can be pre-processed and loaded for further filtering by the GUI. Checklist on the left can then display a forward-searchable list of all discovered

---

1 Note: text pertaining to analysis of a Scamia system blurred
packages, sub-packages, modules, functions, global variables, and detected global variable fields. By this, a user can control which elements will be displayed in the graphical representation. A set of options on the bottom left, \textit{Relationship Discovery}, can be used as an aid in this selection: for selected subsets of elements, automatic search of related elements can be performed. Directions assigned for each relationship category (as shown in Table 5-1) are taken into consideration here, based on checked options (\textit{From} includes dataflow starting \textit{from} selected elements, \textit{To} includes dataflow going \textit{to} selected elements, and \textit{Arg.F.Call Bidirectional} controls whether relationships of ARGUMENT category are treated as bidirectional).

Elements selected here can then be loaded into the next list – a tree view. From here, for each of the non-leaf elements of this tree the user can define whether it will be shown as expanded or contracted to a single element in the final visualization.

Finally, two main types of graphs can be generated. For the purposes of system overview, where usually a large number of elements is selected and lowest-level components are not of interest, a module-level graph can be generated, with modules as nodes and connecting functions as connectors. For views of smaller parts of the system, focusing on tracking particular functions and global variables and things related to them, a function/variable-level graph can be generated, with functions and global variables as nodes and ImF relationship categories as connectors.

\textbf{5.6.2 yEd GraphML Generation and yEd Output}

GraphML is an XML-based format for storing graph data [22]. Its essential components are nodes of following types:

- \textit{node} – graph nodes (vertices); uniquely identified by \textit{id} attribute
- \textit{edge} – graph edges (node connections) ; uniquely identified by \textit{id} attribute, with \textit{source} and \textit{target} attributes denoting source and target nodes for the connections.
- \textit{graph} – the whole graph under which \textit{node} and \textit{edge} nodes are nested, as well as possible sub-graphs if nested under a \textit{node} itself; uniquely identified by \textit{id} attribute

Other nodes and node attributes are available for adding additional graph data (e.g. denoting where to position an edge ending on a node, whether a graph is directed or undirected, etc.). It is also possible to extend it by nodes from a different XML namespace, defined in XML header and denoted by namespace identifier followed by a colon in such nodes. Such extension is applied in yEd-specific GraphML format, to store the additional data that yEd uses for graphical representation of the graph, such as: 2-D node location, geometrical shapes used for nodes, colors, text labels, representation of nested graphs as groups that can be expanded or contracted in yEd application, URLs and tooltips related to nodes in yEd application, etc.
Based on the filtering and customization options set in GUI, a subset of ImF XML data is translated to yEd GraphML format. For function/variable-level graphs, this is performed by creation of:

- nested graph structure based on module name and path information; nested graph structure is also used for global variables of record types with detected fields
- nodes from functions and leaf fields and global variables
- edges from relationships, labeled by relationship category, directed by defined dataflow directions for each category.

![GraphML Example](image)

**Figure 5-5 Example: Function-variable level graph for a small set of related components in EEC3 system**

Some of the components generated from corresponding ImF data are:

- labels – package, module, function, global variable or field names for groups and nodes, connecting functions or relationship categories for connections; information on applied indirection operators extends corresponding labels when present
- shapes – nodes are rectangular by default; on function/variable-level graph, circular nodes are used for leaf global variables/fields to distinguish them from function nodes
- colors – differentiate levels of nesting and types of relationships
- URL and tooltip – URL and Description data from ImF nodes.

However, actual spatial distribution of final graph elements is not performed in this stage. Due to this, generated GraphML file will show the elements one on top the other in the center.

---

2 Actual Scania module, function and variable names obscured (replaced by asterisks to show placement). Edge label font size corrected for readability.
of the pane. The actual distribution of elements is then performed by applying automated distribution algorithms available in yEd application, in combination with manual redistribution, if so desired. This approach was chosen because the problem of distribution of graph elements in a well-organized, uncluttered way, especially for large graphs, is very complex in itself. Options for this, available in yEd application, are of limited use, especially for larger graphs, but they do facilitate the process of element distribution, and can provide acceptable results in combination with manual editing.

For module-level graphs, translation to yEd GraphML is performed by creation of:

- nested graph structure based on module path information in ImF XML files
- nodes from modules (module names)
- edges from relationships with targets in different modules, labeled by the name of the container function for that relationship.

Graphs dealing with system overview that need to show the entirety of a system are very complex. While it is possible to generate such a graph on function/variable level, it will not be of any use for a system of size equivalent to size of EEC3: yEd distribution of components takes a long time, may not complete due to memory overflow problems, the end result is not useful in itself, and given the number of components, manual arrangement of components would be a very complex process. Module level graph generation was developed to deal with this problem. However, even the module level graphs for a system like EEC3 are very complex and require manual editing to render them more usable. Figure 5-6 shows a zoomed-out overview of only a subset of EEC3 connections, with automatic distribution applied and manual rearrangement of topmost packages (layers). Given the number of connections, it would require much more manual intervention to render it usable (all edges and edge labels would need to be redistributed to make it clear how modules are actually connected). A useful aid in this process, and graph overview in general is the inbuilt yEd option of displaying the neighborhood of a selected node, as shown in Figure 5-6.
Figure 5-6 Example: Module level graph, a partial overview of EEC3 system and neighborhood of a selected component.\(^3\)

\(^3\) Actual Scania module, function and variable names obscured (replaced by asterisks to show placement). Node labels represent module names, while longer edge labels represent names of functions linking them.
6 Conclusions

6.1 Results

6.1.1 Review of Implementation Scope Coverage

This implementation covers the scope identified and defined through analysis of the initial description of requirements, analysis of user demands, problem field and available resources.

The idea of developing a tool as a part of long-term development of an incrementally powerful suite was taken into consideration in every step of the process, through focus on a heavily modular structure, design of Intermediate Form (ImF) specification and approach to generation of graphical representation.

Tangible output that can be argued to cover the essential requirements has been made available. It is necessary, however, to reflect on the previously discussed approach to static analysis, in order to be able to properly interpret the nature of the output. The key aspect of this approach was the possibility of analyzing the code in a way that would provide reliable information through a simplified, manageable process, with possible cost on the side of level of depth of analysis and precision of data. The appropriate way of interpreting the data obtained through analysis aimed at covering the defined ImF specification has to take the ImF specification and what and how it was designed to observe into consideration. Without this in mind, paradigms in which apparent shortcomings to implementation could be detected. The best example on which this issue can be observed are variables of pointer types. In theory, they can create dataflow between modules which is not explicitly covered by the current implementation. An experienced developer familiar with such communication between two modules which is not directly discernable in the visualization that relies on ImF could then falsely assert this as an error of the implementation. This is not the case, however – the confusion arises from the fact that the ImF is strictly bound to the simplified definitions of source code structures as represented in source code form, which may differ from interpretations closer to intuitively clear concepts corresponding to theory that may be harder to define and analyze. Note that it would not be hard to extend the ImF to cover these as well: the main problem lies in defining and implementing the analysis that could provide this information. A somewhat trivialized, yet essentially true way of describing this nature of the ImF generation is that “ImF fully covers that and only that which it was specifically designed to cover”, yet by bearing this in mind in interpretation of end results, reliable and arguably significant information about the analyzed system can be obtained. However, the confusion that arises from this issue has to be recognized as a shortcoming of the very approach to implementation taken. In terms of design objectives, this means that dataflow concepts defined do not clearly reflect the requirements. On the other hand, further expansion and development of of ImF specification and analysis performed to obtain it can conceivably done exactly in a way that would reduce the discrepancy between the two perspectives.
With caveats of previous discussion in mind, the implementation can be considered robust in terms of input covered. Obviously, the less the code is reliant on communication that would require more in-depth pointer analysis and special treatment of arrays and function pointers, the more informative the resulting visualization can be. The only MISRA standard rule which was taken as perquisite for accurate analysis of structures addressed by ImF specification is the rule which disallows arithmetic operations on pointers apart from array indexing [2], because meaningful coverage of this type of structure would require a degree of semantic analysis which would not be in accord with current scope of ImF specification.

Finally, performance objectives, up generation of GraphML files have been met. Two-pass analysis of AST XML files through XPath queries executed by C# Xml package, along with lookup of variable and function buffers in second pass, performed on EEC3 system have been shown, though testing, to take time less than or comparable to disk access when reading input and writing output XML files. Key focus of code optimizations was, therefore, reduction of disk access, both through implementation in analysis modules and by minimization of AST XML file sizes. Execution of full analysis of EEC3 systems on average computer configurations in current use within the company can provide visualization from available .i files within two to five minutes. It should be noted that compilation of source code required to obtain .i files has been shown to take an order of magnitude longer. This is well within the stated current practical requirements, for which even solutions allowing for daily automated execution of most time-consuming parts of the implementation were stated as acceptable. With this in mind, potential exponential growth of time required for analysis of larger, more complex systems with more elements and interconnections is not considered as a problem for systems used within the company. A theoretical upper limit to complexity of the analyzed system with respect to use of developed tools is unlikely to be hit by a system in practical use, but only by a system intentionally designed for this (the quickest way to reach this would be to generate a system with maximum module depth achievable within a file system, and then add nodes, linking each new node to all previously added nodes).

However, applicability of yEd for automated distribution of graph components is disputable. Representations of graphs of a very small set of nodes and connections can be developed quickly, with almost no manual intervention required to make them clear and useful. As the number of elements grow, however, time required for execution of automated distribution algorithms grows, and quality of distribution is reduced, invoking the need for manual intervention in order to obtain useful representations, with most complex representations of system overviews likely requiring hours or days of manual redistribution in order to obtain a good quality distribution. For example, analysis of EEC3 system produces around 5000 nodes (functions, variables, fields and modules). Even if we ignore the process of manually designing a viewer-friendly representation, consider that edge distribution could be automated, and uniformly estimate that each node could be located, moved to a desired position and its edges redistributed in around 10 seconds, this would result in around 14 hours of work, though the real figure is likely much higher.
Further increases in system complexity have been shown to result in memory overflows during execution of automated distribution algorithms. One approach to solving this would be to reduce the level of complexity of types of representations aimed at system overview. On the other hand, a better choice might be to abandon the usage of yEd in favor of other tools or implementation of automated distribution algorithms.

6.2 Future Work

6.2.1 ImF Extension/Modification

In previous section, we have discussed problematic aspects of the current implementation of Intermediate Form (ImF), and stated that expansion of ImF under the same principles could be a path to improvement. The key concepts that future expansions of ImF should cover are:

- Intrafunctional dataflow. By actually taking variables local to a function into consideration, including function arguments, a much higher level of detail in terms of variable interdependence could be obtained. As previously discussed, current implementation has low precision with respect to this issue, linking variables only through functions that access them. A type of variable to variable relationship would have to be defined, and analysis of assignments between local and global variables would need to be an intermediate step.

- Deeper type and pointer analysis. Current implementation observes types and pointers only “on the surface”, merely noting data types and indirection operators applied in the code. Even struct types are observed and treated specially only per field access; while this is practical in terms of not cluttering up the visualization, deeper analysis could potentially provide more data.

- Function pointers. This is a commonly used code structure in EEC3, for example, and are currently ignored. Given the way they are defined, it is not hard to foresee how their occurrences would be captured. Currently, a function pointer definition is seen as a variable definition, while function access over a function pointer is ignored given that it is a function-like access to it (i.e. followed by an argument list), yet only functions buffered from function definition discovery are actually analyzed. Examining the variable buffer at function analysis would show what function pointers are, and combined with intra-functional analysis of what is assigned to a variable we would get connections between modules achieved in this way.

- Arrays. In some cases, it may be useful to differentiate arrays from other variables. Even though in practice C arrays do not differ much from general pointers, arrays explicitly used as arrays could be identified.

6.2.2 New Options for Visualization

One of the user requirements mentioned in the interview section was component abstraction, where parts of the system, like RTDB in EEC3, were preferred to be represented differently
from other parts of the system, as interfaces between other parts of the system. The idea is to preserve the connections achieved through RTDB, yet not show RTDB as a separate module. This type of extension would be possible to achieve even without extension of ImF (even though implementation of previously described ImF extensions would be helpful by increasing the precision of obtained connections), by an additional layer of processing in the step of obtaining the representation from generated ImF. Generalized implementation of this would involve allowing selection of parts of the system that should not be represented conventionally, but still analyzed when applying analysis of interconnections, and possibly marked in a special way when showing connections obtained through them.

### 6.2.3 Application of Other Visualization Tools

As seen before, usage of yEd introduces significant limitations to this implementation. For representations of a greater scale, with many elements and connections, distribution algorithms in yEd are of limited use. One option would be to utilize the proprietary yEd API [18]. On the other hand, other tools and formats could be used to achieve better performance, Graphviz’s dot [19] which also includes automated distribution algorithms being the best contender. Finally, if deemed necessary, it would be possible to implement problem-specific distribution algorithms internally.

### 6.2.4 Analysis of Real Code

Analysis of .i files circumvents many of the pre-processing issues like treating preprocessor directives, defines, etc. However, in this way, some data which could be useful for graphical representations is lost. For example, linking components of the representation to the actual lines of code which are represented by them would allow integration with direct code analysis. On the other hand, comments, which are lost in .i format, are sometimes a helpful aid in understanding the code, and may be of use if shown in some way in the representation. ImF was developed with these extensions in mind - URL and Description attributes could be used to convey this type of information. In order to achieve this, code analysis would have to be extended to properly treat all of the code structures dealt with in the preprocessing stage.
References


[14] PRQA Programming Research. QA-C, URL:

[15] Free Software Foundation, Inc. GNU cflow. URL:


[17] CodeSonar® Static Analysis Tool, GrammaTech Inc. URL:

[18] yEd Graph Editor Application Features. URL:


[22] GraphML Working Group. GraphML Specification. URL:

[23] Martin Pruscha, Infrastructure for the Generation of Functional Data-Flow Views for
    Automotive Embedded Systems, Master thesis TRITA-ICT-EX-2013:209, TDIPM, KTH Royal

Appendices

A1. Interview Structure
Interview for Data Flow Visualization Theses
Martin Pruscha, Josip Pantovic

[Presentation of us and topic, relation to ISO26262]

1. Personal information
   a. Name, Office
   b. Job description, with relation to source code

2. System specifications
   a. What is the purpose of system specifications?
   b. How important is adherence to specifications? (SAD, Misra…)
   c. How do you make sure that you adhere to the system specifications?

3. System visualization
   a. Do you create or use system visualizations in any way?
   b. What are the motivations for creating system visualizations?
   c. If you had a visualization system, could you foresee any future new uses of SW-architecture visualizations?
   d. How is system/control-flow/data-flow visualization performed? (Give us an example!)
   e. How important is it that SW-architecture views are consistent with actual implementations?
   f. What tools are used for creating system visualizations?
   g. How much time does it take?
h. How good are the results/what problems are encountered (time consumed, wrong results, not used)?

i. Do you know in-company development of tools for related purposes?

[Presentation of application in its current state and target functionality within the scope of the thesis]

4. Application

a. Given the presentation, what functionality would you like to see implemented, regardless of feasibility?

b. What is your opinion on target functionality? Would the program be useful in practice and how / in which step of your work? [all output including graphs, ImF, abstract syntax tree…]

c. Ideas for improvement and extensions?

d. What shortcomings and (future) problems do you see with our implementation?

e. How does our proposed application compare with existing tools used (e.g. manual drawing or tools available for use, like “Understand”)?

f. Can you quantify the degree of benefits attainable by proposed application (time saved, money for proprietary software saved, etc)?

Would it be feasible to apply recommended changes to coding practices in order to obtain more complete and accurate visualization?