Infrastructure for the Generation of Functional Data-Flow Views for Automotive Embedded Systems

MARTIN PRUSCHA

Degree project in Information Technology Second level, 30.0 HEC Stockholm, Sweden
Infrastructure for the Generation of Functional Data-Flow Views for Automotive Embedded Systems

Martin Pruscha

2014
Abstract

Data-flow visualization for source code can help software developers and software architects to understand code graphically. In this thesis, an infrastructure for data-flow visualization is created to analyze the C source code of an embedded system of a truck. Several commercial and open-source tools for data-flow analysis are investigated and a definition for data-flow is found. A data-flow analysis tool chain consisting of FLex, a lexical analyzer generator, Bison, a parser generator, and a hand-written data-flow analysis is implemented. The tool chain saves data-flow information from the source code into an intermediate representation which can be used to create visualizations. Software developers and architects are interviewed to gather information about how data-flow visualizations are used at Scania and how the tool chain can be improved.
Acknowledgments

- Mark T. Smith Ph.D. for his help with defining the thesis topic and outline and for his ideas on the topic from the view of an ICT product

- Mattias Nyberg, Scania REPA Driver Assistance Systems, Thesis Supervisor for the guidance during the thesis, the help in defining the problems and the many meetings to keep the theoretical thesis aligned with the practical software developers work

- Jonian Grazhdani, Scania NESM Engine Mechatronics, Software Architect, for the information and help with the EEC3, RTDB, system architecture and the RTDB analysis tool

- Bo Neidenström for the RTSM01023-MISRA C Rules for Scania ECU Software and the demonstration of QA-C use at Scania
# Contents

Abstract i
Acknowledgments i
Table of Contents ii
List of Figures iii
List of Abbreviations iv

1 Introduction 1
  1.1 Overview of the EEC3 4
  1.1.1 Abstraction Layers 5
  1.2 Problem Statement 5
  1.3 Goals 5
  1.4 Methods and Expected Results 7
  1.5 Timetable 8

2 Related Work 9
  2.1 Related Work at Scania 9
  2.2 Software for Static Analysis 12
    2.2.1 Parsers for Static Analysis 15
    2.2.2 Parser Generators 17
  2.3 Software for Drawing 18

3 Data-Flow and Data-Dependence 21
  3.1 Data-Flow Definition 21
  3.2 Data-Flow Analysis Methods 24
    3.2.1 Data Dependence and Call Graphs 25
    3.2.2 Basic Blocks and Control-Flow Graph 26
    3.2.3 Liveness Analysis, Use-Define Chains and SSA 27
  3.3 Data-Flow patterns in C 27
    3.3.1 Folder/Architecture Data Flow 29
## List of Figures

1.1 Source code to Visualization: yEd [7] as a drawing backend \hspace{1cm} 3

2.1 Mock-up of Jonians RTDB generator visualization \hspace{1cm} 10
2.2 Schematic view of a manual visualization of a small part of the source code \hspace{1cm} 11
2.3 Example of how parser rules can be edited graphically \hspace{1cm} 14
2.4 ANTLRWorks screenshot showing graphical interpretation of "primary expression" from C parser grammar \hspace{1cm} 17
2.5 Example of a static DOT visualization \hspace{1cm} 19

3.1 Actors and Links Data Flow Graph \hspace{1cm} 23
3.2 Assembler Example \hspace{1cm} 23
3.3 Assembler Data Flow Graph \hspace{1cm} 23
3.4 An example of a RTL CDFG \hspace{1cm} 24
3.5 An example of a source code folder structure/hierarchy \hspace{1cm} 30

4.1 Overview of the tool-chain \hspace{1cm} 33
4.2 Lexical Analyzer explained \hspace{1cm} 34
4.3 Parser explained \hspace{1cm} 37
4.4 VAR_READ and VAR_WRITE recognition algorithm \hspace{1cm} 42
4.5 data-flow algorithm flow-chart \hspace{1cm} 43
4.6 VAR_READ and VAR_WRITE flow chart \hspace{1cm} 44
List of Abbreviations

ANTLR  'ANother Tool for Language Recognition'. 16
API Application Programming Interface. 13, 14, 17, 19
AST Abstract Syntax Tree. 15, 16, 18
Bison Bison parser generator. 17, 19
CDFG Control/Data-Flow Graph. 21
Clang 'C language front-end for LLVM'. 16
DOT DOT graph description language. 17, 18
ECU Electronic Control Unit. 1, 2, 4
EEC3 Exhaust Emission Control Unit 3. 4, 5, 20, 21, 24, 25
EMS Engine Management System. 1
FLex 'Fast Lexical Analyzer Generator'. 17, 19
GCC GNU Compiler Collection. 15, 16
IDE integrated development environment. 12, 14
LHS left hand side. 24
LLVM Low Level Virtual Machine. 16
RHS right hand side. 24
RTL Register Transfer Level. 21
SAD Software Architecture Document. 2, 4, 5

UML Unified Modeling Language. 19
Chapter 1

Introduction

This thesis is written to obtain the academic degree 'Master of Science' in the degree program 'Design and Implementation of ICT Products and Systems' at Kungliga Tekniska Högskolan (KTH) Stockholm and is conducted with the support of Scania CV AB, Södertälje. The thesis supervisors are Mark T. Smith Ph.D., KTH and Mattias Nyberg, REPA, Scania.

The thesis addresses the problem of source code visualization for computer programs. In the case of Scania, a Swedish truck manufacturer, the area of interest is manually written source code for embedded systems such as a microprocessor in one of the trucks Electronic Control Unit (ECU), such as the Engine Management System (EMS).

Scania trucks feature a complex electrical system. The system contains many ECUs which control and manage the vehicle’s subsystems, like the hydraulic actuators, the engine and the exhaust system. Most of these devices use computing circuits like micro-controllers to gather sensor information and control actuators according to a program running on the micro-controller, e.g. a control loop designed in the C programming language. These devices are the vehicle’s embedded systems.

The embedded systems used at Scania are programmed in C. The systems are designed in a modular fashion to be able to abstract low-level functions in a higher level container. The systems which are made of modules will also communicate with each other. The systems structure is defined in the system architecture. The system architecture specifies software components, the software component’s order, hierarchy and communication interfaces and tries to clearly define how the system that is to be developed should look like in the end.

This system architecture can be used as an anchor for the software developer during software development, however, in the case of Scania this system architecture is a by-product of ongoing software development, it has been specified during development and not from the start. Developers try to stick to it but often the
system architecture is not clearly followed or workarounds are made.

The software that is running on Scania’s ECUs is large, in this case more than 100000 lines of code. Many programmers contribute to the software by writing code. All of this makes it difficult to understand the software, while understanding the source code is critical to avoid writing faulty code and to not misuse or corrupt any existing or planned software architecture. Every software ‘bug’ can damage the trucks electrical actuators and result in a failure of mechanical components and probably in loss of control of the truck.

Another area of application for an automated software architecture visualization is software architecture standardization. The source code at Scania is written according to a roughly specified Software Architecture Document (SAD), but the developers follow it only marginally. Developers are only restricted to follow certain coding rules, for example MISRA C [1] to prevent critical software bugs, but the software architecture is not completely defined from the start. Instead, the software could be designed with a large scale software architecture predefined, for example the automotive industry standard AUTOSAR [2]. Visualizing the source code can help to match the existing software architecture against a newly defined software architecture as soon as Scania decides on which software architecture to follow.

Two important steps towards a well defined software architecture or a software architecture standardization are ‘Component based software engineering/modeling’ [3] [4] and ‘Functional Safety’ (ISO26262) [5] [6].

Some ideas behind component based modeling are applied at Scania, e.g. grouping the source code functions that belong to a certain hardware component into one source code file and then to separate this component’s source code and other components by storing the source code files in a folder hierarchy that resembles software components, and in some cases this architecture gets close to object-orientation. However, the Scania coding standards only vaguely specify how programmers should stick to the architecture, and the architecture itself is not fully component-based. This makes programmers write code that groups certain similar C-functions into a file and group these files into a folder structure according to the rules, but interfaces as required by the component based software engineering paradigm can hardly be found, with all the potential problems mentioned above. To develop a more component based software architecture, a graphical representation of the source code would be beneficial, for example UML diagrams of the software interfaces.

ISO26262 requires graphical representations for system documentation and review. To map an existing software to these ideas, the source code will have to be transformed into a human readable view. For Scania this means a shift from an arbitrarily defined system architecture to an architecture which has to fulfill
Both for understanding source code and for fulfilling functional safety requirements, system visualizations are beneficial or required. The topic is significant in terms of development time, efficiency, quality (bugs and faults in programming) and development cost. In all fields, a solution to the problem can make a difference.

Therefore this thesis topic evolved at Scania as a result of a pre-study in functional safety/ISO26262 and a solution to system visualization and a framework for the automatic generation of these visualizations is developed in this report.

The thesis’ goal of creating automatic visualizations of the software system is to help developers to understand their code. The visualization shall present the source code on an abstract, graphical level. This shall facilitate code analysis by abstracting details from thousands of lines of source code into a customizable graphical representation. A prototype that was developed in the thesis is shown in fig. 1.1.

Figure 1.1: Source code to Visualization: yEd [7] as a drawing backend

This thesis will present a solution which is an ICT product according to the contents of the degree program. This involves the analysis of the company’s problems and goals, including interviews with responsible persons, a survey of existing solutions to the problem either as a software solution or as other ways to solve the problem (e.g. manual drawing of visualizations) and finally the development of an ICT product, a software tool chain for source code visualization including a
running demo of the program.

The first chapters of the report deal with identifying the problem of source code visualization and define the goals of the thesis work. The central chapters deal with the implementation/programming of the tool chain. In the end of the thesis, the tool chain is presented.

It proved to be possible to approach most of the identified problems theoretically and to then find a matching goal for it. However, not all goals were reached. The developed tool chains works well up to a certain point, but it fails to deliver all necessary and desired information about the analyzed source code. On the other hand, the approach chosen results in an open, versatile tool chain that has the ability to extract all necessary data from the source code, but the later steps in analyzing the data-flow need to be adjusted and enhanced to get a complete and useful representation of the source code.

1.1 Overview of the EEC3

The Exhaust Emission Control Unit 3 (EEC3) embedded system will serve as an example system for source code analysis. The software architecture of the EEC3 ECU is specified by a Software Architecture Document (SAD) [8]. The SAD explains the EEC3 software architecture by providing an abstract view of the software modules and interconnections, it does not provide rules on how the source code should be written. To get valid information about data-flow and program structure all c source code files which are used in the EEC3 system need to be investigated and checked against the SAD to see how data-flow between modules is programmed. For solving the task of this thesis, it is necessary to first look into all possible data-flow/communication patterns. When all patterns have been analyzed, it will be possible to determine which communication patterns need to be considered for the analysis of the data-flow, probably on different abstraction levels.

Real-world communication in the EEC3 happens only on the micro-controller level. The code for the EEC3 is written in C, the c source files are then compiled and linked into machine code which is executed on the micro-controller at run-time. Each C instruction, such as a definition or assignment of a variable, contains many assembly/machine language instructions. If the code for the EEC3 was written in a hardware-near programming language such as Assembler or machine code in which every instruction on the micro-controller has to be written by the programmer, a much bigger code-base needs to be maintained and the overall programming time and complexity increases. The c-language uses c constructs such as variables, pointers, functions and structures to exchange data, and behind each of these constructs there are assembly/machine code instructions to execute.
these constructs on the micro-controller. Therefore, the C language abstracts the assembly/machine code instructions running on the micro-controller and can be seen as a first layer of abstraction.

1.1.1 Abstraction Layers

In embedded systems, abstraction layers can be used to facilitate programming. An example for abstraction layers is the OSI model [9]. The SAD EEC3 architecture specifies the abstraction layers, which are a guideline for how the system programmers should write their source code. The relations between the different abstraction layers as they have been designed by the system architect can be found in the SAD. "The overall goal is to limit the number of relations and as long as possible reducing the number of multi-directional dependencies" [8]. The abstraction layers will play a role in the visualization of the data-flow, described in section 3.3.1.

1.2 Problem Statement

Source code for ECUs can be very complex. The C functions in one source file can be called from other source files and can return data to the callers. If a system designer has to add a new piece of code or change some functionality in an existing piece of code, he will have to look at all the connections to other code to understand the system behavior when all code constructs are communicating with each other in the complete system. Understanding all the connections between hundreds of C files and the contained functions by reading through the code and resolving all data flow paths, probably writing them down by hand, can take a long time. The system as a whole is hard to understand for the system architect if each module has been designed by different programmers and the coding style may not be completely coherent between programmers. At this time, at Scania software solutions exist that can help programmers to solve this task but they are not used or are only used for creating low-level representations (e. g. on function- or module basis).

1.3 Goals

The main goal of the thesis is to visualize data-flow. A subset of goals to further refine this goal were set at the start of the thesis:

1. Find an abstract graphical representation of an embedded system including all data flows from the given embedded system’s source code (in this thesis
the Scania EEC3 will be used as an example).

2. Automate the creation of this graphical representation.

3. Create an intermediate representation of the source code which can then be used by different visualization techniques to create the final graphical representation.

4. Facilitate the development process of embedded systems at Scania.

5. A "complete" set of all possible data-flow patterns shall be defined.

6. An "intermediate data flow graph" is specified and created where nodes are e.g. functions or communication variables

7. Views are generated based on the users choices, identify needs regarding final views

8. If there is more time, generate views

9. If there is time, use the output of the intermediate representation with available software to create a graphical representation.

10. If there is time, use the intermediate representation with self-developed software and components of available software to refine the graphical representation.

11. The program to be written shall have a good architecture

12. An intermediate representation of the source code for the Scania EEC3 embedded system which can be used by Scania for further research

13. A program which can automatically create an intermediate representation of the source files in an embedded system

14. Evaluate how the written software can be used by system designers in the company and how useful existing software is

15. An evaluation about whether the software written fulfills the defined goals

Additional goals that have been collected during the beginning of the thesis time frame:

16. Find available tools which can solve the problem

17. Shorten development time by a reasonable percentage
18. Reduce development cost

19. Provide a maintainable, well documented solution which can be modified by company programmers

20. A solution which can run on all systems used for development

21. A solution which provides architecture visualization with as small as possible (reasonable) amount of user time

22. A solution which can be used by non-professionals (for example Master Students)

1.4 Methods and Expected Results

Methods that can be used to reach the thesis goals:

1. Research literature on how graphical representations can be created from source code. Expected result: Mathematical/logical process on how source code can be analyzed and abstracted. See chapter 3.

2. Find existing solutions for graphical data flow representation and test them on the available source code. Expected result: Understanding on how the problem is approached by third party software programs. See chapter 2 and section 2.2.

3. Analyze the existing embedded system (EEC3) to see what needs to be done to adapt eventually existing graphical representation techniques to the EEC3 system. It is important to understand the data flow between the modules. Expected result: Understand the system architecture and what problems will be encountered when creating the graphical representation. Find specifications for the different communication patterns in embedded systems currently used at Scania. See section 3.3.

4. Interview system designers to learn about their problems when analyzing data flow. Expected Result: Parameters which constrain the appearance of the graphical representation that needs to be found. Probably several different views are necessary as a result of the interview. The interviews will help to check some of the goals regarding usability. See chapter 5.

5. Create a definition of an 'intermediate representation' of the source code. Expected Result: Mathematical/logical description of a program which analyses source files and creates a low-level abstraction of the data flow. The
graphical representation with further abstraction can then be set on top of this intermediate representation. See section 4.6.

6. Design a program or modify existing software to create the intermediate representation. **Expected Result:** A collection of software, maybe connected by a scripting language or a single program which will create the intermediate representation as a text file or comparable file. Use the output of the intermediate representation with available software to create a graphical representation and use the intermediate representation with self-developed software and components of available software to refine the graphical representation. See chapter 4.

7. Validate that the solution solves the problem. **Expected Results:** Estimate programming time savings by finding out how much programming time is used and how often visualization is needed in this progress. Comparison of the visualization program against other third party software. Interviews with programmers showing a demonstration of the program verifies that the visualization is useful. See chapter 6.

### 1.5 Timetable

<table>
<thead>
<tr>
<th>Date</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-06-25</td>
<td>Start of thesis</td>
</tr>
<tr>
<td>2012-06-29</td>
<td>Structure of the EEC3 embedded system and messaging mechanisms (data flow basics) understood</td>
</tr>
<tr>
<td>2012-07-13</td>
<td>Literature study on intermediate representation and graphical representation finished</td>
</tr>
<tr>
<td>2012-07-20</td>
<td>Mathematical abstraction process found and formulated</td>
</tr>
<tr>
<td>2012-07-27</td>
<td>Testing of available software for solving the problem finished</td>
</tr>
<tr>
<td>2012-08-03</td>
<td>Have an idea on how to modify existing solutions so that they can be used on the EEC3 embedded system</td>
</tr>
<tr>
<td>2012-08-10</td>
<td>Interviews with programmers finished and evaluated</td>
</tr>
<tr>
<td>2012-08-24</td>
<td>Intermediate representation defined, requirements for the program set</td>
</tr>
<tr>
<td>2012-10-15</td>
<td>Program for intermediate representation written</td>
</tr>
<tr>
<td>2012-10-29</td>
<td>All of the above task including two weeks slack finished</td>
</tr>
</tbody>
</table>
Chapter 2

Related Work

2.1 Related Work at Scania

1. Jonian Grazhdani, Scania NESM Engine Mechatronics, system architect: Jonian developed Python scripts to visualize connections between source code parts (modules, see later chapters) which is called 'Real-Time Database (RTDB) signal generator'. The scripts can be run at compile-time by passing a flag to the compilation tool chain.

From an interview with Jonian on his signal generator: The generator searches the source code for keywords defined by Scania coding rules. It scans the code but does not 'understand it', there are for example problems with 'if-else' construct. All found keywords are then connected to each other based on their names. If an identifier is found in two or more places in the source code, the script will write down a connection between these source code modules or files in a xml file. The xml file can then be imported to the "yEd" software (see fig. 1.1) which can be used to create the final visual representation. The output from yEd needs to be adjusted manually, because the visualization created by Jonian's script will have many variables written on the directed edges of the nodes in the diagram, which can be very hard to read (see fig. 2.1).

Limitations of the scripts:

- Does not 'parse' the code, only does string searches
- It can therefore not find data-flow, it can only find a limited data-dependency
- The pre-compiler needs to be run on the source code first, and then the output 'I-files' are used by the script
The script leaves out parts of the source code such as the 'util' and other inter-layer functions that are considered to be not important.

The visualization can only inform the user about certain variables that were named according to the Scania rules for coding. Therefore, the user will miss information about many arbitrarily named variables and needs to know the Scania coding rules to understand the visualization.

Writing the variable names on the directed edges between the nodes makes them hard to read.

Figure 2.1: Mock-up of Jonians RTDB generator visualization

2. Mattias Nyberg, Scania REPA Driver Assistance Systems, Thesis Supervisor: Mattias has created a visualization of a part of the source code manually as a part of another thesis project (see fig. 2.2). The time demand for making the A3-paper size visualization was around three weeks. One of the goals of this thesis is to make a visualization "as good" as the hand-made one, so this hand-made visualization can be used as a model of how the visualization can look like in the end and how the level of detail should be chosen.
However, the visualization is a static drawing and can only be used when printed out, it is too detailed for a PC screen. It only covers a small part of the source code and needs to be redrawn every time the source code changes.

Figure 2.2: Schematic view of a manual visualization of a small part of the source code

3. Jonas Biteus, Scania YSNS Service Support Solutions: Jonas and his thesis students’ work focuses on software fault propagation. A software tool exists to visualize the effect of one or more faults on the whole software system. The user can activate a fault in a software component in a UML diagram-like visualization and then sees how this fault affects other software components. The tool parses a xml-file which contains information about the software components and then displays a visualization. The program can parse code samples but cannot parse real source code, the user has to enter the information about the software components by hand.

4. Another thesis project [10] at Scania focuses on code generation for system diagnosis tests, but it touches the topic of visualization only slightly.
2.2 Software for Static Analysis

Stand-alone programs and plug-ins for Integrated development environment (IDE)s which analyze source code in different ways are available from commercial vendors and the open source community. Most of the programs are specialized on helping developers to find potential bugs, check if the code conforms to the given coding rules ("code conformity checkers") and analyze certain code metrics, such as cyclomatic complexity. All of these tools need to analyze the source code in some way, and to find out if there is an existing software that can create visualizations or could help with one or more steps towards it, a comparison between all tools that could be found is made in this chapter.

Some of these programs, such as the 'Eclipse Metrics Plugin' [11] calculating cyclomatic complexity, use control flow theory in their computations, which is one of the basics that can help in visualizing source code (see chapter 3). All of these programs, including Lint, which is a popular software which looks for potential bugs in source code, belong to the family of static source code analysis tools. Most of the programs mentioned create call-graphs, which are data-dependency graphs and which should not be confused with data-flow visualization. To analyze data flow it is therefore necessary to calculate all data-dependencies and to calculate all possible control flows. This is how the parser tool-chain described in chapter 4 works. The table 2.1 shows a comparison of static analysis tools, all of them feature a C source code parser which means that they are applicable to the C source code that needs to be analyzed at Scania.
<table>
<thead>
<tr>
<th>Software name</th>
<th>Static analysis</th>
<th>Visualization</th>
<th>Focus</th>
<th>License</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA-C</td>
<td>yes</td>
<td>data-dependency</td>
<td>code quality</td>
<td>commercial</td>
<td>[12]</td>
</tr>
<tr>
<td>Parasoft</td>
<td>yes</td>
<td>no</td>
<td>code quality</td>
<td>commercial</td>
<td>[13]</td>
</tr>
<tr>
<td>C/C++test</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coverity</td>
<td>yes</td>
<td>data-dependency</td>
<td>code quality/architecture</td>
<td>commercial</td>
<td>[14]</td>
</tr>
<tr>
<td>Klocwork</td>
<td>yes</td>
<td>no</td>
<td>code quality</td>
<td>commercial</td>
<td>[15]</td>
</tr>
<tr>
<td>Understand</td>
<td>yes</td>
<td>data-dependency</td>
<td>data dependency</td>
<td>commercial</td>
<td>[16]</td>
</tr>
<tr>
<td>Source Insight</td>
<td>yes</td>
<td>yes</td>
<td>data dependency</td>
<td>commercial</td>
<td>[17]</td>
</tr>
<tr>
<td>Rational Quantify</td>
<td>yes</td>
<td>call-graph</td>
<td>profiling</td>
<td>commercial</td>
<td>[18]</td>
</tr>
<tr>
<td>Doxygen</td>
<td>yes</td>
<td>call-graph</td>
<td>documentation</td>
<td>open-source</td>
<td>[19]</td>
</tr>
<tr>
<td>AQtime</td>
<td>yes</td>
<td>no</td>
<td>profiling</td>
<td>commercial</td>
<td>[20]</td>
</tr>
<tr>
<td>Visustin</td>
<td>no</td>
<td>yes</td>
<td>flow-charts</td>
<td>commercial</td>
<td>[21]</td>
</tr>
<tr>
<td>CleanC</td>
<td>yes</td>
<td>call-graph</td>
<td>profiling</td>
<td>free</td>
<td>[22]</td>
</tr>
<tr>
<td>Valgrind</td>
<td>yes</td>
<td>no</td>
<td>profiling</td>
<td>open-source</td>
<td>[23]</td>
</tr>
<tr>
<td>Egypt</td>
<td>yes</td>
<td>call-graph</td>
<td>call-graph</td>
<td>open-source</td>
<td>[24]</td>
</tr>
<tr>
<td>CodeViz</td>
<td>yes</td>
<td>call-graph</td>
<td>call-graph</td>
<td>open-source</td>
<td>[25]</td>
</tr>
<tr>
<td>KCachegrind</td>
<td>yes</td>
<td>call-graph</td>
<td>call-graph</td>
<td>open-source</td>
<td>[26]</td>
</tr>
</tbody>
</table>

Table 2.1: Static Analysis Tools
In the following, the programs that offer solutions to one or more of the thesis problems are described.

1. QA-C
The QA-C "Structure101"-subprogram lets software designers make "Accurate representations of code structural dependencies in slice or hierarchical views with auto-partition feature" [27]. The software focuses on data-dependency analysis and uses a parser, but it can not visualize data-flow as defined in chapter 3 and it has no Application Programming Interface (API). One of the most interesting features is the "Dataflow Defect Detection", but it focuses on finding bugs and not on visualization, too. The program can only be used with a commercial license which moves it out of the thesis focus.

2. Parasoft C/C++test
Parasoft C/C++test is comparable to QA-C but features a "rule wizard" which can be used to graphically specify the rules for detecting errors (see fig. 2.3). The rules are parsing rules, similar to the handwritten rules that Bison, the parser generator later used in this thesis, uses. A graphical program to define these rules could be very useful and is used in other parser generators, too (see the parser generator ANTLR, [28]) and could also make the parser used in this thesis more user-friendly in its configuration (see chapter 4). An evaluation version was obtained.

Figure 2.3: Example of how parser rules can be edited graphically

3. Coverity
Coverity is similar to QA-C. It provides an IDE to visualize code dependencies, but the program is proprietary and is made to "Quickly understand the existing hierarchy and dependencies of large, complex code bases" (see [14]). An evaluation version was requested but could not be obtained.

4. Klocwork
Klocwork is similar to QA-C and Coverity but has no visualization functionality. An evaluation version was requested but could not be obtained.

5. Understand
Understand has been thoroughly tested in Josip Pantovics thesis [29]. Understand can create an automated visualization of the source code, has an
API for accessing all necessary data-flow information from its parser and is therefore a good candidate for solving the thesis problem. However, the visualization that is created by Understand is hard to understand because of its simple layout algorithms, and the program needs to be heavily modified through the API to show the needed information. Too much time would be consumed in modifying Understand, and it has to be expected that because of the commercial license it would be hard for the user to make changes in the software to adjust e.g. the parser to his needs.

6. Doxygen
Doxygen "can also visualize the relations between the various elements by means of include dependency graphs" [19]. However, Doxygen cannot generate data-flow visualizations.

7. Valgrind
Valgrind executes the code that should be analyzed on a synthetic CPU. This could be interesting for future studies about dynamic data-flow (instead of focusing on static data-flow), but Valgrind is too complex and would have to be altered too much to be useful for solving the thesis problem.

### 2.2.1 Parsers for Static Analysis

It became clear that none of programs which output a visualization come close to visualize data-flow as needed to fulfill the thesis goal of visualizing data-flow. However, all these programs feature a parser which provides all syntactical information about the source code to be able to compile the source code into an executable program. The only necessary tool to extract all possible information from source code is a parser which needs to be configured to output this information in a way so that it could be read by a second separate program that does the data-flow analysis. A third program then could take the output from the data-flow analysis and create a visualization. Therefore, two parsers are investigated for their use in data-flow visualization.

**GCC**

GNU Compiler Collection (GCC) is an open-source compiler tool-chain which can output the Abstract Syntax Tree (AST) when used with special compiler flags (command line: "gcc -fdump-tree-original-raw fileToWrite"), see [30]. The AST output of a small program is shown in listing 2.2:
Listing 2.1: Hello World AST example
1    #include<stdio.h>
2    int main(int arg_count,char ** arg_values)
3    {
4        printf("Hello\nWorld\n");
5        return 0;
6    }

Listing 2.2: AST output
1    ;; Function main (null)
2    ;; enabled by -tree-original
3    @1    bind_expr  type: @2    body: @3
4    @2    void_type  name: @4    algn: 8
5    @3    statement_list  0 : @5 1 : @6
6    @4    type_decl  name: @7    type: @2    srcp: <
        built-in>:0
7    @5    call_expr  type: @8    fn : @9 0 :
        @10
8    @6    return_expr  type: @2    expr: @10
9    @7    identifier_node  strg: void  lntgt: 4
10    @8    integer_type  name: @12    size: @13    algn: 32
11
12    ... continues up to @62

The AST output has to be analyzed by a separate program to be able to get not only data-dependency (Call Graph, as in [30]) but data-flow information. It is decided that GCC, with its very large code-base, is too hard to modify to get data-flow information in the limited scope of this thesis. However, GCC is widely used and the demand for a data-flow visualization is apparent from developer discussions in online forums, which makes it a good starting point to develop a data-flow analysis tool that could be maintained and developed by a large developer community.

LLVM/Clang

Low Level Virtual Machine (LLVM) is a compiler infrastructure that can be used together with "C language front-end for LLVM" (Clang) to compile C source code. The compiler is relatively new so that there is not much literature about how to create data-flow visualizations from LLVM. It has to be expected that the amount
of work to be put into LLVM and Clang is at least as high as for GCC. Therefore, LLVM is not investigated in this thesis.

2.2.2 Parser Generators

ANTLR

'ANother Tool for Language Recognition' (ANTLR) is a parser generator that 'generates a parser that can build and walk parse trees' (see [28]). It can be used to parse C code. One important feature that other parser generators such as FLex/Bison lack is that the user can design the parser grammar visually using a grammar design tool called "ANTLRWorks" fig. 2.4. Visually editing the parser grammar instead of writing it by hand, as done in this thesis work, could help end-users who are unfamiliar with parser grammars.

Figure 2.4: ANTLRWorks screenshot showing graphical interpretation of "primary expression" from C parser grammar

FLex/Bison

The 'Fast Lexical Analyzer Generator' (FLex) [31] is a lexical analyzer generator that can be used together with the Bison parser generator (Bison) [32] to create a parser tool chain. FLex and Bison are well documented programs, with FLex written in 1987 and Bison written in 1990. Both programs turned out to be open-source, easy to understand, easy to modify and easy to interface to other hand-written code. They are the two most basic tools that programmers could use to create a lexical analyzer/parser toolchain. That is why FLex and Bison were used to approach the thesis problem (see chapter 4).
2.3 Software for Drawing

C source code has a hierarchical structure, as described in the parser section of this thesis. That means that for example the contents of a function block in C, which can be for example variable declarations and function calls, should be visualized in a hierarchical way, too. A variable assignment in a function should therefore be displayed as "encapsulated" by the function block. Not many programs for visualization can automatically display such hierarchical or encapsulated nodes, and the ones that can, use a 'graphical programming language' such as DOT graph description language (DOT) or yEd which is then 'compiled' (DOT) or translated on the fly (yEd) into a visualization. Having a graphical programming language or an API is a must to interface the visualization software to the parser, which should in turn be able to modify the visualization by modifying the underlying source code of the graphical programming language. The following programs were investigated for visualizing data-flow:

1. yEd
   yEd is a proprietary software that can visualize hierarchical, UML-like diagrams. Its underlying language where all visualization parameters are defined is GraphML. yEd was used in Josip Pantovics thesis [29] to visualize the output of this thesis' parser tool chain. yEd is the only software that provides an acceptable, user-customizable (drag-and-drop) visualization, but interfacing the parser tool chain to the GraphML language was not possible in the limited time of the thesis and is dealt with in [29].

2. DOT
   DOT is used in this thesis to visualize the AST that is output by the parser. However, DOT is a programming language (listing 2.3) which is compiled into a static visualization which cannot be manipulated by the user while looking at it, only by changes in the DOT source code (see fig. 2.5). That excludes DOT from being a visualization back-end for the parser tool chain.

Listing 2.3: DOT example

```
1   graph graphname {
2       a -- b -- c;
3       b -- d;
4   }
```
3. Gephi
Gephi is a visualization back-end for DOT. Gephi can display DOT files, but its cloud-oriented layout algorithms have problems with hierarchically ordered nodes, therefore it cannot be used for visualizing the hierarchical structure of C source code.

4. Borland Together
Borland Together is a Unified Modeling Language (UML) modeling tool. It could provide a good UML-style visualization, but it is proprietary.

5. Lucidchart
Lucidchart has the same functionality as Borland Together and is proprietary, too.

6. Enterprise Architect
Same as Lucidchart.

7. Dia Diagram Editor
The Dia Diagram Editor is open-source, but its output is too unrefined to be helpful for data-flow visualization.

Most of the software which combines a parser, static analysis and visualization such as QA-C is proprietary and cannot be used without a license, which makes it impossible to use them for this research project. Demo-versions of QA-C, Parasoft C++-test and Understand were obtained and tested. None of the programs can provide detailed information about data-flow because most of them focus on coding rule checkers to help the programmer to find bugs. Programs that check for coding rules and coding standards (such as MISRA C [1]) violations are already used at Scania, therefore a combination of the mentioned programs and a source code visualization back-end could be very interesting for Scania. Most of the more complex programs use proprietary parser and data-flow engine (such as QA-C’s
'Structure101' [27]) which gives access to all necessary source code information, but only Understand features an API that could be used to access this information and to program another backend that displays this information.

In the end, the FLex/Bison lexical analyzer/parser tool chain is choosen to approach the problem of visualizing data-flow from C source code. The tool chain is fully open-source, therefore fully modifiable and has a good documentation.
Chapter 3

Data-Flow and Data-Dependence

The term 'Data Flow' has many definitions in literature which defines data-flow according to the problem to be investigated. To be able to do a data-flow analysis, data-flow needs to be defined. This chapter explains the term data-flow and presents a definition for it.

In the end of the chapter, the definition of data-flow is used on the EEC3 code to find a subset of EEC3 data-flow patterns. These data-flow patterns need to match the communication patterns in the EEC3 that are defined by the EEC3 architecture. A table shows mappings of data-flow patterns to communication patterns in the EEC3 source code.

3.1 Data-Flow Definition

Data-flow itself exists in every computer program. The earliest programming languages ('low-level' programming languages) were used for writing programs for sequential execution and the programmer had to write down each operation that should be executed on the processor in machine code. Later, programmers could use the low-level programming language Assembler to translate instructions into machine code. The programmer had to stick to a certain grammar that was defined for every processor and that incorporated all possible instructions in machine code. Data-flow was easy to define on a machine code level, for example writing and reading to and from a memory location on a micro-controller, because everything that happened in the micro-controller directly corresponded to the machine code. However, already at this point with rather small program memory on a micro-controller it was probably not easy to understand or to visualize all possible data-flow connections because of readability problems.

The complexity of computer programs increased to a degree where such programming is no longer feasible because it takes too much time to write each and
every single instruction. Instead, the machine code instructions were abstracted into 'high-level' languages like C [33]. A function call in C is comparable to a 'JUMP' instruction in Assembler, but the C instruction contains in turn many Assembler instructions, e.g. for memory handling. On the C-language level, it is most important to see the data flow that results from the C language instructions. The underlying Assembler instructions should be hidden from the programmer to speed up development, but he can still access them (e.g. via 'deassembling' the C code into Assembler instructions during debugging) to see the data flow on this level. Different programming paradigms like object-oriented languages and parallel programming languages have lead to more abstract data-flow definitions, even though the real data-flow will still take place on the machine code (from the programmers point of view) or even at a lower-level, the hardware or transistor level of the processor (from the hardware point of view).

The EEC3 programming language is C, which as a structured programming language that abstracts much information compared to machine code or Assembler instructions. [34, pg. 713] explains the different abstraction levels by the examples of Fortran and Algol: The more basic operations that the programming language abstracts into its syntax and the further away it is from individual machine code statements the more complex the definition for data-flow and the more complicated is the analysis of data-flow. The most basic data-flow can be found in machine code and Register Transfer Level (RTL), object-oriented programming languages can have abstract definitions of data flow.

[35, pg. 941] introduce a 'Data Flow Concept', where a bipartite directed graph with two types of notes, 'links' and 'actors' is used. These nodes are connected by arcs which are 'channels of communication', see figure 3.1. This resembles the concept of Petri-nets where the basic items are called places, transitions and arcs [36]. R. E. Filman explains how a data-flow model can be derived from a Petri-net model in four steps [37, p.114].

A data-flow can consist of the following basic items [35] and can be visualized as in fig. 3.1:

- **Actors/Places**: This is one of the states that the program can be in, e.g an addition in Assembler language or the start state of a called function on a more abstract level

- **Links/Transitions**: They trigger the data-flow, e.g. on Assembler level a single CPU tact, on a more abstract level e.g. a function call

- **Arcs/Edges**: They define the origins and destinations of the data-flow between actors or places

For the assembly language a data-flow for moving two constants to two registers and adding them with the MOV and ADD instructions could look like figure 3.2.
Figure 3.1: Actors and Links Data Flow Graph

```
MOV r1, 4
MOV r2, 3
ADD r3, r1, r2
```

Figure 3.2: Assembler Example

Figure 3.3: Assembler Data Flow Graph
For logic circuits, e.g. FPGA, a RTL Control/Data-Flow Graph (CDFG) can be defined, for example for a Hardware Description Language (in this case VHDL) [38, 39]. [39] claims that "there is no widely accepted format for representing CDFGs", therefore the authors CDFG definition is complete for a subset of VHDL only. The paper also presents different 'representations' of the CDFG (see fig. 3.4), which makes the approach to data-flow visualization comparable to the goal of this thesis, only that the target language and platforms are different. Both papers limit the use of their approach to data-flow visualization, where [38] states the following:

"Also, the chances for errors in the resulting CDFG increase with the increasing complexity of the code"

Figure 3.4: An example of a RTL CDFG

To sum up, data dependency means that two objects depend on each other, but the data flowing between these two objects can be unknown. Data-flow can be described as a transfer of data between two dependent nodes or source code objects and is a mix between a control dependency graph and a control flow graph [35]. A similar definition of data-flow can be found in the concepts of the "Definition-Use Chain" and "Liveness Analysis" in the next chapter, which lead to a final definition of data-flow for this thesis.

3.2 Data-Flow Analysis Methods

Source code analysis is divided into static analysis and dynamic analysis. Static analysis extracts information from the source code at compile- or parse time, dynamic analysis gathers information during run-time. The underlying system is a
micro-controller-based embedded system and relies in contrast to a typical desktop application on real-time physical sensor values such as exhaust temperature. For a run-time analysis, the program needs to be compiled, a connection to the ECU hardware needs to be established and the micro-controller needs to be programmed. When the system runs it has to be able to gather real-world sensor values to compute all possible paths and states that can be reached in the program. These sensor values are only available if the EEC3 is mounted in a truck. Alternatively an offline simulator which simulates the micro-controller can be used together with a database which contains all possible sensor values. These values can then be used when the simulator executes to simulate a real world environment. As the number of sensors for the EEC3 is large, the database will require much development work. Furthermore, a complete simulator tool chain is not available at this point of time for the used micro-controller. A similar micro-controller simulator could be used but the results can be offset. Therefore, the dynamic analysis will not be a topic in this thesis, but research on dynamic data-flow analysis is encouraged because it could not only give all possible data-flow but could provide all data-flows that happened during time of execution of the program, which do not need to be all data-flows that have been detected by static analysis.

3.2.1 Data Dependence and Call Graphs

As described in section 3.1, data-flow can be calculated by first calculating all data-dependencies and then analyzing the control flow of the system. There are three data-dependency types: True, Anti- and Output dependency [40, 41].

- True data dependency, also called 'Flow-dependency' or 'read-after-write hazard', means that a value is read after it has been written to.
- Antidependency occurs when a value is written after it has been read.
- Output dependency occurs if a value is written to in multiple locations.

Additionally to these three basic types, another type can be found:

- Control dependency

The form of the dependency depends on the sequence of execution in a computer program. If the program executes all statements in a row and allows no jumps between statements, there will be only true data dependency. Even in a single C source code file, the execution sequence of the code statements does not always need to be the same. If pieces of the program can be executed in an arbitrary fashion, for example when function calls happen depending on a random if-condition, all four data-dependencies can exist.
An example is given in listing 3.1, where different execution sequences produce different results:

```
Listing 3.1: Data Dependency Example

1 func1()
2 {
3     a = b;
4 }
5
6 func2()
7 {
8     a = c;
9 }
```

The thesis scope is limited to static source code analysis, that is why the execution of the program cannot be predicted (a dynamic analysis would be necessary) and the three forms of data-dependency can not be separated because the program execution sequence is unknown. Therefore, if the term "data-dependency" is mentioned from now on it will incorporate all four forms of data-dependency. This means that all data dependency types need to be considered for analyzing data-flow, too. Data-dependency can for example be visualized in a Call Graph.

Call graphs contain information about dependencies in source code, the basic elements are nodes and edges [42]. Each edge can be written as \( edge(f, p) \) where function \( f \) calls function \( g \). Grove and Chamber [42] propose an 'Inter-procedural Analysis' for data-flow analysis. Inter-procedural analysis consists of two tasks, first a call graph is constructed at compile-time, and then the call graph is examined to extract the desired program data-flow information from the call graph.

### 3.2.2 Basic Blocks and Control-Flow Graph

A basic block is a sequence of code that has one entry point, the first instruction executed, and one exit point, the last instruction executed. It can have many predecessors and successors [43]. Basic blocks can be used to split a program into separate pieces in which the code execution is linear, that means that all instructions inside a basic block will be executed after each other, from entry point to exit point. This makes the concept of basic blocks useful to data-flow analysis, because if all basic blocks are known, a control-flow graph can be created 'in which the nodes represent basic blocks and the edges represent control flow graphs' [43]. As mentioned in section 3.1, data-dependency and control-flow can be combined to calculate the final data-flow information.
3.2.3 Liveness Analysis, Use-Define Chains and SSA

There are many more kinds of analysis in Compilers that use data-flow information from data-dependency and control-flow analysis as basic information. These further steps are out of the scope of the thesis but shall be mentioned for completeness:

- Liveness Analysis [44]
- Use-Define Chains [44]
- SSA [45]

To conclude, a tool-chain to visualize data-flow needs to be able to obtain information about

1. Data-dependence
2. Basic blocks
3. Control-flow

3.3 Data-Flow patterns in C

Before starting with the development of a data-flow analysis tool-chain, the patterns that shall be detected need to be specified. Because of the thesis’ time constraints only a subset of all theoretically possible data-flow patterns is chosen. The subset of patterns that can be found in table 3.1 have been found in the EEC3 source code and were then discussed with software engineers at Scania (see chapter 5), but a proof of completeness is out of this thesis’ focus. A more wholesome set of patterns can be found in [29].

The subset of patterns is used as a requirement for the software implementation in chapter 4 to find data-flow in the EEC3. The data-flow patterns need to be detected by the final implementation which needs to be able to find all important data-flow patterns in the EEC3 example source code.

Table 3.1 lists the subset of C data-flow patterns that can be found in a typical EEC3 source code file. In some cases, the data-dependency can only be computed when the basic blocks and the control-flow is known, as in case 'function call with argument'. The argument passed to the function can be read or written or both inside the function or it is not used at all, depending on how the function executes. In the case of a 'function call with argument and return value', the return value depends on the content of the function which depends on the argument passed to
the function. Such 'dependency chains' can be of arbitrary length, and the data-
dependency and data-flow needs to be analyzed in an iterative way. In this way,
most of the patterns can be combined with each other to create other forms of
data-dependencies and data-flows, which proved to be one of the major challenges
for the software implementation. Data-flow created by pointers is very hard to
understand, it can analyzed by a 'Points-to' or 'Alias Analysis' ([44], page 370 and
following). This proves to be a complex implementation and could therefore not
be addressed in the implementation.

The table can be read as follows:

- \texttt{var1<-var2}
  Data-dependency: \texttt{var1} depends on \texttt{var2}.
  Data-flow: data flows from \texttt{var2} to \texttt{var1}, \texttt{var2} is read and \texttt{var1} is written.

- \texttt{var<->func()}
  Data-dependency: the value of the variable depends on the function, and the
  value of the function depends on the variable.
  Data-flow: data can flow in both directions.
### Data-flow pattern

<table>
<thead>
<tr>
<th>Data-flow pattern</th>
<th>Example code</th>
<th>Data-dependency</th>
<th>Data-flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable assignment (local-local)</td>
<td>var1 = 5; var2 = var1;</td>
<td>var2&lt;-var1</td>
<td>var2&lt;-var1</td>
</tr>
<tr>
<td>Variable assignment (local-global)</td>
<td>locVar = glob-Var;</td>
<td>locVar&lt;-globVar</td>
<td>locVar&lt;-globVar</td>
</tr>
<tr>
<td>Function call</td>
<td>func1();</td>
<td>no dependency</td>
<td>no (control-flow)</td>
</tr>
<tr>
<td>Function call with return value</td>
<td>var = func();</td>
<td>var&lt;-func()</td>
<td>var&lt;-func()</td>
</tr>
<tr>
<td>Function call with argument</td>
<td>func(var);</td>
<td>func()&lt;var</td>
<td>func()&lt;var</td>
</tr>
<tr>
<td>Function call with argument and return value</td>
<td>var1 = func(var2);</td>
<td>var1&lt;-func(), var1&lt;-var2, func()&lt;var2</td>
<td>var1&lt;-func(), func()&lt;&gt;var2</td>
</tr>
<tr>
<td>Function call with pointer argument</td>
<td>func(&amp;p);</td>
<td>alias analysis</td>
<td>alias analysis</td>
</tr>
<tr>
<td>Field access (Structures)</td>
<td>var = struct.field;</td>
<td>var&lt;-struct.field</td>
<td>var&lt;-struct.field</td>
</tr>
<tr>
<td>Arrays</td>
<td>var = array[i]</td>
<td>var&lt;-array</td>
<td>var&lt;-array</td>
</tr>
</tbody>
</table>

Table 3.1: Data-flow patterns

Certain C syntax that can lead to data-flow could not be dealt with in the scope of this thesis. These are:

- Pointers, function pointers (requires alias analysis)
- (Dynamic) Memory allocation

### 3.3.1 Folder/Architecture Data Flow

Additionally, the EEC3 source code is ordered in folders which represent layers, managers and modules. Those folders are ordered in a parent-child system on the file system. This structure is determined by the software architecture and coding rules. As most software projects are stored in a certain way on the file system, it is necessary to define an abstract architecture related data-flow which depends on how the source code is organized in the file system to be able to visualize the system architecture on top of the source code data-flow. The software implementation uses a 'folder crawler' which recursively steps through all source code folders and calls the lexical analyzer/parser on each source code file which is
found. The information about the folder is then used when the intermediate form is computed, see chapter 4.

Figure 3.5: An example of a source code folder structure/hierarchy
Chapter 4

Implementation

The software implementation connects several components to form an infrastructure to analyze program data-flow. To be able to read in code (lexical analysis), a lexical analyzer is required. To understand the syntax or the ‘meaning’ of the code, a parser is required. The parser output is analyzed for data-dependency, building blocks and control-flow analysis are not a part of the implementation. This is why the implementation can only deliver a part of the complete data-flow analysis. Furthermore, the implementation does not include a data-flow visualization. The implementations output is used in Josip Pantivics thesis work to create a data-dependency visualization [29].

The most important outcomes of the implementation are an interface to other data-flow analysis programs (its ‘intermediate representation’), a fast lexical analyzer and parser implementation, and the capability of analyzing not only the programs internal data-dependencies but also architectural dependencies that arise from how the code is organized on the file system. The tool-chain (fig. 4.1) consists of open-source software except the visualization software yEd. The tool-chain can therefore be used in other thesis projects without licensing problems. Table 4.1 lists the tools used in the final implementation. The tool-chain is developed with the Linux distribution Kubuntu, compiled with the compiler GCC and debugged with the GDB debugger. The tool-chain code is attached to the thesis on CD, and can be compiled under Linux with GCC and Flex and Bison installed. The tool-chain can be run from a console (for example from the Linux ‘Bash’ shell) by calling the executable with the following three arguments, separated by space:

1. Source code input top folder path
2. I-files input folder path
3. Output xml file path
<table>
<thead>
<tr>
<th>Tool</th>
<th>Function</th>
<th>Version</th>
<th>Ref</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLex</td>
<td>Lexicographic analyzer</td>
<td>2.5.37</td>
<td>[46]</td>
<td>section 4.2</td>
</tr>
<tr>
<td>Bison</td>
<td>Parser generator</td>
<td>2.5</td>
<td>[47]</td>
<td>section 4.3</td>
</tr>
<tr>
<td>AST analyzer</td>
<td>Generates the AST from Bison</td>
<td>self-written</td>
<td></td>
<td>section 4.4</td>
</tr>
<tr>
<td>Dependency detector</td>
<td>Detects the data-flow dependencies by analyzing the AST</td>
<td>self-written</td>
<td></td>
<td>section 4.5</td>
</tr>
<tr>
<td>Intermediate representation</td>
<td>Contains the data-flow information in a XML-file</td>
<td>self-written</td>
<td></td>
<td>section 4.6</td>
</tr>
<tr>
<td>GraphML</td>
<td>Used for translation of the abstract intermediate representation XML file to the visualization/graph-oriented GraphML</td>
<td>-</td>
<td>[48]</td>
<td>not implemented in this thesis, see [29]</td>
</tr>
<tr>
<td>Visualization Back-end</td>
<td>Displays the information encoded in the intermediate representation</td>
<td>-</td>
<td>[7]</td>
<td>not implemented in this thesis, see [29]</td>
</tr>
</tbody>
</table>

Table 4.1: Tool-chain details
4.1 Source Code/Precompiler/i-Files

In the early stages of development of the tool-chain it is used on a single C source code file from the EEC3 system. As Scania sticks to the MISRA-C [1] coding standard, all types used need to be typedef’s. An example for a typedef is the code in listing 4.1. All types in the source code that are typedef’d in header files are missing from the source code file and can not be seen and analyzed by the lexical analyzer and therefore need to be manually replaced by their original C types. Rewriting every single source file is not feasible for the whole software system, therefore the output from the Scania precompiler is used. The precompiler places the content of the header files, including the typedef’d types, at the start of the source file and saves the file with a .i extension. This way the type can be seen by the lexical analyzer and the lexical analyzer can process the file.
Listing 4.1: Typedefs according to MISRA-C

1 In the header file:
2
3 typedef unsigned char uint8_t; Where 'unsigned char' is
the original C type.
4 typedef unsigned int uint16_t;
5
6 In the source file:
7 uint8_t var1_u8 = 8;
8
9 Need to be rewritten to:
10 unsigned char var1_u8 = 8;

4.2 The Lexical Analyzer: FLex

FLex [31] is a lexical analyzer/scanner/tokenizer generator that generates a lexical
analyzer which is used to read in character after character from the source code text
file and combines them into tokens that are passed to the parser. The grammar
that is used for the lexical analyzer is a grammar close to the original C grammar
[49]. The lexical analyzer (yylex()) is called by parser (see section 4.3).

The grammar has no support for typedefs (only a source code skeleton is avail-
able), preparser commands, line numbers and comments. Support for typedefs
is added by a check_type function added to the lexical analyzer grammar. If
the lexical analyzer has passed two 'identifier' tokens directly after another to the
parser, the parser recognizes this when it happens for the first time as typedef
(see listing 4.2) instead of issuing an error (two identifier tokens following each
other are not allowed in the C-grammar). The parser adds the typedef type to
a list in the lexical analyzer grammar by calling the 'check_type' function in the
lexical analyzer grammar (see listing 4.3). The next time that the lexical analyzer encounters two identifier tokens, it looks up the first identifier in the typedef list and passes a typedef token followed by an identifier token to the parser.

Listing 4.2: Parser function to capture typedefs

```cpp
if (strcmp(yytname[decl->type_number_of_production], "declarator"))
{
    if (find_child(decl, child_node, "IDENTIFIER"))
    {
        typedef_started = false;
        new_types.push_back(child_node->value_of_production);
    }
}
```

Listing 4.3: Lexical Analyzer function to list typedefs

```cpp
{L}({L}|{D})*
{ count(); return(check_type()); }

int check_type()
{
    string s;
    for (list<string>::iterator iter=new_types.begin(); iter!=
        new_types.end(); iter++)
    {
        s = *iter;
        if (!s.compare(yytext)) {
            // printf("type name: %s\n", yytext);
            return(TYPE_NAME);
        }
    }

    return(IDENTIFIER);
}
```
When programming the tool chain, it proved to be convenient to automatically call the FLex parser generator for each compilation of the tool-chain. Therefore, the following commands were added to the compilation makefile:

```
1 #build the lex.yy.c file which implements the lexical analyzer
   function yylex:
2   flex $(shell find $(CND_BASEDIR) -name *.lex)
```

### 4.3 The Parser: Bison

Bison is a parser generator that generates a parser from a given parser grammar. The parser analyzes the syntax of the lexical tokens that it acquires by calling the lexical analyzer. It takes the tokens from the lexical analyzer and understands what function, what syntax they have based on the sequence of the tokens. It accepts tokens until it can ‘reduce’ the acquired token into a ‘C element’, such as a global variable declaration or a function definition (see fig. 4.3).

Steps to use Bison [50]:

- Write a lexical analyzer to process input and pass tokens to the parser
- Write the grammar specification for Bison, including grammar rules, yy-parse() and yyerror(). Jeff Lee’s ANSI-C grammar [51] is used as a basic grammar for Bison, though it was designed for the older parser generator tool ‘Yacc’. New grammar rules to be able to parse the EEC3 source code, including comments and precompiler instructions, were added to the grammar. Parts of the data-flow analysis related functionality have been written directly into the C grammar (see the following sections and the code attached to the thesis)
- Run Bison on the grammar to produce the parser
- Compile the code output by Bison
- Link the object files to use the parser in the tool chain

The parser tries to reduce all tokens from the source file into a ‘translation unit’. If the parser succeeds and does not find any syntactical error in the token sequence (for example if the source code contains a bug that does not fit into the parser’s C grammar), the translation unit represents a complete, parse-able source code file. A part of the parser grammar that shows a variable assignment, for example of the type ‘z = 1’ where z is a variable, is shown in listing 4.4. Z is in
this case an IDENTIFIER token and reduced to a primary expression, the equals sign is reduced into an assignment operator and in a second step into a assignment expression and the ’1’ is a CONSTANT primary expression. The final production is an an assignment expression, that means that the parser ’understood’ that the code contains an assignment.
The standard Bison C parser grammar can only be used to check a source code file for a correct C syntax. To be able to analyze data-flow, that can for example happen in case of an assignment expression, the grammar needs to be modified. Additional code in the parser production code lines, which are also called parser actions, construct an AST (such as the function 'create_node_new').
4.4 The Abstract Syntax Tree Analyzer

The AST represents the abstract syntactic structure of the C source code file, its nodes are the productions that the parser processed. The complete AST will contain all productions that the parser found in the source code file in a hierarchical parent/child format. The whole source code file is represented as a translation unit production node in the AST, its children are the nodes that the parser has reduced into the translation unit production and their children are again productions of one or more children. This is a convenient format for analyzing source code because aside of all C constructs it also contains hierarchical or 'scope' information encoded in the parent/child node structure. For example, if a variable is assigned a value in a function, the function as a parent node will have an assignment expression node as a child which will have all other productions involved in the assignment as children, and one of the final child nodes that has no children will be an 'IDENTIFIER' node with information about the variable name. The AST is implemented in C++ which enables the AST nodes to be C++ objects, which contain information about the type of the production, the productions value if available (for example a CONSTANT value) and the line number of the production in the source code (see listing 4.5).

Listing 4.5: AST node in C++

```c++
1 node::node(int exp_type_number, char *exp_value, int exp_line_number)
2 {
3     type_number_of_production = exp_type_number;
4     sprintf(value_of_production,"%s",exp_value);
5     line_number_of_production=exp_line_number;
6 }
```

The AST information is then analyzed by a 'Dependency Detector' for data-flow. The data-flow information is then translated into an intermediate representation.

4.5 The Dependency Detector

The dependency detector consists of a number of C++ functions in the parser that search the AST for data-flow patterns. Due to the thesis time limitation, the detector searches only for the most important data-flow patterns specified in section 3.3:
- Variable read
- Variable write
- Function calls

In a first pass through the AST, C++ objects are created for all of the three patterns that were detected. In a second pass, data-flow information is added to the objects. If a variable write is found on the Left hand side (LHS) of an equals sign and a variable read is found on the Right hand side (RHS) of the equals sign, the variable write object will be assigned a connection to the variable read object, and the variable read object a connection to the variable write object. To interface the data-flow information to other programs and the visualization tool, an ‘intermediate representation’ is filled in with the data-flow information. The dependency detector is a recursive tree-walking sub-tree matcher, more on this can be read in section 4.8.

### 4.6 Intermediate Representation

Together with Josip Pantovic [29], a format for an intermediate representation is created. The requirements for the intermediate representation are:

- Store data in a hierarchical format because of the hierarchical structure of the AST and the C source code
- The information in the intermediate representation should be easily accessible so that information can be quickly extracted without focusing on writing an analysis tool for the intermediate representation

As a first experiment, the information is stored in a text file. The hierarchy is represented by adding carriage returns and indentations to the text. Analyzing the text file requires a hand-written text parser again, so that this approach is discarded.

As a result, the XML [52] format is chosen. XML is a hierarchical format and stores information in an easily accessible way. Many tools exists for writing and analyzing XML, and with TinyXML [53], an open-source XML tool that can be integrated into the tool chain by including a C source and a header file, XML files can be created and the data-flow information can be written to it.

The full specification of how data-flow information is stored in the intermediate representation XML file is given in the next section.
<table>
<thead>
<tr>
<th>Data-flow</th>
<th>Description</th>
<th>XML representation</th>
</tr>
</thead>
</table>
| VAR_WRITE       | A value is assigned to a global variable via assignment operators (=, +, -, etc.) | `<p:Function...>
    <p:Relationship
        p:Category="VAR_WRITE"
        p:Target="...v"/>
</p:Function...>` |
| VAR_READ        | A global variable's value is read (all occurrences except when it is directly assigned to via '=' ) | `<p:Function...>
    <p:Relationship
        p:Category="VAR_READ"
        p:Target="...v"/>
</p:Function...>` |
| RETURN          | Function is invoked and its return value is actually used in some way by caller function | `<p:Function...>
    <p:Relationship
        p:Category="RETURN"
        p:Target="...f"/>
</p:Function...>` |
| ARGUMENT        | An argument (one or more) is passed to a function                            | `<p:Function...>
    <p:Relationship
        p:Category="ARGUMENT"
        p:Target="...f"
        p:Content="Vars: z"/>
</p:Function...>` |
| VAR_INIT        | Global variable initialized via another global variable (in declaration outside function) | `<p:Variable...>
    <p:Relationship
        p:Category="VAR_INIT"
        p:Target="...x"/>
</p:Variable...>` |

Table 4.2: Data-flow patterns captured in the intermediate form XML file

### 4.6.1 File Format

The specification for the intermediate form is developed in a way that makes it easy to write extensions for future developers. The following intermediate form is developed together with Josip Pantovic and can also be found in [29], where a data-flow visualization tool-chain based on the information saved in the intermediate form is developed. Not all of the data-flow patterns specified in section 3.3 can be found in the intermediate form due to time constraints, in the future the parser and the intermediate form need to be adapted to detect all possible data-flow patterns (see [29]).

The data-flow patterns that can be found in table 4.2 are translated into the XML format specified in appendix B. For details and code examples of the data-flow patterns, see table 3.1.
4.7 Visualization

From all programs analyzed in section 2.3, yEd proved to be the best solution. The visualization part of the thesis project is part of Josip Pantovics thesis [29].

4.8 Tool-Chain Performance

4.8.1 Quality of Data-flow Analysis

In early development stages, a simple, recursive data-flow detection algorithm was hard-coded to only find VAR_READ and VAR_WRITE data-flows. The simple algorithm works in one pass:

1. In one pass, run through all found VAR_READ and VAR_WRITE nodes and connect them if they occur in constellations that indicate a data-flow, e.g. an ‘assignment_expression’ with VAR_WRITE on the left hand side and VAR_READ on the right hand side

2. Write the found data-flow connections into the intermediate form XML file

In the latest implementation of the data-flow algorithm, the detection is not only limited to VAR_READ and VAR_WRITE data-flow but can match any pattern that the programmer would like to detect. Patterns that shall be detected can be defined as node-objects which represent AST sub-trees, and then passed to
Figure 4.5: data-flow algorithm flow-chart
Figure 4.6: VAR_READ and VAR_WRITE flow chart
a tree-matching function (see section 4.4. The more complex algorithm works in two passes:

1. In a first pass, find VAR_READ and VAR_WRITE AST nodes

2. Store the nodes in a look-up table

3. In a second pass, run through all found VAR_READ and VAR_WRITE nodes and connect them if they occur in constellations that indicate a data-flow, e.g. an 'assignment_expression' with VAR_WRITE on the left hand side and VAR_READ on the right hand side

4. Write the found data-flow connections into the intermediate form

Even though the tool-chain is very flexible, the quality or performance of the data-flow detection algorithm depends on how the patterns are programmed. The prototype nature of the tool-chain programmed can lead to unexpected behavior, for example memory leaks an pointer problems, that lets the program crash. However, FLex and Bison work without problems but their grammars have to be adjusted if certain pre-compiler or non-standard C source code shall be input. In summary, the tool-chain implementation is a proof-of-concept with a working prototype but work needs to be done on the data-flow recognition algorithms. The mix of the plain C FLex and Bison with a self-written object/node-oriented AST and data-flow recognition algorithm written in C++ worked very well. The concept of an abstract syntax tree with parents and children suits an object-oriented C++ implementation very well.

4.8.2 Computational Performance

For the end-user, the execution time of the program matters most which can be measured in CPU performance. Other performance measurements such as memory consumption and amount of function calls can be used for optimizing the tool-chain performance but are of secondary importance. During the development, the tool-chain performance is acceptable in a way that running it on the EEC3 sample code does not take longer than a few seconds.

The performance of the tool-chain can be measured as a whole and for its components. As it proofs to be complex task to measure the tool-chain single components performance, such as FLex and Bison, numbers are only given for the whole tool-chain.

The lexicographic analyzer FLex performs a constant numbers of operations per input symbol, this is still valid for the modified grammar for FLex used in this thesis. The execution time of FLex is linear proportional to the amount of
input symbols, roughly the same is valid for the Bison parser. More about the performance of FLex can be read in the FLex manual, see [54]. The AST analyzer creates one node for each reduction that the parser finds, which means that its performance is also linear proportional to the length of the input.

The expected bottleneck of the tool-chain is the recursive tree-walking data-flow analysis. As the data-flow analysis algorithm is not complete and works only for the VAR_READ and VAR_WRITE data-flows, the performance of an extended, complete production-quality tool-chain will be an issue. Because of the incompleteness and the impact of the data-flow detection algorithm, the performance is analyzed for the tool-chain WITHOUT the data-flow analysis part. However, performance data from FLex, Bison and the AST generation is obtained.

The performance of the tool-chain is tested with the GNU gprof profiling utility which is part of the GNU Binutils [55]. As a comparison to gprof, gperftools (Google Performance Tools) [56] is intended to be used, but because of linking issues and bad documentation the tool does not create any profiling information.

gprof Performance Testing

Setting up the gprof profile is done according to [57], the tool-chain has to be linked and compiled with the option `-pg`. The tool-chain is then run as usual but automatically generates a file that contains the profiler information. gprof can then be invoked on this file and generates a 'flat profile' and a 'call graph' annotated with the time spent in each function, see appendix C.

The tool-chain with Gprof is run on a

- Dell XPS l502X, i7 CPU, 8 cores with 2001Mhz and 8GB memory
- Linux version 3.2.0-58-generic-pae
- gcc version 4.6.3

It can be seen that the tool-chain total execution time (with the EEC3 code as input) is around three seconds (in this case 2.98 seconds).

In the following, the total execution time in percent is listed for different parts of the tool-chain:

- The lexer functions check_type() (18.5%), yylex() (14.8%) and count() (5.2%) 38.5%
- The main parser function yyparse() 15.1%
The hand-written AST functions connected to the AST node generation `std::list (9% + 5.5% + 4.9% + 2.2%) and create_node_new() (5.4%)`, total 27%. The missing 19.4% are spent in smaller tasks related to the AST generation, which makes a total of 46.4%.

The hand-written AST generation functions require almost half of the programs execution time, which indicates that either the FLex/Bison implementations are very fast or the AST generation algorithm implemented in C++ is inefficient compared to the mature C implementations of FLex and Bison.

The performance analysis shows that the basic functions of lexicographic analysis, parsing and AST generation take around three seconds. Any further data-flow algorithms will add CPU time on top of this, extending the tool-chain execution time depending on how large the data-flow patterns that are to be searched by the recursive tree-matcher are.

gprof proofs to be an easy to use tool and can be recommended for further optimization and analysis of the tool-chain.
Chapter 5

Interviews

As soon as the main parts of the software tool-chain are implemented, interviews with software engineers were done to determine how a visualization can be used at Scania, what kind of data-flow programmers would like to have visualized and how large the time-savings during debugging or drawing visualizations by hand can be. This input shall be used to verify that the way that the implementation will analyze and visualize the data-flow matches the expectations of the software engineers.

Seven software developers and one system architect were interviewed. Except the system architect, all of the developers develop in C for systems alike the EEC3. For the interview questions asked see appendix A.

The interview has two sections, a system visualization and a implementation related section.

5.1 System Visualization

These questions try to estimate in which situations the developers use a visualization and where additional visualization can be helpful. The final visualization tool chain can then be customized to fit into the existing practices. The answers for this section were very different depending on what the interviewed person worked with. A typical software developer would use visualization like a call-graph, preferably directly in the IDE that he uses to develop the source code. Software developers also need to document the source code, typically by adding comments directly to the source code but also in external text document were visualizations are required to convey the source code meaning to other developers and system architects which are and can not be familiar with every aspect of source code parts that they do not develop themselves. For the implementation to be useful for source code developers this means that the visualization tool chain
needs to replace a hand-made visualization which is able to display all details of
the source code, such as data-flow between variables. The interviews also show
that the most difficult and time-consuming task that programmers are concerned
with is the ‘tracking’ of signals in the source code. This can be for example sensor
information such as the ambient temperature, that is acquired in one source file,
filtered in a another source file and then distributed to a couple of other source
files by a pointer reference. The problem is to track in which source files the sensor
value could have been affected, recalculated or filtered. Jonian Grazhdani’s pro-
gram, see section 2.1, is a tool that was created just for this purpose. A data-flow
analysis tool that can give all data-dependencies between the source code files is
considered to be very helpful in this case by the interviewed developers. One more
point that developers are considered with is adherence to coding standards such
as MISRA C [1]. This is mostly out of scope of the thesis topic and is not possible
to display with the current implementation. Furthermore, tool to analyze source
code for coding rules violations are already in use at Scania. A few developers use
graphical programming languages to generate or simulate source code, there is a
model for the EEC3 that is developed in Matlab/Simulink for example. This is a
potential competitor for an automatic visualization because if code is visualized in
a model already, there is no need to analyze the generated code from this model
again. Anyhow, most of the models do not deliver production-quality source code
yet so a visualization tool chain could still be useful.

In contrast to a software developer, a system architect would answer that he
needs to create abstract visualizations of how the system, not the source code,
works. A system architect will typically use a graphic design tool such as a UML
modeling tool like yEd or Enterprise Architect to visualize system architecture
manually and can be disturbed by an automatic visualization that has too many
architecture-unrelated details. On the other hand, the interviewed system archi-
tects did not have much information on how their models are realized in the source
code, and in some cases they never read parts of the source code because this is out
of their responsibility. However, the source code developers often do not stick to
the system architecture and for system architects, a visualization could help to see
how the real source code architecture differs from their models. One interviewee
stated exactly this by saying that the visualization could ‘connect developers to
architects’.

In all interviews it is mentioned that the visualization could be helpful in
explaining the system to newcomer developers or architects.

In one case it is mentioned that visualizations are needed for code reviews,
which are in turn needed for Scania to fulfill a coming software standard ISO26262
‘functional safety’. A visualization software could in this case help Scania to ful-
fill ISO26262 goals. Another aspect of visualization use mentioned in the same
interview is ‘model based development’, an uprising trend in automotive software design that requires the creation of visualizations.

A set of questions deals with time demand and accuracy of the visualization. For all software developers interviewed, the amount of time or working hours that they need to create a visualization of source code manually does not matter. That means that there is always enough time to complete the task even without an automatic visualization tool. One reason is that all changes in the source code are documented in text documents which replace the visualization. If a visualization is necessary, it should however be completely accurate as all data-flow information should be detected to make accurate estimations. According to the interviews, software developers, in contrast to software architects, are dealing with highly detailed visualizations of a small area of code, for example several functions or modules that work together.

For system architects in contrast to software developer, it is more important to be able to deliver complete visualizations, not only a detail, of the software architecture fast because understanding the software architecture from source code is very time demanding, up to three weeks for a single code module consisting of several source files. For the software architects it is also relevant that an automated visualization can be run often to capture (unwanted) changes in the architecture or architecture violations that could not be seen if the visualization need to be made by hand and take a very long time to make.

5.2 Implementation

In the implementation part of the interviews, the interviewed persons are given a short introduction to how the tool chain can be used and how the output will look like. The questions in this part of the interview are designed to get feedback and ideas for improvement from the developers. From the analysis of the interviews it turns out that the interviewed persons are mostly skeptical about the quality of any data-flow visualization, not only the presented tool-chain, in terms of usability and correctness of the data-flow information. A reason for this could be that many programmers are not familiar with data-flow analysis and are not presented with any proof of correctness of the visualization of the data-flow analysis algorithm during the interviews. Especially one senior software developer was concerned that this flaw of missing specifications and proofs could render the whole tool-chain unusable for any serious software development. Some programmers would like to have an easily manipulate-able (for example drag&drop/double click expansion of a data-flow node) visualization which can be modified by company or ECU specific input files. Some developers emphasize again the importance of ‘signal-trace-ability’ which is a feature that is not planned to be incorporated in the tool chain.
One developer has been involved in a thesis project in which a parser was created that can parse C code for ECU fault diagnosis keywords. This developer would like to see a link to the text documentation of the source code in the visualization, an interesting feature which is implemented in Doxygen already and is not a part of this project. However, by the lexical analyzer passing line-number information to the parser, a similar feature could be implemented based on the source code text file locations of data-flow. As a side notice, it is mentioned in the interviews that the documentation and architecture of the source code (for example EEC3) was created after the source code development and therefore is hardly correct. This can be seen by comparing for example the most up-to-date EEC3 source code with the software architecture: drawings in the architecture documents are not up-to-date and the source code violates the architecture in some cases. Here, the tool chain could be used to visualize the current architecture and to rewrite the architecture documents.

To sum up the interviews: all interviewed persons show interest in the tool chain and its further development. Most of the interviewed persons have an idea where a visualization could help in their work, but some of the developers were concerned with the tool chain accuracy. Only two developers do not think that the tool chain at its current status could be helpful, mostly because they either only rely on the source code itself and do not create any visualizations, or they create visualizations very seldom.

Ideas for further improvements from the interviews are:

- Make it possible to 'track' a signal through the code and source code files.
- Extract only the architecture of the source code by means of externally supplied architecture definition files. An architecture description language could be useful.
- Create a documentation from the source code which incorporates data-flow information (see Doxygen).
- Modify the data-flow detection algorithm by external files. This is a part of Josip Pantovics thesis in which he uses XPath for this purpose [29].
- Link the visualization closer to ISO26262 and model based development. This is a very abstract goal and needs further research.
Chapter 6

Conclusion

The goals that were defined in the beginning of the thesis have not all been fulfilled. As expected when setting the goals, for some tasks there was not enough time, especially the point of ‘If there is time, use the output of the intermediate representation with available software to create a graphical representation’. Especially the goals that deal with creating an automated visualization have not been reached, as the main focus of the thesis had to be put on source code analysis or the ‘front-end’ of the visualization tool-chain. However, in the part of source code analysis and data-flow analysis, the goals have been reached. A thorough review of existing software applications that can analyze source code was made, and a definition of data-flow was extracted from available literature. All instruments required for a data-flow analysis have been combined in a lexical analyzer/parser/data-flow detection tool chain, they are fully customize-able, easily understandable and open-source. By reusing existing grammars, the tool-chain has been held as standardized as possible. The most problematic theoretical part of the thesis proved to be the definition of the data-flow patterns. It is a topic that is still researched, and the data-flow patterns that are represented in chapter chapter 3 and that can be detected by the tool chain represent only a small percentage of all possible data-flow patterns. The tool chain lacks a mathematical/logical proof of correctness of its data-flow analysis algorithm, which had to be held very simple due to the time constraint on the thesis. Further investigation on how data-flow analysis is done in production quality compilers, such as GCC, would certainly have been helpful. The goal of the program to have a good architecture has been fulfilled, by using standard available software components with a good documentation such as FLex and Bison, and by using C++ as a programming language which works well with creating abstract syntax tree objects.

To be able to evaluate how the tool chain can be used by developers at Scania, interviews were conducted and the results can be used in further research. The way that the tool chain approaches the problem is appreciated by the interviewed
persons and attracts great interest, however the tool chain can be further refined according to the interviews. From the interviews it can be gathered that the visualization, when complete, can facilitate the development process.

An intermediate representation was created in XML in a format that can be easily used to interface to data-flow analysis programs of all kinds. However, the algorithms that search the code for the data-flow information, the dependency detector, is only capable of finding very basic data-flow information.
Chapter 7

Future Work

7.1 Visualization

A great part of work has to be done on visualizing the data-flow saved in the intermediate representation. A good solution to this is presented in Josip Pantovics thesis [29]. The visualization in Josips thesis is done in yEd, which is a proprietary program and probably this could be replaced by another visualization program, like Gephi or DOT which were mentioned in section 2.3. To get a user-oriented visualization, more interviews should be conducted, and a demonstration version of the tool chain could be shown and the user interaction could be recorded to optimize the visualization, and different visualization perspectives for software developers and software architects.

7.2 Data-flow analysis

The tool chain is designed without support for pointers and function pointers. Additional data-flow analysis function needs to be implemented in the tool chain to catch these data-flow connections, too. As the implementation of these features will lead to a tool chain that in its functionality comes close to a compiler (without the machine-code generation back-end and without optimizations), it would be more feasible in the amount of time and amount of work spent to use a production quality compiler such as GCC or the emerging LLVM (with Clang), as demonstrated in section 2.2.1. The compilers AST could be used instead of the tool chain generated one by using certain compiler flags. This could tackle the preprocessing step that is necessary to generate the .i-files, it could get rid of maintainability issues because a production compiler grammar is usually continuously updated by its developers, and is well documented.
Bibliography


57


[52] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. URL: http://www.w3.org/TR/xml/ (visited on 01/14/2014).


Appendix A

Interview Template

1 Interview for Data Flow Visualization Theses
2 Martin Pruscha, Josip Pantovic
3 [Presentation of us and our topic, relation to ISO26262]

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

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

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

[Presentation of our application in its current state and target functionality within the scope of the thesis]
4. Our 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 our proposed application (time saved, money for proprietary software saved, etc)?

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

Intermediate Form XML Schema

Listing B.1: Intermediate Form XML Schema

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <xs:element name="Module">
    <xs:complexType mixed="true">
      <xs:sequence maxOccurs="unbounded" minOccurs="1">
        <xs:element maxOccurs="unbounded" minOccurs="0" name="Variable" type="p:Variable"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="Function" type="p:Function"/>
      </xs:sequence>
      <xs:attribute name="BaseName" type="xs:string" use="required"/>
      <xs:attribute name="Path" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

61
<xs:element maxOccurs="unbounded" minOccurs="0" name="DataFlow" type="p:DataFlow"/>
</xs:sequence>

<xs:attribute name="Name" type="xs:string" use="required"/>
<xs:attribute name="Type" type="xs:string"/>
<xs:attribute name="File" type="xs:string"/>
<xs:attribute name="Line" type="xs:integer"/>
</xs:complexType>

<xs:complexType name="DataFlow">
    <xs:attribute name="Category" type="p:category" use="required"/>
    <xs:attribute name="DataType" type="xs:string"/>
    <xs:attribute name="Target" type="xs:string" use="required"/>
    <xs:attribute name="Content" type="xs:string"/>
    <xs:attribute name="Indirection" type="xs:integer"/>
</xs:complexType>

<xs:simpleType name="category">
    <xs:restriction base="xs:string">
        <xs:enumeration value="RETURN"/>
        <xs:enumeration value="VALUE_ARG"/>
        <xs:enumeration value="POINTER_ARG"/>
        <xs:enumeration value="VAR_READ"/>
        <xs:enumeration value="VAR_WRITE"/>
        <xs:enumeration value="CONST"/>
    </xs:restriction>
</xs:simpleType>
</xs:schema>
# Appendix C

**gprof Profiling Results**

This is an excerpt from the gprof profiler output file:

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>self seconds</th>
<th>calls</th>
<th>s/call</th>
<th>total time</th>
<th>seconds</th>
<th>s/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.46</td>
<td>0.55</td>
<td>0.55</td>
<td>999896</td>
<td>0.00</td>
<td>0.00</td>
<td>check_type()</td>
<td></td>
</tr>
<tr>
<td>15.10</td>
<td>1.00</td>
<td>0.45</td>
<td>319</td>
<td>0.00</td>
<td>0.00</td>
<td>yyparse()</td>
<td></td>
</tr>
<tr>
<td>14.77</td>
<td>1.44</td>
<td>0.44</td>
<td>2759729</td>
<td>0.00</td>
<td>0.00</td>
<td>yylex()</td>
<td></td>
</tr>
<tr>
<td>9.06</td>
<td>1.71</td>
<td>0.27</td>
<td>104070673</td>
<td>0.00</td>
<td>0.00</td>
<td>std::list&lt; std::string, std::allocator<a href="">std::string</a> &gt;::end()</td>
<td></td>
</tr>
<tr>
<td>5.54</td>
<td>1.88</td>
<td>0.17</td>
<td>103013413</td>
<td>0.00</td>
<td>0.00</td>
<td>std::_List_iterator<a href="">std::string</a>::operator++(int)</td>
<td></td>
</tr>
<tr>
<td>5.37</td>
<td>2.04</td>
<td>0.16</td>
<td>10090755</td>
<td>0.00</td>
<td>0.00</td>
<td>create_node_new(int, node*, ...)</td>
<td></td>
</tr>
<tr>
<td>5.20</td>
<td>2.19</td>
<td>0.15</td>
<td>13905080</td>
<td>0.00</td>
<td>0.00</td>
<td>count()</td>
<td></td>
</tr>
<tr>
<td>4.87</td>
<td>2.33</td>
<td>0.14</td>
<td>104013309</td>
<td>0.00</td>
<td>0.00</td>
<td>std::_List_iterator<a href="">std::string</a>::operator!=(std::_List_iterator<a href="">std::string</a> const&amp;) const</td>
<td></td>
</tr>
<tr>
<td>4.03</td>
<td>2.46</td>
<td>0.12</td>
<td>103411327</td>
<td>0.00</td>
<td>0.00</td>
<td>std::_List_iterator<a href="">std::string</a>::operator*(int) const</td>
<td></td>
</tr>
<tr>
<td>2.18</td>
<td>2.52</td>
<td>0.07</td>
<td>10091864</td>
<td>0.00</td>
<td>0.00</td>
<td>std::list&lt; node*, std::allocator&lt;node*&gt; &gt;::M_create_node(node* const&amp;)</td>
<td></td>
</tr>
<tr>
<td>1.85</td>
<td>2.58</td>
<td>0.06</td>
<td>105070569</td>
<td>0.00</td>
<td>0.00</td>
<td>std::_List_iterator<a href="">std::string</a>::List_iterator( std::detail::_List_node_base*)</td>
<td></td>
</tr>
<tr>
<td>1.68</td>
<td>2.62</td>
<td>0.05</td>
<td>10091864</td>
<td>0.00</td>
<td>0.00</td>
<td>std::list&lt; node*, std::allocator&lt;node*&gt; &gt;::M_insert(std::_List_iterator&lt;node*&gt;, node* const&amp;)</td>
<td></td>
</tr>
<tr>
<td>1.01</td>
<td>2.65</td>
<td>0.03</td>
<td>10537808</td>
<td>0.00</td>
<td>0.00</td>
<td>std::allocator&lt;std::List_node&lt;node*&gt; &gt;::allocator()</td>
<td></td>
</tr>
<tr>
<td>1.01</td>
<td>2.69</td>
<td>0.03</td>
<td>10537808</td>
<td>0.00</td>
<td>0.00</td>
<td>std::List_base&lt;node*, std::allocator&lt;node*&gt; &gt;::List_base()</td>
<td></td>
</tr>
</tbody>
</table>
% the percentage of the total running time of the
time program used by this function.
cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.
self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.
calls the number of times this function was invoked, if
this function is profiled, else blank.
self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.
total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
function is profiled, else blank.
name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.34% of 2.98 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>99.2</td>
<td>0.00</td>
<td>2.95</td>
<td>1/1 main</td>
<td>&lt;spontaneous&gt;</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>2.95</td>
<td>1/1 crawler::fill(char*, char*, package_node*) [3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/27 package_node::package_node() [348]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/1 package_node::write_to_xml(char*) [405]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-----------------------------------------------

| [2]   | 99.2   | 0.00 | 2.95     | 362 parser::parse_file(char*, module_node*, char*) [2] |
|       | 0.45   | 2.49 | 319/319  | yyparse() [4] |
|       | 0.00   | 0.02 | 319/319  | detector::define_sub_tree(node*) [31] |
|       | 0.00   | 0.00 | 319/319  | std::list< |
|       |        |      |          | std::string, std::allocator<std::string> > |
|       |        |      |          | ::clear() [293] |

-----------------------------------------------

|       | 26     |      |          | crawler::fill(char*, char*, package_node*) [3] |
| [3]   | 99.2   | 0.00 | 2.95     | 1+26 crawler::fill(char*, char*, package_node*) [3] |
|       | 0.00   | 2.95 | 362/362  | parser::parse_file(char*, module_node*, char*) [2] |
crawler::noneligible_file(char*) [231]
crawler::eligible_file(char*) [232]
module_node::module_node() [258]
crawler::trim_module_name(char*) [259]
__gnu_cxx::__normal_iterator<char*, std::string>::operator-(int const&) const [261]
std::list<module_node, std::allocator<module_node>> ::push_front(module_node const&) [275]
crawler::ignore(char*) [297]
package_node::package_node() [348]
std::list<package_node, std::allocator<package_node>> ::push_front(package_node const&) [360]
crawler::fill(char*, char*, package_node*) [3]
parser::parse_file(char*, module_node*, char*) [2]
yyparse() [4]
yylex() [5]
collect_typedef_info(node*) [23]
node::node(int, char*, int) [15]
std::list<node*, std::allocator<node*>> ::push_back(node* const&) [64]
std::list<node*, std::allocator<node*>> ::clear() [69]
node::~node() [72]
string_to_number(std::string) [125]
yydestruct(char const*, int, YSTYPE*, YYLTYPE*) [227]
yy_lex() [5]
check_type() [6]
count() [13]
include() [39]
yy_get_next_buffer() [129]
yy_get_previous_state() [132]
yy_wrap [76]
yy_get_buffer_stack() [404]
yy_load_buffer_state() [226]
yy_create_buffer(_IO_FILE*, int) [394]
std::list<std::string, std::allocator<std::string> >::end() [8]
std::_List_iterator<std::string>::operator++(int) [11]
std::_List_iterator<std::string>::operator!=(std::_List_iterator<std::string> const&) const [14]
std::list<std::string>::begin() [63]
detector::define_sub_tree(node*) [31]
yy_parse() [4]
create_node_new(int, node*, ...) [7]
std::list<node*, std::allocator<node*> >::push_front(node* const&) [9]
node::node(int, char*, int) [15]
std::list<std::string, std::allocator<std::string> >::push_back(std::string const&) [48]
check_type()

std::list<std::string, std::allocator<std::string>> ::end()