



UPPSALA
UNIVERSITET

IT 16 023

Examensarbete 30 hp
Maj 2016

Architectural model synthesis from source code using Simulink and Hierarchical Function Call-Graphs

Maksim Olifer

Institutionen för informationsteknologi
Department of Information Technology



UPPSALA
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet
UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
<http://www.teknat.uu.se/student>

Abstract

Architectural model synthesis from source code using Simulink and Hierarchical Function Call-Graphs

Maksim Olifer

Modern software systems developed in the automotive industry are very complex. In order to analyze, understand, and document these software systems, architectural models

of the systems at different abstraction levels are used. However, these models are typically ambiguous and inconsistent with the implementation. This thesis presents an approach to construct an unambiguous model of C code in an automatic manner, with a focus on architecture consistency by employing the Simulink environment extended by

an external custom GUI. Such approach also facilitates compliance with the functional safety standard ISO 26262 that requires models of software systems (including legacy code), where the models capture both its behavior and structure.

More specifically, we develop a method for hierarchical modelling in Simulink and describe mapping to the actual C code architecture expressed by distinct abstraction levels (e.g. layers, modules, functions). Although Simulink is capable of handling modular structures, there is a lack of proper visual representation support, which at the moment can reflect only inter-functional dependencies in terms of caller-to-callee relations, omitting any hierarchical view of abstraction layers. An attempt to extend graphical features of Simulink had several drawbacks and performance issues.

As an enhanced solution, external GUI was developed for enabling a "complete" representation of the code architecture. For that purpose, reverse engineering approaches

were employed with a help of LLVM compiler infrastructure and poolalloc project. A further analysis of LLVM IR allowed extracting a function call graph, including indirect function calls with a satisfactory precision (which can be possibly improved). For a better performance, the resulting graph was placed in a database, which allowed to dynamically select particular parts and relations of the graph.

The developed tool-chain was evaluated on a two production software system called COO7 and GMS. This thesis was done at Scania CV AB in Södertälje, Sweden.

Handledare: Jonas Westman
Ämnesgranskare: Philipp Rümmer
Examinator: Justin Pearson
IT 16 023
Tryckt av: Reprocentralen ITC

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Simulink as a choice of modeling formalism	4
1.3	Task Description	6
1.4	Overview of the solution	7
2	Background	8
2.1	What does a model represent?	8
2.2	Model Properties	9
2.3	Embedded software at Scania	11
2.3.1	Scania CAN bus network architecture	11
2.3.2	Software structure	12
2.3.3	Programming language guidelines	13
2.4	Simulink	15
2.4.1	Modularity in Simulink	16
2.4.2	Code Transformation using LLVM	18
2.4.3	Data Structure Analysis	19
3	Related Work	22
4	Solution Design	25
4.1	Representation of Function-Call Graph	26
4.1.1	Hierarchical Function-Call Graph	26
4.2	Modeling of C-code in Simulink	28
4.2.1	Mapping between C-code and Simulink	28
4.2.2	Constraints of Simulink	29
4.2.3	Possible solutions	30
4.2.4	Extension of the graphical representation in Simulink	31

4.2.5	Compromise between behavior expressivity and pre- senting structure	33
5	Implementation	35
5.1	Transformation pipeline	35
5.2	Code compilation to LLVM IR	36
5.2.1	Call-Graph Recovery	37
5.2.2	Web GUI	39
6	Evaluation	42
6.1	Level of Formality	42
6.2	Transformation from C-code to Simulink	42
6.3	Architecture Recovery from C-code	43
6.4	Redundant Concepts	44
6.5	Automation of Models Mining	44
6.6	Graphical Representation	44
6.7	General Opinion	45
7	Conclusions	46
7.1	Further Development	47
	Bibliography	48
A	Visualisation abstractions	52
A.1	Definitions	52
A.2	Hierarchy of ports	53
B	Data-flow mechanics	54
B.1	Direct Assignment	54
B.2	Function parameter passing	55
B.3	Function return value	56
B.4	Indirect assignment	57
B.5	Modify-by-reference	57
B.6	Function call with side-effects	58
B.7	Environmentally tied variables	59
B.8	Modification through interrupt	59

Chapter 1

Introduction

A modern automotive vehicle is a complex technical system, containing many electronic, mechanical, and software parts. Typically, a vehicle can contain 50 ECUs (Electronic Control Units) controlling a large number of distributed functions (e.g. engine and transmission systems, brake module, door lock), many of which are safety-critical. Seamless integration and coordination of these modular systems in multiple ways that are consistent and allow to achieve desired properties is an extremely sophisticated task.

1.1 Motivation

Generally, in software development process it is very common nowadays to use models of the system at various abstraction levels to analyze, understand, and document the system. A well-known modeling concept used at different abstraction levels is *component-based modeling* where an isolated element of the system is modeled by an interface and its behavior. Interface and behavior might imply various meanings (e.g. set of operations, set of signals) depending on a context of a specific modeling framework. For instance, an interface could be described by a set of input and output port variables, while behavior could be a set of all possible execution traces over these variables respectively [1].

Components are widely used in well-known Modeling Languages (MLs), such as Modelica and SysML, and tools (e.g. Simulink) [2], [3].

Considering the complexity of these automotive systems, functional safety might be an issue. With respect to this, special functional safety standards

are applied in the automotive industry. One of the widely-used standards is *ISO 26262* [4], and it concerns the functional safety of electrical and electronic systems including their life cycle and aims to mitigate hazardous situations.

ISO 26262 requires that system requirements are decomposed starting from top-level requirements, i.e. *Safety Goals* (SG), down to software (SW) and hardware (HW) requirements. Hence, requirements at all levels, excluding SG, must also be "allocated", i.e. assigned, to a specific element of the system in architecture. At a software and hardware levels, these elements of the system are defined as "*technically and logically separated elements*" [4]. One way to model these elements is using *components* that offer a clear separation between the elements themselves and their environment.

Accordingly, in ISO 26262 decomposition of requirements is an integral part of a top-down development approach, where components can be used all the way down to the generation of code, with for example Simulink. However, given that a large portion of the code is still written manually, and considering legacy code, it is important to guarantee that this code fulfills respective functional safety requirements. In fact, there is not much support for this today.

In the recent years, a *Model Base Development* (MBD) [5] became leading approach for software systems development in various industries [6], e.g. automotive. Models' behaviour can be simulated without linkage to any target hardware, and their functional requirements can be verified in a more accurate and structural way. Moreover, various formalisms, e.g. Matlab/Simulink, provide special certified tools for that purpose, e.g. Simulink Verification Tools Qualified to ISO 26262, facilitating the creation of a safety-critical software. The resulting model can be converted to the actual code for a target platform. Such code has a sufficiently good quality and fulfills requirement constraints posed on the system. Although the generation process might be customizable to a great extent, the produced code is usually unstructured, hardly readable by a human, and optimized only for machine execution. Theoretically, it could be possible to apply reverse engineering techniques to translate this generated code back to the model, but it is unlikely that the resulting model would be the same. However, a more reasonable task might be acquiring a model from an existed manually-written code. Obviously, manual conversion of the code to MBD is not easy, error-prone and time-consuming, raising demand in automated solution.

In general, possible mixture of *Model Base Development* and common software development (SD) could result in the following potential scenarios:

- Manually written code. There are no models used in this scenario; all the systems are implemented manually. One method to ensure the safety of the software is to employ continuous integration (CI) and continuous delivery (CD) approaches (these approaches can be effectively used also with models).
- Model-based development. This way of development is a target of many large organizations since it eliminates any need to write code manually, improves the speed of implementation as well as the quality of produced software. The number of people who can be involved in actual production expands significantly, without being limited by pure software programmers. High-level models are usually easy to create, comprehend and modify. Furthermore, most of the modeling support verification in accordance with established functional safety standards applied in automotive industry.
- Combination of two previous scenarios. This case is quite common and is an intermediate transition from manually written code to MBD. The main challenge here is to transfer all legacy code to the models. This scenario can tag Scania's current state of development.

One possible approach to begin with the issue is to allow a systemic decomposition of an architecture and requirements in parallel as indicated in Figure 1.1. In this scheme, an element \mathbb{E} represents a model of a software/hardware object on different levels of the architecture described by its interface and behavior, whilst a guarantee R is a requirement on respective \mathbb{E} . Therefore, this decomposition is naturally combined with components, which might be evaluated in order to make modeling of (implemented) code feasible.

Summarizing the discussion, modeling of source code can bring several potential advantages:

- A generated model might be more comprehensible than a written C-code, facilitating modular development and improving overview of the complex relations.
- System software requirements might be mapped in a more natural fashion (e.g. one-to-one), using hierarchical structures.
- Executable models support richer capabilities for analysis, e.g. validation and verification.

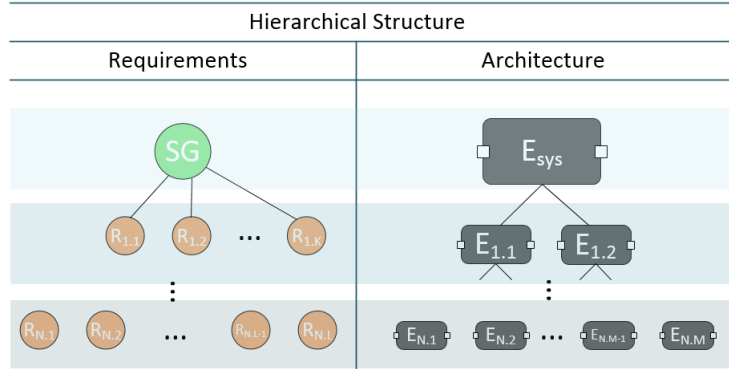


Figure 1.1: Hierarchical structure.

- A model itself can serve as a good source of documentation.

At the same time, the choice of *component-based modeling* is not casual and formed by the following key factors:

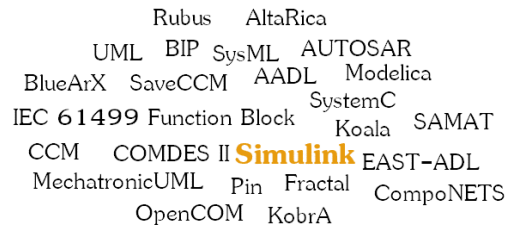
- Component-based models are an appropriate formalism to capture the process of decomposing requirements and architecture in parallel, as required by ISO 26262. This has been shown in several *contract* articles [1].
- It is one of the most common formalism and accepted in the industry.
- Modeling formalisms exist to capture systems as component models, but not to model implemented software as it is. Adding the ability to model software using components, would mean that the formalism can be used at all levels. If different modeling formalisms are used at different levels, the model transformation is needed.

Although the methods developed in this thesis are concerned specific case of Scania's software, they can be used in other contexts as well.

1.2 Simulink as a choice of modeling formalism

Many formalisms are used to model software systems (Figure 1.2). Our aim is not merely a graphical visualization. For the future extensions, a high emphasis should be put on the ability to model behavior that can be executed

(in accordance with ISO26262). Therefore, all formalisms that cannot provide this feature must be declined. After this filtering, the number of remaining options is not large (e.g. Simulink, Modelica, LabVIEW). However, why should we select especially Simulink as a modeling formalism and not any other alternatives?



A word cloud of modeling formalisms. The words are arranged in a roughly circular pattern. The word 'Simulink' is highlighted in orange and is positioned in the center. Other words include: Rubus, AltaRica, UML, BIP, SysML, AUTOSAR, BlueArX, SaveCCM, AADL, Modelica, SystemC, IEC 61499, Function Block, Koala, SAMAT, CCM, COMDES II, EAST-ADL, MechatronicUML, Pin, Fractal, CompoNETS, OpenCOM, and Kobra.

Figure 1.2: Examples of modeling formalisms.

Scania actively integrates *Model-Based Development* approaches to software design process. As a result, *Simulink* is widely adopted by the company as a main formalism for MBD. At the same time, Simulink is a common choice of automotive companies due to its expressiveness that is essential for system design. Moreover, one of the principals features of Simulink is an ability to construct hierarchical models, which can be simulated and executed, supporting multiform analysis, e.g. validation and verification (V&V). Considering that fact, it is important for the company to investigate a feasibility of using Simulink as a modeling formalism to capture software architecture as hierarchy of components and data flow between them (for a further extensions).

Altogether, the rationales for selecting Simulink are presented in the following list:

- Ability to construct hierarchical models.
- **Behavior modeling.**
- **Simulation.**
- Verification.
- Adopted by Scania for MBD.
- Widely used in automotive industry.

1.3 Task Description

From what has been described above, it follows that the one of the main targets of the project is to investigate the possibility to model implemented software using component-based approach. This goal can be accomplished by evaluating the ability of Simulink to model an architecture of low-level (implemented) C-code for an example system at Scania. In this context the following research questions arise:

- What are the levels of formality supported? (e.g. can the (hierarchical) model be simulated?)
- What is a transformation from C-code to Simulink? To answer the question, there is a need in discussions relating to a level of granularity, aspects of imperative languages that can be captured with data-flow, consistency of semantics.
- How an architecture can be recovered from C-code? Defining a proper methodology to extract architectural artifacts from the source code is a vital step in the performed modelling.
- What concepts are redundant? Formalisms (e.g. Simulink) can expose reach set of functionality handling a variety of tasks. However, not all of the instruments might apply to the current problem, and therefore, it could be useful to highlight it explicitly.
- Can the task to acquire the models be automated? For Scania, it is highly desirable to have an automated solution to be able to cover a vast repository of legacy source code.
- Will it be possible to build a graphical representation, which would capture all required information? Unambiguous and clear visualisation is a key point to make it compliant with functional standards as well as comprehensible for a human.

Along with the research topics, the following practical goals can be highlighted:

- Pick architectural code representation for evaluation (e.g. function call graph, data flow graph, control flow graph).

- Define an appropriate graphical representation that could reflect properties of the model in a flexible way (e.g. with filtering).
- Find an efficient way to extract this architecture automatically from the source possibly containing millions lines of code with intensive usage of complex structure and pointers.
- Synthesize a model in Simulink capturing extracted architecture.
- If Simulink is not capable of capturing all dependencies, try to:
 - Extend Simulink environment.
 - Implement a lightweight solution to visualize required code structures, having Simulink as a back-end for the possibility of behavioural modeling.
- Perform benchmarks of the solution.

1.4 Overview of the solution

The accomplished work has identified an automatic approach for capturing software architecture in the model. Along with this, principal weaknesses in displaying structural dependencies were revealed in Simulink. An attempt to realize internal improvements within the formalism exposed a number of shortcomings that prompted the use of an external (web-based) solution for visualisation. Architectural model mining was performed by static analysis of source code intermediate representation produced by *Low-Level Virtual Machine* (LLVM).

Chapter 2

Background

Before starting the process of solution design, it is essential to understand the working environment, determining its initial conditions and induced limitations. Hence, a discussion of software systems features in Scania and general principles for achieving modularity in Simulink will take place in this chapter.

2.1 What does a model represent?

A general model can be viewed as another (possibly abstract) representation of some object. The object in its turn can carry a multitude of meanings. For instance, (executable) software code could be thought of as a model, modeling objects of the physical world. Likewise, a behavior of the imperative code might be captured with a help of state chart, which is also possible to qualify as a model. In the case of object-oriented program, it is highly common to construct a class diagram giving a good insight into software structure and dependencies. Therefore, should we consider the latter example as a model as well?

For a better comprehension it would be expedient to obtain a more accurate meaning of the model within this thesis. Since the aim of the work is to model a programming code written in C, let us assume that the model here will be considered as a semantically consistent representation of the source code. Such representation could complement the object (C-code) by adding desired properties (e.g. contracts, graphical visualization, overview of hierarchical dependencies) and possibly canceling a set of native features (e.g.

imperative nature, organization, memory management). At the same time, we can define a model as a *complete* if the translation to the model is reversible, i.e. for a given model of software it should be possible to generate a new code with the same behavior as in the original program, but with no requirements for preserving of the exact structure. For example, to achieve reversibility in our case, it is necessary to model not only a structure, but also a behavior (Figure 2.1). However, since we concern only with architectural models in the current work, our models are not complete models.

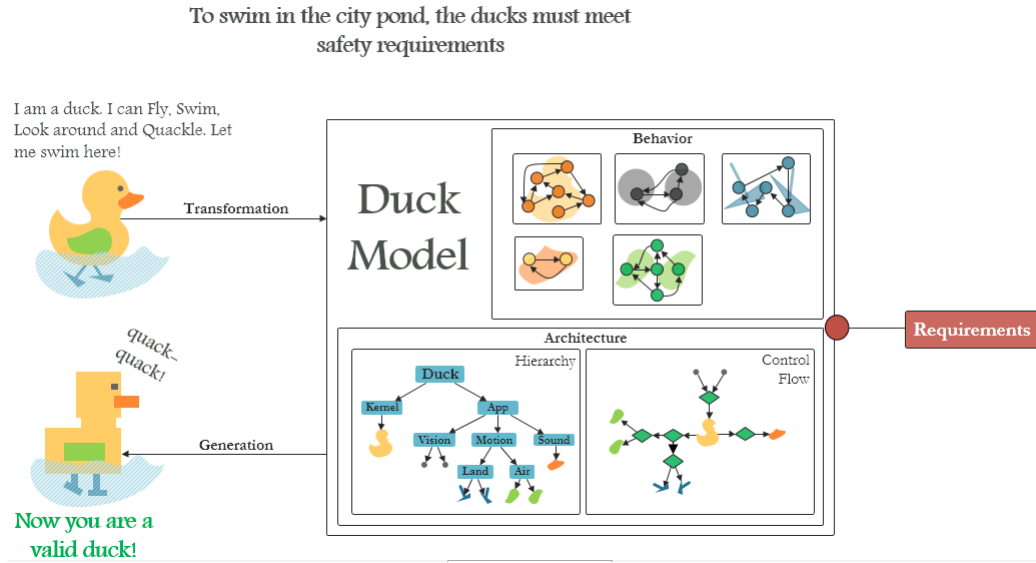


Figure 2.1: Illustrative example of reversible modeling.

2.2 Model Properties

The first and quintessential step in the translation of legacy code is comprehension of the mapping process from C construct to modeling formalism. A straight way of translating would be modeling a heap memory with state chart, representing sequential execution as it is done with the program counter in imperative languages. Although this would imitate an imperative programming style, the produced result would not be adequate to work with, making this kind of model pointless for developers. On the contrary, what developers are highly interested in is getting a concise overview of the code consistent

with the hierarchical model that would be easy to maintain, and that also could be used for requirements mapping.

From the previous discussions, we can identify two main requirements that the resulting model should satisfy:

- Architecture consistency. A hierarchical decomposition of legacy code should be retained.
 - E.g.: the layered code architecture should be equivalent to the model's architecture.
- Behavior consistency. The semantics of legacy code should be captured in the model.
 - E.g.: under the same environmental condition, an input stream of the component (i.e. data sequence over time) should produce the same stream history.

A practical example of behavior consistency is modeling C-code functions or higher-level system abstractions using (Simulink) components. The main idea is to ensure that the component inherits the same behavior as a modeled function. Consistency in behavior implies that the same input stream of data must produce the same output stream of data in all instances of the component (Figure 2.2). Although the system can be either discrete or continuous, the principle should remain unchanged.

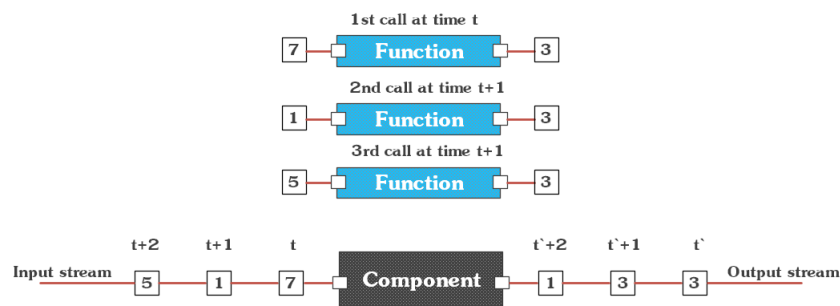


Figure 2.2: Behavior consistency between component and respective function.

It is worth noting, an interface of a component might not be always equivalent to an interface of a respective function since a behavior of the function

cannot be completely expressed only by observing its ports. Therefore, *program state* and possible *side-effects* should be taken into account and exposed to the interface of the component.

Since the behavior consistency involves dynamic properties of the model, it will be considered in the project only as a future extension.

2.3 Embedded software at Scania

As it was mentioned, Scania produces heavy trucks and buses usually equipped with several ECUs. Most ECU software is proprietary and implemented in-house. Therefore, Scania has a large storage of legacy code.

2.3.1 Scania CAN bus network architecture

The interaction between ECUs in Scania vehicles is organized with a CAN-bus network (Figure 2.3). This network consists of 3 sub-nets (in most cases), and available ECUs are arranged within them. There is also one particular ECU called coordinator (COO) that handles all the communication process.

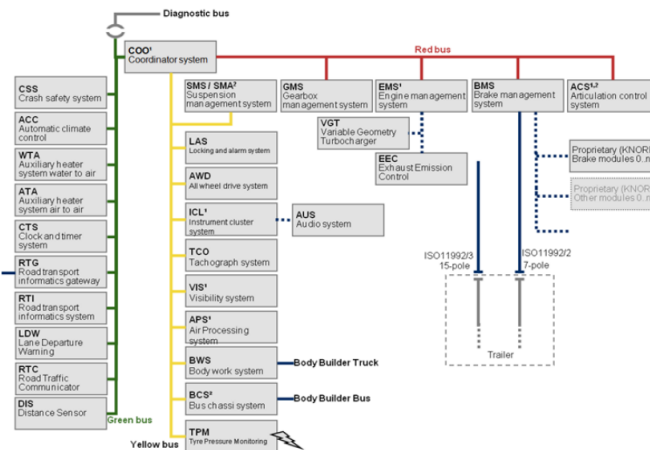


Figure 2.3: Scania CAN bus network architecture.

2.3.2 Software structure

Embedded software at Scania is built in a modular fashion, encapsulating low-level details in a higher-level abstractions. For instance, the COO7 system contains layers represented on Figure 2.3. This layered structure features a clear software architecture, where every layer serves distinct purpose:

- APPL – consists of software modules that handle the trucks behavior.
- MIDD – responsible for handling the inputs from sensor.
- DRIV and BIOS – contain HW related code.
- RTDB – used for communication between APPL and MIDD layers.

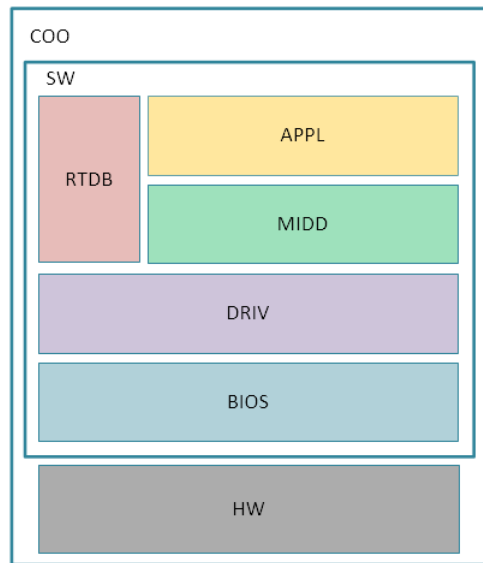


Figure 2.4: Example of software structure at Scania (COO7)

The software layers are decomposed further into modules, and modules consist of functions. This hierarchy might be inherited from the source code organization contained in folder structure within a file system. Furthermore, these folders and files can comply with certain naming conventions. Another option for the organization is, for example, a code repository decomposed within an architecture browser that might feature different hierarchical structure compared to the former alternative. For the sake of simplicity, we will

consider only the first option, but there will be no principal distinctions for the approach in general.

Thereby, an ECU's layered structure can be represented as a folder tree structure (Figure 2.5), where any layer corresponds to a folder and might include other layers (i.e. sub-folders) and/or modules. One or several C-files with a respective H-file and an optional calibration file (a place to declare global variables and macros accessible within a certain module) form a module.

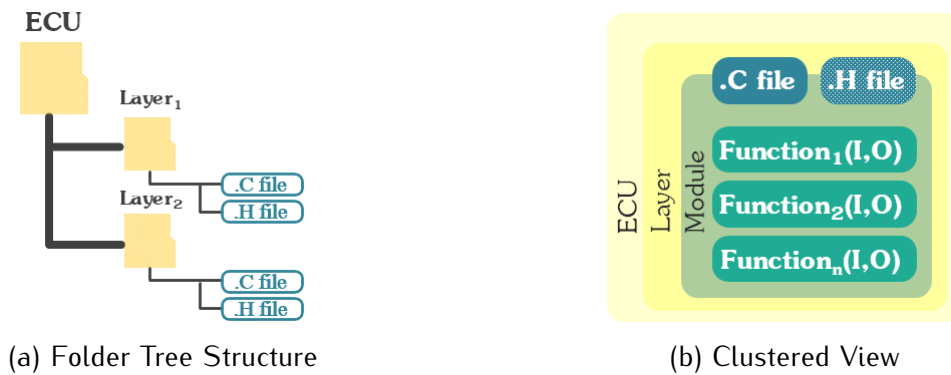


Figure 2.5: Layered structure of ECU

2.3.3 Programming language guidelines

A significant part of software for embedded systems used in Scania is developed in the *C language*. The C language adopts the *imperative paradigm*, where data paths are hidden from the observer by a set of operations over a mutable state. Subroutines in C are represented by functions, containing executable code, which can be referred, i.e. invoked, by their names. A function could have input and/or output parameters. The former are indicated by input arguments, and latter by the return statement. Although there is only a single output from a function, a return variable might consist of a complex structure. Functions with no return value, i.e. void functions, usually have *side-effects*. The side-effect is a change of the program state (e.g. global/static variables, modification by pointers) beyond the scope of the function. A simple example of side-effect is presented below:

Sometimes it can be nontrivial to analyze the impact of side-effects on the data flow between different functions and modules, since they are always

```

1  /* Global variables */
2  int speed, time;
3  int callCnt;
4
5  /* Retrieve the current distance. */
6  int getDistance(){
7      int distance = speed * time;
8      callCnt++; // side-effect, changing global variable
9      return distance
10 }
11
12 /* Some scope. */
13 {
14     int distance = getDistance();
15 }

```

Listing 1: Example of side-effect.

implicit and their roots can be hidden within sequences of function executions [7]. In addition, input parameters to functions might be passed by reference, and that allows variables to escape their original scope where they were created and affect other parts of the system. In contrast, in purely functional languages (e.g. Haskell) data flow is explicit.

An unconstrained usage of C language features (e.g. `goto`) might be fraught with risky and unspecified behavior. Moreover, since some bad programs might legitimately survive compilation, the compilation is definitely not a quality measure [8].

Therefore, there are MISRA-C:1998 (MISRA – The Motor Industry Software Reliability Association) [9] rules (developed by MISRA) applied by Scania in order to derive a safer subset from ANSI-C [10]. MISRA-C is proposed for usage in *safety-critical* systems, and in general reflects common sense rules for a good programming practices, intended to eliminate undefined behavior by avoiding legal constructions that could lead to such behavior (e.g. [11]). There are 127 MISRA rules (93 mandatory and 34 advisory). Generally, three types of commonly used rules might be found in programming language guidelines [12]:

- Category A – rules concerning development style.
- Category B – rules avoiding certain language features.
 - Category B.1 – rules associated with failure, but with no support of measurement metrics (i.e. no quantitative evidence, e.g. does not cause a failure), e.g. "goto statement shall not be used".
 - Category B.2 – rules associated with failure supported by measurement metrics, e.g. "division by zero".

According to [12], the safest subset should contain rules only from category B.2, e.g. *EC* in [13]. It can be observed that MISRA-C rules are focused on category B, with a bias to B.2. Nevertheless, several problems with the rules were indicated in the MISRA-C:1998 standard [12]: incorrectness, redundancy, decidability, cross-talk, and atomicity. Likewise, these problems have not been solved in the second edition of MISRA-C (2004) [8].

To check conformance of written C-code with the standards (MISRA and other internal guidelines), Scania integrated static analysis tools such as PC-lint and QA-C.

Overall, it can be concluded that deep analysis should be performed for MISRA-C rules used in the company in order to refactor and complement them, acquiring a safer subset of C. Obviously, there is a direct correlation between well-definiteness of behavior implemented in C language and ability to model it.

2.4 Simulink

Simulink, developed by MathWorks, started as a pure simulation environment and lacks many desirable features of programming languages [14]. Simulink is based on *synchronous data-flow diagrams* [15], where an individual system block transforms input data to output data. Systems are constructed from various blocks that could be built-in blocks (e.g. arithmetic/logic operations) and/or composite blocks, e.g. *subsystems*. In contrast to exclusive data-flow formalisms (e.g. Lustre), Simulink extends the concept of data-flow by allowing to bypass a block scope using global data storage elements (i.e. "Read" and "Write" blocks). The behavior of these elements is derived from a sophisticated simulation algorithm and might be not completely deterministic [16].

2.4.1 Modularity in Simulink

MathWorks has created established approaches for building hierarchical models in Simulink comprising many layers [17], where an individual element, i.e. component, can be modeled by the following means:

- **Subsystem (S)** – is an independent encapsulation of a set of blocks. Therefore, a subsystem will constitute a higher level of the hierarchy.
- **Library (L)** – is a reusable model that could be instantiated in other blocks.
- **Reference Model (RM)** – is a separate model that could be referenced within other models.

According to the comparison among design components on Figure 2.6 (based on [17]), a subsystem component cannot be reused in multiple places and, therefore, is not suitable for capturing C-code following best design practices that proclaim modularity and code duplication avoidance. On the contrary, very similar concepts of library and reference model are well designed to support multiple instances. The primary difference between them is in internal exception mechanism and optimization, which makes reference models more preferable for usage.

Requirement/Feature	S	L	RM
Component reuse			
Memory performance			
Noninlined S-functions			
Signal property inheritance			

Figure 2.6: A concise comparison among hierarchical components in Simulink.

All of the mentioned blocks allow conditional execution (so-called *Triggered Subsystems*). A triggered subsystem is executed at precise moments in time while a common type of Subsystem is "continuous" (Figure 2.7). The decision on which type to select might be based on modeling guidelines (e.g. MAAB [18] proposes a decomposition of the model into several layers, where an (optional) separate layer handles all the triggering activity) or software architecture.

2.4.1.1 Limitations of Simulink

It is usually argued that Simulink has no explicit semantics, but it might be expressed from the other point, – Simulink has many semantics (depending on user-configurable options); they are not formal, and can be defined as "what is given by the simulator" (e.g., what is observed in terms of inputs/outputs). [16] For this reason, the following restriction should be applied:

- Simulink has a continuous-time semantics [16] that is common for control theory systems, only the discrete-time subset of Simulink modeling will be considered within the current project.
- Simulink semantics depends on the simulation method. Hence, the models should be confined to a discrete fixed-step solver.

Additionally, it could be reasonable to follow an established modeling standard such as MAAB (*MathWorks Automotive Advisory Board*) [18] developed in collaboration with a group of the major automotive companies (e.g. General Motors, Ford, Toyota). These guidelines stimulate the creation of consistent and more readable automotive models with well-defined interfaces. According to MAAB, the model hierarchy should correspond to the functional structure of the control system (*rule : db_0040*). Therefore, models can be organized using a hierarchical structure presented in Figure 2.7, containing the following layers:

- Top layer – contains input and output variables together with a short overview of the model.
- Trigger layer (optional) – indicates the timing for periodic processes, using Triggered Subsystems. If needed, priority might be assigned to each Subsystem for a definition of execution order.
- Structure layer – presents a modular decomposition of the system.
- Data flow layer – describes algorithmic computations.

Likewise, NASA developed the Orion GN&C standard for their exploration spacecraft. The new standard updates and extends MAAB guidelines with emphases to readability and code generation.

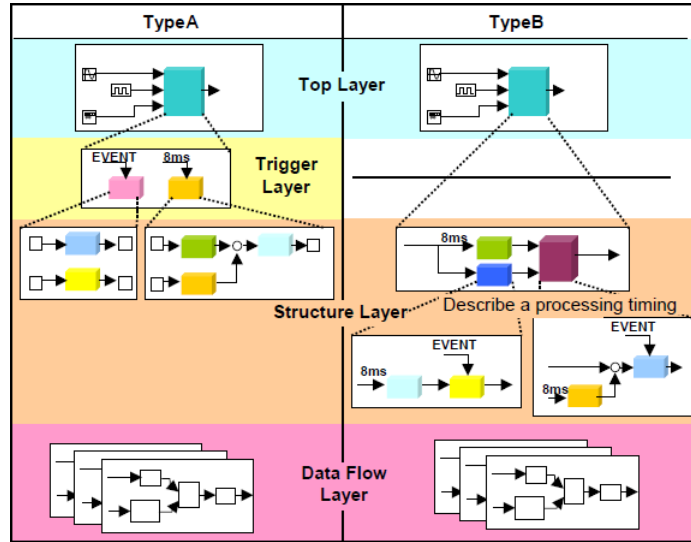


Figure 2.7: MAAB Model Architecture Decomposition. Type A: with trigger level. Type B: without trigger level.

However, it can be observed that it might not be possible to follow the proposed architecture decomposition, especially when a model separation to Structure and Data Flow layers could break consistency with the code.

Nevertheless, MAAB standard provides a solid base for best practices in MBD, and some of them might be applied for modeling of legacy code.

2.4.2 Code Transformation using LLVM

Low-Level Virtual Machine (LLVM) is a compiler infrastructure designed for analysis and optimization of arbitrary programs at all stages of a software lifetime (e.g. in compile-time, run-time or offline). Our analysis of source code will be based on its intermediate representation.

Intermediate representation (LLVM IR) provided by LLVM is source-language-independent and based on a (strict) *RISC*-like instruction set, extended with higher-level information (e.g. type system, control flow, data flow). The instruction set includes conventional processor operations (just 31 opcodes in total due to unambiguity and overloading for operations), avoiding only machine-dependent constraints. Most of the instructions are in the three-address form: one or two input operands producing a single output. Thereby, LLVM does not capture high-level language features, nor platform-dependent

features. The former basically releases any restriction on higher-level languages that can be possibly compiled to LLVM IR, comprising a program in a uniform way [19].

LLVM uses *Single Static Assignment* (SSA) [20] form for its typed virtual registers (but not for memory locations) that hold values of typed primitives (e.g. int, float, pointer) [19]. SSA implies that any virtual register can be written once, in a single instruction. The registers can exchange data among them using load/store commands with type pointers, i.e. there are no direct accesses to memory.

Each function in LLVM IR is represented by a *Control Flow Graph* (CFG), which is a set of *basic blocks*, where each block is a sequence of instructions [21]. Although LLVM contains inbuilt inter-procedural analysis generating a call graph, it supports only explicit function calls. Since function pointers are frequently used in Scania's embedded SW, it is crucial to include them into recovered architecture as well.

2.4.3 Data Structure Analysis

To address the problem of function pointers presence in a call graph, various approaches might be used for revealing possible targets of function call sites. A trivial solution would be to detect possible targets based on a signature (i.e. argument and return type) of a function, whose address has been taken. This naïve method can work with simple programs, but might fail to analyze more or less complex systems. For the latter case more advanced techniques are used that apply existing alias analysis that could help finding points-to sets for every function pointer. There are two main alias analysis approaches:

- *Andersen's* algorithm [22]. It has good precision, but slow speed.
- *Steensgaard's* algorithm [23]. It enables faster analysis at the expense of precision loss.

Previously, LLVM included tools for supporting alias analysis, but later they were moved to a project named "Automatic Pool Allocator" [24]. This project delivers features for *data structure analysis* (DSA). Data Structure Analysis is a *context-sensitive, field-sensitive and flow-insensitive* pointer analysis which identifies data structures on the heap [19]. This analysis is used to capture the layout of memory objects in a program, supplementing LLVM with important information.

The DSA algorithm supports the full generality of C/C++ programs and is based on *Tarjan's Strongly Connected Component* (SCC) [25] finding algorithm, which allows us to discover an accurate call-graph incrementally on-the-fly. The *current modification* of the SCC algorithm is:

- Context-sensitive. The algorithm computes an answer for every call-site, distinguishing the behavior of a function.
- Field-sensitive. The analysis keeps track of structure fields.
- Flow-insensitive.
 - Ignores the control-flow graph, assuming that statements can be executed in any order, any number of times.
 - Delivers a single solution (i.e. points-to graph) for the whole program.

To perform an analysis, DSA computes a *Data Structure Graph* (DS graph) for each routine in a program, where a single node represents memory object that is referenced by a single pointer. Also, it is able to handle incomplete programs in a (conservatively) correct way on intermediate phases of the analysis. For each DS node, the algorithm detects possibly missing functions. If there are no such functions, the node is marked as "complete". Otherwise, at the later point, the node can be complemented with additional data (e.g. edges, flags, type) and/or merged. At the same time all "collapsed" nodes (due to incompatible types in operations) are treated in a safe, but not field-sensitive, way.

DSA creates and refines DS Graph for each function by the following steps:

- "Local Analysis" phase computes a Local DS graph for each function, without any information about callers and callees.
- "Bottom-Up" (BU) analysis phase is used to eliminate incomplete information due to callees in a function, by incorporating information from callee graphs into the caller's graph (creating a "BU" graph).
- "Top-Down" (TD) phase eliminates incomplete information due to incoming arguments by merging caller graphs into callees (creating a "TD" graph). the TD phase can traverse the SCCs of the call graph directly using Tarjan's algorithm

The actual program representation is inspected only during Local Analysis Phase; the other phases are based solely on DS graphs. The BU and TD phases operate on the Strongly Connected Components (SCCs) in the call graph.

Likewise, [19] highlights a number of key aspects of the DSA algorithm that the DSA algorithm: low memory usage, speed, and scalability. These properties make the algorithm a good candidate to use in this work.

Chapter 3

Related Work

In fact, many commercial software tools supporting the translation from C code to Simulink such as Modelify [26]. This product is created by "Ensoft" and uses Atlas' (another company's system) eXtensible Common Software Graph (XCSG) representation to perform automatic conversion from C code to Simulink. The company claims a high-fidelity behavior of generated model, which is partly done by a functional testing with no description of other methods. Although the conversion is automatic, an expert is needed in order to customize produced models in accordance with modelling style and architecture, and ensure their quality. Common customizable alternatives could be the following: use of libraries v.s. model references, or data stores v.s. additional inputs and outputs. However, all these features are impossible to verify, since there is no demo version provided by the company, and the information is very limited. A similar situation applies to other commercial software in this area.

A good overview of other tools that also support verification of the code was performed in the previous theses at Scania, e.g. in [27] and [28]. A number of disadvantages were revealed during this study that prevented usage of proprietary solutions (e.g. not possible to extend, time and resources to adopt, preferences for in-house solutions).

As regards the open source tools (e.g. cgraph), they are not universal enough and mostly intended for generation of specific ad hoc software representations, while we are interested in the general solution featuring extensibility and flexibility.

Furthermore, some research has been done relating to the topic in several articles. In [29], the author developed a tool (C2Model) for the pure source code translation, not aiming to cover the logic and functionality. Therefore,

this solution could assist only in the routine transformation of basic primitives and interfaces. Another article [30] is based on the previous work and extends it with conformance to *MathWorks Automotive Advisory Board* (MAAB) guidelines [18] together with support for function calls and arrays conversions. A modular structure of a model is created based on a function call graph (FCG). Along with the model generation, a produced tool (C2Sim) generates test suite to ensure correctness of a translated Simulink model. However, this tool does not handle C pointers and recursive function (an input language is ANSI-C). An annotation approach for legacy code in XML format is proposed in [31] as an intermediate stage before translation to Simulink model.

Therefore, drawbacks and limitations mentioned above impelled several master students, who worked on the similar topic in the previous years at Scania, to develop a custom toolchain and proposed various methods for its implementation. The short outline of their work focused mainly on the source code processing is presented below:

- Martin Pruscha developed a tool that was based on Flex/Bison to parse code structures into an intermediate form. This intermediate form was in XML format and efficiently captured *abstract syntax tree* (AST). For further processing, XML AST was traversed with XPath queries to extract data flow and control flow.
- Josip Pantovic extended previous work and offered a new format of *intermediate form* (ImF) to make it intuitively linked to the structures in the desired visualization. In this thesis, use of pointers was addressed only partially, while function pointer analysis was ultimately considered out of the scope. Likewise, yEd visualization tool introduced significant limitations to the implementation.
- Oscar Molin used an open-source srcML tool to obtain a selected AST in XML-format with the source code linked as a text. The achieved result has not covered the entire generality of C-language constructions, omitting structure fields, arrays, function pointers.

A characteristic feature of the presented studies is the transformation of the source code to intermediate representation (IR) preliminary to the analysis step, where an individual custom IR was suggested in every case. Obviously, due to time constraints for a thesis, developing an efficient IR is an extremely ambitious task. Hence, all the attempts turned out to be limited and could

handle a restricted part of the C language, performing only certain predefined types of analysis. At the same time, the introduction of new capabilities could require a (significant) modification of IR.

In contrast, in this thesis, we explore the possibility of the *Low-Level Virtual Machine* (LLVM) to carry out effective static analysis of the source code. LLVM is an established compiler environment applied by several competent companies, e.g. Adobe, Apple and Google. LLVM provides its own IR with a rich set of tools for flexible and powerful analysis of software systems. Thereby, it would be reasonable to evaluate its facilities aiming for a tangible result.

Furthermore, to resolve the issue with function pointers and structure fields support (which was not fully addressed in the previous works), we apply Data Structure Analysis (DSA), which is context-sensitive and field-sensitive, from the *poolalloc* project that can be integrated into the LLVM system.

A detailed explanation about LLVM can be found in the section, concerning implementation part.

Chapter 4

Solution Design

Based on the previous discussions, the main objective of this thesis is to propose a solid foundation for translating C-code into Simulink in a consistent way, putting emphasis on the graphical visualization. As a proof of concept, this work is supposed to consider only a distinct type of software architecture that is a hierarchical function-call graph.

The solution design section can be rationally organized into three phases:

1. Modeling. This phase illustrates the process of mapping between C-code structure and Simulink model.
2. Representation. To represent the hierarchical dependencies along with function-call relationships two main directions are taken here for evaluation:
 - (a) Internal extension of Simulink graphical representation.
 - (b) External graphical interface implementation.
3. Algorithm. An automatic architecture recovery from the source code using static analysis tools will be the main concern of the current phase. A proposed approach will be tested through implementation.

A practical outcome delivered within this thesis will comprise a combination of all presented phases.

4.1 Representation of Function-Call Graph

The scope of the thesis covers only a particular case of the architecture, which is a function-call graph that reflects static dependencies between functions in a C program. As it was discussed, functions can be related to different modules and layers, exposing a hierarchy of abstractions. The actual production software systems in Scania incorporate a vast number of distributed routines and their connectivity at various levels. Therefore, an integral part of the software modeling is finding an efficient representation of the considered type of architecture.

4.1.1 Hierarchical Function-Call Graph

There are many established visualisation approaches created for various assignments. For the case of structured data elements with relations between them, ordinary graphs might be applicable as a fundamental structural representation.

4.1.1.1 Function-Call Graph

Assuming that no hierarchical information is needed, there an established and *natural* approach exists to capture function-call dependencies using *Function-Call graphs* (FCG). A FCG is a directed graph, where each edge indicates that some function *A* calls other function *B* (Figure 4.1). A dashed edge shows indirect function calls. Since C code at Scania is followed by MISRA-C rules and does not contain recursion, FCG should be acyclic.

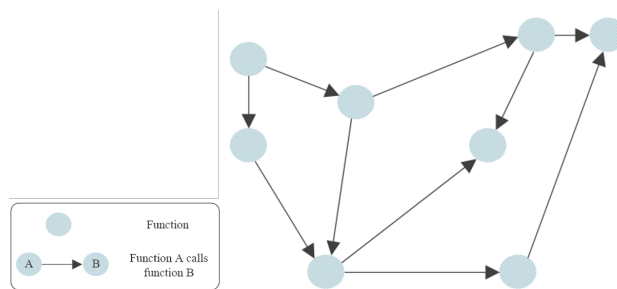


Figure 4.1: Function Call-Graph.

This graph is a good starting point, featuring a simple and clear visualization. Furthermore, it can be easily extended to capture one level of abstraction using the clustering technique.

4.1.1.2 Clustered Function-Call Graph

In a *Clustered Function-Call graphs* (C-FCG) vertices can be grouped into clusters [32]. Thereby, clusters might serve as a mean to organize function by modules as it is depicted on Figure 4.2.

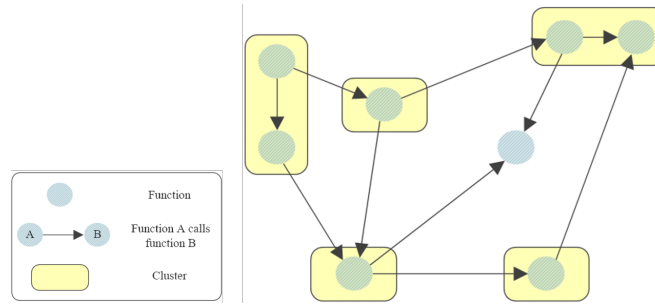


Figure 4.2: Clustered Function-Call Graph.

It should be noticed that not all vertices must necessarily be embedded in clusters, e.g. a node can indicate an external function. However, it is clear that no overlaps between clusters are possible (i.e. when one node is a part of several clusters).

Although this method deals with only one level of abstraction, it can be developed further to cope with arbitrary hierarchical structures.

4.1.1.3 Hierarchical Function Call-Graph

In *Hierarchical Function-Call graph* (H-FCG) clusters can be combined into others clusters (Figure 4.3). This approach can be employed to represent graphs on various levels of abstractions, exposing a semantic of relations between nodes. Moreover, it can effectively handle large graphs, making the navigation easy by reducing a visual complexity [32].

Another important feature of clustering is an ability to perform *filtering* and *search*. In visualization, filtering commonly denotes the removal of elements from the view, and search commonly denotes the emphasis of an element or group of elements [32].

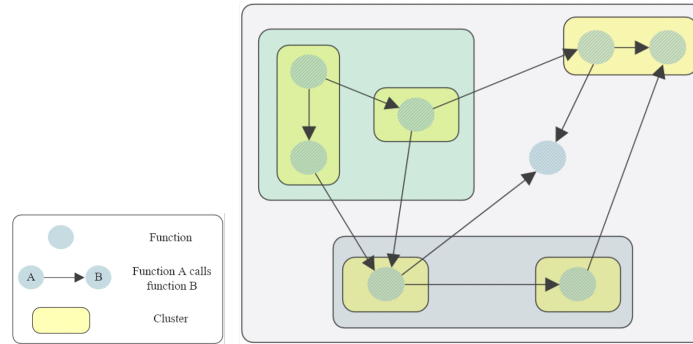


Figure 4.3: Hierarchical Function-Call Graph.

In general, clustered graphs are widely used in many application, such as networks visualisation and social networks.

4.2 Modeling of C-code in Simulink

A model consistency with the source code requires retaining the hierarchically layered architecture of software. As it is described in the previous section, a *hierarchical function-call graph* could allow us to capture the desired structure of software, and this structure is needed to be created in a Simulink model.

4.2.1 Mapping between C-code and Simulink

We aim at using component-based modeling in Simulink to capture the software architecture of the systems in Scania. Considering the source code on a coarse-grained level, an ordinary C function can be viewed as an atomic element in the hierarchy. The function can be explicitly mapped to a component in Simulink, defined as model reference or library. In a simple case, input arguments would correspond to input ports of a component while a set of return variables to its output ports respectively. However, in the C language, an input argument of a function does not necessarily indicate the same direction of data flow. For instance, a pointer argument can naturally serve even as an output channel or a bi-directional channel for the data stream. In addition, considering an interface of a given opaque component that models a C function, both control flow and data flow are hidden from the observer. To supplement the expressivity of the model, it is required to

unpack this "*black-box*". Moreover, taking into account possible *side-effects* of C functions, external data flow could take place within the nested component without any reference to the interface of the parent component, i.e. its flow is not dependent on (host) component inputs and/or does not affect on outputs. As a result, the extracted model might not satisfy traceability requirements of components. Yet, since the behavior and data flow modeling is out of the scope of this thesis, we omit the disclosure and mapping of a function's data flow junctions to ports of components (though general examples can be found in Appendix A). In contrast, the focus will be solely on the structural organization of components used for modeling.

The next step is to model a relationship between atomic elements. As it was decided, a function call relation (function-call graph) can be considered as a basis for modular decomposition of C program. Apparently, a nested function invocation (i.e. caller-callee relation on Figure 4.4a) might be illustrated as an *inclusion* dependency between individual components (Figure 4.4b). In C code, one function can call another (directly or indirectly) incorporated function only when its context is executing. In a similar way, a sub-component can be executed when the parent component is in the active state. Thereby, this *caller-callee* relation is a conceptually admissible mapping between C-code and model.

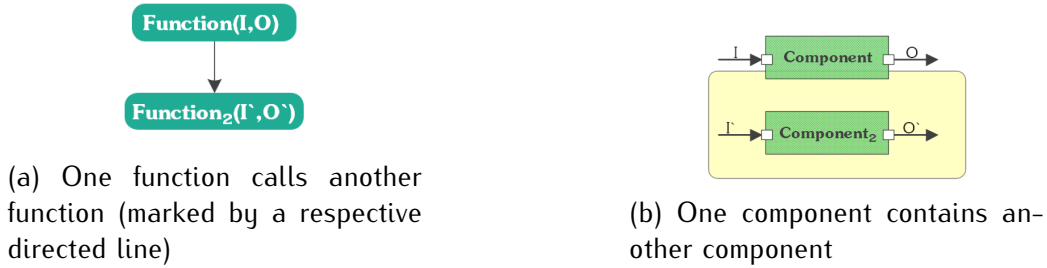


Figure 4.4: Mapping caller-callee dependency between C and Simulink

4.2.2 Constraints of Simulink

Figure 4.5a depicts a sample part of a (Simulink) model that captures a COO7 system. It can be observed that the model consists of several components, describing coincident C functions. Navigating in the native Simulink environment (denoted as "Normal view"), it is liable to correlate only local relation-

ships between components, not being able to have an overview of the entire picture. For this reason, Mathworks introduced an alternative "Dependency view" (Figure 4.5b) that is equivalent to FCG (function-call graph) and effectively assists in comprehension of the model's modular structure, where each component is unique and arrows characterize *inclusion* dependency. However, neither of the options provides any information about connection of an individual element to a certain module, i.e. there is no clear separation between internal and external function calls. For example, from the Figure 4.5b it is not possible to know whether these components belong to the same or different modules (we cannot rely on the naming convention).

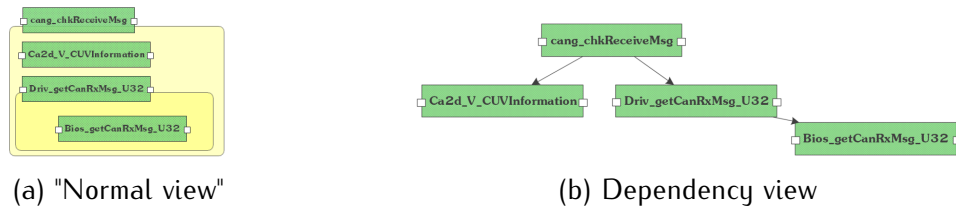


Figure 4.5: Simulink views

Obviously, since higher abstraction levels designate another type of the relationship, which is "parent to children", they cannot be reflected in FCG conceptual view. Thereby, the modular organization of C code is not retained in Simulink models using a standard set of tools.

4.2.3 Possible solutions

In order to expose this hidden hierarchical architecture, we propose two approaches to handling higher abstractions in Simulink:

- Annotations. Additional details can be attached to every block in Simulink using either its description field or internal structure to keep user related information. Consequently, every component might be annotated with a specification of a binding to its parent module/layer, allowing programmatic processing of this information.
- "Virtual" components. Complementary components with no real behavior might be introduced to the model to represent abstraction layers. Such component can contain other sub-components, i.e. indicating a parent-to-children relation, replicating an software architecture to a full extent.

Although the presented methods have nothing to do with a graphical representation, they provide the opportunity to capture desired modular structure of software using meta-data as it is shown on Figure 4.6. Immediate practical solution to the visualization is examined in the Representation section.

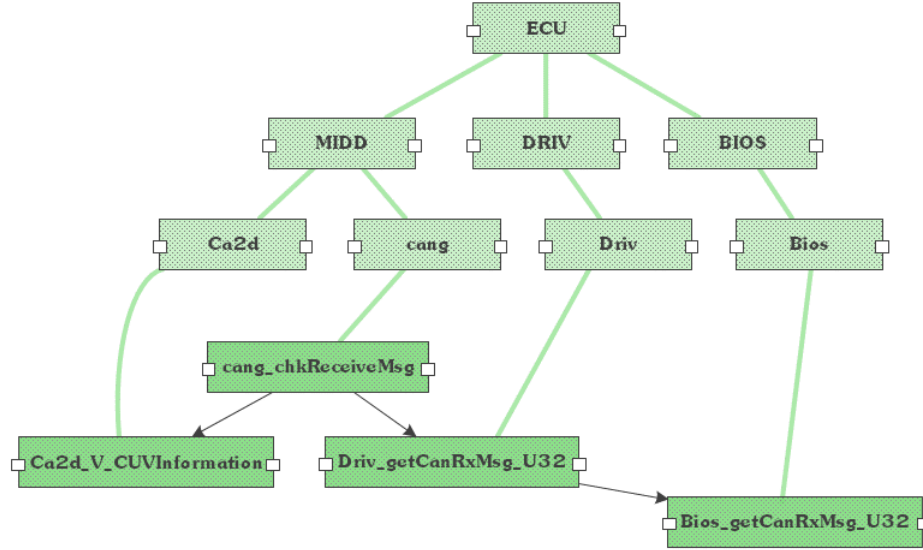


Figure 4.6: Modeling hidden relations in Simulink.

4.2.4 Extension of the graphical representation in Simulink

The Simulink modeling environment is integrated into Matlab workspace, supporting development with Matlab code. Knowing what kind of representation we are aiming to achieve based on previous discussions, we can try to enhance a hierarchical model in Simulink with all abstraction layers and hidden dependencies.

It is important to notice that Simulink is not a drawing tool, and it does not allow to draw lines and create custom shapes. Moreover, strict limitations on programming capabilities exist in Simulink, where all the allowable functionality is squeezed into the scope of modeling, simulation and analysis. For instance, there are no built-in button controls provided, and even a "click" callback is missing for Simulink components. To handle this particular case, it is a common approach to using "open" callback (activated by double click)

for some block, imitating a button push by that. Thereby, many similar *"tricky"* attempts have been made to overcome current restrictions.

The lack of an opportunity to draw lines, describing semantic relations between components, can be compensated by textual information. It implies that a component interface must be enhanced to a composition of its own interface and all interfaces of its internal components. This feature is implemented using component masking, when an ad hoc uniform mask can be applied to any component, revealing all desired information about nested dependencies (all the way down) to its interface automatically (Figure 4.7), - green and pink ports with respective annotations. Hence, the relations of the considered component are depicted on the one screen followed by exactly the same pattern when representing deeper levels of their relations.

The exposed dependencies can use the clustering representation, and the connectivity between components can be established by observing annotations and matching their interfaces. In order not to violate the working ability of the system, components destined to express relationships are *commented*, i.e. Simulink is not supposed to execute or enable them during the simulation.

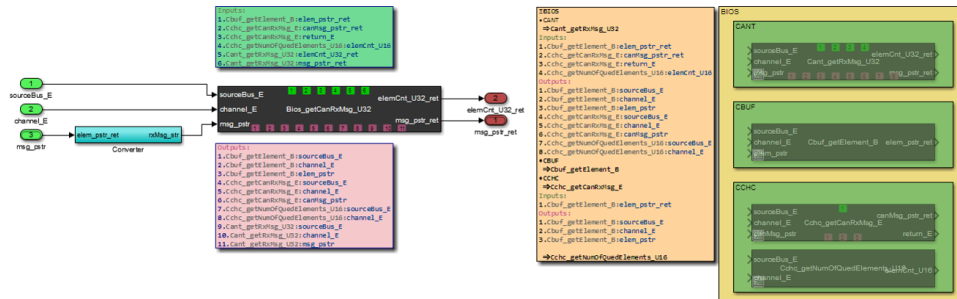


Figure 4.7: Example: Internal extension of Simulink graphical representation.

As a result, the delivered solution was able to perform automatic transformation of hierarchical Simulink models to models with explicit dependencies (Figure 4.7). At the same time, the approach taken uncovered a number of issues that could not be addressed constructively.

4.2.4.1 Shortcomings of the approach

- Tangled view. It is complicated to track visual relationships between components only based on textual annotations.

- Performance issues. Although commented blocks were used to display dependent components, they influenced the performance (e.g. save and update operations). The reason was that Simulink still handled commented blocks as a part of the model, making sluggish response from the environment.
- Highly restricted API. An approach of interaction with a model through callbacks is strictly limited.
- No drawing features. It is not allowed to create custom shapes.
- Unsafe Simulink programming. Use of Mask callbacks, where the callback modifies entities outside of its scope may result in unexpected behavior [33].

Overall, the results made it clear that there is a need for more powerful graphical visualisation tool to capture hierarchical function-call graph.

4.2.5 Compromise between behavior expressivity and presenting structure

Although Simulink does not support the desired expressivity level, it is, undoubtedly, a solid platform for modeling and analysis, e.g. allowing programmers to execute models and supporting modular design in a hierarchical fashion. This key functionality is part and parcel of complying with ISO26262. Hence, having the Simulink environment as a *back-end* of the model responsible for its behavior, it might be acceptable to resolve the lack in representation views by employing external means to form a *front-end*.

The choice of visualisation tool must first be justified according to universality and customizability to handle any types of architectural design. This decisively rejects proprietary products, making a shift to the open-source segment. To fulfill our basic needs, the front-end is supposed to have an ability to handle big architectural visualisations, supporting zooming, dynamic filtering, and interactivity. An additional emphasis on easy accessibility and usability of the system promotes web-based solutions as an eligible candidate to accomplish the goal (Figure 4.8).

The architecture recovered from software can be effectively applied to construct both structured Simulink model and corresponding visual represen-

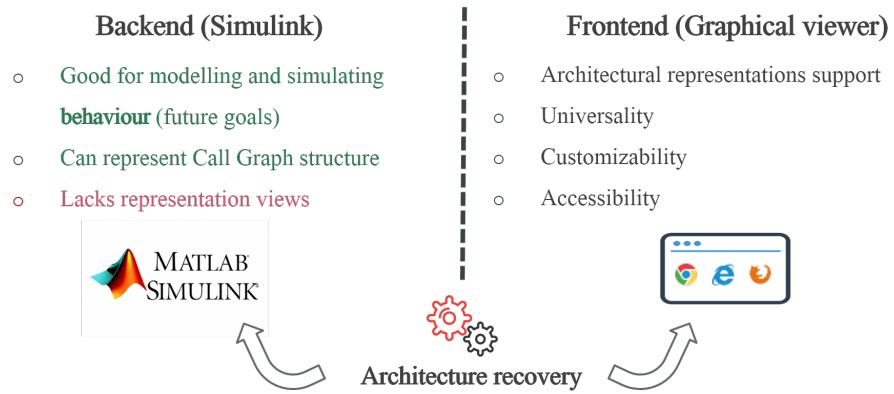


Figure 4.8: Compromise between behavior expressivity and retaining structure.

tation. The way of interaction between front-end and back-end might be realized through *MATLAB API* [34] or *MATLAB COM SERVER* [35].

Chapter 5

Implementation

After consideration of C-code modeling and the eligible way to represent its function-call graph, this chapter offers a practical solution to architecture recovery. Even though the algorithm would deal with only one type of architecture, the developed infrastructure can serve as a solid base for further extensions.

5.1 Transformation pipeline

In order to automate the mining of architectural models from the source code, we designed a new toolchain that may be seen in Figure 5.1.

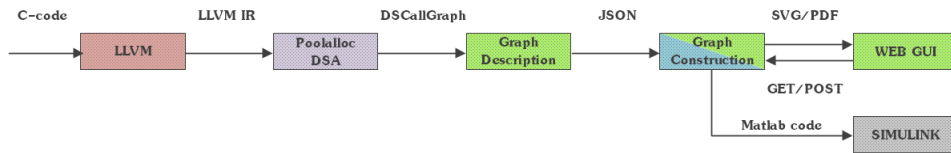


Figure 5.1: Implementation tool-chain.

Thereby, the data transforming pipeline is described by the following units:

1. LLVM. With an assistance of front-end tools included in LVMM, C-code is translated to LLVM IR.
2. DSA. *Poolalloc* project is employed to derive a call-graph from the source code represented as a DSCallGraph structure.

3. Graph description. Desired dependencies are extracted from DSCall-Graph and merged with the hierarchical structure of abstraction layers, producing an H-FCG as output in JSON format.
4. Graph construction. The graph is converted into an MySQL database with a further ability to query dynamically specified sub-graphs.
5. WEB GUI. The received graph in SVG format is visualized on the client side.
6. Simulink (not implemented). Building a hierarchical model in Simulink corresponding to the recovered architecture (i.e. H-FCG). Also, there should be an ability of interaction between Simulink and Web GUI.

A substantial advantage of the presented pipeline is a rational decomposition of key operations among units, which allows us to access and modify any state of the transformation in isolation from the whole system. Once the back-end is customized (from "LLVM" to "Graph Description"), it can supply a multitude of self-sufficient front-end services. Therefore, back-end and front-end are not dependent on each other and can be independently developed further.

5.2 Code compilation to LLVM IR

The Clang project, which is a part of LLVM, implements all front-end-related steps and can be used to compile source code to LLVM IR. In practice, Clang is able to conduct the full compilation, including back-end steps [36]. Thereby, using **clang** command with "-emit-llvm" parameter can produce either bitcode (BC, encoded LLVM IR) or assembly (LL, human readable form) files (determined by the presence of "-S" parameter). Considering the fact that Scania's systems contain a huge number of C-code files, it will be an awkward solution to process them individually. A good way to solve it might be performed by a linker (**llvm-link**) tool. This tool merges all the compiled files together into a single LLVM bitcode (or assembly) that encompasses all inputs. After that, the whole software system is treated as a top-level container that is equivalent to a *module* in LLVM terms. A module keeps track of global variables, functions, and a symbol table. LLVM comprises a distinct Module pass to run

a custom analysis for a module in a convenient way. However, another optimisation step should be taken on IR, which must convert it to SSA form before carrying out any analyses. This step is done with **opt** tool and "-mem2reg" option. An overall sequence is outlined on Figure 5.2.

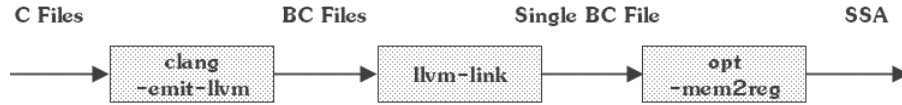


Figure 5.2: Compilation to LLVM IR.

It is worth noting, although Clang is capable to translate programming code written in assembly language using GAS syntax, Scania relies on a proprietary *Wind River Diab* compiler that features another assembly syntax and, therefore, is not supported by Clang. To address this problem, a custom translator should be implemented. However, due to time limitations, it was not implemented. Thereby, assembly dependencies were neglected.

5.2.1 Call-Graph Recovery

The call-graph mining from the existing source code is implemented with a help of LLVM Module analysis pass, which runs on LLVM IR. The process can be described in three major steps:

- Extraction caller-callee dependencies from DSCallGraph structure generated by "Top-Down" (TD) DSA pass.
- Obtaining layered structure (i.e. folders hierarchy) of software and merging it with the dependencies from the previous step.
- Generation of two output JSON files for nodes (i.e. functions) and edges (i.e. dependencies).

A significant remark should be made regarding DSA algorithm in the *poolalloc* project. The evaluation of this analysis on the source code in Scania deduced that results for indirect inter-procedural relations are pessimistic due to the complexity of the structures with pointer fields in use. Proceeding further, a special example was created to highlight the issue (Listing 2) and committed to the git repository of the *poolalloc* project [24].

```

1  typedef void (*tX)(int a, int b);
2  typedef void (*tY)(int a);
3
4  typedef struct {
5      tX p; // decoder for a msg
6      int n;
7  } msg;
8
9  static void A1(int a) { }
10 static void B2(int a, int b) { }
11 static void C2(int a, int b) { }
12
13 tY q;
14
15 static void decode(tX decoder_f){
16     decoder_f(1,2);
17 }
18
19 int main(void) {
20     msg *a = malloc(sizeof(msg));
21     q = &A1;
22     a->p = &B2;
23     decode(&C2);
24
25     return 0;
26 }

```

Listing 2: An example for which DSA produces erroneous results.

The code in Listing 2 fails to analyze indirect function calls correctly, giving an output as in Listing 3.

From the sample code, we can observe that **decode** function has two redundant callees (A1 and B2). This situation happens when addresses of A1 and B1 are taken within **main** function. While a discussion of the question on a mailing list, one of *poolalloc* developers pointed out that call graph reported by TD is the same as computed during BU, and BU cannot determine the indirect call in this case for A1 and B1, but can only target C2. At the moment, the issue is still open, and the solution is in progress.

```
main: [malloc decode]
A1: []
B2: []
C2: []
decode: [A1 B2 C2]
```

Listing 3: Execution results.

5.2.2 Web GUI

Web development is rapidly growing and becoming more advanced. A wide variety of instruments can be used for *full-stack* development. For this project, we propose an extremely popular *PHP-MySQL-JS* source language chain due to several reasons:

- Ease of deployment. Even many ready-made solutions already exist, e.g. XAMPP.
- Open source.
- Detailed documentation and large community.
- Multitude of frameworks for extension.

The function-call graph recovered from software using LLVM is stored in two JSON files (nodes and edges). With a help of a PHP script, the content of these files is automatically parsed and inserted into MySQL database using two respective tables. Although MySQL is not the best solution to store and process graphs, the query performance can be improved by precomputation of nodes hierarchical relations and caching. Considering the number of nodes (>5000) and edges (>30000) for the test case, the response speed has not exceeded 1s.

Having the function-call graph saved in the database, it is possible to select and render dynamically any part of it. The selection is based on the four parameters:

- Name – name of the function to evaluate.
- Depth – desired level of dependencies depth.
- Type – format of the output (SVG, PDF).

- `isIndirect` – whether to show indirect function calls.

Before sending a resulting graph to a client, the graph must be properly arranged. Graph layout is not a trivial task, and it is done with a help of two tools **fdp** and **sfdp** from Graphviz package. These tools support clustered graphs described in DOT language and produce acceptable solutions. Moderate size graphs are processed using *fdp*, while large-scale graphs are rendered with *sfdp*. Likewise, the server is able to transfer the final representation in two formats: SVG and PDF. For the large scale graphs, only the latter option is available, since processing a huge SVG interactively with JavaScript on the client side might be totally inefficient [37].

The client side is written in JavaScript using *D3.js* framework for manipulation of hierarchical data bounded to DOM. The interface is shown on Figure 5.3 and divided into two parts:

- Navigation panel (left side). The panel reflects the hierarchical structure of software layers using bars that can be expanded or collapsed by clicking, intuitively similar to folder browser. This method accelerates the process of function finding within structures of any complexity. A call-graph might be displayed for any chosen function. In addition, there are two controls for setting parameters located on the left:
 - "Indirect function" switch (top). It defines whether to show indirect function calls.
 - "Depth level" switch (bottom). It sets the desired level of dependency depth that is displayed, making visible only a respective number of nested function calls.
- Visualisation panel (right side). A function-call graph for a selected function with applied parameters is rendered within this panel. All layers of abstractions are marked with a title and various colors. The current view supports zooming (mouse wheel), moving (hold right-click), and observation of local dependencies (all parents and children chains) highlighted with color for a chosen function ("ESC" – to cancel). Likewise, there is a small button (top-right corner), which opens a graph in PDF format.

The developed visualisation had a positive feedback from employees and other master students at Scania, but no formal survey was carried out. It is advisable to fill this gap in the future.

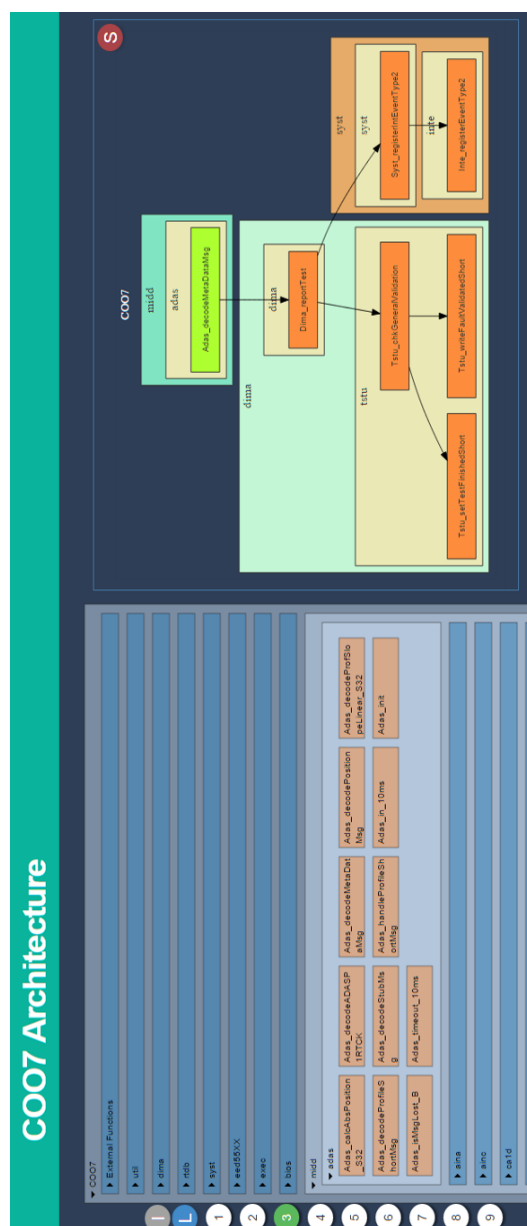


Figure 5.3: Web GUI.

Chapter 6

Evaluation

The current thesis work was motivated by notable advantages that software modeling can bring. Along with this discussion in the introduction chapter, a set of key criteria was defined to evaluate legibility of performing the task with proposed use of Simulink as a modeling formalism. These measures effectively give an evaluative instrument to comprehend an integrity and seamlessness of the presented solution. Furthermore, basic benchmark has been done around the solution to provide an approximate performance estimation.

6.1 Level of Formality

Simulink does not require any changes in order to translate its models to executable code. This fact implies that Simulink avoids the ambiguity and contains enough information to enable model implementation. Having said that, the executable behavior is based on a set of settings applied to a particular model allowing a varying level of formality. Therefore, this variability of semantics in the presented work is constrained by persistent contexts (e.g. continuous-time, fixed-step solver).

6.2 Transformation from C-code to Simulink

While Simulink supports a coarse-grained granularity, the architectural modeling is performed at a fine-grained level. A smallest identifiable and reusable part of a program written in C is a function, which is taken as an atomic ab-

straction element. A function declaration is easy to capture with component blocks provided by Simulink (when taking a definition also into consideration, behavior aspect should be expressed). Yet a hierarchical structure obtained in Simulink is not flexible enough to expose software architecture with all required dependencies explicitly.

6.3 Architecture Recovery from C-code

A static analysis of original program code in complex systems can turn to be highly sophisticated and inefficient task. Improved re-engineering techniques propose carrying out a processing of intermediate representations of software systems instead. Under the circumstances, a versatile and powerful solution is presented by LLVM. Its primary value lies in the fact that it can embrace multitude code transformations accompanied by desired extensions to the core LLVM IR of the code. That is to say, giving a full traceability of code compilation, allowing to recover automatically high-level abstractions from the source code.

To give an idea about the efficiency of the transformation, we performed a benchmark of two systems: COO7 from Scania and Trogdor (an open source text adventure engine [38]), which was done using the virtual machine with the following specifications: 1 core CPU with 1697 Mhz and 2GB of RAM, Linux version 3.19.3-3-ARCH, llvm version 3.6.0. The results are shown in the table 6.1. For every system, there is an indication of a total number of nodes with a number of atomic elements (i.e. functions) in the brackets, while the rest are higher levels of abstractions. The figures show that the magnitude of COO7 nodes is almost eight times greater than in Trogdor. Likewise, links represent the number of interconnection between the nodes with a designation of indirect connections (i.e. by pointer) in the brackets. It is interesting to see that around 80% of function calls in COO7 are performed through pointers, compared to only 13% in the second system.

Table 6.1: Source code transformation results.

System	Nodes (func)	Links (ptr)	Execution Time
COO7	2624 (2439)	14977 (10284)	44.176s
Trogdor	337 (294)	1010 (132)	5.463s

The measured execution time it takes to translate COO7 is eight times

higher than for Trogdor transformation. Although the dependency might seem linear for this case, with an increase of nodes and relations, the growth in time is expected to be exponential.

In summary, preliminary results look acceptable, and possibly could be improved in the future by reconsidering every step of transformation and removing redundant passes.

6.4 Redundant Concepts

With respect to Simulink, this aspect is illustrated hazily for a beholder. A Simulink modeling environment contains multiple tools and utilities, which can be viewed as a point of redundancy. There is usually a need in using Simulink standard libraries for any most of modeling processes; however many other tools might never be used (e.g. tracing utilities, 3D/2D graphical simulators, debuggers, specialized toolboxes). Considering the number of features and exemplification of the modeling formalism with only one architectural representation (i.e. HFCCG), not even touching behavioral mapping, it is highly challenging to anticipate redundant concepts carefully. For this reason, these concepts should be reviewed in the future to satisfy the needs.

6.5 Automation of Models Mining

The process of (architectural) model construction from source code in C is proven to be achievable in an automatic way. Although there was an exemption introduced in assembly code due to the syntax differences, this issue can be addressed by writing a custom parser for another assembly language standard within LLVM infrastructure. Likewise, not only structural aspects of the code are possible to capture by the toolchain, but it can cover behavioral aspects as well.

6.6 Graphical Representation

A curtailed graphical expressibility of Simulink is the main impediment to retaining software structure along with the dependencies in a model. This

circumstance prompted to use an external extension to the graphical visualization of software structure, leaving all back-end modeling tasks (e.g. behavior modeling) to Simulink. Thereby, created web front-end can capture dependencies of HFCCG in a holistic and uncluttered way.

6.7 General Opinion

Even though Simulink is confined in some representative functionality, not allowing to retain software architecture to a full extent, its advantages outweigh the shortcomings, urging a supportive solution. There are no prerequisites to beware of jumble caused by using external tools in combination with Simulink: seeming initial wonky connections have a good prospect to accrete into established form.

Overall, the evaluation of posed criteria gives a promising approach to address the modeling of source code written in C.

Chapter 7

Conclusions

From the outcome of our investigation that has been undertaken it is possible to infer that manually written C-code hierarchical structure can be successfully captured with Simulink, extended by the external visualization tool, in an automated fashion. The developed system has been successfully tested on two functioning software systems in Scania (GMS and COO7) and one open-source text-based game engine Trogdor.

The work concerned only a particular type of architecture, i.e. hierarchical function-call graph; however, the developed solution should be applicable also to any other architectural design. The findings suggest that this approach could also be useful for fine-grained analysis of the source code that is essential for behavior modeling.

The discussion presented in this thesis revealed several vital conclusions that are consolidated in the following list:

- Simulink lacks expressivity in capturing the modular structure of the manually written C-code. Internal extensions are hardly possible due to the proprietary of the product.
- LLVM is a powerful ecosystem that suits well for a wide variety of reverse engineering tasks (and much more) by employing static analysis.
- There is a need for well-defined approaches for addressing the complexity of large graphical representations overloaded with information. These methods should enable customizable visual abstractions, emphasizing certain details and features.

It should be admitted, that not all goals defined in the introduction of the work have been fulfilled. Due to lack of time, an automatic synthesis of a Simulink model from the extracted architecture along with an integration between front-end and back-end was not implemented. The graphical representation was chosen based on the best practices, but there were no interviews organized to evaluate and refine it. Regarding the front-end part, although it can be improved with many features in many ways, the goal was to provide a basic core that can be easily extensible.

7.1 Further Development

Future study of the issue would be of Scania's interest. The next stage of the work might concern extension of architectural representations that can be recovered from the source code (e.g. control flow, data flow). Clearly, further research will be required to address the behavior modeling issue and validation of the resulting system. The behavior modeling can be possibly done in two methods: using adopted and imported to Simulink C files or using a direct transformation of C-code constructions to Simulink blocks. It should be admitted that both solutions are not trivial.

Besides, some practical improvement might be suggested to the current system:

- Replace MySQL with a graph database (e.g. Neo4j) for the better efficiency.
- (Possible) Improvements for indirect function call analysis.
- Generate structured model for Simulink and link it with the front-end.

Bibliography

- [1] Jonas Westman, Mattias Nyberg, and Martin Törngren. Structuring safety requirements in iso 26262 using contract theory. In *Proceedings of the 32Nd International Conference on Computer Safety, Reliability, and Security - Volume 8153*, SAFECOMP 2013, pages 166–177, New York, NY, USA, 2013. Springer-Verlag New York, Inc.
- [2] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hnětynka, and Michal Malohlava. Comparison of component frameworks for real-time embedded systems. In Lars Grunske, Ralf Reussner, and Frantisek Plasil, editors, *Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin Heidelberg, 2010.
- [3] CESAR project. Survey report on modeling languages, components technologies and validation technologies for real-time safety critical systems — v 2.0. http://www.cesarproject.eu/fileadmin/user_upload/CESAR_D_SP3_R1.2_M2_v1.000.pdf, 2010.
- [4] ISO International Organization for Standardization. Iso 26262 road vehicles functional safety part 1-10. 2011.
- [5] M Broy, S Kirstan, H Krcmar, and B Schätz. What is the benefit of a model-based design of embedded software systems in the car industry? *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 343–369, 2011.
- [6] Joy Lin. Measuring return on investment of model-based design. https://www.mathworks.com/solutions/model-based-design/mbd-roi-video/Measuring_ROI_of_MBD.pdf, 2011.

- [7] N. Wilde and R. Huitt. Maintenance support for object oriented programs. In *Software Maintenance, 1991., Proceedings. Conference on*, pages 162–170, Oct 1991.
- [8] Les Hatton. Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. *Information and Software Technology*, 49:475–482, 2007.
- [9] MISRA Ltd. Development guidelines for vehicle based software. <http://www.misra.org.uk>.
- [10] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.
- [11] A. Koenig. *C Traps and Pitfalls*. Pearson Education, 1988.
- [12] Les Hatton. Safer language subsets: An overview and a case history, MISRA C. *Information and Software Technology*, 46:465–472, 2004.
- [13] Leslie Hatton. EC – A measurement based safer subset of ISO C suitable for embedded system development. *Information and Software Technology*, 47:181–187, 2005.
- [14] I. Safaka M.Bozga V. Sfyrla, G. Tsiligiannis and J. Sifakis. Compositional translation of simulink models into synchronous bip. 2010.
- [15] R. Lublinerman and S. Tripakis. Translating data flow to synchronous block diagrams. In *Embedded Systems for Real-Time Multimedia, 2008. ESTMedia 2008. IEEE/ACM/IFIP Workshop on*, pages 101–106, Oct 2008.
- [16] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time simulink to lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, November 2005.
- [17] Mathworks. Componentization guidelines. <http://se.mathworks.com/help/simulink/ug/model-architecture-guidelines.html>.
- [18] S. Lehman. *Controller Style Guidelines for Production Intent Development Using MATLAB, Simulink, and Stateflow*. Mathworks, 2012.

- [19] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. See <http://llvm.cs.uiuc.edu>.
- [20] M. Belyaev and V. Tsesko. Llvm-based static analysis tool using type and effect systems. *Automatic Control and Computer Sciences*, 46(7):324–330, 2012.
- [21] Chris Lattner and Vikram Adve. Llvm: A compilation framework for life-long program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.
- [22] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.
- [23] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.
- [24] LLVM. Poolalloc. <https://github.com/llvm-mirror/poolalloc>, 2015.
- [25] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.
- [26] EnSoft. modelify. <http://www.ensoftcorp.com/modelify>.
- [27] Martin Pruscha. Infrastructure for the generation of functional data-flow views for automotive embedded systems. Master's thesis, KTH, School of Information and Communication Technology (ICT), 2013.
- [28] Josip Pantovic. Automated data dependency visualization for embedded systems programmed in c. Master's thesis, KTH, School of Information and Communication Technology (ICT), 2013.
- [29] V Zsombori. Transformation of C Code to Matlab/Simulink Models. *Daimler Chrysler SIM Tech*, 2005.

- [30] Indranil Saha Saha, Kuntal Chakraborty, Suman Roy, B. Vardhan Reddy, Venkatappaiah Kurapati, and Vishesh Sharma. An approach to reverse engineering of c programs to simulink models with conformance testing. In *Proceedings of the 2Nd India Software Engineering Conference, ISEC '09*, pages 137–138, New York, NY, USA, 2009. ACM.
- [31] Chethan Kotekar. An approach to translate legacy 'C' code to Simulink® model using XML. *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, 2012.
- [32] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, January 2000.
- [33] Mathworks. Best practices for masking. <http://mathworks.com/help/simulink/ug/best-practices-for-masking.html>, 2015.
- [34] Mathworks. Matlab api for other languages. <http://se.mathworks.com/help/matlab/programming-interfaces-for-c-c-fortran-com.html>.
- [35] Mathworks. Matlab com automation server. <http://se.mathworks.com/help/matlab/call-matlab-com-automation-server.html>.
- [36] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.
- [37] Stephen Bannasch. Svg path benchmark. <http://bl.ocks.org/stepheneb/1296930>, 2015.
- [38] crankycyclops. Trogdor. <https://github.com/crankycyclops/trogdor>.
- [39] Joseph Derek Morrison. *A scalable multiprocessor architecture using cartesian network relative addressing*. PhD thesis, 1989. MAS.

Appendix A

Visualisation abstractions

An example of basic graphical framework is described in this appendix. It was not a main focus of the thesis and therefore, only a small draft is presented.

A.1 Definitions

Individual element

is an item from the set of the software system parts, e.g.: ECU, Layer, Module, Function. An individual element can be organizational (ECU, Layer, Module) or computational (Function).

Component

represents an individual element of the system. A component is called atomic if it represents a computational unit.

Port

defines a point of interaction between a component and its environment. A component may have multiple ports.

Interface

a set of ports of the component.

Connection

is a communication channel between components. Two types of connections are supported: data flow and function invocation. A data flow connection allows computational data exchange between components. A function invocation connection allows data (strobe) exchange to indicate triggering of one component by another.

Architectural model

is a set of components and connections.

Architectural unit

can represent Component, Port, or Connection.

Abstraction of the architectural unit

is a unit derived from original unit by grouping/generalizing properties of original unit.

Dependency graph

is a unit derived from original unit by grouping/generalizing properties of original unit.

A.2 Hierarchy of ports

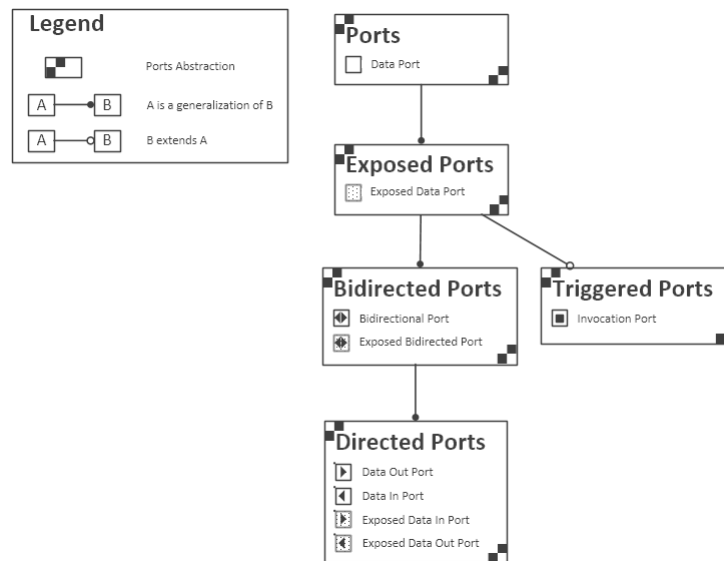


Figure A.1: Hierarchy of ports in visualisation framework.

Appendix B

Data-flow mechanics

This Appendix relates to behavior modeling and presents rough ideas on such modeling, which were based in collaboration with Joakim Gustavsson. Due to the time constraints there was no possibility to develop it further.

Tracking of data flow in C programs is not a trivial task. In order to grasp fundamentals of behaviour capturing, relevant data-flows in C code are analyzed and summarized in further cases, where each of them is accomplished with a respective Simulink component-based model. Individual elements in the models arranged in the way that data always flows from left to right.

B.1 Direct Assignment

Direct Assignment - a given variable is assigned a new value using the assignment operator '=' (Listing 4). Thereby, data flow from the right side of the assignment operator to its left side.

```
1  /* Direct Assignment */  
2  a = 5;
```

Listing 4: Example of direct assignment.

This case can be modeled by labeling the connection (channel) between data source (e.g. constant 5) and data sink (e.g. port x of some component F) as it is depicted on Figure B.1. Following this, all possible sinks dependant on the certain source should have a direct connection to this source.

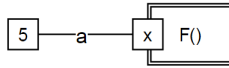


Figure B.1: Direct assignment is modeled by connection labeling.

Another option to model direct assignment is by using a data storage element in Simulink, which results in "imperative way" of data-flow modeling as it is shown on Figure B.2.

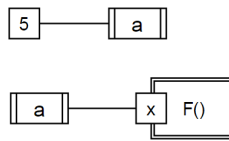


Figure B.2: Direct assignment is modeled by data storage.

B.2 Function parameter passing

This type of flow is defined by data that is passed by value, i.e. the actual value of the input parameter is copied to the stack and its modification does not affect the original data. (Listing 5).

```

1  /* Calculate fuel consumption */
2  int calcConsumption(int coef);
3
4  {
5      int coef = 10;
6      int calcConsumption(coef);
7  }

```

Listing 5: Example of function parameter passing.

Data-flows into a subsystem block can be modeled by input ports of this block similarly to function parameters of C function (Figure B.3).

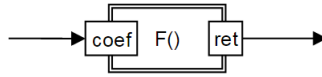


Figure B.3: Function parameter passing is modeled by input ports.

B.3 Function return value

Function return value is a value returned by a function 6). According to MISRA-C, a function should have a single return statement.

```

1  /* Global variables */
2  int speed;
3
4  /* Returns the current speed */
5  int getSpeed(void){
6      return speed;
7  }
8
9  /* Some scope */
10 {
11     int currentSpeed = getSpeed();
12 }
  
```

Listing 6: Example of function return value.

Data-flows out of the function can be modeled by output ports of the component. The current example (Figure B.4) models a parameter-less function dependant on the global variable and does not have an input data flow.

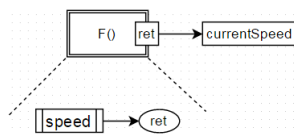


Figure B.4: Function return value is modeled by output ports.

B.4 Indirect assignment

Indirect assignment defines a case when other variables use data that is derived from a given variable 7).

```
1 int Func(int a){  
2   b= a + 5;  
3   return b;  
4 }
```

Listing 7: Example of indirect assignment.

Indirect assignment is transformed to a model as a modification of the flow, e.g. "increase by 'b'" (Figure B.5).

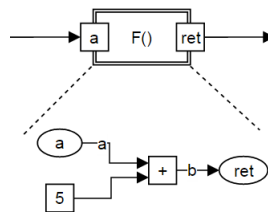


Figure B.5: Indirect assignment is modeled by flow modification.

B.5 Modify-by-reference

There is a common practice in C language to modify a value of the variable by passing it to a function by reference (i.e. as a pointer), which should update its value 8). In contrast to the copy-by-value, only the address of the input parameter is copied to the stack. One practical example of usage is referencing real-time database array elements.

Thereby, if a constant pointer is passed to a function as an input parameter and the function modifies the input parameter, this constant pointer should be exposed as an input port of the component.

```

1 void MultByTwo(int *a){
2     *a = *a << 1;
3 }
4
5 /* Some scope */
6 {
7     int b = 5;
8     MultByTwo(&b);
9 }

```

Listing 8: Example of modify-by-reference.

B.6 Function call with side-effects

This type of flow usually takes place in void functions, which changes global variables and/or calls another function with side-effect 9.

```

1 /* Global variables */
2 int speed;
3
4 /* Updates the speed with an offset */
5 void updateSpeed(int offset){
6     speed += offset;
7 }
8
9 /* Some scope */
10 {
11     int v_dif = 5;
12     updateSpeed(v_dif);
13 }

```

Listing 9: Example of function call with side-effects.

It should be noticed that general data-flow models do not appear to fit well with programming languages involving side-effects, since it can produce read-write races and other subtle timing bugs [39]. However, as mentioned

previously, Simulink enhances dataflow paradigm, though making it not completely deterministic.

Conceptually, a previous modeling approach, i.e. modify-by-reference, might be similarly used to model the current example.

B.7 Environmentally tied variables

A practical example here is memory-mapped IO, where the value of a variable can be set by events in the environment, such as sampling period of sensors (Listing 8).

```
1 #define SENSOR_ADDR 0xDEADBEEF
2
3 int *fuelLevel = (int *) SENSOR_ADDR;
```

Listing 10: Example of environmentally tied variables.

In this case, one variable (could be local/global) behaves like a pointer to another variable. This situation might be handled by a global data storage provided by Simulink, where one data storage is in sync with the another (Figure B.6).

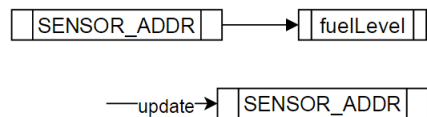


Figure B.6: Sync data storage in Simulink.

B.8 Modification through interrupt

In real-time system a task execution can be interrupted by a processor or peripherals. When an interrupt occurs, ISR (interrupt service routine) is invoked by an interrupt handler to process it. ISR in its turn might change environmental variables used by the interrupted routine (Listing 11).

```

1  /* Global variables */
2  static volatile int power;
3
4  void ISR_stop(void){
5      power = 0;
6  }
7
8  /* Some scope */
9  void MoveForward(int a){
10     a = power;
11     // An interrupt comes here
12     // now, a != power
13 }

```

Listing 11: Example of modification through interrupt.

There is a special Real-Time™ toolbox available in Simulink for supporting various real-time features. These models should be created for a dedicated target platform, since they manage HW characteristic, which cannot be abstracted away.