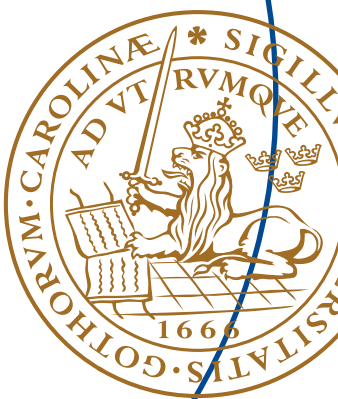
Master's Thesis

Variability modeling in automotive embedded systems

Daniel Hilton



Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, June 2014.

Variability modeling in automotive embedded systems

- from specification to implementation

Daniel Hilton
daniel.hilton@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisor: Anders Ardö, Associate Professor, Ph.D,
Dept. Electrical and Information Technology

Advisor: Mattias Nyberg, Expert Engineer, Ph.D,
Scania

June 19, 2014

Printed in Sweden
E-huset, Lund, 2014

Abstract

To satisfy the needs of the highly competitive and customer oriented automotive market most manufacturers target the specific needs of their prospective customers by creating a highly flexible product line. Over the past two decades the complexity of automotive vehicles has grown rapidly [1]. As vehicle functionality has increased, so has the amount of possible configurations that a vehicle can have. This is handled by introducing variation points in the embedded software in order to adapt its use to the different configurations. However this introduction is often given incidental treatment by developers which leads the system architecture becoming increasingly inconsistent and highly complex over time. As a consequence of this the impact of a vehicle configuration becomes extremely time-consuming and hard to evaluate. To be able to meet the increasing demands on functional safety in the automotive embedded systems the impact of this variability must be known and evaluated.

This thesis aims to explore the possibility of recovering a model which describes the software variability within Scania's embedded systems. The method that is used involves the definition of a model through using the information and data that is retrieved from the process of architecture recovery. This results in a model that provides an architectural overview of where the vehicle configuration choices affect the software of the embedded systems within Scania. The conclusion is that although the recovered model provides information about the variability there are large limitations when recovering variability from legacy systems through architecture recovery.

Keywords: ISO26262, Variability, Architecture recovery, EAST-ADL, Embedded systems.

Acknowledgments

During the course of this thesis many paths were investigated and found to be dead ends. The notion of variability is still very much an area of active research and hard to grasp in a consequent way. A hope is that this report highlights both the use of architecture recovery for gaining vital knowledge about the variability in embedded systems and also the scope of the problem that I have tried to solve.

I'd like to thank my supervisors Mattias Nyberg and Anders Ardö for their constructive feedback and support during this thesis. A very special thanks also goes out to Cecilia Haugland for being a constant support and source of inspiration during the last three and a half years.

Table of Contents

List of Figures

Chapter 1
Introduction

1.1 About Scania

Scania CV AB is situated in Södertälje, Sweden, and was founded in 1891 [2]. From the very beginning Scania have concentrated their development efforts in the heavy transport segment. Today Scania is one of the worlds foremost manufacturers of bulk transport vehicles and has production units in multiple locations across the globe.
Approximately 3 300 people, representing 9% of the total workforce of Scania, work within *research and development* (R&D). Within Scania there is a holistic approach to R&D where they focus more on methods rather than results. They emphasize the importance of lessons learned within the organization. As is illustrated in 1.1, the R&D process follows four main branches; research and advanced engineering, concept development, product development and product follow up. The whole process of developing a new product starts in the research and advanced engineering branches where state of the art concepts are researched and tried. If the results of the research are deemed to be promising the concepts are developed as a part of the yellow arrow branch. This is effectively product predevelopment. If this stage shows results the product is moved to the green line of development where it is subsequently released to customers.

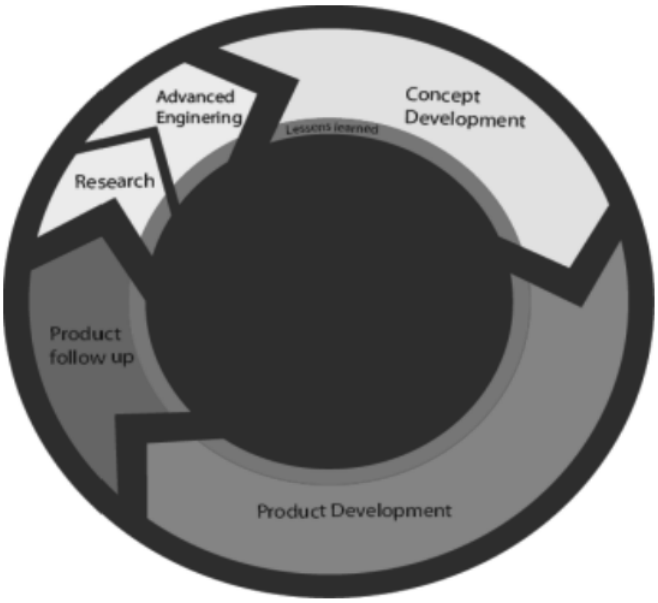


Figure 1.1: The development cycle within Scania

The research and development processes that exist within the yellow arrow process are divided between three main departments, the N department which focuses on power train development, the Y department which focuses on vehicle definition, continuous improvement and product quality and the R department which focuses on truck, bus and cab development [3]. This thesis was conducted under the R department, in the REPA group which focuses on Advanced Driver Assistance Systems and product pre-development.

Scania introduced the concept of modularity in their vehicles during the 1940s to increase customer choice and satisfaction. The current modular system is divided into four main parts; cabs, engines, transmissions and chassis [4]. Customers are able to combine modules from the different parts in order to build a vehicle that suites their needs.

1.2 Motivation

Over the past two decades the complexity of automotive vehicles has grown rapidly [1]. This is driven by the significant increase in the number of computer based functions that are embedded in vehicles. These functions aim to improve passenger comfort, safety and economy [5]. The introduction of more functionality has led to an increase in the amount of possible configurations that a vehicle can have. When ordering a vehicle from Scania the customer must make around 200 different choices in order to fully define the desired functionality of the vehicle. These choices create in excess of 10 000 different vehicle variants. Managing the resulting differences between the vehicle variants, referred to as product variability, is a key success factor of the modern automotive business [1].

As it is not feasible to create a completely new embedded system for each possible vehicle variant there is a need for enabling the reuse and adaptation of common system components. This is frequently done by creating software product lines. The philosophy is to exploit the commonalities between software products while at the same time preserving the ability to vary the functionality between the products [6]. In other words software product lines are groups of applications that come from the combination and configuration of generic system components.

One way of achieving the efficient derivation of system variants is by delaying customization details to a late stage of the production process. This can be achieved through the introduction of variability in the software of the embedded systems. Variability describes the ability of an artifact or of a system to be used in different contexts by changing or customizing some characteristic or aspect of it. The changeable characteristics are located somewhere in the system artifacts and are realized with what is commonly called variation points. Variation points identify all places where members of a product line may differ from each other. The difference may be in the existence of a certain software module or the difference in parameter values. Thus they are used to alter the systems behavior depending on the desired configuration of functions later in the production process [7]. The whole collection of variation points within a embedded system is typically referred to as the embedded systems variability.

High variant complexity can often be found in the development of automotive embedded systems. These systems often control safety critical functions with hard runtime constraints. In parallel to the increasing system complexity the funcional safety in vehicles is becoming more and more important. This is exemplified by the introduction of stricter safety constraints by the ISO 26262 standard. The overall aim of the standard is to address possible hazards caused by the malfunctioning behavior of electronic embedded systems within automotive vehicles. The standard requires that each configuration of a vehicle is to be verified in order to produce convincing arguments of its functional safety [8]. In order to provide this the impact of each different vehicle configuration on the embedded system needs to be identified.

This is a problem at Scania as the introduction of variability in a system is often given incidental treatment by its developers. This leads to a variation point often being introduced based on heuristics or expert knowledge [5] where no defined architectural model is followed. As a result of this the system architecture

becomes increasingly inconsistent and highly complex over time. As a consequence the impact of a vehicle configuration is extremely time-consuming and hard to evaluate. The evaluation requires the ability to reason about the software on an architectural level, presently within Scania this is not possible. The ability to do this is however widely recognized as having potential to further and improve many aspects of the software development processes ranging from the detection of errors and inconsistencies to the better evolution, reuse and understanding of a system [9]. Due to these factors a representation of the system variability on the architectural level is considered as a necessity by Scania.

An approach at creating this representation is to migrate to a top-down, model based, development process. The product line variability is then explicitly defined on multiple levels of abstraction. However this solution is currently unfeasible for large legacy systems such as are found in Scania as it is both time consuming, costly and can face objections within the organization. Another solution, that is not as obvious, is to take a bottom up approach and recover information about variability within Scania's product line by using existing data sources. The aim of this is to produce a well defined architectural model of the variability on different levels of abstraction. This approach is commonly referred to as architecture recovery.

1.3 Goal and problem statement

The overall purpose of this thesis study is to analyze variability across the development life-cycle of the automotive embedded systems within Scania. The analysis should result in the recovery of a variability model that can be used to further the knowledge of the embedded systems and their architecture. The model should support important notions about variability that are to be identified by analyzing common variability concepts and current architecture description languages. The possibilities to automate the task of populating the recovered model shall be demonstrated. The thesis documents a case study at Scania CV AB.

With this in mind the following questions will be addressed in this thesis:

- What is the possibility of recovering a model which describes the software variability within Scania's embedded systems?
  - How to identify the variation points that are used to configure the variability?
  - How to represent the variation points in the model?
  - What are the levels of abstraction needed?
  - How can the concepts of a theoretical architecture description language (ADL) be applied to the recovered model of variability within Scania?
  - To what extent can the model assist developers in fulfilling aspects of the ISO26262 standard?
- What is the possibility of populating the recovered model?
  - How to determine and retrieve the data needed to populate the variability model?

  - What is the possibility of automating the population of the model from the existing data structure at Scania?

1.4 Delimitations

As the scope of this thesis is large certain constraints have been placed on the present analysis.

- The case study within Scania will be carried out on only two embedded systems, the coordinator and the engine management system. However as these two embedded systems are developed within different departments they are very different from each other and should provide a good general picture of the viability of the solution.
- The variability within the order production system is assumed to be known. Therefor the vehicle specification (V-spec) is used as a starting point of abstraction. This is due to the limited access that is given to these preceding systems within Scania.
- The study of architecture recovery is done on the application layer of Scania's embedded systems. This is due to the vehicle features being implemented in the software components in the AUTOSAR standard, which reside in the application layer.
- Only static analysis is done in order to extract information from the source code.
- Given that the thesis was performed in Scania it is natural to perform the variability study for this case only. As Scania is an established and prominent company that uses proven standards and development methods it is expected that the general concepts and results can be carried over to other scenarios.

Chapter 2
Background

The ability to reason about software on an architectural level is widely recognized as having potential to further and improve many aspects of the software development processes, from the detection of errors and inconsistencies to the better evolution, reuse and system understanding [9]. Since the 1960s cost, quality and time to market have been the main concerns in software engineering [6]. These concerns are addressed by exploiting the similarities in a set of products and building very flexible product lines. The flexibility creates a need to be able to easily tweak a product and allow it to vary depending on the specific usage. This introduces variability into the product and thus a need to represent this on an architectural level arises. The main problem lies in the fact that this representation seldom exists or is up to date and has to be recovered directly from the systems implemented artifacts - a demanding process commonly referred to as architecture recovery [10]. This case study aims to demonstrate an approach at retrieving information about software product line variability from real life embedded systems in a way that can provide the information needed to facilitate the high-level representation of the systems and their related variability.

2.1 Related work

In [11] the challenges of ISO26262 are shown. The paper discusses on a very abstract level what has to be done in order to bring product lines and functional safety together. The conclusion that is drawn is: Tool support is essential and the representation of features and where they impact artifacts within the system is a powerful way to provide sophisticated guidance to safety engineers. Further the concept of architecture recovery is widely researched and discussed in many papers. In a paper by J.Garcia et.al. [12] a comparative analysis of software architecture recovery techniques is made. It is disclosed that many of the techniques use patterns as a common way of identifying and extracting the relevant information. In a paper by A. Lozano [13] a comprehensive study is conducted in order to provide an overview of techniques used for the detection of software variability in source code. The paper concludes that although there exists literature describing how to implement variability in source code very few approaches for the recovery of variation points exist. The one that was identified involved analyzing applications that extend the same object oriented framework and using the results to find

possible variation points. This approach is however not applicable to Scania. Literature regarding mining for variants, which implies assigning a high-level concept to the product variability, in order to trace the implemented variation points to their configuring feature was not found at all. In an article by Mengi et.al. [14] the concept of developing a variability model that represents the implementation in source code is researched. The approach identifies the need of a connection between the feature model and the implementation in source code, however they only recognize preprocessor directives as a way of implementing variation points in source code and rely on a top down approach at handling variability. The extraction of preprocessor based variability is examined by Sincero et.al [15] where they define a method of detecting inconsistencies in the usage of the preprocessor directives. This approach does not use information from multiple levels of abstraction, however a conclusion is that the extraction of variability information can be used in a variety of ways to improve software product line development.

2.2 ISO 26262

The ISO26262[8] standard is a recently introduced functional safety standard that is aimed for use in the automotive industry. The standard is an adaptation of the IEC 61508 standard to comply with the application domain requirements of *electronic/embedded* (E/E) systems in road vehicles. The standard is aimed at encompassing all aspects of development during the safety life cycle of systems comprised of electrical, electronic and software components. The standard defines functional safety as the absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems [16]. The main goal of functional safety is thus to make sure that installed electronics and software does not fail in a dangerous way.

The challenges the standard attempts to address are comprised of the following [17]:

- The safety of new E/E software functionality in vehicles.
- The trend of increasing complexity, software content, and mechatronics implementation.
- The safety risk from both systematic failure and random hardware failure.

These challenges are handled in the ISO standard by providing appropriate development and verification processes in order to achieve system safety. The introduction of the standard suggests an increase in the regulation and control of the safety requirements within the automotive industry as a whole. The main consequence of the standard is that all individual components of each produced vehicle variant shall be statically verified for their safety. The interaction and dependencies between components shall also be verified in the same manner. The safety activities in ISO start with a hazard analysis. This analysis is based on the intended use of the system. At this point the use can already differ between different system variants. The scope of the hazard analysis should cover all possible use cases in all possible variants [8]. The standard recognizes two main ways of configuring the software of an E/E system to allow for variability within a vehicle. These

are the use of C preprocessor directives such as #if, #ifdef to enable conditional compilation of the ECU source code as well as the use of configuration/calibration data to parameterize a system [18].

The specific relevant clauses regarding variability that were identified as being of importance for this thesis are 5.4.7, C.4.6 and C.4.7. Clause 5.4.7 in part 6 discusses the choice of implementation language and coding style. The Mirsa C language standard [19], which Scania has as a policy to follow, is explicitly mentioned as an example of a good coding standard. The main points of recommendation that are to be followed in the software development are:

- Enforcement of low code and structural complexity
- The use of established design principles
- The use of an unambiguous graphical representation

In part 6 - product development at the software level clause 5.4.3 states that "...if developing configurable software annex C should be applied". This annex encompasses system variability and the verification that is required [18]. Clause C.4.6 discusses configuration data and its specification. It states that the configuration data associated with software components shall be specified to ensure the correct operation and expected performance of the configured software. This shall include:

- the valid values of the configuration data
- the intent and usage of the configuration data
- the range, scaling, units, if applicable with their dependence on the operating state
- the known interdependencies between different elements of the configuration data
- the known interdependencies between configuration data and configuration data

Clause C.4.7 states that verification should be performed in order to ensure that:

- the use of configuration values within their specified range.
- the compatibility with other configuration data.

The standard further specified that the verification of configuration data can be performed with application specific software verification. Verification of configuration data can include checking the value ranges of the configuration data or the interdependencies between configuration data.

As of today Scania is not affected by the standard as it does not apply to vehicles in the weight class in which they manufacture (>3500 kg) [8]. Despite this Scania believe that the standard will eventually be extended to encompass vehicles which they produce. This assumption is supported in a research report by VDC research where they indicate that adherence to ISO 26262 is expected to increase significantly in the near future [20].

2.3 Architecture recovery

The process of architecture recovery aims at reconstructing viable views of a software's architectural composition [21]. It is a technique that uses development artifacts such as source-code and documentation to enable the reconstruction of the software architecture. Architecture recovery can be used for many different purposes, and its usage in order to understand architectural dependencies in embedded software is also identified by the Software Engineering Institute of Carnegie Mellon (SEI) [22]. In their technical report on architecture reconstruction guidelines they identify five phases of software reconstruction, in the present thesis the guidelines for two of the phases are used.

- Information Extraction: This phase involves analyzing the different existing development artifacts and constructing a model that captures the elements of interest and the relations between them. In order to do fact extraction from source code they identify the usage of parsers and *abstract syntax tree* (AST) analyzers. The AST-analyzers can be used to parse the code and build an explicit tree representation of the captured information. The tree structure can then be traversed in order to identify and extract select pieces of architecturally relevant information. When implementing the information extraction phase they recommend using the "least effort" extraction method.
- Database Construction: This phase is aimed at persistently storing the extracted architectural information in a way that reflects the actual architecture of the system. The recommendations that are given are to consider the database design carefully.

These two phases were identified due to the scope of this thesis being the modelling and extraction of currently existing variability.
The other phases involved creating applications/views that use the populated model, which is not encompassed in this scope.

2.4 Software product lines

To satisfy the needs of the highly competitive and customer oriented automotive market most manufacturers target the specific needs of their prospective customers by creating a highly flexible product line. This enables the mass production of vehicles that are built in accordance to customer wishes. A product line is a collection of closely related products that exhibit variations in their supported features [23]. In the automotive industry, where embedded systems are almost exclusively used to enable these features this diversity in the products presents a serious problem. The problem is related to the development of software to enable a flexible product line. This type of development is a lot more complex and demanding then the development of traditional single purpose systems. The complexity can be attributed to multiple factors such as often occurring intertwined products, features and production deadlines.

To solve this problem software product lines are used in companies as a way to provide a set of reusable assets for the related groups of software assets in a

product line [23]. The main difference between software product line engineering and conventional software engineering is the presence of variation in the software assets. The variation is introduced during the development stages of the software product lines life-cycle by implementing variation points. These represent options on how the software will behave during execution. The introduction of these variation points thus provides mechanisms to provide delayed design decisions. These decisions are then made at some later point during the production process in order to define the desired behavior of the finished product. The behavior of the software is thus altered through specifying the variation points behavior. The time at which the configuration decision for a variation point is made is referred to as the *binding time*, after this point in time the variation point is considered to be *bound*.

2.4.1 Features

Features are used to describe the differences in products that are created from the same product line. They give a highly abstract overview of the variability between the products [23]. Each feature represents some functionality that is enabled through software. The features of a product are normally visualized in a feature diagram, as can be seen in fig. 2.1. The general principal is that a single feature at a specific level of abstraction can be specialized through the decomposition into groups of less abstract features at a lower level [24]. For example the engine feature in fig. 2.1 can be decomposed into what type of engine it is. A distinction is also made between features that are mandatory and optional [13]. Mandatory features represent functionality that is always required to be present in some way while optional features represent functionality that may or may not be present. In figure 2.1 the mandatory features are the transmission and engine systems while the extra functionality represent optional features.

2.4.2 Variants

Variants are the result of allowing for product flexibility through delayed design decisions. The configuration of the variable features results in the derivation of a product variant [24].

The mandatory features can be decomposed into alternative features. The alternative features implement the mandatory feature in different ways depending on the configuration of the system. In fig 2.1 the alternative features are the features that are present at the abstraction level below the mandatory features. The alternative features can in their turn have variability dependencies on other features. These dependencies can be that they *require* or *exclude* other features. In fig. 2.1 the selection of a automatic transmission excludes the selection of a gas engine. The alternative features can also be mutually exclusive or inclusive, in fig. 2.1 the automatic and manual decomposition of the Transmission feature are mutually exclusive, thus only one of them can be included in a vehicle variant. The features that are mutually inclusive can all be selected at once, and don't have dependencies on each other. For example the cruise control feature can have both the predictive and adaptive features selected as extras in a vehicle.

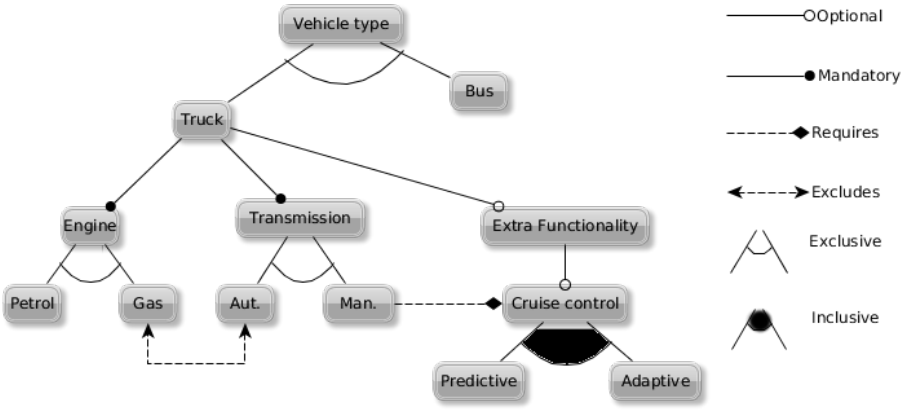


Figure 2.1: The features related to a vehicle.

2.4.3 Variation points

Variation points represent the mechanisms needed in order to realize the implementation of delayed design decisions in a product line. The optional and variable features are thus explicitly represented in the software implementation through the usage of variation points. To sum it up variation points describe where variations occur between different variants of a product. They describe what the choices are and how they are related to each other.

Techniques for realizing variation points in software product lines

There exists a multitude of ways for realizing the explicit representation of variability in software product lines. The realization can be done during different stages of the development process. An example of two different development stages and a subset of the identified realization techniques as identified in [7], are presented below :

- Compilation:
  - Condition on Constant: The intent of this realization technique is to support several ways to perform an operation, where only one of these ways will be present in the bound software system. This is done through the conditional statements that are a part of the C preprocessor directives. The directives can be used to alter the architecture of the system by opting to include one file over another or using another class or component. The consequences of using the preprocessor directives to realize variability can lead to an explosion of potential execution paths making the software maintenance and bug fixing an arduous task.
- Runtime:

– Condition on variable The intent of this realization technique is to support several ways to perform an operation, where only one of these ways will be present in the bound software system, however the condition can be re-bound post compilation. This is enabled by providing the functionality to assign a value to a variable post compile time. This technique uses pure programming constructs to change the behavior of the system. The consequences with this technique is the possibility for the programming constructs that are used to control a specific feature to be spread over a large amount of source files. This leads to it being hard to get an overview of the variability.
– Variant component implementations The intent of this realization technique is to support several coexisting component implementations of the same architectural element where only one of the component implementations will be present in the bound software system. This is enabled by implementing several component implementations that adhere to the same component interface but provide different behavior. The decision of what component implementation to use can be done at startup, where the decision is made by specifying a startup parameter to the system.

2.4.4 Variability

Variability is the ability to change the behavior of a system through customization. The concept of variability can thus be thought of as defining the required features/characteristics that the system is to support and in what variants these can be configured [25]. The concept of improving variability within a system implies making it easier to do certain kinds of changes to the vehicle configuration. Early in development all possible systems can be built. Through each step of the product life-cycle the variability within the system is constrained. This is done through the binding of variation points until finally at runtime there is exactly one system.

In an article by Bosch et.al [26] they highlight several problems with handling the introduced variability within software product lines, these are:

- Knowledge gap: when large scale systems are to be designed and implemented there is often a gap in the knowledge between the domain expert and the engineers that implement the system. This gap between high level design and low level implementation can result in an error prone and inconsistent implementation. The knowledge gap also makes the introduction of new variation points an arduous task to perform.
- Traceability: The traceability, which is the ability to trace the artifacts of the high level design down to the implementation level, is often lacking within a system architecture. This in its turn leads to few people having a full understanding of the system variability.
- Scattered variation points: A feature in the system at the requirements level can often lead to multiple variation points that reside in different modules in the final implementation. There can also be modules that contain

variation points which are affected by multiple features. As a result the identification of all code that is dependent on a feature, which is a necessity for maintenance, is hard to realize.

2.5 Variability within Scania

At Scania the product line variability is handled with *functional product characteristic* (FPC)-codes. These codes describe a products functional characteristics and encompass all aspects of a vehicle on all granularity levels. An FPC code has different executions that can be chosen. The whole collection of executions make up the FPC:s variants. The concept of this can be seen in figure 2.2 where a FPC code is decomposed in to its mutually exclusive variants. If a product specification contains the FPC-1 in variant A we know that the product is a truck. This also means that FPC1-B or FPC1-C cannot be present in the specification if it is to be correct.

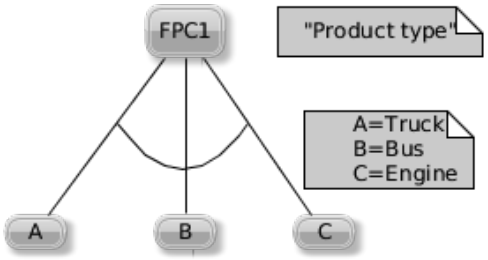


Figure 2.2: FPC decomposition into its variants

2.5.1 From order to vehicle specification

When a vehicle is ordered from Scania the customer has 200 different configuration choices to make. These choices are necessary in order to fully configure the vehicle and its features, adapting it to the customer needs. The configuration choices that are made are used as a basic input for the generation of the corresponding set of FPC-codes and their variants. The initial set of FPC codes describe what features that were chosen by the customer. These are used as the input to an application called SMOFS (Scania Manufacturing and Order Feature System). The SMOFS application generates a larger set of FPC codes dependent on the feature constraints, where an example can be seen in fig. 2.3. In the figure it can be seen that the selection of FPC1-B also prescribes the selection of FPC4-B.

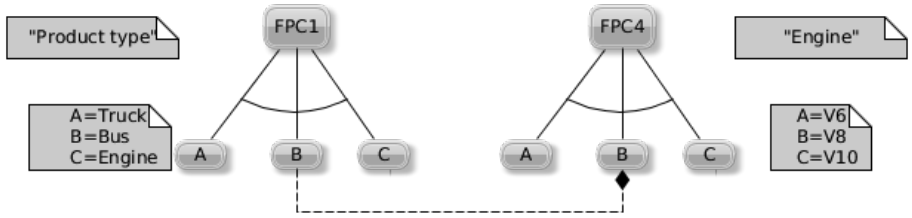


Figure 2.3: Feature constraints between FPCs

The set of FPC codes are output from the SMOFS application specify the vehicle and all of its features, taking into account the feature constraints that are not configurable by the customer. These FPC codes make up what is known as a *chassi-specification* (C-spec) [27]. The chassi-specification that is produced does not consider the dependencies that exist between the selected features and their required hardware. As the FPCs are used also for the specification of the vehicle hardware, on a lower level of abstraction then the features, these need to be generated. For example a truck(FPC1-A) with a diesel engine (FPC2-B) also needs a diesel particulate filter attached(FPC33-D). The dependencies between features and hardware components are handled by the *translation control register* (TCR) that generates all of the FPCs needed to build the vehicle. The output is then sent to the *variant control register* (VCR) which validates the data. After this stage in the production the result is a VCR-controlled specification, commonly known as a *vehicle-specification* (V-spec).
As the V-spec only contains abstract descriptions of the hardware that is required for a vehicle there is a need to convert these FPC descriptions into actual component codes. The Mainframe application IBM-MONA is used in order to convert the V-spec into a component specification. The V-spec is also run through the construction structure (KS) part of Spectra in order to add information about the necessary cable lists that are needed. These define how the components are to be connected in a vehicle. The resultant vehicle definition contains all information that is needed in order to build the specific vehicle. The whole process can be seen in figure 2.4.

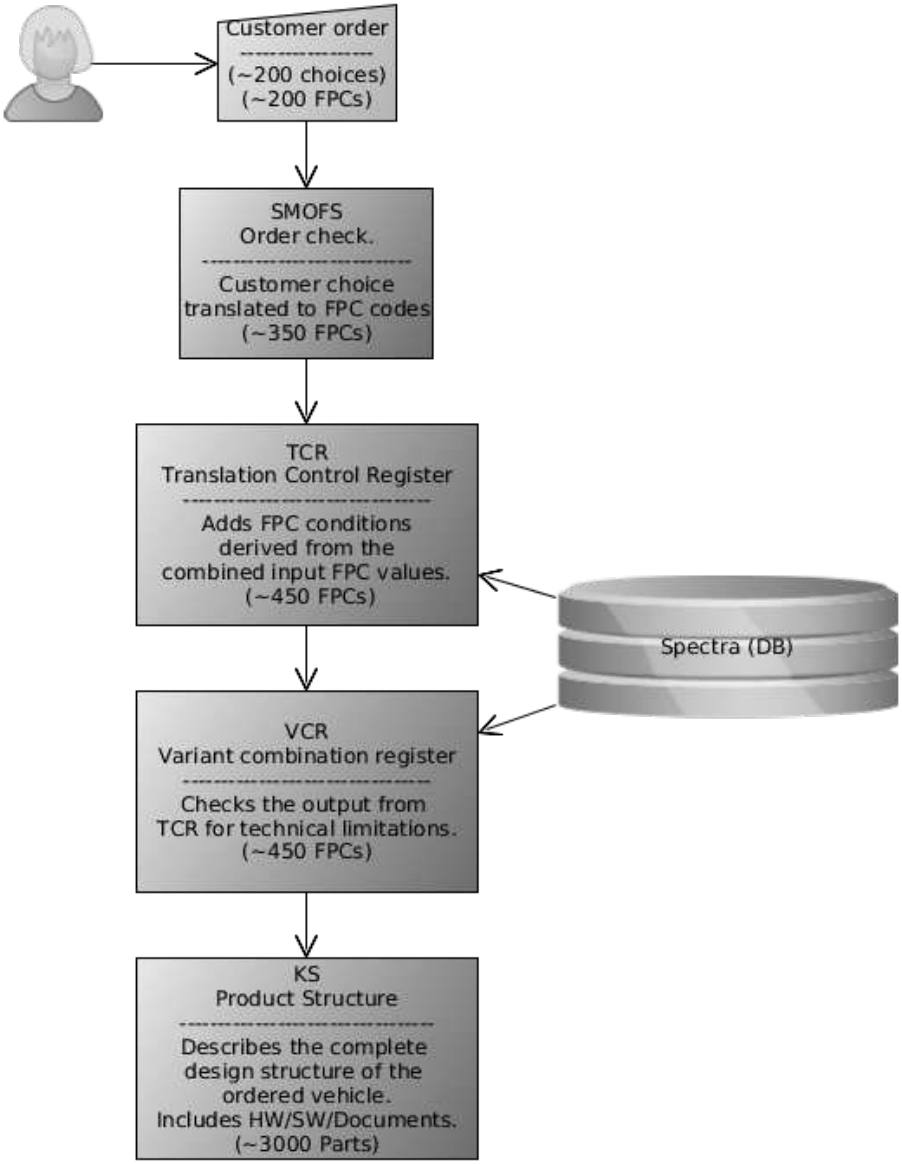


Figure 2.4: FPC and component generation process within Scania

2.5.2 SOPS file

For each vehicle that is built within Scania a configuration file is generated, this file is called the *Scania Onboard Specification* (SOPS) file [28]. It is used as a specification for the vehicle variant and its features. It contains different blocks

that are used to specify the necessary configuration items that are needed in order to reproduce the vehicles E/E system configuration. The blocks that are contained in the SOPS are:

- **FPC block**: This block contains a comprehensive list of FPC-codes and their variants. Further it contains all of the FPC variants that are needed to fully parameterize a vehicles electrical system.
- **XPC block** : This block contains converted numerical values of the FPC codes. These values are to be used in e.g. vehicle parameter adjustment.
- **Cablelist block** : This block specifies the used cable harnesses in a vehicle, their connection ends and attached hardware component-codes.
- **ECU block** : The ECU block contains information of the connected ECUS in the vehicles electrical network.

2.5.3 Electrical system within Scania vehicles

The electrical systems within Scanias vehicles are mainly driven by *Electronic Control Units* (ECUs). The ECUs are embedded systems that control one or more of the electrical system or subsystems in the vehicle. They are responsible for enabling the vehicle features that are to be supported as well as reading the sensors and actuators that are connected to them.

More than 30 different ECU families exist within Scania, these ECUs are distributed over three controller area network buses (CAN-buses) of different criticality. Each ECU in the system has different responsibilities and handles different vehicle functionality, thus not all ECUs will be present in a vehicle. The main ECU:s in a Scania vehicle that are always present are the Engine management system (EMS), Gearbox management system (GMS) and the Coordinator system (COO). The Coordinator system has the role of coordination of the different buses of the vehicle CAN network and thus acts as a message gateway as seen in fig 2.5.

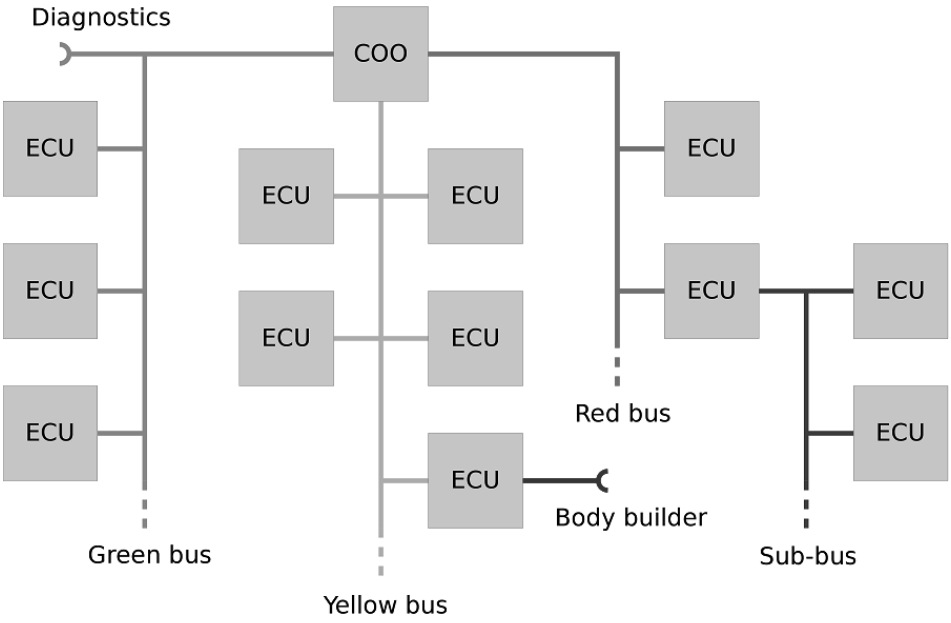


Figure 2.5: The ECU network where the COO ECUs role can be seen

Each ECU can be configured to support a various range of features. They are built as monolithic systems. This means that all of the source code is present in all variants of the configured ECU, independent of what features it is to support. In order to configure the ECUs and enable their re-usability and adaptability they are subject to parametrization during the vehicle construction phase. The parameters that are set during this phase determine what features the specific ECU should support. This determines the vehicle variant.

2.5.4 Variability within Scania's electrical systems

Within Scania an ECU can provide a large range of different features as can be seen in fig. 2.6 [29].

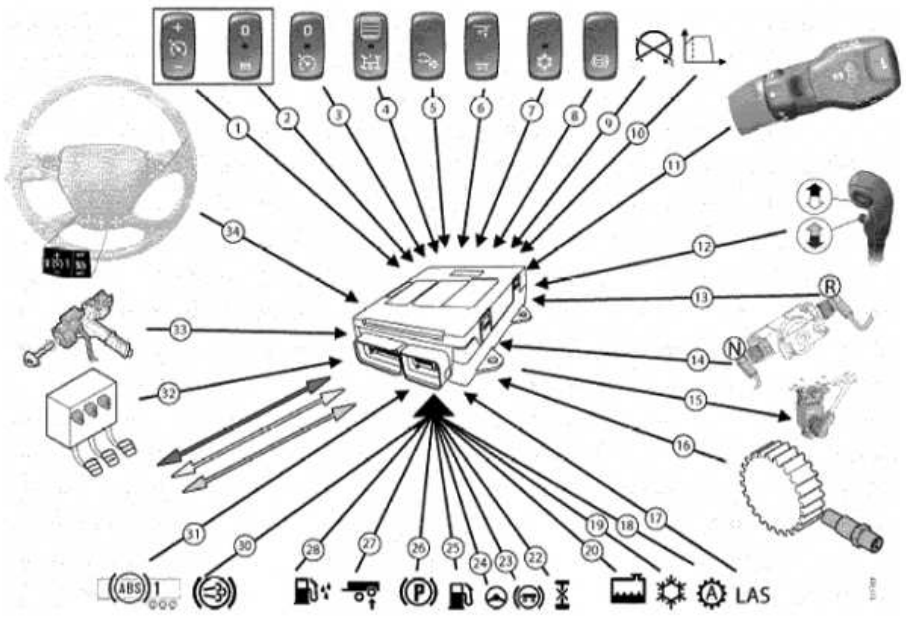


Figure 2.6: Functionality provided by the COO7 (Not complete).

The features and their functionality are implemented in the software of the ECU. When an ECU is used in a vehicle not all of the features that it supports will necessarily be enabled. The features will not always behave in the same way either. For example there can be a cruise control feature that an ECU is to enable. This feature can be enabled/disabled depending on the customers decision when ordering. The features behavior can also be configured by what market the vehicle is sold to. In USA the feature may require the usage of a radar to determine its speed while in the EU it may use the GPS. This variability in what features are to be supported and in what way motivates the introduction of delayed design decisions in Scania's embedded software.

As Scania's ECUs act like monolithic systems, the delayed design decisions are introduced as variation points directly in the systems source code. What subset of features a specific ECU supports is determined by the later parameterization of the ECU software, this binds the variation points and creates exactly one system.

Parameterization within Scanias ECU:s

The parameterization process is complex and the mapping between the features that are to be supported and their corresponding effect on the variation points within the architecture is not well documented within Scania. When analyzing the parameterization process a distinction is made between off-board and on-board parameters, which can be seen from figure 2.7. The off board parameters can be seen as configuration placeholders that are used to to configure the on board parameters of a vehicle. The on board parameters in Scanias ECUs are bound during the parameterization process that happens pre-runtime. The on-board parameters are mapped to variables that reside in the sourcecode of the ECUs. In the parameterization process the memory locations where the variables are stored are assigned a value. This is done through the use of external tools to access the memory. As far as the compiler is concerned the variables are in fact variable. However, as the memory locations where they are stored can only be written externally they are in fact constant after the parameterization process. The product line variability is achieved through using the parameterized variables to alter the behavior of the ECUs software components. This is done by controlling the programs execution flow through common programming constructs. An example of this could be an if-statement where the outcome of evaluating the conditional expression has a dependency on the parameterized variable [27].

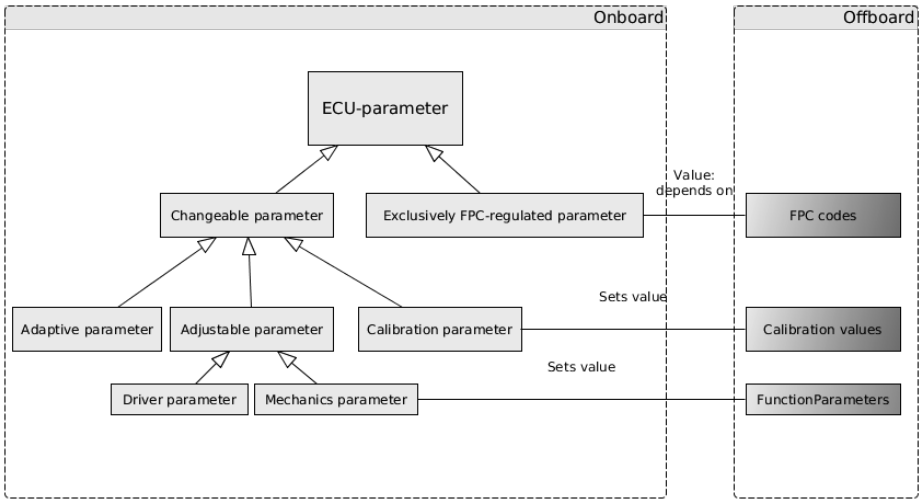


Figure 2.7: On-board and Off-board parameters.

As can be seen from the figure the on-board parameters are referred to as ECU-parameters within Scania [30]. The term ECU parameter is used as an umbrella term to describe all parameters that are utilized within an ECU to effect the software behaviour [31]. ECU parameters can be divided into two different subcategories [31]: *Exclusively FPC-regulated Parameters* (EFP) and changeable

parameters. EFP are parameters that obtain their associated value exclusively through the specified FPC codes in a vehicles SOPS file. As they are bound prior to runtime, through the parameterization process, they are what affect a vehicles embedded systems feature-dependent behavior. As the EFP values are bound through the specification of the vehicles FPCs they do not change during the vehicle lifetime. An EFP value represents a vehicle feature in software. The FPCs that specify the configuration of an EFP are of a more specified nature than the feature that the EFPs describe, thus there is no top down process of specifying and configuring variability within Scania.

The second category of ECU-parameters, the changeable parameters, all have the property that they can be changed during runtime. This can be done for example through the signals that a system recieves from sensors and actuators. As they are not bound prior to runtime they do not have a direct affect on a systems feature-dependent behavior.

The process of parameterizing an ECU to obtain different software variants can be seen in fig. 2.8. One physical ECU can be associated with many different variants of the software depending on what features that are selected. The parameters with *parameter ids* (PIDs) that are set by the specification of functions through FPCs are the parameters which contribute to the software product line variability within Scania products.

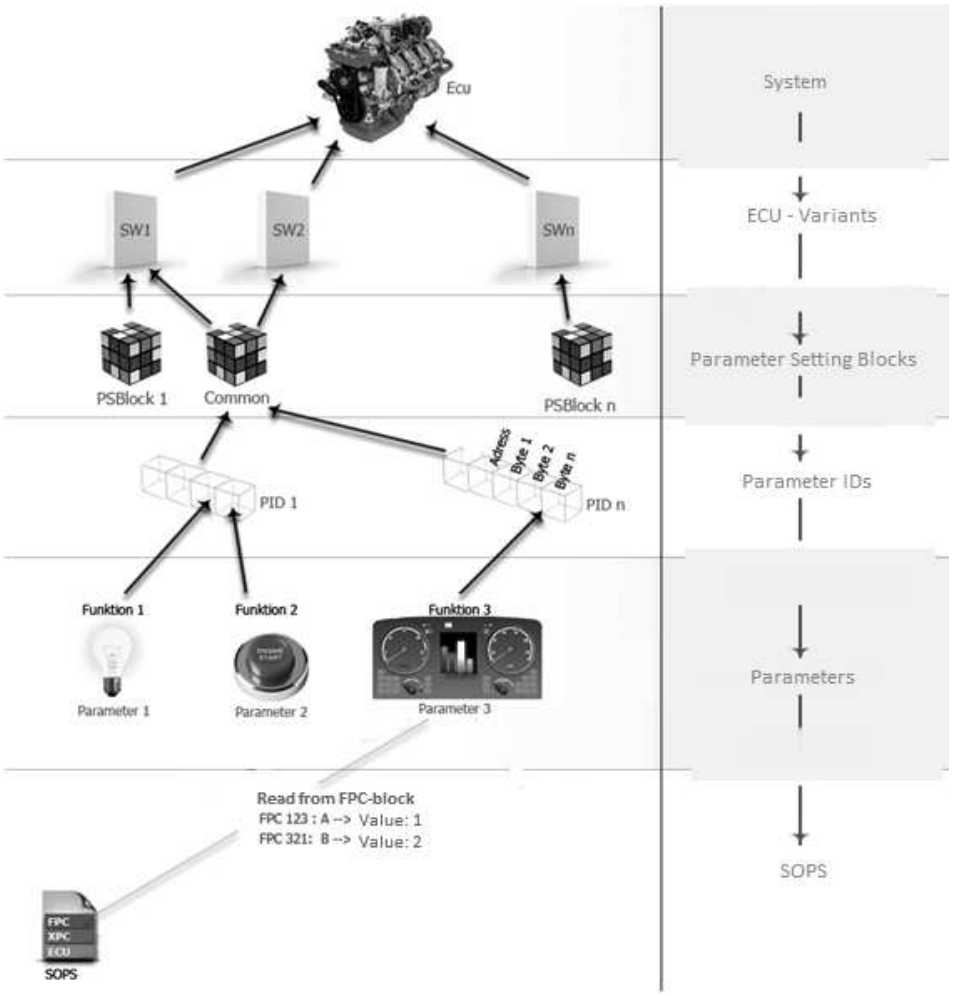


Figure 2.8: Simple configuration model of the embedded software within ECUs

2.6 Theoretical variability modeling approaches

Within theory there are many approaches at modeling variability in software product lines. The approaches are part of what is commonly referred to as *Architecture Description Languages* (ADLs). After analyzing the ADLs evaluated by Sinnema and Deelstra [6] along with the *Electronics Architecture Software Technology* (EAST) -ADL [32] and *Architecture Analysis and Design Language* (AADL) [33] models a decision was made to investigate the EAST-ADL variability model and its application to the domain of Scanias software product line variability. The decision was motivated by the fact that the EAST-ADL model is both developed for application within the automotive industry and is backed by many of the large companies within this domain such as Volvo, AUDI, BMW etc. As the model is developed through case studies within the domain of automotive engineering it was deemed more interesting to Scania than other languages. The later versions of the model are also aligned with the ISO26262 standard [32].

2.6.1 EAST-ADL

The East-ADL language is aimed at describing automotive electrical and systems in a comprehensive way and with sufficient detail to allow modeling for analysis, documentation, synthesis and design of complex embedded systems [32]. This aim is realized through the utilization of an information model that captures engineering information in a form that contains all aspects of a vehicle represented in a standardized way. The different components of the language are shown in fig. 2.9 [34].

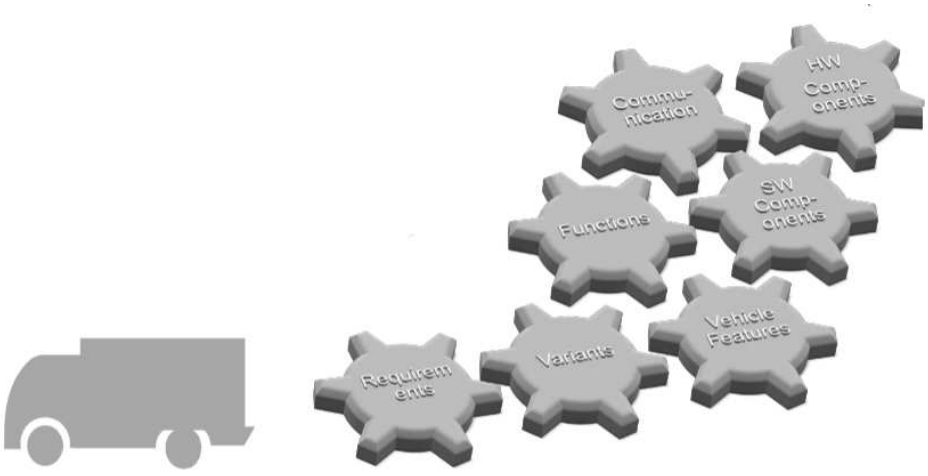


Figure 2.9: The different elements of the EAST-ADL model

The language introduces abstraction levels to allow for the reasoning about product line features on each level. The abstraction levels are however only concep-

tual; the modeling elements used are organized according to the system artifacts, which may span one or more layers [32]. A feature is a concept that is highly abstract. To see how the feature effects a system it needs to be decomposed into the artifacts that it has an effect on. In figure 2.10 one can see the concept of abstraction where the complexity increases when the abstraction decreases.

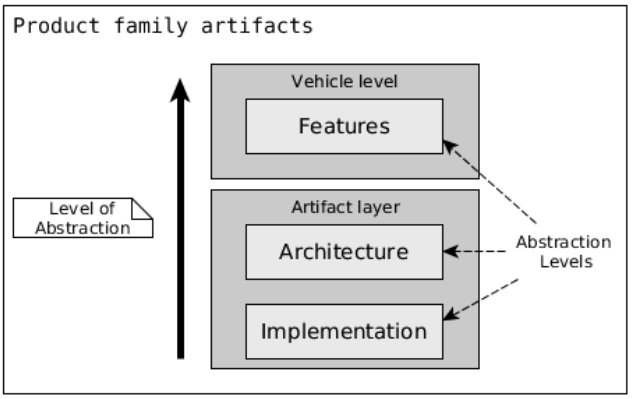


Figure 2.10: Abstraction levels and their decomposition.

The four abstraction levels covered by EAST-ADL are the vehicle, analysis, design and implementation levels. The abstraction layers that are present on a lower level than the vehicle level are commonly grouped into the artifact-level.

- Vehicle level - This layer represents the characterization of a vehicle by a means of its features. It contains feature models which describe the decomposition of system characteristics [32], in these models each vehicle feature denotes a functional characteristic of the vehicle. This level of abstraction thus states what features the vehicle consists of, however it does not represent how these features are realized and what effect they have on the lower levels of abstraction.
- Arifact layer - This layer is decomposed in many sub-layers describing the different artifacts of the system.
  - The analysis level gives an abstract functional description of the E/E system. This level realizes the vehicle functionality based on the vehicle features and requirements, independent of implementation details, thus this layer defines the embedded system from a functional point of view [35]
  - The design level describes the concrete functional definition of the vehicle. This encompasses the functional definition of the application software and their behavioral description however it excludes software implementation constraints [32].
  - The implementation level contains the software based implementation of the system where the system elements defined by the AUTOSAR standard are used used for the representation of the low-level software architecture. Traceability is supported from the implementation level artifacts to the vehicle level elements of the model.

Variability model

The EAST-ADL model has an extension to manage variability. This model originates at the vehicle level where the vehicle features and their variability are represented. A variability in this sense is a part of the complete vehicle system that changes in the different variants of the complete system. The vehicle level of the model is aimed at providing a highly abstract overview of the variability in the defined system together with the dependencies between these variabilities [35].

An example of this is: The speed may or may not be controlled by the use of the cruise-control feature. At this level the impact that this variability has on the system is not defined, however the fact that this variability exists within the system is defined through the introduction of an optional feature called e.g.Cruise Control. From this the following definitions are made [32]: *Variability* - an aspect of the system which changes between variants. *Abstract Feature* - show that the system has such a variability but not how/where the variability affects the system.

Feature models in EAST-ADL are used in order to define the systems commonalities and/or variabilities. On the vehicle level the *Core Technical Feature Model (CTFM)* is used to define the complete systems global variability from a

technical perspective. In addition to this there can be multiple *Product Feature Models* (PFM) that are used to define subsets of the CTFM. These are used to create a particular viewpoint of the system. This can be seen in fig. 2.11 where the CTFM is present along with a PFM that contains the features that are to be configured by the customers.

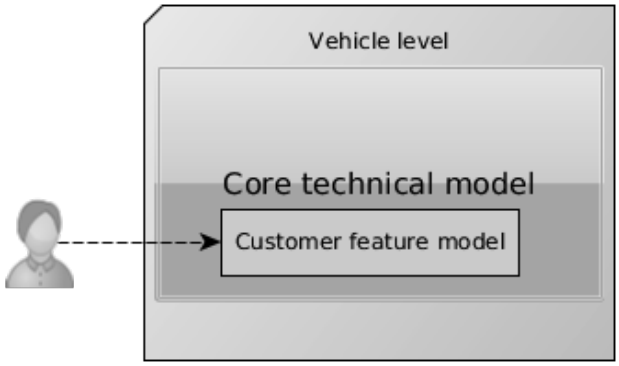


Figure 2.11: The CTFM and the Customer feature model

Feature models within the EAST-ADL language are also used on the artifact layers of the model, where they obtain a much more concrete meaning [35].

The details of how a variability affects and is realized in a system is of more focus when managing variability in other areas of the system development process. It is important to know how and where a vehicle feature affects the other system development artifacts. It is thus not sufficient to only describe that a variability exists within the feature model of a vehicle, it is also important to describe in what way a variation in the feature model affects and modifies the corresponding lower-level artifacts. An important aspect of the EAST-ADL language is that it promotes vehicle safety in general through enabling traceability between the different abstraction layers of a system. This is realized through the usage of ”realizes” relationships in the model [32].

The main components of the variability model are thus the feature models, the low-level artifact variation points and the corresponding realizes relations between these levels.

Configuration decision modeling

Within the variability model of the EAST-ADL language the concept of configuration decision modeling is introduced. In this modeling the configuration (selection/de-selection) of features that are contained in a feature model F1 is defined as the result from the configuration of another feature model F2. The configuration decision model is the link from F2 to F1 which enables the derivation of the model F1s configuration from the specified configuration of model F2 [35].

The concept of configuration decision models can be seen in figure 2.12. Here we can see that the configuration of the CTFM is derived from the user configuration of the User Feature Model. The configuration decisions are made by the configuration decision model that is related to the two models. The configuration models can also be used to drive the configuration of a feature model on a lower level of abstraction, as is also seen in fig 2.12.

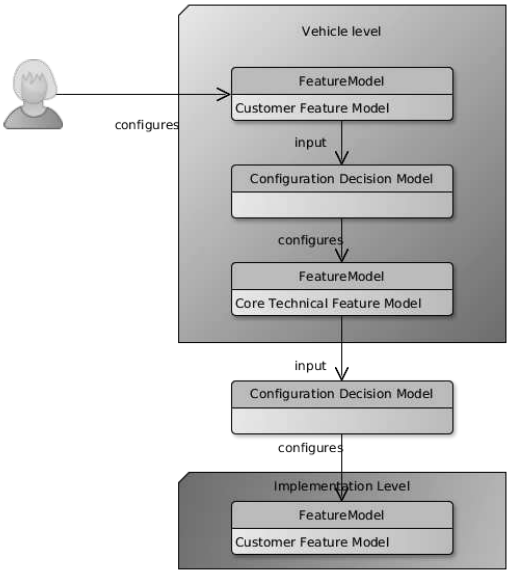


Figure 2.12: The usage of Configuration Decision Models

Configuration decision models are thus used to define how a high level feature configuration affects the binding of the variability in the lower-level components that implement the feature. This means that the variability management on the implementation/artifact level, which is done by the definition of variability points, is driven by the variability that is defined on a higher level of abstraction. The main driver for the binding of the implementation level variability is the configuration that is captured on the highest level of abstraction, namely the vehicle-level.

2.7 AUTOSAR

The *Automotive Open System Architecture* (AUTOSAR) is a defacto standard within automotive software system development. The standard is the result from the cooperation between the major entities within vehicle manufacturing [36].

The principal aim of the standard is to master the growing complexity of automotive electronic architectures [37]. The need to build a common architecture became compelling for a number of reasons such as defining a common understanding of how ECUs co-operate on same functions. The common architecture is aimed at separating the software from the underlying hardware in order to allow an increase software reuse between vendors and within systems.

The idea is to make Software Components (SW-C) re-usable and to standardize their interfaces so that they are independent of the external usage context such as what the underlying hardware is. This idea should result in a system that is flexible and easily scalable since the addition, removal and relocation of software components does not require any modification to the underlying HW system [38].

With this being the case it will be possible to reuse stable and well-tested SW-Cs. This will further a systems safety and result in more reliable vehicle systems. Another goal is to reach the point where enough software and hardware suppliers abide to the standard so that it becomes possible to mix software and hardware solutions from these suppliers seamlessly. The resulting infrastructure should be aimed mainly at facilitating the SW-C allocation and not their optimization.

Within the autosar standard their are three layers defined:

- Basic software layer
- Real time environment
- Application layer: The application layer consists of software components that realize vehicle functionality. All of the component re-usability that the AUTOSAR standard aims for is found in this layer.

Software components according to the autosar standard are comprised of:

- Application - is an atomic SW-C which realizes part of or all of a vehicle function.
- Composed
- Sensor-actuator
- Calibration - provides other SW-Cs with calibration data to configure them. Within this SW-C the parameter interface is found and is used by one or multiple SW-Cs to calibrate their internal logic.
- Service component
- ECU abstraction component
- Complex device driver component

2.8 Software control flow

The control flow of a program generally refers to the order in which the statements of a program are executed or evaluated. A control flow statement is a programming construct that results in a choice being made as to which of two or more paths should be followed in the program execution [39]. The choice of path results in the behavior of the system being altered.

The statements that depend on the evaluation outcome of a control flow statement are said to have a control dependence to the control flow statement. As an example the statements that are executed inside the block scope of an if statement are dependent on the conditional statement of the if statement. In the same way the statements inside the block scope of a switch statement are dependent on the evaluation of the case within the switch statement.

2.9 Graph databases

Graph databases2.13 are unlike relational databases as they are based on the concept of graph-theory. In this context a graph is a collection of vertices's and edges, which may more simply be referred to as nodes and relationships. The graph structures that are represented use nodes to represent the objects that one wants to keep track of. The relationships are what connect nodes to each other. By building graphs from these two components meaningful data structures may be represented in a natural way which allow for easy traversal. The properties linked with the property graph model are:

- Nodes contain properties (key-value pairs)
- Relationships are named and directed, and always have a start and end node
- Relationships can also contain properties

The figure 2.13 shows a representation of the components in the graph model and how they transform to real life data structures.

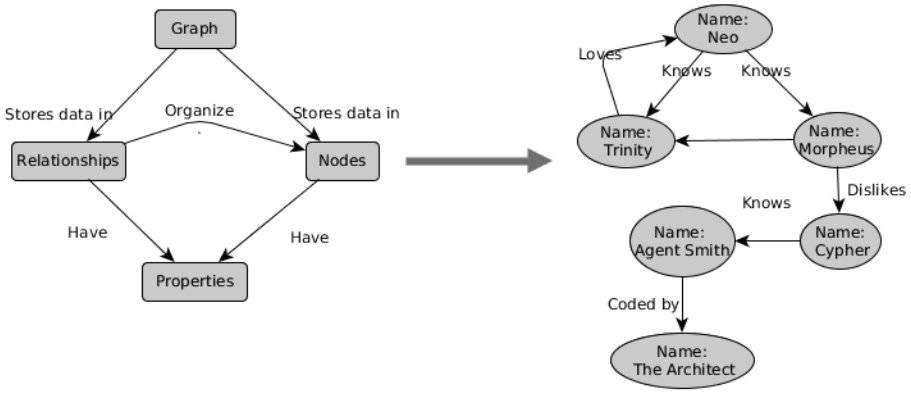


Figure 2.13: The concept of nodes and edges building a graph

As can be seen from figure 2.13 graph databases are a powerful tool to model data structures in a natural way. The graph structure also enables graph-like queries, for example computing paths between nodes and evaluating relationships between nodes. Through the assembling of nodes and relationships into connected structures, graph databases enable the construction of models which map closely to the problem domain. These resulting models are both simpler and and more expressive than the models produced when using traditional relational databases [40].

Chapter 3
Method

The method that is used investigates the possibility of recovering a model which describes the software variability within Scania's embedded systems through the use of architecture recovery. In order to derive an accurate model which can also be automatically populated an iterative approach at building the model is taken. This approach involves defining the model based on the information extracted during the architecture recovery process. As the theoretical models that were investigated all demand a top down development process, where the variability information is modeled before implementation, only the general concepts that are identified will be incorporated in Scania's model.

3.1 Variability model goals

3.1.1 EAST-ADL - defining concepts for modeling

As a means of constructing a variability model at Scania EAST-ADL and its variability model was used as a reference for gathering modeling concepts [32]. As the EAST-ADL model has been developed through case-studies at multiple large automotive companies the concepts are identified to be of importance also for Scania.

It was identified that the goal of the EAST-ADL model is to create an overview of feature-level variability in a production line and how/where it affects development artifacts within the software product line. From the analysis of the EAST-ADL model that was done as a part of the background study three concepts were identified as being important to model in order to fulfill this goal.

- Abstraction levels: As the vehicle configuration is done on a different abstraction level than the software configuration it is important to represent both of these levels. This representation will give an architectural overview of the variability in the implementation and also its relation to the variability in product configuration.
- Traceability: This is what enables a system to be represented in a coherent way on different levels of architectural abstraction. Traceability is thus important to have in a variability model that is to represent different levels of abstraction and their dependencies.

- Configuration decision modeling: When a vehicle configuration is made on an abstract feature level the configuration decisions are affected by constraints between features. These constraints are important to include in a variability model as they convey information about the resulting constraints in the software configuration. The concept can be seen in fig. 3.1 [41].

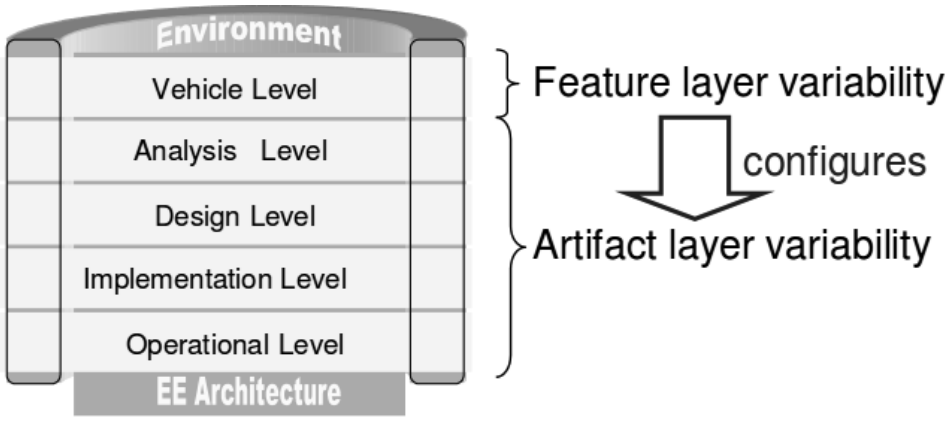


Figure 3.1: EAST-ADL variability

The evaluation of the EAST-ADL modeling framework resulted in two goals being defined for the variability model at Scania: The first goal is that the model should represent the configuration decision model that is used on the FPC level and in what way it impacts the implemented variation points. The second goal is that the implementation level variation points are to be modeled. The modeling of dependency between FPC level elements and the effect on the implementation level variation points is to be realized as fig 3.2 [42] shows.

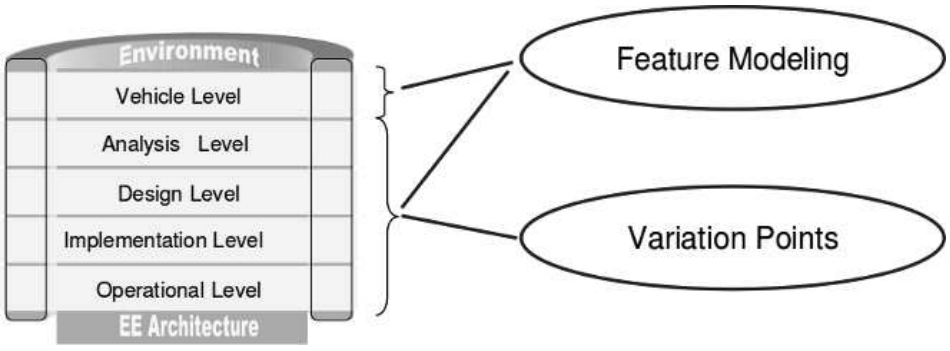


Figure 3.2: The model shall represent both artifacts from feature modeling and their connection to implementation level variation points

3.1.2 ISO26262

From the ISO26262 standard [18] it was identified that to enable the hazard analysis the possible effect of different configurations on a system must be represented. Thus an exact representation of the influence of different variants needs to be determined. In order to do this the model must help in the determination of where the configuration choices affect the system variation points. This means that the configuration on the FPC level of abstraction must be traceable down to the variation point which they affect. The model should also help in the verification of the correct usage of the configuration data, thus the model should represent how the configuration data is used in the variation points.

3.2 General tool chain for architecture recovery of variability

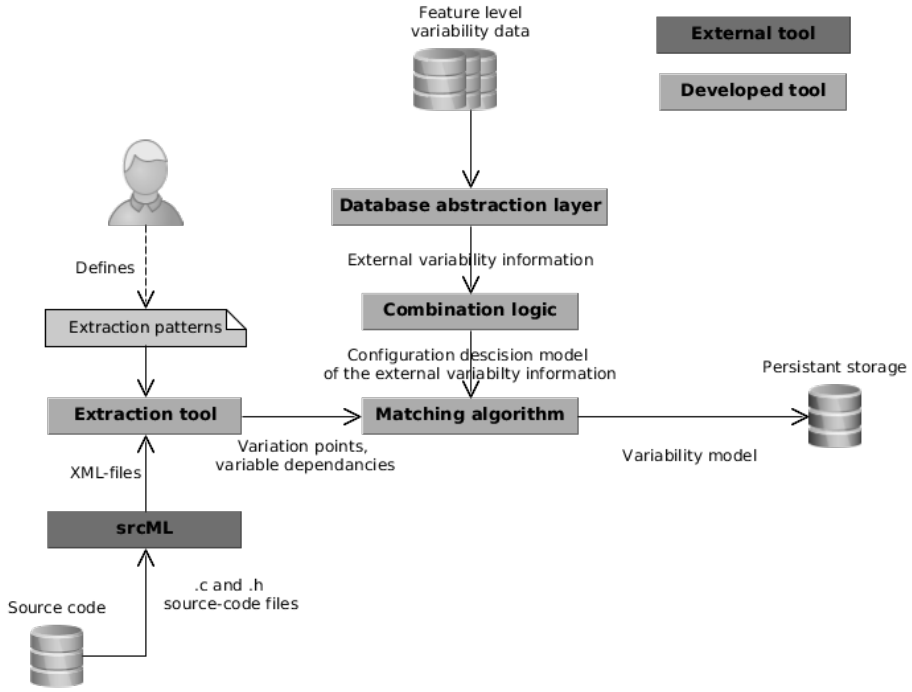


Figure 3.3: The tool chain that is used in the solution concept

The approach for extracting information regarding variability that was developed is shown in fig. 3.3. The approach is aimed at enabling architecture reconstruction of information regarding the variability within parameterized embedded systems at Scania. The means of the proposed approach is to make it possible to describe

at what locations in the implementation configuration dependent variability exists and under what configurations the variations can be applied.

The steps of the tool chain were developed using the information and recommendations in the technical report on architecture recovery by SEI [22]. The information is extracted from both source code and external data sources. This is done in order to create traceability between the different levels of variability abstractions that are present within Scania's software product line. To enable the extraction of information about the variability on the implementation level certain extraction patterns must be defined by the user. Extraction patterns are commonly used in architecture recovery approaches [12] as they can convey information about the system that cannot be captured in fully automated approaches. By using these patterns the tool will extract information regarding the variability within the source code. The information is then combined with the externally retrieved information in order to populate the variability model of the software product line. The populated model is to be used as a means of providing an architectural overview of the variability within Scanias software product line.

3.2.1 Inputs

- The inputs into the extractor are user-defined extraction patterns for identifying how the system handles the retrieval of the parameterized variables within the ECUs.

3.2.2 Tool chain components

Software extraction tool

The information that is extracted from the implemented software is to give an architectural overview of the FPC dependent variability of the source code. This is where the configuration of an FPC affects the software implementation. Following this a definition of exactly what needs to be extracted is made.

During the process of identifying the information to extract interviews were held with developers from Scania. These interviews as well as the analysis of the ECU source code resulted in identification of the realization technique that is used to enable software variability in Scania's parameterized E/E systems. This is done by implementing explicit variation points in the source code of the ECUs. The variation points alter the runtime execution flow of the system. The behavior of the variation points that realize the ECU systems variability is determined by variables that are bound to a value during the parameterization process. We call these *parameterized variables*. A parameterized variable is defined as a variable that is associated with a constant value pre-runtime through the parameterization process of the ECU. This means that these variables receive a constant value post compile-time but pre-runtime. The parameterized variables have a 1-1 mapping with the exclusively FPC regulated parameters that are used in the parameterization process. The EFP parameters, which are external parameters, can thus be coupled to the concept of a feature as they configure and describe a functionality that is implemented in software.

The variation points that are related to the ECU systems variability within Scania are also defined. As the variation points in parameterized systems are to affect the behavior of the system, the concept of control flow within a program was first analyzed. Within the C programming language the control flow statements were categorized by the effect that they have on the execution of the program [43]:

- unconditional branch or jump - Continuation of execution at a different statement within the program.
- conditional branch - The execution of a set of statements only if some condition is fulfilled.
- loop - The execution of a set of statements zero or more times, until some condition is fulfilled.
- subroutines, coroutines and continuations - The execution of a set of distant statements, after which the flow of control usually returns.
- unconditional halt - The termination of the program execution.

From the said interviews with developers as well as information obtained through internal documents, source code and external information [7] the conditional branch statements of the C programming language were identified as the implemented variation points that are used to change the systems control flow. The use of these was attributed largely to these types of control-flow statements being explicit in their representation of an alternative path of execution.

Conditional dependency for the conditional branch control flow statements is defined in the following way:

- If a variable is used in a conditional expression, the consequent blocks that are dependent on this conditional expression have a conditional dependency on this variable. The conditional branch statement itself is also dependent on this variable.

In the C programming language the conditional branch statements are made up of the if and switch programming statements [43]. For these statements conditional dependency is defined as follows, this is depicted in fig. 3.4:
if statements:

- the if block depends on the condition in the if statement.
- the else if block depends on the if condition, any preceding else if conditions and the current else if condition.
- the else block statements depend on all preceding if/else conditions.

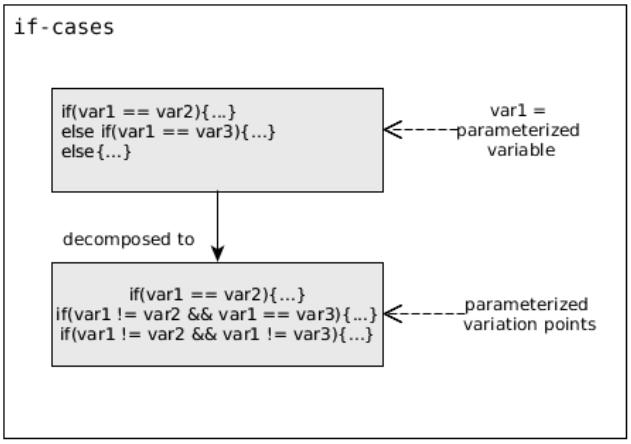


Figure 3.4: The decomposition of if-cases

switch statements:
Can be modeled as multiple if statements where the default expression is the else for all of the resulting if statements. Switch statements without breaks after a case are banned by multiple coding standards such as MISRA(15.2), thus this case is not considered.

With this in mind the following definition is made: The conditional branch statements of interest to extract are those that are conditionally dependent on a parameterized variable. The control flow of these statements will be configured during the parameterization process. The conditional branch statements are split into two categories:

- Fully bound: If the evaluation outcome of the conditional branch statement only has dependencies on parameterized variables or constants the statement is considered to be bound during the parameterization process. In this case the control flow of the program at this point can be fully derived. We call this type of variation point a *fully bound* variation point. Further all of the control flow statements that are bound prior to runtime define a vehicles static variant.
- Partially bound: If the conditional branch statement depends on the comparison of a parameterized variable with some variable that is not a parameterized variable or a constant the variation point is only deemed to be partially bound. The information gathered from this type of variation point is that the control flow of the execution has a dependence on the parameterized variable. We call this type of variability point a *partially bound* variation point. As this type of control flow statement is not bound prior to runtime it does not contribute to a vehicles static variant.

From the previous definition of variation points a categorization of the extracted information about the variation points was made:

- Binds - The variation point is fully bound. This type of variation point is used to create the vehicles static variant. The variation point can be fully modeled and the usage of the configuration data can be extracted for evaluation.
- Affects - The variation point is partially bound, thus the evaluation of the variation point can only deduce that the the parameterized variable has an affect on the program behavior. The usage of the configuration data can not be evaluated through static means of extraction.

From this evaluation it was defined that the software extraction tool is to extract the conditional branch statements that have a conditional dependency on a parameterized variable. These variation points are defined to be parameterized variation points.

As the parameterized variation points may not have a direct dependence on a parameterized variable the concept of data dependence between variables must also be considered. This is as the value contained in the parameterized variable can be assigned to a different variable before being used in a conditional branch statement. When this is the case the variation point has a data dependence to the parameterized variable, thus it is a parameterized variation point.

The following concepts about data dependence between variables within a program were used in this thesis [43]: In the case that a variable is defined all variables that read from that variable, through assignment, have a data dependence to that variable. As the C language execution is linear one can follow the data flow in a top down manner, this assumption is based on the goto statements that exist within C not being used. This assumption is made after considering the rules defined in the MIRSA-C (Point 14.4) programming standard [19].

Variables to the left of an assignment operator have a data-flow to them from every variable on the right hand side. In the context of C, this could be phrased like: The lvalue of an assignment expression is dependent on the rvalues of that expression.

As C allows pointer arithmetic this was also considered and evaluated. Pointer arithmetic within C-programs is difficult to analyze by static means. This is due to their global nature. This allows them to operate outside of language construct boundaries such as scope[1] [43].

This is handled by the extractor by the definition of the following rules: When a pointer is defined in a local function scope and is initialized through a call to one of the extraction patterns that are recognized by the extractor the data-flow can be handled in an analogous way to when the value is assigned to a local variable. This assumption is made because the parameterized variable that is pointed to retains its constant value throughout the systems operational lifetime. When the pointer in used in a local function scope the extractor can detect if it is re-assigned. If the reassignment is not to another parameterized variable it looses its association

[1]Scope is a region of the program where a defined variable can have its existence and beyond that the variable can not be accessed.

with a parameterized variable and is no longer of interest to the extractor. The pointer is only considered to be a data-bearer for the value of which it points to, therefore it only realizes this value when dereferenced.

If we follow this flow of data within a certain function scope we can trace the variables used in a conditional branch statement to their initial value assignment. If this initial assignment is to the value of a parameterized variable, the variable has a data-dependence on the parameterized variable. The conditional branch statement is then recognized as a parameterized variation point. With these definitions one can determine where the parameterized variation points within the implemented source code are. These can then be extracted.

External extraction tool

The external extraction tool shall use the development data sources that are considered to hold relevant information for the variability model. The sources that are used will thus be sources that hold information about the variability on the FPC level. The sources that are used should enable a FPC level configuration decision model to be constructed. The resulting model should enable the evaluation of how different FPC configurations impact the system.

Evaluating relevant data sources within Scania

The data sources that are identified in this step are to be used in order to derive information about the variability in the software product line.
The technical regulation for product data management within Scania [44] states " ...to fully parameterize a vehicle's ECUs, only the FPC codes which are present in the vehicle SOPS file are needed". The data sources that are used must therefore contain information about these FPC codes and their mapping to the external parameters. These parameters are mapped to the parameterized variables within an ECU.

Through interviews that were conducted with developers from different departments within Scania as well as the extraction of data from multiple sources the PSM database was identified to be the relevant source of information to used as input to the extractor. The PSM database is a XML document database containing information about the external variability of the systems within Scania, including FPC codes and external parameters.

Information matcher

The information matcher is used to relate the parameterized variation points from the source code to the information that is extracted from the external data sources. The matcher is aimed at creating realization links between the different abstraction levels. This is to enable the traceability and configuration decision modeling between the abstraction levels.

3.3 Tool chain for architecture recovery and modeling of variability

3.3.1 Technology used

The ECU source code is parsed with the open source parser srcML [45]. The XML files that are output from the srcML parser are used as the input to the source code extractor. The extractor uses the XML addressing language XPath to extract information about the systems variability [46]. The variability information that is extracted is stored in a Neo4J [47] graph database where it can be queried and visualized. By using a graph-database for persistent storage the variability model that is developed can be represented in an explicit way. This is as the structure and derived model of the variability in the different abstraction levels can be represented in a natural way. This way of representing the data is intuitive and serves the purpose of creating a comprehensible variability model. Using srcML in combination with XPath allows for the structured querying of the source code in order to extract the data needed to represent the variability. The usage of srcML allows for an AST tree to be used for fact extraction from the source code, as recognized in the technical report by SEI [22]. A short description of the technologies is made in appendix C.

3.3.2 Software extraction tool

Inputs

- The files that are output from the srcML parser are used as input to the extractor. These files contain the abstract syntax tree of the corresponding source code file in XML format.
- The extraction patterns that are used in as input to the tool are the identified processes of retrieving the parameterized variables within the source code modules that the application layer consists of.

Process

When extracting information about variation points within the implementation of the embedded systems a challenge is how to handle the variation in implementation style between the different ECUs.

The COO- and EMS- ECUs are developed within two different departments. The development guidelines and methods are very different between these departments. This leads to their system architectures and implementation style also being very different. Within the COO and the EMS the parameterized variables and their representation/usage in the implementation are handled in different ways. The solution to the problem of differing implementations was utilizing user-defined patterns of what the accessing methods of these variables are within the specific ECU implementation. These patterns are used as input to the extractor in order to create starting point for the extractor to traverse from.

- COO: Within the COO the parameterized variables are contained in the *Real Time Data Base* (RTDB) layer of the software. This layer serves as a common data storage layer for global variables within the COO. The usage of the parameterized variables within the COO follows a distinct pattern. In this pattern they are used in a function call to a function which extracts the boolean value of the variable. The extractor identifies the function call and the corresponding parameter which is the parameterized variable. The boolean values data flow is then traced to determine whether it is used in a control flow statement.

  Each parameterized variable in the COO is related to an unique ID, which is used in the parameterization process to identify the variable. The variables and their unique IDs are stored in a source code file that contains two C type arrays, as seen below.

```
 /* List of IDs for objects. Should be sorted in ascending
     order */
static const tU16 kwdb_id_comId_aU16[] =
{
   ((tU16) 0x0070), /* RTDB_IMMO_WORKSHOP_CODE_E          */
   ((tU16) 0x0071), /* RTDB_IMMO_ACCESS_DATE_E            */
}

/* List of definition for ID-objects. Sorted according to
    list of IDs */
static const tKWDB_OBJDEF_STR kwdb_od_comId_astr[]=
{
   {
      KWDB_DB_BAR_OBJECT_E,              /* objectType_E  */
      (tU32) RTDB_IMMO_WORKSHOP_CODE_E,  /* ix_object_U32 */
      SESS_BF_ALL_SESSION_STATES_U32,    /* bf_rdSession_U32
          */
      (tU32) 0,                          /* bf_orSession_U32
          */
      SESS_BF_UNLOCKED_IS_U32,           /* bf_prSession_U32
          */
      KWPC_NO_FINGERPRINT_E              /* fingerprint_E */
   },
   {
      KWDB_DB_BAR_OBJECT_E,              /* objectType_E  */
      (tU32) RTDB_IMMO_ACCESS_DATE_E,    /* ix_object_U32 */
      SESS_BF_ALL_SESSION_STATES_U32,    /* bf_rdSession_U32
          */
      (tU32) 0,                          /* bf_orSession_U32
          */
      SESS_BF_UNLOCKED_IS_U32,           /* bf_prSession_U32
          */
      KWPC_NO_FINGERPRINT_E              /* fingerprint_E */
```

```
    }
}
```

The first list defines the unique ID of each parameterized variable. This id is used in the parameterization process to link an external parameter with its corresponding internal parameterized variable. The second list contains a C-type struct that defines each parameters properties. The property which is of interest in the name of the parameterized variable that is related to the unique ID.

This style of implementation is very structured and ultimately leads to the code being more simple to extract information from. As the IDs that are used in the programming of the variables are explicitly defined in the source code the mapping between the parameterized variables and the external parameters that are used for parameterization can later be made. Thus from the extracted information from the COO the variation points that are dependent on a parameterized variable can be extracted and used to populate the variability model.

- EMS : The EMS ECU does not utilize the RTDB layer for storing the parameterized variables. Instead it has a simple implementation of a file system. This consists of C-type structs which are used as the containers for the variables. To retrieve a parameterized variable a utility function is used. This function obtains a leash that points to the base of the C-struct that contains the variables. This leash is then used to retrieve the desired parameterized variables in the struct, as shown below:

```
 void Fuel_50ms(void)
{
  /* Inputs: */
  const tFILE_EOLPARAM_STR* const e2EolData_pstr =
      File_rdEolParam_pstr();

  /* Locals: */
  tFILE_FUEL_TYPE_E           engineFuel;

  /* Get Eol fuel parameter */
  engineFuel = e2EolData_pstr->s_EngineFuel_E;
}
```

The function call that is used in order to retrieve the leash is used as a pattern to determine where a parameterized variable and its corresponding value is retrieved. In the EMS implementation of the parameterized variables the variables are not explicitly linked to unique IDs. The variables are linked to specific addresses within memory through defining what is known as pragma expressions in C. Pragma expressions give the compiler/linker special instructions for the compilation of the file in which they appear. These expressions within the EMS determine where the memory needed to store the parameterized variables is allocated. This information is thus con-

tained in the linker directives. These directives were not possible to retrieve and use for this theses. Therefore the names of the variables were used as IDs. The struct which contains the externally programmable variables can be seen below:

```
 /* PRQA S 639 QAC_BLOCK */
/*!@brief End of line configuration parameters stored in E2 */
typedef struct
{
    tFILE_COOL_LEVEL_E s_coolLevelEnable_E;      //!< Coolant
        level sensor.
    tFILE_CYL_BALANCING_E s_cylinderBalancingCtrl_E; //!<
        Cylinder balancing control.
    tFILE_FAN_TYPE_E   s_fanControl_E;           //!< Fan
        control 0=Mecanical 1=Electric Behr 2=Elect BorgWarner
        3=Hydr
    tFILE_GEARBOX_TYPE_E s_gearboxType_E;        //!< Type of
        gearbox. OPTICRUISEWITHOUTCLS_E=0
    tFILE_VEHICLE_TYPE_E s_vehicleType_E;        //!< Type of
        vehicle.
    tFILE_DISPLAY_TYPE_E s_ccDisplayType_E;      //!< Show the
        cruise control reference speed in the ICL display
    tFILE_DISPLAY_TYPE_E s_EngineFuel_E;         //!<The fuel
        type which is to be used in this vehicle
}
```

From the defined patterns of how to identify a parameterized variable within the ECUs the extractor follows the data flow of the parameterized variables value and extracts the variation points which depend on this value. The whole parameterized variation point is extracted, where all of the variables or constants that are evaluated in the conditional expression and their corresponding evaluators are modeled.

The concept of identifying a parameterized variation points can be seen below for a simple case in the EMS:

```
   void Fuel_50ms(void)
{
    #define FILE_FUEL_FAME_E 0;

    /* Inputs: */
    const tFILE_EOLPARAM_STR* const e2EolData_pstr =
        File_rdEolParam_pstr(); //Get the leash.

    /* Locals: */
    tFILE_FUEL_TYPE_E           engineFuel; //Local variable.

    /* Get Eol fuel parameter */
    engineFuel = e2EolData_pstr->s_EngineFuel_E; //Assignment of
```

```
        parameterized variable value to a local variable.

    /* If COP mode, use diesel values even if Fame is selected in
        Eol */
    if ( (engineFuel == FILE_FUEL_FAME_E) ) //parameterized
        variation point, dependant on the value of the
        parameterized variable.
    {
        engineFuel = FILE_FUEL_DIESEL_E;
    }
}
```

Here it can be seen that the local variable engineFuel has a data dependence on the parameterized variable s_EngineFuel_E. The value of the engineFuel variable is then used in a variation point where it is compared against the constant value of the FILE_FUEL_FAME_E. As all of the variables used in this parameterized variation point are constant. The variation point is thus determined to be fully bound.

Output

The extraction tool creates a clean abstract representation of the parameterized variation points. The information obtained through the architecture recovery from source code results in a representation of the variation points that are dependent on one or more parameterized variables. The variation point and its conditional constraints is also parsed in order to deduce whether the variation point is fully bound or partially bound. The different levels of information that is extracted can be seen in fig. 3.5.

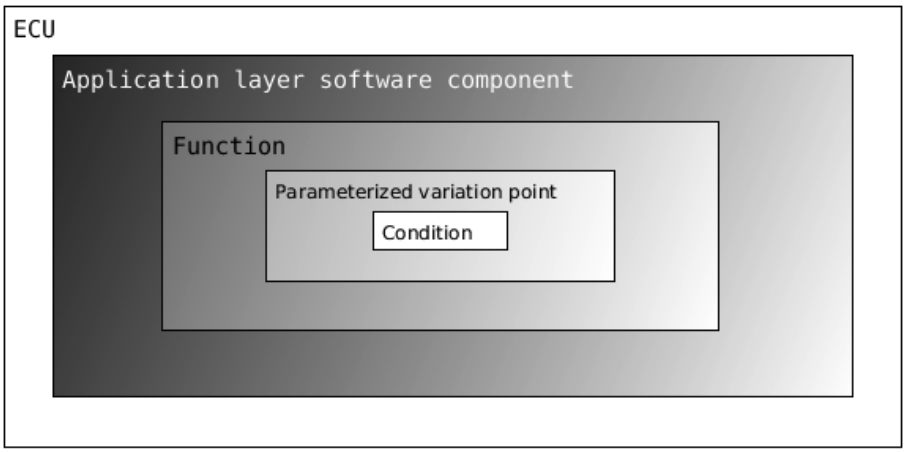


Figure 3.5: The different levels of extracted information from the source code.

From the information obtained by the architecture recovery performed in this stage a model of the parameterized variation points in the implementation was derived. The meta model is shown in fig. 3.6.

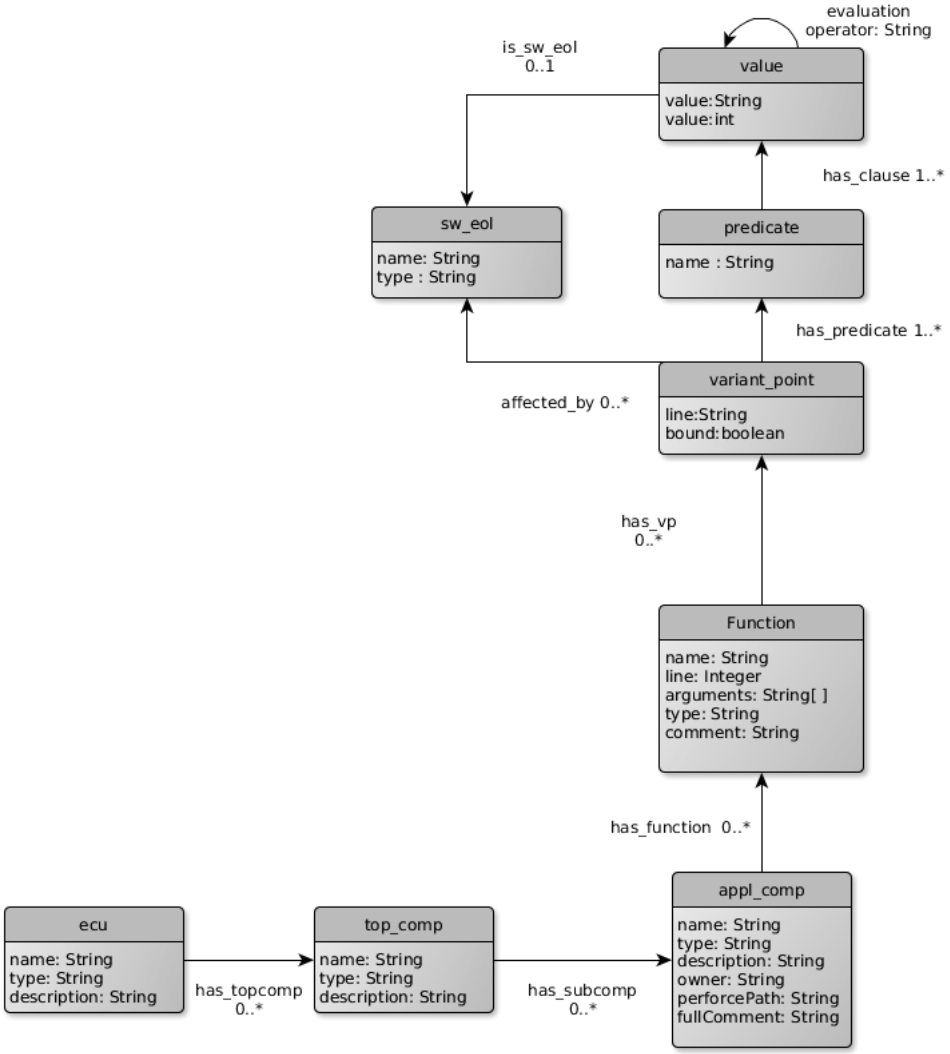


Figure 3.6: The meta model of the implementation level variation points.

The different nodes of the meta model are:

- sw_eol: This node represents the parameterized variable within the source code.
- variant_point: This node represents the parameterized variation point in the software implementation.
- predicate: this node represents a complete predicate that is used to determine the behavior of a variation point. The has_clause relationship repre-

sents the individual clauses of the predicate. If all of the clauses are evaluate to true the predicate evaluates to true and thus the variability point evaluates to true. In C-speak the has_predicate relationship represents the || relationship(s) in a conditional statement and the has_predicate relationship(s) represent the &&. An example of this can be seen in fig. 3.7 where the conditional in the variant_point node is modeled.

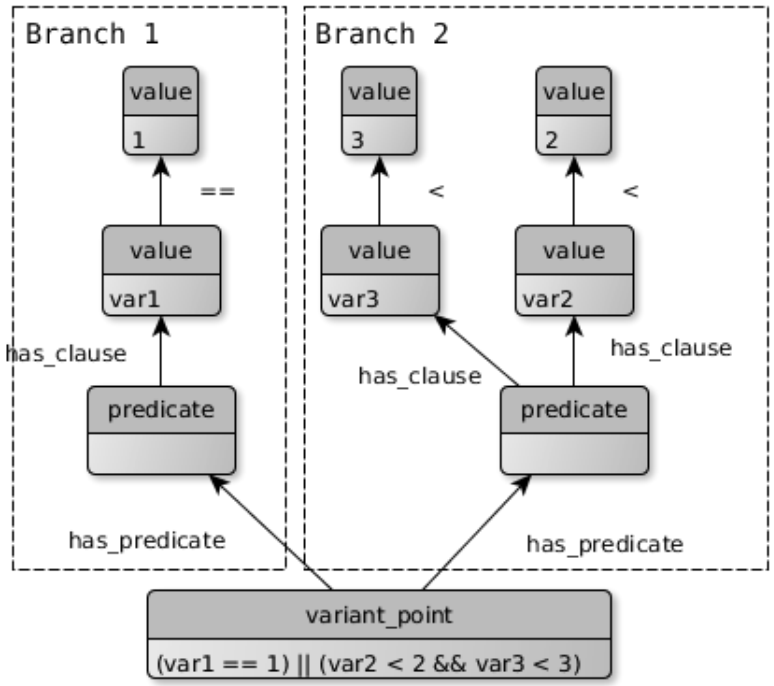


Figure 3.7: An example of the modeled variation points.

- value: this node contains a value, it can be a parameterized value, which then has the relationship is_sw_eol to a sw_eol node. It can also be a constant value or a placeholder for a dynamic variable. The operator relationship contains the boolean operator used to evaluate the conditions between two values.

The resulting populated model is shown in fig. 3.8.

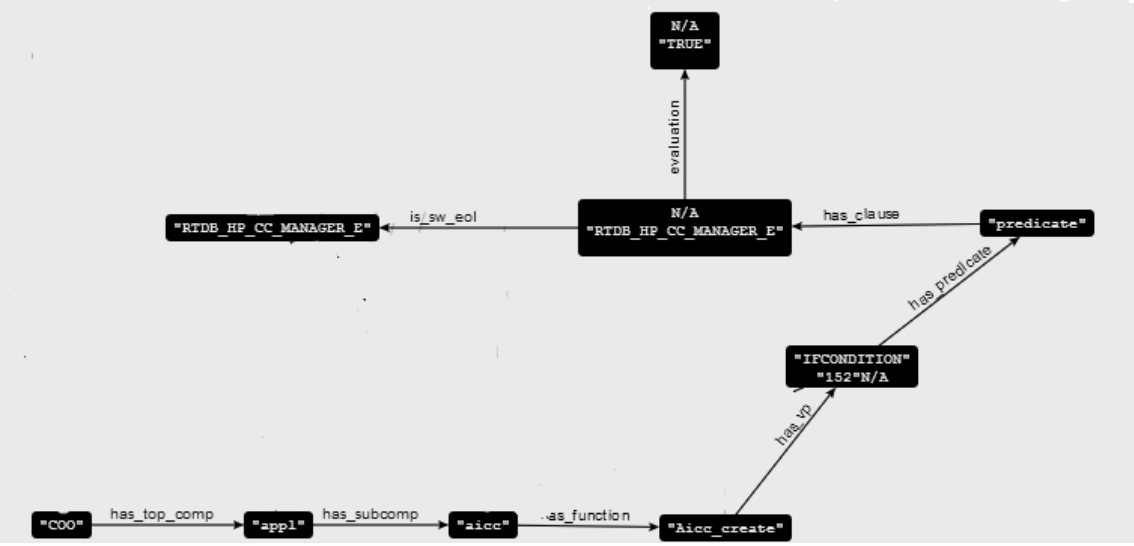


Figure 3.8: The internal variability that is extracted.

In fig. 3.8 one can see that the whole variation point and its location in the system architecture is modeled. Thus the location of the variation point is captured by the model. The conditional evaluation is also modeled where the variation points parameterized variable(s) is linked to a general representation of the parameterized variable node through the is_SW_eol relationship. This enables the architectural overview of all variation points that are dependent on the parameterized variable as seen in fig 3.9. The evaluation relationship has an attribute that is the relational operator e.g. ==,<=. The value nodes that have a constant value, such as the node denoted as "True" in fig. 3.8 also have their related constant value e.g. 1 as an property in the node. The representation of the conditional in the variation point is done to enable an analysis of the way that the parameterized variable (configuration data) is used in the implementation. For example if a conditional is dependent on the evaluation of "par_var == 4" where the variable par_var is a parameterized variable the validity of the evaluation can be analyzed by looking at all possible values that par_var can obtain through parameterization. If par_var can only ever be set to the value 1 or 0 the expressions that have a conditional dependency on par_var are in fact dead code.

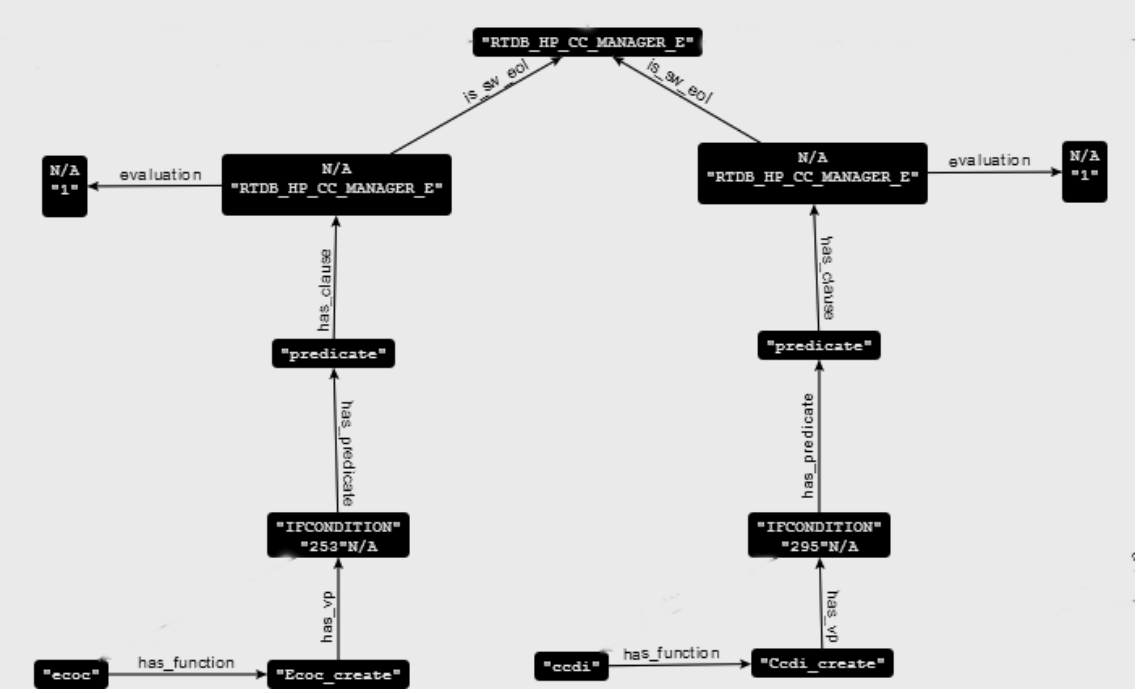


Figure 3.9: Multiple software modules that are dependent on the same parameterized variable.

3.3.3 External extractor

Inputs

The extractor used the PSM database as its input.

Process

The information contained in the PSM database was extracted. Through the development of algorithms to analyze the data a configuration decision model for each external parameter was constructed. The configuration decision model can be used to derive the configuration of the external parameters from the configuration of FPCs. The logic that is required to build the configuration decision model turned out to be very complex. Through generating the algorithms necessary to build a configuration decision model it became evident that variability within the software product line is handled in a implicit way within Scania. The external parameters that are mapped to the parameterized variables had a one-to-one mapping with vehicle features. However the FPC codes that are used to specify the configuration of the parameters are not allocated to only one level of abstraction. This conclusion was drawn after observing that there was sometimes

a one-to-one mapping between an FPC and external parameter, however there could also be a n-to-one mapping between FPCs and a external parameter.

There could also be multiple configuration decision models that were used to configure an external parameter. This result of the representation and configuration of variability made it clear that Scania does not use an explicit model for representing a feature configuration, this is instead done in an implicit fashion that relies heavily on the domain expertise of each developer. The way that configuration decisions are handled within Scania is a result of the implicit way that variability is handled. This is also what makes it hard to deduce how FPC configurations affect the parameterized variables. An example of how the combination of multiple FPCs affect an individual feature is exemplified in fig. 3.10.

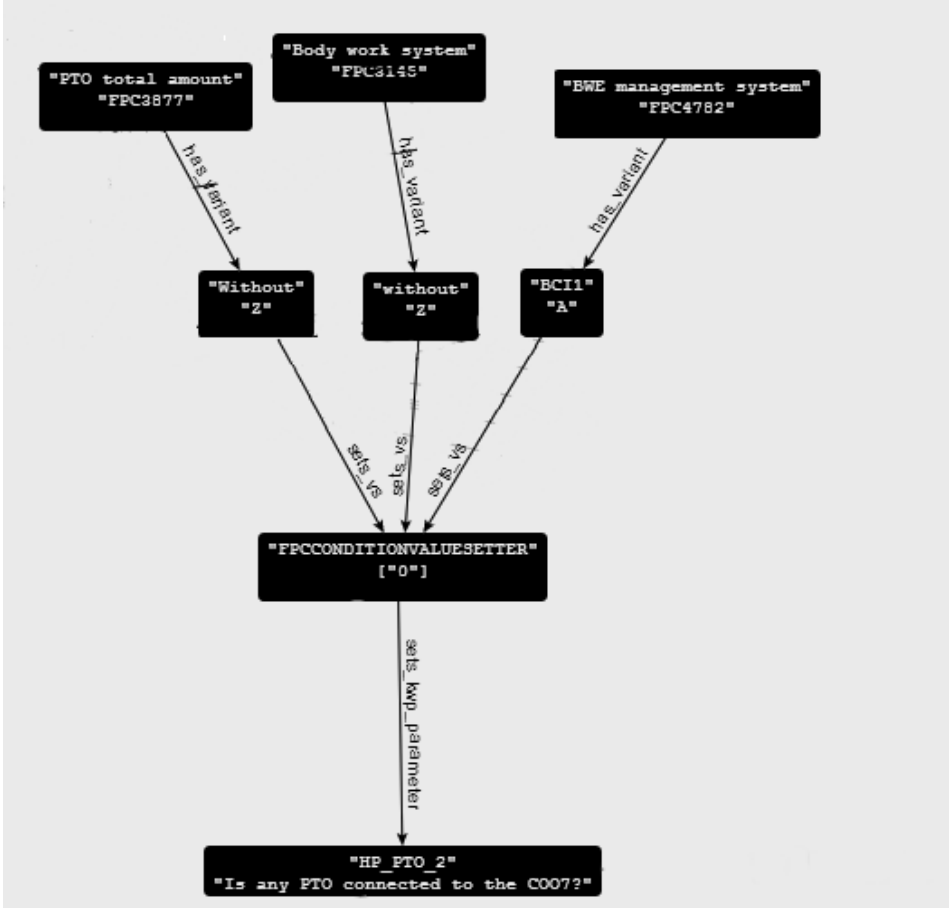


Figure 3.10: Multiple FPCs defining the configuration of one vehicle feature.

Fig. 3.10 shows that in the COO there is a feature of a "power take out", represented by the node "HP_PTO_2", that can be present or not. This is configured

by the customer. In the configuration decision model for the related external parameter its existence is specified by three different FPC codes: 3077, 3145 and 4782. One describes the total amount of PTOs, one describes the body work system and one describes the bodywork communication interface. The configuration of the PTO feature is dependent on the configuration of all three FPC codes. This is different to having an explicit FPC that describes the existence of a PTO, as would have been the case in a theoretical feature model, such as EAST-ADL.
From the information about the external parameters and their dependencies to FPCs the meta-model in fig. 3.11 was constructed.

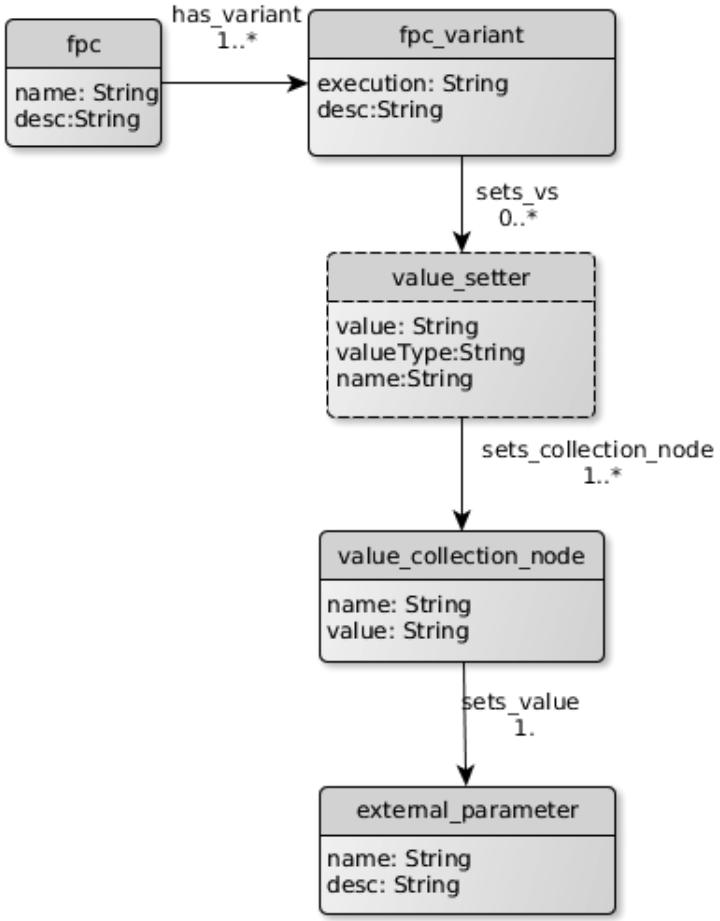


Figure 3.11: The meta-model for the external variability.

The different nodes of the meta-model and their relationships are:

- fpc: Contains the top-level FPC and a description of it. An FPC-execution

is created by following the has_variant relationship to the fpc_variant node.

- fpc_variant: This node contains an FPC-variant. This node represents the configuration of an FPC.
- value_setter: This node defines a configuration decision for the external_parameter node that is related to. The external_parameter that has a relationship with the value_setter obtains the value that is defined in the node only if all of the FPC-executions that are related to the value_setter node are selected in the current configuration. In fig. 3.10 the HP_PTO_2 external parameter will only receive the value 0 if the chosen configuration contains FPC3877Z, FPC3145Z, FPC4782A. Thus there is an "and" relationship between between FPC-executions that are related to the same value_setter node.
- value_collection_node: This node serves to collect all value_setter nodes that result in the parameterized variable obtaining the same value.
- external_parameter: This node represents the external parameter that is related to an internal parameterized variable. The parameter can be related to many value_setter nodes, where only one of them can be active at once. Thus there is an "or" relationship between the value_setter nodes related to one external_parameter as these are mutually exclusive. As the value of the external parameter is configured by FPCs this node represents the result of a configuration decision.

The resulting populated model can be seen in fig. 3.12.

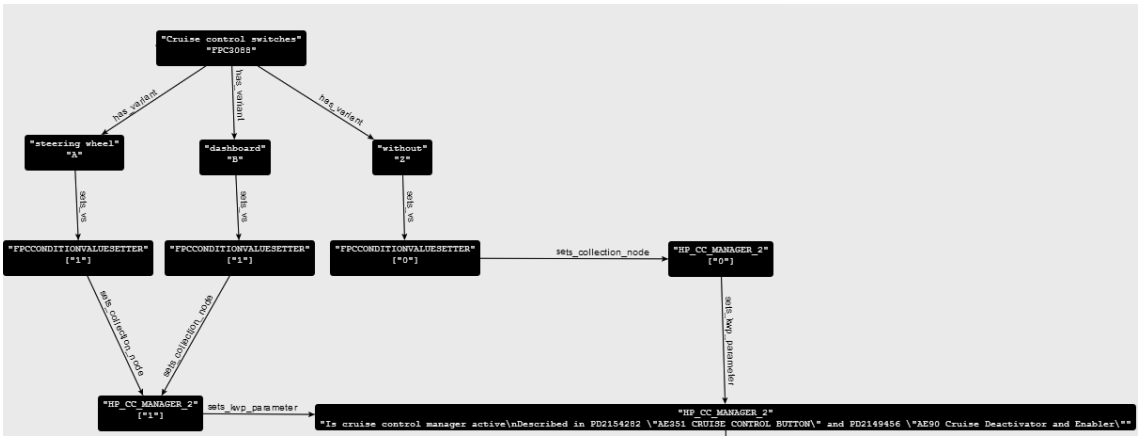


Figure 3.12: The external variability that is mined.

Here it can be seen that even though the FPC executions are mutually exclusive they can lead to a external parameter having the same value, eg. If FPC3088 is selected in the variants A or B the external parameter HP_CC_MANAGER_2 is set to 1.

Outputs

The output from the external extractor is the external parameters and their value-dependencies on FPC codes. The logic of how a parameter obtains a value from the configuration of the FPCs is also extracted. This information is enough to model the FPC level variability of the software systems and the configuration decision models that exist between the FPCs and external parameters.

3.3.4 Matching algorithm

The information about the internal variability points and the parameterized variables that they depend on needs to be combined with the information about the external variability that was extracted. This stage creates the realization links that are specified in the EAST-ADL variability model and that lead to traceability of variability through the different levels of abstraction.

Inputs

The inputs of the matching algorithm are the information models of the variation points and the FPC to external parameter dependencies.

Process

As the embedded systems within Scania are parameterized through the programming of static memory within the ECUs the programmer must write values to addresses within the memory. If the information about what addresses the external parameters are written to and what addresses the parameterized variables are linked to is available, the information can be fused by creating links using the addresses. This method of creating the links ensures the correctness of the mapping between the external parameters and their realization in the internal parameterized variables. If the information is missing, as is the case with the EMS ECU, a fuzzy string matching algorithm based on the Levenshtein distance algorithm [48] is used. This maps the external parameters name with the internal parameterized variable name. This matching through utilizing naming conventions is not as reliable as the mapping of addresses, and only works in the case that the external and internal parameters follow the same naming conventions.

Outputs

The output from the matcher is the recovered model of variability within the software product line. This model of variability enables the configuration decision modeling and traceability of FPC-codes to the parameterized variables. These variables are then used to determine where the FPC configurations effect the behavior of the ECUs. As the FPCs are a result of feature configuration this result proved very good. In fig. 3.13 the dashed line shows the separation of the externally and internally extracted information. The realized_as relationship is thus the equivalent of the realization links that were previously identified as being of importance for the variability traceability. This type of link maps the external

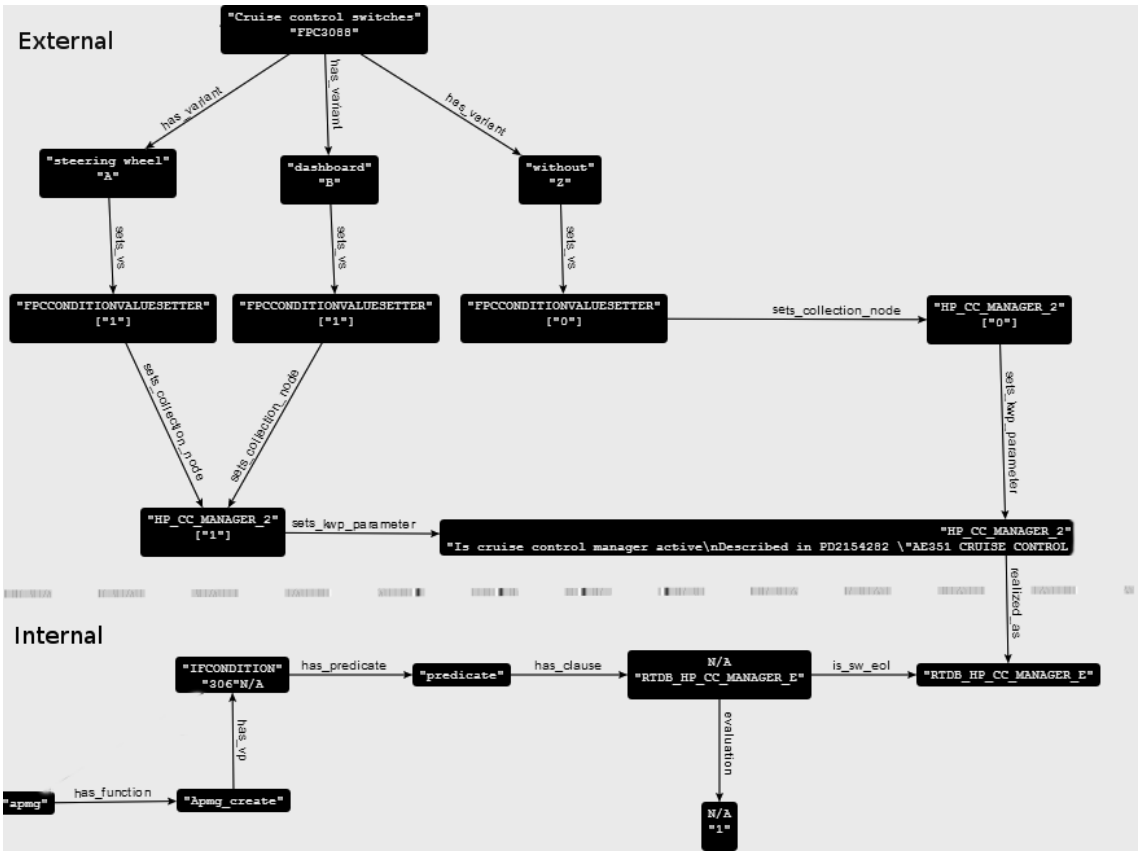


Figure 3.13: The combination of the internal and external extracted information.

Chapter 4
Results

This thesis resulted in a method of recovering and modeling variability within parameterized embedded systems at Scania. It defines the identification of relevant variation points and their subsequent modeling. The tool chain that was implemented provides the necessary steps needed in order to recover information about the variability on multiple abstraction levels within the software product line at Scania. The resultant model, which can be seen in appendix B.1, can be used in order to improve software product line development. This is done through creating an architectural representation of variability on different abstraction levels. The model represents where FPC configuration decisions impact the variation points of the ECUs. From Scanias point of view the result was very satisfactory and the populated model is already being used in different development projects.

The variation points in the parameterized ECUs were identified as being conditional branch statements that have a conditional dependence on one or more parameterized variables. Further the identification of fully and partially bound variation points was made. Through this identification a concrete meaning was given to the variation points within Scanias software product line. Each variation point is represented explicitly as a node in the model. This node has a relation to the statically set variable(s) that affects it and is also related to the function and subsequently the software module where it resides within the source code. This representation is due to Scanias use of parameterization, where the variation points affect only a part of the software module where they are implemented. This representation enables the traceability of where a parameterized variable affects the ECU behavior through the parameterized variation points having a conditional dependency to the parameterized variable. The variation points conditional statement is also modeled. This representation is aimed at enabling the verification of how the configuration data is used in the variation points.

The abstraction levels that are needed in the model are:

- The level where a vehicle configuration is specified, this was identified as the FPC level within Scania.
- The level that represents the impact that the configuration decisions have on the system, this was identified as the the implementation level within Scania.

By representing these two levels in the model the configuration decisions that are

made on the FPC level can be traced to where they impact the ECUs behavior on the implementation level.

The concepts that were identified from EAST-ADL model are represented in the developed variability model for Scania in the following way:

- Abstraction levels:
  The model represents two levels of abstraction. The FPC level and the implementation level.
- Traceability:
  The traceability between abstraction levels is enabled in the model by the realized_as relationship between the external_parameter node and the sw_eol node. This traceability is what makes the architectural representation useful for the reasoning about the system on different levels of abstraction. Through enabling the traceability the model gives a good overview of the FPC related dependencies between the software modules. This information provides an understanding of the systems complexity due to the implementation of variability. To enable the traceability between the abstraction levels the need for a method to match the data that is recovered from the different levels of abstraction was identified.
- Configuration decision modeling:
  The model of variability represents the configuration decision models on the FPC level of abstraction. This representation shows how the configuration of a parameterized variable is dependent on the configuration of FPCs. By enabling this the correct usage of the configuration data in the variation points can be evaluated. The dependency between parameterized variables can also be analyzed through analyzing the FPC dependencies. For example two parameterized variables that are dependent on two different executions of the same FPC are mutually exclusive.

The model enables the representation of all three concepts that were identified as being important in the EAST-ADL model. As the EAST-ADL model and its use has been investigated through case studies in numerous automotive companies the resulting variability model at Scania is considered satisfactory.
The model succeeds in enabling the determination of where the configuration choices affect the systems variation points. This representation was identified as being important when hazard analysis of different system variants is to be conducted as specified in the ISO 26262 standard. Further the model assists developers in fulfilling the clauses regarding the verification of configuration data in the following way:

- The use of configuration values within their specified range:
  By modeling the variation points and their conditional evaluation in the database this verification can be made on all of the bound variation points. This limitation is due to the nature of static parsing of the source code, where dynamic variables that obtain a value during runtime can not be determined. In fig. 4.1 one can see the possible values that a parameterized variable can have. These are represented as the nodes “s_vsrType2_E” with values 1, 2 and 3. Though knowing the possible values that the parameterized variable

s_vsrType2_E can obtain through configuration the conditional clause can be analyzed. If for example the value of s_vsrType_E could only be set to 3, the statements with a conditional dependency on the variation point would have been dead code.

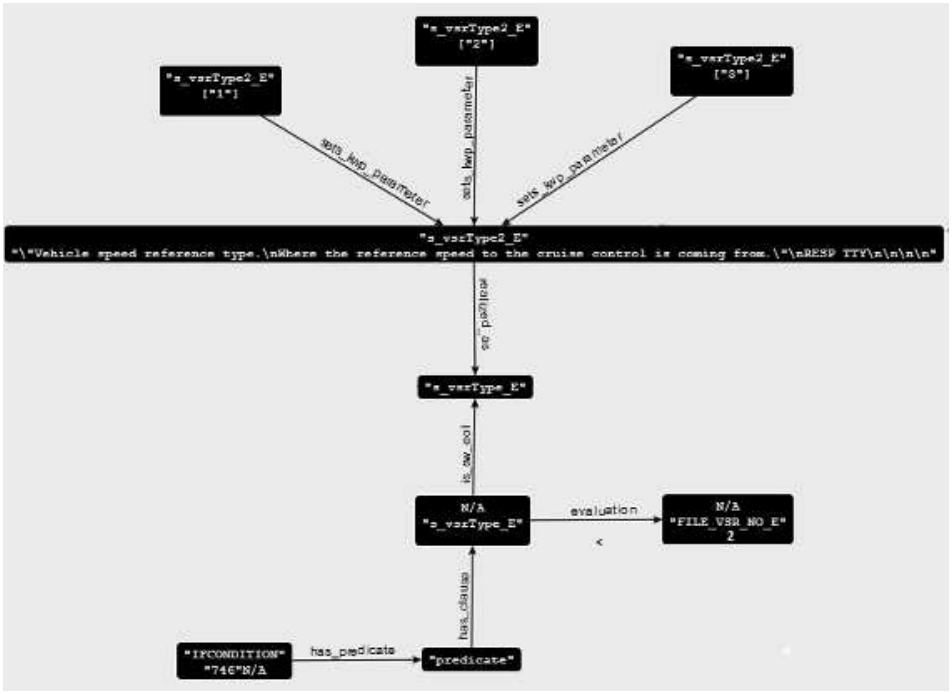


Figure 4.1: The verification of the correct usage of the configuration data

- The compatibility with other configuration data:
  Through the realization links between the FPC and implementation level one can evaluate the FPC-constraints between different parameterized variables and their value. This representation of information provides the developer with an overview of the constraints that can be used in the subsequent evaluation of the feature compatibility.

The population of the model was automated. The data that was needed to populate the model was data about the identified variation points in source code and the related FPC level information. This resulted in the tool chain using the source code of the ECU systems as well as a database containing information about FPC level variability. With the populated model the traceability of FPC configurations and the impact on the system can be analyzed as seen in fig 4.2. This figure shows where the configuration of FPC3088 has an affect on the system behavior. The configuration affects the variation point on line 306 in the function apmg_create in the software module apmg.

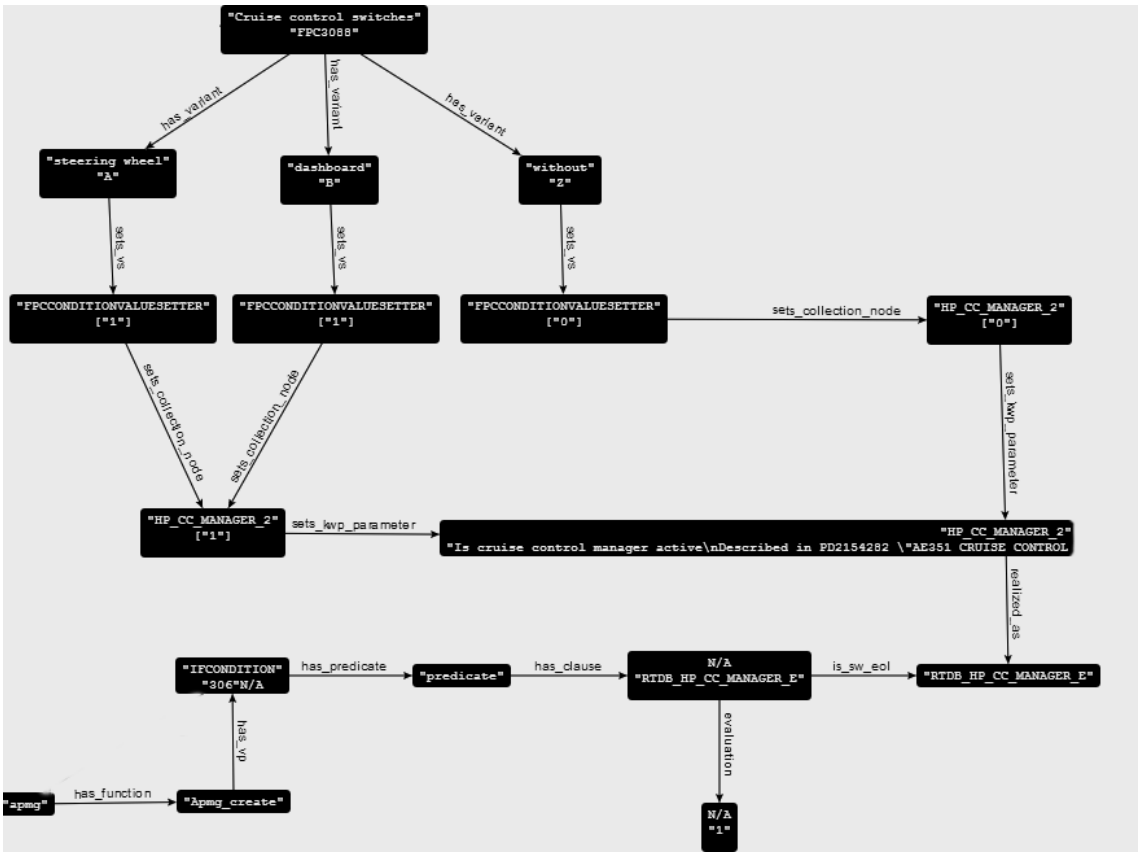


Figure 4.2: The traceability from FPC to internal variation point

The full configuration decision model of the HP_LDW parameter can be seen in fig 4.3. The configuration outcome of the model will affect the variation point on line 177 in the software module ldws. By analyzing the model one can determine that the only time where the variation point will be active is when the Scania lane departure warning and the Forward looking camera are both present, as seen in figure 4.4. Thus the lane departure warning feature requires both that the feature is active and also that there is a Forward looking camera present. The LDW feature has a “requires” dependency on the forward looking camera.

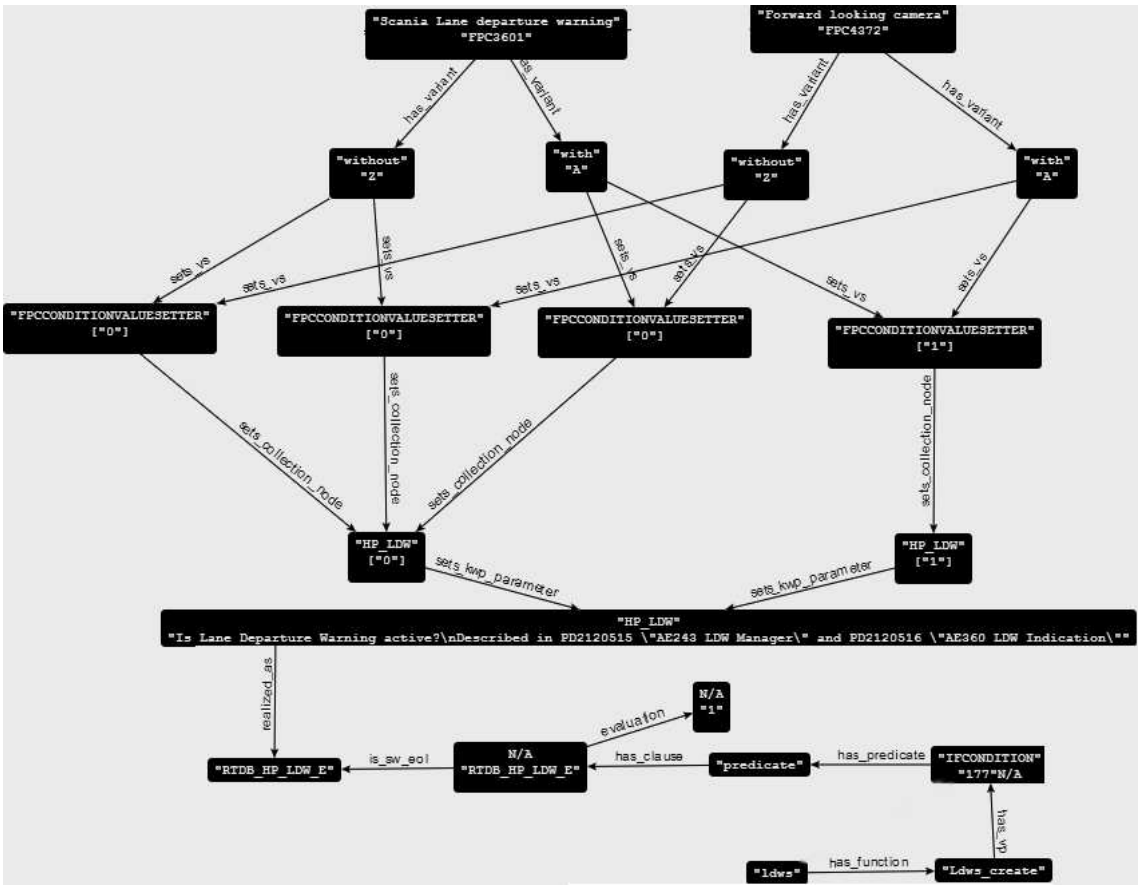


Figure 4.3: The configuration decision model for the "HP_LDW" feature.

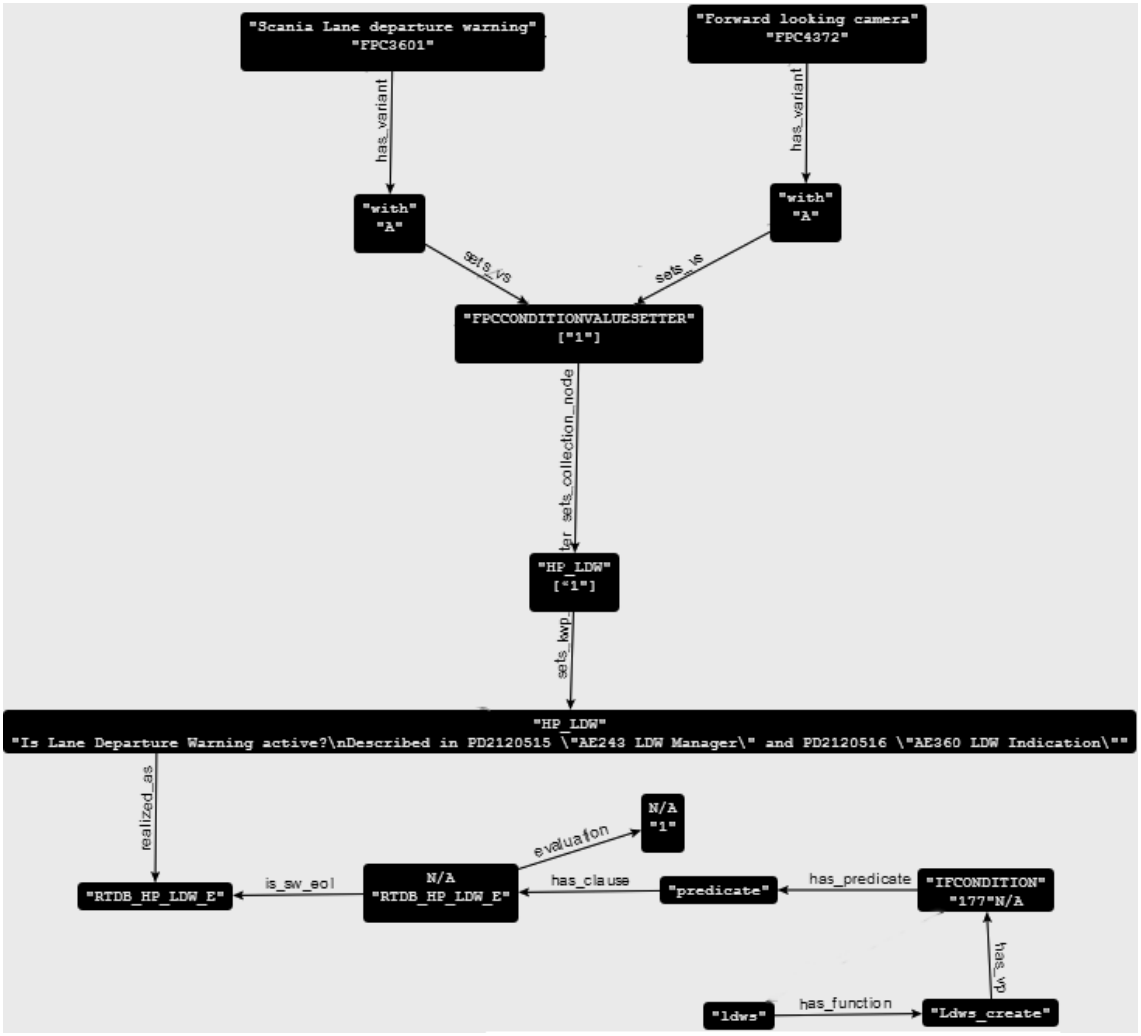


Figure 4.4: The result from analysis of the "HP_LDW" feature.

This type of tracing and analysis helps solve the problems that were identified in the report by Bosch [26] [7]. Through the architectural representation of the system variability that the model provides the knowledge gap that is discussed in the article by Bosch is minimized. This is done by creating an intuitive feeling for how the variability in the product line affects the embedded system ECUs. The usage of a graph-database enables the explicit representation of the model also when populated. As an example an overview of all FPCs that affect the behavior of the ECU can be seen in fig. 4.5.

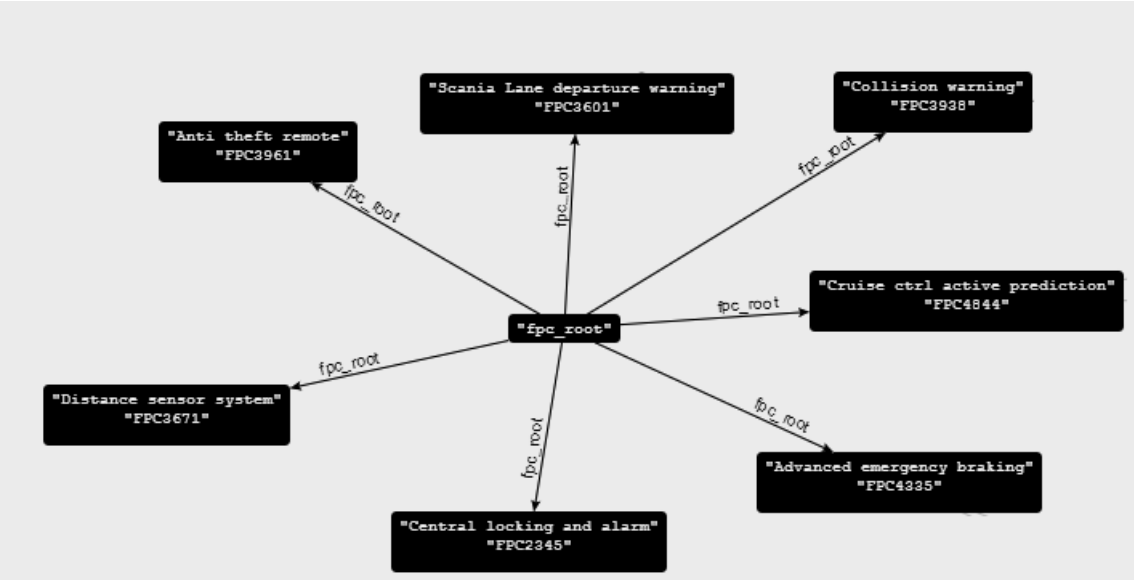


Figure 4.5: All of the FPCs that affect a certain ECU

Further the scattering and tangling of feature dependencies throughout different source code modules can be identified. This is seen in fig.4.6. Here one can see that the configuration of the parameterized variable RTDB_HP_CC_MANAGER_E affects the behavior of four different source code modules: apmg, ecoc, ccdi and aicc. Thus the analysis of the impact that the feature configuration has on the system can greatly be simplified.

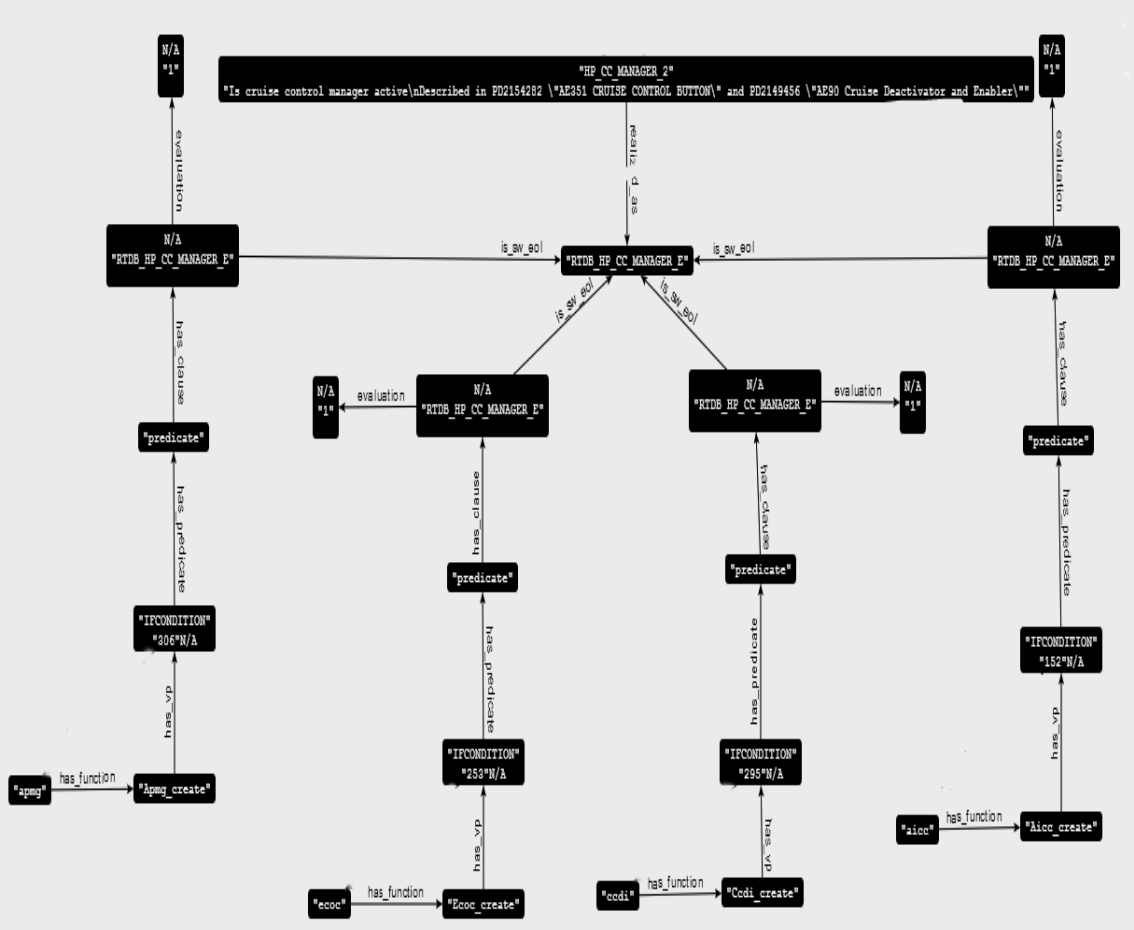


Figure 4.6: Database information about the scattering of a feature in different software modules

Chapter 5

Discussion and future recommendations

In order to retrieve a model that describes the software product line variability in Scania a qualitative study was made of how variability is handled in the development of the product line. The study was conducted by gathering relevant information and performing architecture recovery in order to confirm the validity of the information. The result of this study has been used in order to create a model of the variability in Scanias software product line. The model can be used to give developers and product stakeholders an architectural overview of the software and in how it is currently affected by the product line variability. The model can be used as a powerful tool to help developers and test engineers in their understanding and analysis of how variability is implemented and handled in Scania's development process.

5.1 Evaluation

5.1.1 Method evaluation

The method that was used to build a model of the variability relied heavily on an iterative approach at development. The information and data that was recovered during the architecture recovery process was used to build the model of variability in parallel to the recovery. This method produced a populated model that is used to solve actual problems within Scania. A drawback with using this method is that the model is very influenced by the actual data that was discovered during the process of architecture recovery. The model may thus have been more complete if it had been developed to encompass the theoretical usage of variability within Scania. However the usability of the model could have been affected if this was the case. This is the problem that has been identified with other theoretical models that need a large amount of structured information about variability to be useful in their representation. Research related to the architecture recovery of variability proved to be pretty much nonexistent to date. This led to the identification of the important recovery stages being hard. As a result of this - existent knowledge about and theoretical models of - variability was researched and evaluated during the initial development process. The process identified important concepts to include in Scania's variability model. These concepts along with the information obtained by analyzing the ISO26262 standard provided a concrete way

of evaluating the model in every iteration of development. The iterative method where architecture recovery and variability modeling were done in parallel proved necessary. This was due to the deep understanding of variability that is needed in order to construct a functioning model that can to be applied in real life. By constantly being able to update and evaluate the model after current knowledge the result proved to be satisfying, both in the subjective eyes of Scania and through the evaluation of the objective concepts that were defined in the initial development process.

5.1.2 Model evaluation

The model that was developed represents where FPC configurations affect the system software modules. It has succeeded in furthering the comprehension and potential analysis of where product variability affects the embedded system ECUs. The model is thus sufficient for the current basic needs that the model is aiming to address. As the process of configuring the FPCs from the customer configuration is not modeled there is a risk that the model will have to be extended in the future. Also the representation of the variation points and their conditionals is very basic. Further the analysis of the usage of configuration data that can be made based the represented information in the model is limited. As the model was developed based on the recovered data within Scania this was the largest limitation to its scope. Despite these limitations the model can be used as a powerful tool for supporting the verification of requirements in the ISO26262 standard, however it is not sufficient to fully automate this verification.

5.1.3 Tool chain evaluation

Scalability

The tool chain that is implemented is mostly limited by the usage of patterns to deduce the static variables. This approach is feasible where there are strict coding regulations of how to handle the parameterized variables, such as in Scania. However, in a case where the static variables are not retrieved with the usage of utility functions the extraction approach will have to be a lot more comprehensive.

Reliability

After the application of the tool chain on two complex ECUs and manual verification of the results, the reliability of the tool chain proved to be good. As the intended application area is to provide additional information to engineers and to create an architectural abstraction of the systems, the reliability is not as critical as that of a tool used for formal verification.

Validity

The validity of the tool chain has been manually investigated. Although unit tests on the tool chain have been implemented, due to the limited time of the thesis a full system test and validation was not done. The manual investigation of validity

was instead conducted for a number of cases and within the results there was no discrepancies between the information that was manually extracted and the information that was automatically extracted.

Risks

Since the source code being analyzed is manually written there are a lot of discrepancies in the style of implementation. Since the amount of source code files being analyzed is vast these problems have been overcome by the extraction tool utilizing targeted extraction. This targeted extraction reduces the overall amount of code being evaluated for extraction. This minimizes the risks with the extraction as the amount of analyzed code is dramatically decreased.

5.2 Recommendation to Scania

The extraction of this variability data from the source code enables the following conclusions to be drawn about the variability within the implementation level of Scanias ECUs: The separation of FPC dependent variability into different software modules is followed in the COO, however in the EMS there are many cases of variability tangling. This is where one source code module and its control flow is dependent on more than one parameterized variable. This way of implementation is contradictory to the AUTOSAR method where there is a n-1 mapping between SW-Cs and features. Also feature scattering was identified in both the COO and EMS implementations. This entails that a feature can create dependencies in many different software modules which leads to a more complex system architecture. The differing implementations and handling of variability throughout different development departments makes it hard to construct one model of the variability. Thus a recommendation is to define guidelines on how variability is to be handled and specified within the software product line in Scania. The use of FPCs to determine the existence of features is very implicit and the developers should be given integrated tool support when determining this. In the future a recommendation is to either replace the FPC system or create an explicit feature model of these. This model will make it easier for both developers and product stakeholders to use the FPCs in a consistent way over all phases of product development.

5.3 Future work

The future work within the field of variability in Scania can consist of investigation into a more explicit way of representing variability throughout the whole development chain. The usage of FPC codes needs to be investigated and guidelines on their usage in different scenarios need to be improved. To complete the architecture recovery chain and enable the representation of customer choices the process leading up to the the vehicle specification (V-spec) needs to be investigated and integrated into the existing model.

5.4 Conclusion

This thesis demonstrates the use of architecture recovery to model variability within large legacy product lines. It shows how the traceability between different levels of abstraction can be reconstructed and the expressiveness of a model that contains this information. Throughout this thesis it became more and more evident that variability within software product lines is a very complex and large topic where there is a lot of future work to be made. The increasing complexity of a system and its architecture as a result of introducing variability is a fact. This can be seen by looking at the retrieved model and the evaluation of its contents. Often however this is not recognized by developers or system architects. This lack of acknowledgement leads to a rapid increase in a systems complexity and thus a degradation of the overall system quality. Further the ability to perform system verification is greatly hampered by this. Even though the variability model that was the result of this thesis only contains a small subset of the total variability within Scania, the information is enough to have a direct impact on the future quality of Scania's products. Hopefully it can result in increased system safety. This result must surely be motivation enough to treat the future introduction of variability in systems as a first class concern of modern software development and system safety.

5.5 Example of model usage

The information that the model contains proved to be useful in many different applications. For example the data is currently used for examining what configurations affect what source code modules. In fig. 5.1 one can see the application components in the COO. All of the dashed nodes can be configured to be active or not depending on the vehicle configuration.

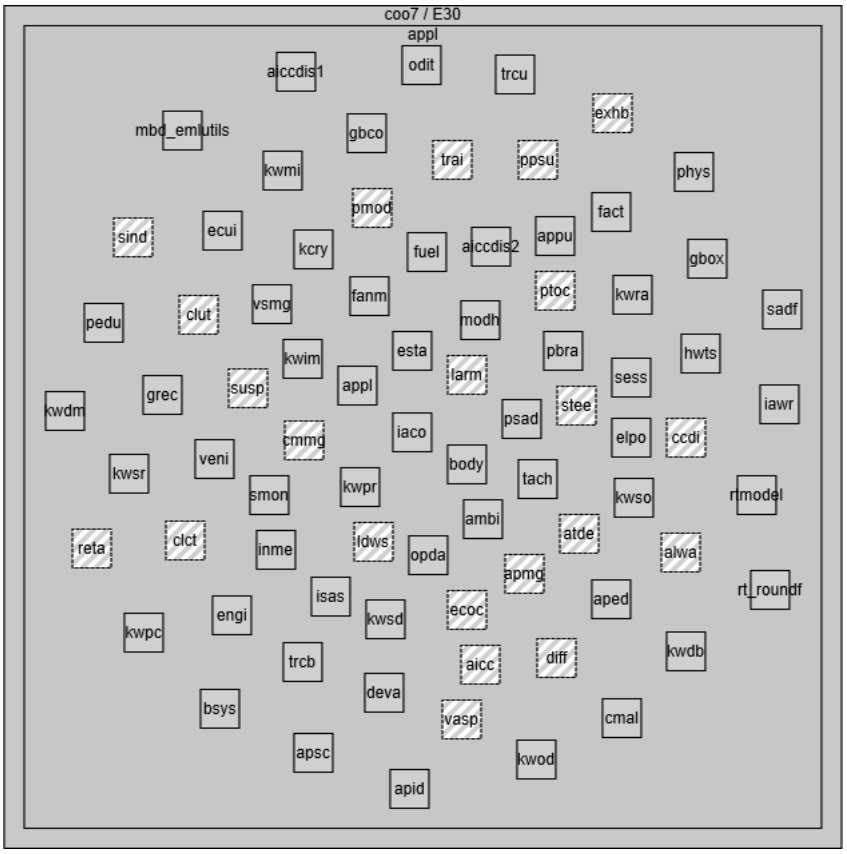


Figure 5.1: The variability in a unconfigured system

When a vehicle configuration file (SOPS) is specified the variability in the system can be reduced to the figure 5.2.

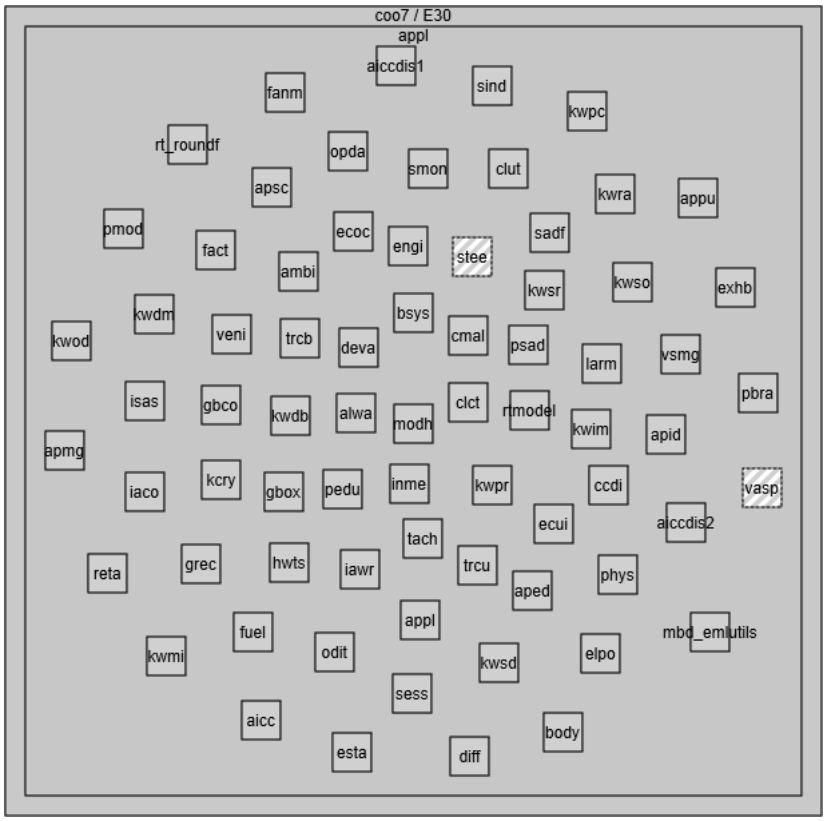


Figure 5.2: The verification of the correct usage of the configuration data

In fig. 5.2 the two dashed nodes are still variable. This means that either the configuration data that was used as input is incomplete or the actual use of configuration data within the software modules is wrong. By providing this type of sophisticated guidance and visualization support to the safety engineer the artifacts that are not relevant for the performance of safety verification can be identified. As many of the functional safety analysis activities that are defined in ISO26262 are hardly automatable this type of guidance is a key aspect of supporting tools and models.

Appendix A

Abbreviations

- AADL - Architecture Analysis and Design Language
- ADL - Architecture Description Language
- AST - Abstract Syntax Tree
- AUTOSAR - Automotive Open System Architecture
- C-spec - Chassi Specification
- COO - Coordinator ECU
- CTFM - Core Technical Feature Model
- EAST - Electronics Architecture Software Technology
- ECU - Electronic Control Units
- EFP - Exclusively FPC-regulated Parameters
- EMS - Engine Mangagement System ECU
- E/E - electronic/embedded
- FPC - Functional Product Characteristic
- PID - Parameter ID
- PFM - Product Feature Model
- SMOFS - Scania Manufacturing and Order Feature System
- SOPS - Scania Onboard Specification
- TCR - Translation Control Register
- VCR - Variant Control Register
- V-Spec - Vehicle Specification

Appendix B
Figures

The complete variability model B.1.

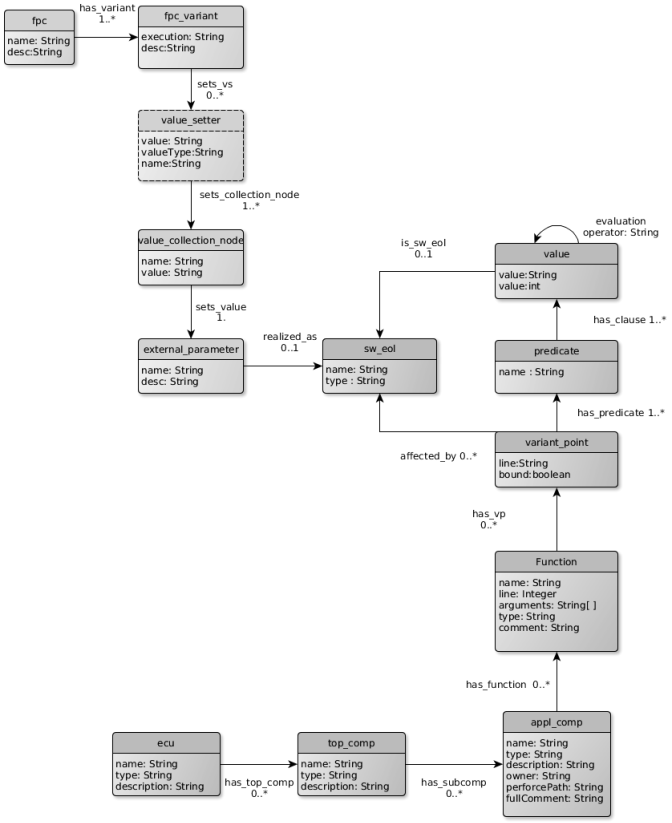


Figure B.1: The full meta model.

The architecture browsers representation of software modules when no configuration data is specified B.2.

Figure B.2: The variability in a unconfigured system

The architecture browsers representation of the software modules when configuration data is specifie B.3.

Figure B.3: The verification of the correct usage of the configuration data

Appendix C

Used technologies

- srcML: srcML is a markup language that displays a document oriented XML representation of the source code input. As the plain-text representation of source code is limited in addressing and providing context to locations within source code the srcML tool attempts to solve this by providing an enhanced XML view of the source code which allows access to all levels of information in the source code i.e. lexical, structural, semantic and documentary.[49] The source code is wrapped with information from the abstract syntax tree while retaining all of the original text in the input file. This approach enables querying access to the source code and its corresponding syntactic structure.
- XML: The Extensible markup language is a textual markup language that provides a document markup format that is both human and machine readable.[50] The language is documented and described in numerous publications and specifications by the World Wide Web Consortium (W3C). The overall design goals with the XML markup is simplicity, generality and usability.
- Neo4J: Neo4j is a graph-database which is ”embedded, disk-based, fully transactional Java persistence engine that stores data structured in graphs rather than in tables” [47]. The database provides fast querying and traversing of nodes and their corresponding relationships. Further it is opens source and easy to scale up in a horizontal fashion.

References

[1] Francoise Simonot-Lion. Special section on in-vehicle embedded systems. Technical report, 2009. IEEE Transactions on Industrial Informatics 5, 4.

[2] Scania in breif. http://www.scania.com/scania-group/scania-in-brief/. Accessed: 2014-05-23.

[3] Research and development structure at scania. Internal document. Accessed: 2014-03-23.

[4] Scania products. http://www.scania.com/products-services/trucks/main-components/. Accessed: 2014-03-26.

[5] Steffen Thiel and Andreas Hein. Modeling and using product line variability in automotive systems. Article, Robert Bosch Corporation, 2002. IEEE Software archive Volume 19 Issue 4, July 2002.

[6] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. Research report, Department of Mathematics and Computer Science, University of Groeningen, Department of Mathematics and Computer Science, University of Groeningen, PO Box 800 , 9700 AV Groningen, The Netherlands.

[7] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. Technical report, Software Practice and Experience,(35).

[8] ISO/FDIS. Road vehicles — functional safety. Technical report, ISO/FDIS, 2011. International standard 26262 (2011).

[9] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. Technical report, 1995. IEEE transactions on Software Engineering, vol.4, no.21.

[10] Ghulam Rasool and Nadim Asif. Software architecture recovery. Technical report, 2007. World Academy of Science, Engineering and Technology 4.

[11] Danilo Beuche Michael Schulze, Jan Mauersberger. Functional safety and variability: can it be brought together. Technical report, 2013. SPLC '13 Proceedings of the 17th International Software Product Line Conference.

[12] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. Technical report, Computer Science Department, University of Southern California, Los Angeles, CA 90089, USA. Wilfrid Laurier University, 75 University Avenue, West Waterloo, Ontario, ON, N2L3C5, CN, 2007.

[13] Angela Lozano. An overview of techniques for detecting software variability concepts in source code. Article, Université catholique de Louvain (UCL), Institute of Information and Communication Technologies, Electronics and Applied Mathematics (ICTEAM), Place Sainte Barbe 2, B-1348 Louvain La Neuve, Belgium, 2011. ER'11 Proceedings of the 30th international conference on Advances in conceptual modeling: recent developments and new directions.

[14] Cem Mengi. Model driven support for source code variability in automotive software engineering. Research report, Aachen University.

[15] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder Preikschat. Efficient extraction and analysis of preprocessor-based variability. Article, Freidrich-Alexander University, Department of Computer Science, 2010. GPCE '10 Proceedings of the ninth international conference on Generative programming and component engineering.

[16] ISO organization. Road vehicles — functional safety — part 1: Vocabulary. Technical report, ISO/FDIS, 2011. First edition 2011-11-15.

[17] Assessment of the iso 26262 standard, "road vehicles – functional safety". SAE 2012 Government/Industry Meeting January 25, 2012. Accessed: 2014-03-09.

[18] ISO Organization. Road vehicles — functional safety — part 6: Product development at the software level. Technical report, ISO/FDIS, 2011. First edition 2011-11-15.

[19] Mirsa-c:2004 coding standard. LDRA Ltd. Version 2.5.5, Accessed: 2014-02-17.

[20] Automated defect prevention for embedded software quality. http://alm.parasoft.com/embedded-software-vdc-report/. Accessed: 2014-04-06.

[21] Stéphane Ducasse and Damien Pollet. A process-oriented software architecture recovery taxonomy. Technical report, LISTIC - Language and Software Evolution Group, Université de Savoie, France, 2007. Accepted to CSMR 2007.

[22] Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture reconstruction guidelines, third edition. Technical report, Carnegie Mellon Software Engineering Institute, 2003. CMU/SEI-2002-TR-034 ESC-TR-2002-034.

[23] Günter Böckle Klaus Pohl and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, 1998.

[24] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. On the notation of variability in software product lines. Research report 02/01, Blekinge Institute of Technology, Department of Software engineering and Computer Science, BTH, 2001.

[25] Matthias Galster. Variability in software systems - a systematic literature review. Article, IEEE Transactions on Software Engineering.

[26] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Klaus Pohl, and J.Henk Obbink. Variability issues in software product lines. Article, Department of Mathematics and Computer Science, University of Groeningen, Department of Mathematics and Computer Science, University of Groeningen, PO Box 800 , 9700 AV Groningen, The Netherlands, 2001.

[27] Mikael Johansson. Product data information. Internal document about Product Data Information, 2005.

[28] Technical regulation, ecu development. Internal regulations for Scania Implementation, dec 2005.

[29] Technical product data, system description coo7. COO7 product data sheet., nov 2007.

[30] Pse user manual. Internal Scania Manual, dec 2009.

[31] Oscar Blomkvist. Technical regulation, regulatory system for ecu-parameters. Internal regulations for Scania Implementation, dec 2009.

[32] EAST-ADL Association. East-adl – domain model specification. Technical report, 2013. Version V.2.1.12.

[33] AADL. The sae architecture analysis and design language (aadl) standard. Technical report, SAE.

[34] East-adl introduction, east-adl overview. MAENAD project concept Presentations. Accessed: 2014-05-01.

[35] Hans Blom, Henrik Lönn, and Frank Hagl. East-adl – an architecture description language for automotive software-intensive systems white paper. Technical report. Whitepaper version M2.1.10.

[36] Autosar faq. http://www.autosar.org/index.php?p=1&up=6&uup=1&uuup=0. Accessed: 2014-04-08.

[37] Simon Fürst. Autosar – a worldwide standard is on the road. Technical report, BMW Group.

[38] Autosar technical overview. http://www.autosar.org/index.php?p=1&up=2&uup=0. Accessed: 2014-04-08.

[39] The c book. http://publications.gbdirect.co.uk/c_book/chapter3/flow_control.html/. Accessed: 2014-05-08.

[40] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly, 2013.

[41] East-adl introduction - east-adl variability. MAENAD project Concept Presentations. Accessed: 2014-04-06.

[42] Managing complexity of automotive electronics using the east-adl. Slides from the UML & AADL Workshop, July 14 2007. Accessed: 2014-05-01.

[43] Stephen Kochan. *Programming in C - A complete introduction to the C programming language*. Sams Publishing, 2004.

[44] Mikael Johansson. Product data information. Internal document about Product Data Information, page 36, 2005.

[45] Src2ml. http://www.sdml.info/projects/srcml/. Accessed: 2014-03-26.

[46] W3c recommendation xpath. http://www.w3.org/TR/xpath20/. Accessed: 2014-05-08.

[47] Neo4j information site. http://www.neo4j.org/learn/graphdatabase. Accessed: 2014-05-08.

[48] Levenshtein distance, national institute of standards and technology. http://xlinux.nist.gov/dads//HTML/Levenshtein.html. Accessed: 2014-04-26.

[49] Michael Collard. Addressing source code using srcml. Technical report, Department of Computer science, Kent state university., 2005. IWPC'05 Working session paper.

[50] W3c xml. http://www.w3.org/XML/. Accessed: 2014-05-09.

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2014-402

http://www.eit.lth.se