Master Thesis

In

Concept to store variant information gathered from different artifacts in an existing specification interchange format

For fulfillment of the academic degree
M.Sc. in Automotive Software Engineering

Submitted by: Samridhi Langer
Matr.-Nr. 391027
29 January, 1993
Delhi, India

Supervisors: Univ.- Prof. Dr. Wolfram Hardt
Dipl.-Ing. (B.A) Timo Vanoni

Date of submission: 31 August, 2016
Samridhi Langer

Concept to store variant information gathered from different artifacts in an existing specification interchange format

Master Thesis, Faculty of Computer Science, Department of Computer Engineering
Technische Universität Chemnitz, August 2016
Declaration

I hereby declare that this master thesis in topic “Concept to store variant information gathered from different artifacts in an existing specification interchange format”, is entirely the result of my own work and it has been written by me in its totality. Also, I certify that I elaborated this research independently. The work is based on the foundation of the information sources and literature used in the thesis that I have faithfully and properly cited.

Stuttgart, September 13, 2016

Place, Date

Samridhi Langer
Acknowledgement

The serenity that ensues the fulfilment of any work is inadequate without conveying the gratitude to all those people who helped you make it all possible. There are some special people without whose constant guidance and support, success may not have been achieved with same spirits.

I would like to express my deepest of regards to my professor Univ.-Prof. Dr. Wolfram Hardt and my advisor Dr. Ariane Heller. Their constant enlightenment on the subject and persistent motivation helped through each stage of my master thesis.

My profound thank you to the department of ProductLine Embedded Systems, Vector Informatik GmbH for giving me the opportunity of doing my master thesis in their teams which lead my way into enrapture developments. My manager Mr.Hartmut Hörner, Senior Manager Software Development, Vector Informatik GmbH whose constant expertise on the subject was very valuable for my work during the thesis.

Ardent thank you to my mentor, Mr.Timo Vanoni, Senior Software Development Engineer who helped me steer my way through the entire master thesis and help enact my thesis in a very productive manner. I would like to give earnest of regards to Mr.Markus Schwarz, Senior Software Development Engineer for his perseverance at each stage and for his useful opinions that helped me look at things with a wider perspective.

I must acknowledge my friends Aakash, Akhil, Charu, Janavi, Rajat and Shashank without whose editing assistance I would have not finished this thesis.

Lastly, I would like to thank my parents Mr.Rajesh Langer and Mrs.Ritu Langer for always supporting and encouraging me to do everything that felt unfathomable. I can’t thank them enough for everything they have done for me.

A genuine thank to everyone for helping me successfully complete my master thesis.

Place: Stuttgart
Date: September 13, 2016
Abstract

Any software development process deals with four main artifacts namely; requirement, design, implementation and test. Depending upon the functionality of a particular product there might be variants present in these artifacts. These variants influence all the artifacts involved in a software development process. Data in the higher level artifact affects the data present in the further artifacts and is also refined when we move towards the lower level of abstraction. This thesis deals with the handling of all the variant information present in all the artifacts. Verification and consistency checks on this information were to be automated for making the development process easier.

The results achieved during this thesis discuss the solutions for the problem of inconsistent variant information present in all the artifacts. By defining the extension of the intermediate format to support the variant information at Vector Informatik GmbH this problem has been resolved. The data used during the development is the variant information.

The generic intermediate format has been extended in a way so that it can further support a variety of use cases. Along with the formulation of a format, documentation of variant information and methods to extract variant information form C source code are also discussed.

**Keywords** - variants, software engineering, configuration variants, generic
# Contents

1 Introduction 1
   1.1 Problem Description and Purpose ............................................. 1  
   1.2 Thesis structure ................................................................. 4

2 State of the Art 5
   2.1 Intermediate format and variants ............................................ 5  
      2.1.1 VSF ................................................................. 5  
      2.1.2 Variants ............................................................. 7  
   2.2 Variant Handling in General ................................................... 8 
      2.2.1 SPES Framework ......................................................... 8  
      2.2.2 Integrated Feature Modelling ........................................... 10 
      2.2.3 Modelling Variability with UML ....................................... 11 
   2.3 Storage of Variant Information ............................................ 14 
      2.3.1 Requirement Interchange Format ....................................... 14  
      2.3.2 Variant handling in simulink ......................................... 15  
   2.4 Extraction of variant information from code ............................... 17 
   2.5 Tools and Technologies ........................................................... 18

3 Concept 22
   3.1 Fundamentals ........................................................................... 22 
      3.1.1 Abstraction levels .......................................................... 22  
      3.1.2 Variant information in different artifacts ............................ 24 
   3.2 Concept Overview ................................................................. 28 
      3.2.1 Concerns ........................................................................ 28 
      3.2.2 Use-cases ...................................................................... 29 
   3.3 Proposed Approaches ............................................................... 31 
      3.3.1 As global as possible ....................................................... 31 
      3.3.2 As local as possible ......................................................... 32 
   3.4 Satisfiability Checking ............................................................... 33 
   3.5 Detailed design ....................................................................... 37

4 Implementation 40 
   4.1 Extension of VSF ................................................................... 40 
   4.2 Handling of presence conditions .............................................. 42 
      4.2.1 PC from VSF to internal model ......................................... 43
# List of Figures

1.1 Variant - An Example .................................................. 1  
1.2 V-Model ................................................................. 2  
2.1 File model of VSF .................................................... 6  
2.2 Example of VSF file ................................................. 7  
2.3 Example of an ADAS variability model ......................... 9  
2.4 Integrated Feature Modelling ..................................... 11  
2.5 Feature Model using UML .......................................... 12  
2.6 Use of Variation Points and Optional Elements ............ 13  
2.7 Meta model of ReqIF Format ..................................... 15  
2.8 Example of the extension of ReqIF ............................. 16  
2.9 Approach Overview .................................................. 16  
2.10 Before and after partial processor ............................. 18  
2.11 Enterprise Architect ............................................... 19  
3.1 Structural Variability ............................................... 22  
3.2 Code example depicting structural variability ............... 23  
3.3 Functional Variability ............................................... 23  
3.4 Code example depicting functional variability ............... 24  
3.5 Example of a functional variant in requirement artifact .... 24  
3.6 Concrete example of how requirements are depicted ....... 25  
3.7 Example of a variant in design ................................... 25  
3.8 Sequence diagram having alternative and optional blocks 26  
3.9 State machine showing configuration variants ............... 27  
3.10 Example of a variant in implementation ..................... 27  
3.11 Specific code example depicting variability ................. 27  
3.12 Overview of new concept ......................................... 29  
3.13 Use Case 1,2 and 3 .................................................. 30  
3.14 PC in the same artifact ............................................ 30  
3.15 Keeping PC as global as possible .............................. 32  
3.16 Disadvantage of keeping PC as global as possible ....... 33  
3.17 Keeping PC as local as possible ............................... 34  
3.18 Meta model of the PC object .................................... 37  
3.19 File model of the VSF after adding PC object ............... 39  
4.1 Structure of a PC node ............................................... 40
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Example of a PC node</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of a constraint node</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Value node added to the parameter</td>
<td>42</td>
</tr>
<tr>
<td>4.5</td>
<td>Nodes in VSF</td>
<td>43</td>
</tr>
<tr>
<td>4.6</td>
<td>Expressions from VSF to internal model</td>
<td>44</td>
</tr>
<tr>
<td>4.7</td>
<td>Types of boolean expressions</td>
<td>44</td>
</tr>
<tr>
<td>4.8</td>
<td>Supported binary operators</td>
<td>45</td>
</tr>
<tr>
<td>4.9</td>
<td>Supported unary operators</td>
<td>45</td>
</tr>
<tr>
<td>4.10</td>
<td>Visitor implementation for internal model to Z3</td>
<td>47</td>
</tr>
<tr>
<td>4.11</td>
<td>Relation of one function to another</td>
<td>49</td>
</tr>
<tr>
<td>4.12</td>
<td>Requirement tracing to design artifact</td>
<td>50</td>
</tr>
<tr>
<td>4.13</td>
<td>Function with one return type and two parameter types</td>
<td>51</td>
</tr>
<tr>
<td>4.14</td>
<td>Function with two different parameter types</td>
<td>52</td>
</tr>
<tr>
<td>4.15</td>
<td>Result when context is fed with PC equals !D</td>
<td>53</td>
</tr>
<tr>
<td>4.16</td>
<td>Visitor for relation, property and object</td>
<td>54</td>
</tr>
<tr>
<td>5.1</td>
<td>Relation under a condition and a function with relation to two other functions</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>Two functions tracing to the same requirement example</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Examples depicting occurrence of Tautology and Contradiction</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Parameter types in different configurations of D and E</td>
<td>54</td>
</tr>
</tbody>
</table>
Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAS</td>
<td>Advance Driver Assistance System</td>
</tr>
<tr>
<td>API</td>
<td>Application Interface</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
</tr>
<tr>
<td>BDD</td>
<td>Boolean Satisfiability Problem</td>
</tr>
<tr>
<td>CAD</td>
<td>Component Abstract Design</td>
</tr>
<tr>
<td>CDD</td>
<td>Component Detailed Design</td>
</tr>
<tr>
<td>CDK</td>
<td>Component Development Kit</td>
</tr>
<tr>
<td>CIA</td>
<td>Change Impact Analysis</td>
</tr>
<tr>
<td>CLI</td>
<td>Common Language Infrastructure</td>
</tr>
<tr>
<td>CREQ</td>
<td>Component Requirement</td>
</tr>
<tr>
<td>DPLL</td>
<td>Davis-Putnam-Logemann-Loveland</td>
</tr>
<tr>
<td>EA</td>
<td>Enterprise Architect</td>
</tr>
<tr>
<td>HIS</td>
<td>Herstellerinitiative Software</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>MAX-SAT</td>
<td>Maximum Satisfiability</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PC</td>
<td>Presence Condition</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development</td>
</tr>
<tr>
<td>ReqIF</td>
<td>Requirement Interchange Format</td>
</tr>
<tr>
<td>SAT</td>
<td>Satisfiability</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability Modulo Theories</td>
</tr>
<tr>
<td>SPES</td>
<td>Software Platform Embedded Systems</td>
</tr>
<tr>
<td>SPL</td>
<td>Software Product Lines</td>
</tr>
</tbody>
</table>
UML  Unified Modelling Language
VSF  Vector Specification Format
V-Model  Verification and Validation model
XML  Extensible Markup Language
# 1 Introduction

A simple scenario that describes a variant can be; a car which is a common base, with two stakeholders having different requirements in terms of doors of the car. To depict the basic idea of what a variant can be, we consider a situation where the first stakeholder needs one door in his car and the second stakeholder needs three doors in his car, all the other functionalities in the car except the doors can be reused. See Figure 1.1.

A company has several products and a particular product may have more than one stakeholders. Depending upon the different requirements of stakeholder’s variants of a particular product are built.

![Figure 1.1: Variant - An Example](image)

Main focus, while adapting variants should be on reusing the parts of the old variant and only adding the parts that fulfil the new requirements of the stakeholders. So, only the parts that are changed in the variant must be the focus rather than focussing on the product as a whole.

## 1.1 Problem Description and Purpose

### V-Model

There are various methodologies for software engineering. A software process model is an abstract representation of a process with four different perspectives such as specification, design, validation and evolution. Some of the general software process models are:

- Prototype Model
- Spiral Model
1 Introduction

1.1 Problem Description and Purpose

- Iterative Model
- Rapid Application Development (RAD) Model

The process model that is used for the development at Vector Informatik GmbH is the Verification and Validation model (V-Model). To understand where all the variant information could be present we need to first understand the V-Model. Figure 1.2 depicts the V-Model. V-Model is a graphical representation of the development life cycle of a system which summarises the main steps to be taken during the development of a system. It is a software development life cycle methodology that describes activities which are to be performed for the development of a product. Four different artifacts are focussed in the diagram namely; requirements, abstract/detailed design, implementation and test. See Figure 1.2.

First step in developing a product is to analyze the product requirements clearly and have a clear idea of what we want from the system that is being developed. Along with the requirement specifications the acceptance test plan is also created. The abstract design focuses on the overall system architecture and design. It provides an overview of the overall system, platform and solution. Test cases with respect to the system tests are created here. Then comes the detailed design where the actual software components are designed which defines the detailed logic of each and every component in the system and the unit/integration test cases that are developed in this phase. The last development stage is implementation where all the software development
takes place. Once all the software development is done, the right side of the V-Model comes into action. Here, all the test cases that were written during development are executed. This method is not only simple and easy to use but also saves a lot of time as all the test cases are made during the development stage itself. It also helps in early defect tracking. The model is rigid so it can be broken down into 4 different artifacts and is easy to manage, for example, if there are some defects or problems detected in the abstract design stage the changes can be made at that level itself. It works well for medium sized projects where the requirements are clearly defined and understood.

The data that is exchanged from one level to another is present in different formats and the usage of that information is also different. The information that is present in each stage is discussed at length in the upcoming chapters. This thesis focuses mainly on the variant information present in each artifact.

**Problem Description**

In order to support the variety of use cases, software products implement different variants of functionalities which are selected based on a configuration. These variants influence any of the artifacts involved in a software development process, such as specifications, implementations or test. All the artifacts have variant information present in them. Specification of the previous activity in each artifact listed on the left side of the V-Model is refined as we move downwards. Moving from the requirement artifact to the design or implementation artifact there is a refinement of the variant information i.e. the variant information becomes more detailed and elaborated when you move from one artifact to the other. Also, if there is some variant information present in the requirement artifact it should also be present in the architecture, design or implementation artifact i.e. the variant information present should be consistent everywhere. For these types of cases the input of one artifact can be verified against the other.

**Purpose**

Analyzing where all is the variant information located now we have to see how it is currently being handled. Presently, only informal documentation of the variant information is available. Component developers and the architects also need the variant information as a reference to understand the product as a whole. This is important because if there is some change at the component level they need to understand what is been affected by that i.e. there should be a possibility for impact analysis. It can also be done with the informal documentation of the variants that we currently have but right now we can do it manually which is rather difficult. If there is a formal definition for the variants this process can be automated. A formalised documentation of the variant information must be made so that different operations on the variant information can be carried out. Some of the operations are listed below.

- A possibility for verification checks on the variant information can be carried out. An example of a verification check can be done to verify whether the conditions responsible for the presence of a particular variant within the same artifact are consistent or not.
• Variant information located in the earlier artifacts should be same but more detailed in the later artifacts. If this is possible, we can say that the variant information across the different artifacts is consistent. So there should be a possibility for this as well.

• Static code analysis considers only one variant for now but in future, should consider multiple variants.

1.2 Thesis structure

The structure of this Master Thesis is formulated in the following chapters:

• **Chapter 2** describes the literature research on variants, how they are handled on different levels and what are the problems that arise while doing that, how is the storage of variant information done including the interchange formats and lastly it discusses about all the variant information that is found on the code level. Under tools and technologies various methodologies and platforms that could be used for the thesis are discussed.

• **Chapter 3** lays the base of the Master thesis. This chapter describes the state of the art i.e. the existing state of the problem and how is it handled currently. All the possible solutions to the problems are also discussed. Finally, one of the solutions of how the format extension should be done is chosen.

• **Chapter 4** defines how the concept that is finalised has to be implemented. This involves the explanation of tooling that was implemented to achieve the desired goal. It also explains the strategies that were implemented to attain the final objective of the thesis. Usage of the tooling is also discussed in detail.

• **Chapter 5** outlays the verification of results of the implementation. Some use cases are described in order to make sure that the tool is working fine. It also helps to know some limitations if they are present.

• **Chapter 6** presents the conclusion to this work. Also, it tells about how the tooling could be used in the future.
2 State of the Art

This chapter thoroughly explains the following topics as part of the state of the art and literature research:

- Intermediate format and variants
- Variant Handling in General
- Storage of Variant Information
- Variant Information in code

These topics are the prerequisites for understanding the variants in general and how are they handled by the approaches that currently exist. It gives a clear idea of all the topics listed above. The first section is the current state of the art at Vector Informatik GmbH and the following sections are the literature research that has been done with respect to this thesis.

2.1 Intermediate format and variants

2.1.1 VSF

**VSF** is a generic format used at Vector Informatik GmbH to store all kinds of specifications [1]. It is called a generic format because it can not only be used in the automotive industry but all the other industries as well. It is important to understand because the extension of the VSF format is supported in this thesis. So in order to extend **VSF** one needs to understand the format first. Figure 2.1 shows the model of VSF Format.

An example of how a **VSF** file looks like is shown in Figure 2.2. Some important properties of this format are:

- It is hierarchical in nature
- It is a generic file format because it can be used in any type of industry apart from Auto-
- motive
- It is easily extendible, as the object node in **VSF** can be extended easily. All the artifacts like parameters, relations are directly placed under the object node which makes it easier to edit
• All the information with respect to a single object is directly available along with extra information

Figure 2.1: File model of VSF

Description of the format is given as follows:
• Each VSF file contains multiple objects
• An object can have properties, relations, reference and metrics
• There are many different kinds of objects, some of them can be seen in objectKind enumeration
• Some of the relation kinds covered in the model can be seen in relKind enumeration
• Some of the reference file kinds can be seen in refFileKind enumeration
• The objId under the relation node, refers to the id of an object. It can be seen with the dotted lines between objId and id

```xml
<vsf>
  <prop name="Origin">Ba2VsF</prop>
  <prop name="CreateDate">2016-05-20 22:12:29</prop>
  <object kind="Section" name="Design">
    <prop name="Title">DrvTrans__coreLinuxAas</prop>
    <prop name="Version">5.01.00</prop>
    <prop name="Origin">BA Work.Component</prop>
    <prop name="CreateDate">2016-05-20 22:12:29</prop>
    <prop name="Kind">ComponentSpec</prop>
    <object kind="Component" name="DrvTrans__coreLinuxAas">
      <prop name="ComplexityClass">Normal</prop>
      <prop name="Layer">EcoAbstractionLayer</prop>
      <prop name="ModuleID">64</prop>
      <prop name="ModuleName">LIN Transceiver Driver</prop>
      <prop name="ProvidedByVector">true</prop>
      <prop name="Responsible">visapp</prop>
      <prop name="SafetyLevel">None</prop>
      <object kind="Section" name="CAB">
        <object kind="Package" name="Functional" id="{A5D62BEF-6135-4640-023A-1896398B1697}">
          <prop name="Id" kind="Id">DSGN-LinTrov22376</prop>
          <prop name="BA_GUID" kind="Id">{A5D62BEF-6135-4640-023A-1896398B1697}</prop>
          <prop name="Desc">![CDATA[These are the functionalities of the component.]]></prop>
        </object>
        <object kind="DesignFeature" name="Get operation mode" id="DSGN-LinTrov22862">
          <prop name="Id" kind="Id">DSGN-LinTrov22862</prop>
          <prop name="BA_GUID" kind="Id">{D0152138-B433-4864-9D0C-0669C812011F}</prop>
          <prop name="Desc">![CDATA[To get the current operation mode of a specified LIN transceiver hardware, the service function LinTrov_GetOpMode() has to be called. This function is called by LinIF.]]></prop>
        </object>
        <rel kind="Trace" objKind="CRBQ" objName="CRBQ-620" objId="CRBQ-620" />
      </object>
    </object>
  </object>
</vsf>
```

**Figure 2.2:** Example of VSF file

### 2.1.2 Variants

In the starting stages of the thesis, definition of the term ‘Variants’ had to be formed. There are several definitions of variants but the main task here to understand was ‘What are variants with respect to this thesis’. So, a definition of variants was to be formulated. The conclusion was ‘variants are derived from the common base and have stakeholder relevant properties but differs from other variants derived from the same common base’.

Variants can have a condition and a configuration. Different conditions can lead to the same variant, for example, Condition 1 or Condition 2 result in a behavior, even if either of the con-
dition is true the behavior will be same. Which condition it results in is always affected by the configuration that is set. Variants can only be present if a particular condition is true. PC is a subset of configurations in a particular artifact which should be true for a variant to be present.

Many different types of variants are available but, here only configuration variants are taken into consideration. Configuration variants are those which do not change during the runtime and are not associated with binding times. Binding time for an attribute is the time at which binding occurs i.e. the association of an attribute with a program component at a particular time.

2.2 Variant Handling in General

This section describes the variants in general which are present in many application domains, for example, automotive industry from a very basic level and how they are handled in different situations, for example, when the number of variants increase. Two different approaches for handling these kinds of situations are being discussed in the following sub sections.

2.2.1 SPES Framework

This subsection mainly focuses on the extension of Software Platform Embedded Systems (SPES) modelling framework in order to structure the overall variability model. According to [2], management of variants of embedded software in different application domains becomes increasingly important to address the demands of various stakeholders like customers, users etc. This variability of the embedded software is considered with respect to the artifacts that are created during the engineering process. The artifacts that were considered here are requirements, functional design, logical architecture and technical architecture. Continuous management of these variants of the embedded software throughout the whole life cycle is required.

SPES modelling framework is developed in the recent years by an industry and a consortium of 21 partners from Academia. It aims at supporting model-based engineering of embedded software. It is built upon 2 principles namely: “separation of concerns” and “divide and conquer”.

The principle separation of concerns distinguishes between the 4 viewpoints: requirements, functional design, logical architecture and technical architecture. These viewpoints focus on a set of role-specific concerns. For instance, the viewpoint requirement focuses on the concerns of a requirement engineer. Layers of granularity are represented by disintegrating the embedded software into fine-grained blocks like subsystems, components etc. The coarse-grained problems are step wise decomposed into the fine-grained problems. The principle of divide and conquer is executed by numerous layers of system granularity.

Originally variability variants were not considered in SPES. Therefore, they proposed a general solution of how to extend the SPES framework by adding an explicit variant management to it. Variability has a cross-cutting nature with respect to embedded software, so the concept of
variability perspective that is orthogonal to viewpoints and granularity layers is introduced.

**Approach**

As it is not sufficient to rely on just the SPES viewpoints for the structuring of the variability perspective, an approach to structure the variability based on role-based variability concerns was introduced by [2]. This approach was explained by giving an industrial example of an Advance Driver Assistance System (ADAS) variability model. Figure 2.3 shows the ADAS variability model. 2.3 represents a small part of a variability model as the complete model comprises of several hundred features. Multiple roles are involved in the engineering and marketing process of the ADAS. Some of these roles are used in this diagram. The system architect is responsible for designing and maintaining the architecture of a system and is responsible for the internal characteristics and variability of it. In the Figure 2.3 set of role based variability concerns associated with the role architecture are named as 'Sys.Arch'. In contrast, the requirement engineer manages the requirement of specific markets. In this figure set of role based variability concerns associated with the role requirement engineer are named as 'RC.CD.Re.Eng'. Thus, based on the concerns of an architect and requirement engineer two different variability viewpoints can be defined. Each one has a unique name and focuses on role specific concerns such as technical variability or the market variability in terms of behaviour or functionality. Hence, both the viewpoints listed above use the role specific subset of the variability information documented in the variability model and use the same model for representing the variability information.

![Figure 2.3: Example of an ADAS variability model](image-url)
2.2.2 Integrated Feature Modelling

This subsection deals with the variant traceability in Software Product Lines (SPL). It is crucial in the context of different tasks, for example, Change Impact Analysis (CIA) especially in complex global software projects. Restrictions and dependencies between the variants must be handled when the traceability concepts are extended by automated variant configuration mechanisms. Increasing numbers of software variants require inspection of traceable software variants over the life-cycle.

**Fundamentals**

In the automotive domain, stakeholder requirements lead to the increasing number of product variants. According to [3] understanding variability in a SPL, development process of engine control unit and its embedded software system should be analysed. This process involves development phases such as requirement engineering, specification, design, implementation, integration and testing. It also comprised number of abstraction layers such as system and subsystem and it also considered some artifacts namely requirements, function models, test specifications, regression test or calibration parameters.

To support the variability information this existing development process was extended with an integrated feature model. This integrated feature model has the above mentioned phases, abstraction levels and development artifacts.

**Approach**

While extending the development process some challenges were faced by [3]. One of the most important challenges was to handle restrictions and dependencies between the variants. The documentation itself will be insufficient if the number of variants increase in the future, for example, if a new requirement emerges, a new function is implemented for that variant. Apart from this there are several tracing information’s like requirement, design artifacts, code artifacts and test artifacts of that variant. Several information’s also contain some restriction and dependencies, for instance, function A requires hardware platform B. All these information’s are still documented but not in an integrated and explicit form i.e. all the information is there but it is scattered all over and is not available at one place in an integrated form.

Increasing software complexity due to increasing number of software variants as well as related restrictions and dependencies of software artifacts lead to difficulties in configuration tasks of one software variant.

Some challenges emerged during the extension of the existing development process. First challenge was the documentation of binding times. Figure 2.4 [3], illustrates common binding times in an SPL. To overcome this challenge they realised some connectors between the artifact types and feature model. As there can be a number of variability techniques, tools and artifacts this was a time consuming challenge. Second challenge was to supplement binding times by variation point configuration which was also fulfilled by the approach stated for the first challenge. Third challenge was with respect to change management. Over a period of time features have been added, removed or merged. This has to be considered in feature models. To overcome
this challenge they added additional release features, relations and restrictions to the integrated feature model. Features like split, rename and merge were still untraceable. Forth challenge was the evolution of artifacts and the consideration of related variation points. To document this further meta information was added to the artifact models.

2.2.3 Modelling Variability with UML

With the increase in number of variants in a product or a component, a need to deal with variability in an abstract and clean way becomes important. Feature modelling implements this in a hierarchical manner. It takes features into consideration and aims at expressing concepts, feature interdependencies and variability in order to capture the overall picture. Unified Modelling Language (UML) is a standard modelling language that describes semantics, rules and a notation specified by a meta-model. Currently, it is able to model a wide range of software systems but it is not suited for modelling groups of related systems or components.

Modelling variabilities and commonalities help in identifying the common parts and provide the possibility of development with re-usability. This is the most important part to keep in mind while developing variants. Variability helps to distinguish the members of a particular family from each other that further needs to be modelled and separated from the common parts. There is no consistent and uniform way for modelling variabilities and commonalities.

Variability in UML

As a standard UML is well suited to exchange ideas between stakeholders but it does not provide necessary means to describe variability. The approach by [4] describes an extension to overcome this shortcoming. It proposes a consistent way of modelling the variability using predefined
modelling elements and also using some advantages of feature model to extend the UML models to support variability. Three extensions are basically introduced:

- Feature Model extension is to introduce feature diagrams into UML so that it provides a way to systematically organize the variabilities of a group of software systems.
- An extension for the explicit representation of variation points. This shows the alternatives in the analysis and design of software.
- Third extension is about the optional elements provided specially for the situations where variation points are not applicable example in attributes, associations, operations.

**Feature Model Extension**

A feature is basically a significant characteristic of a system that is relevant for a stakeholder. The Figure 2.5 [4] is an example depicting the usage of all primary elements used for modelling in UML. Four types of features are mainly distinguished here: mandatory, optional, alternative and external. Features in the diagram are organised in a tree. Root is constructed from the composition or generalization relationships. The type of relationship to be used depends upon the identified relationship between features. These relationships are modelled with a UML notation.

![Figure 2.5: Example of a Feature Model using UML](image-url)
Composition rules are formed in form of cross tree constraints. Mutual exclusion (‘mutex’) and ‘requires’ dependencies are hard constraints. According to the semantics in UML constraints can also be distinguished as unidirectional or bidirectional. Every feature node can be represented with additional attributes also known as tagged values. Each node can be completed by addressing the binding time that can be chosen from installation or runtime.

**Variation Points**

Variation points can be applied to all classifiers and generalisable elements in UML. Extension for variant point basically consists of two main parts:

- An explicit marking of variation point
- Different ways to bind this variation

The location is marked with a stereotype «variationPoint» and contains several attributes to describe variation in detail as shown in the Figure 2.6. The mandatory connection from each variation to its variation point is modelled with generalization, template binding or with dependencies.

In addition, the constraints for the variants can also be modelled. At present, Mutual Exclusion (‘mutex’) and ‘requires’ is used for it. These constraints can be applied between variants and other modelling elements but not between variants and their variation points. These constraints can also be used for optional elements.

**Optional Elements**

When the notion of variant point is not applicable then the optional elements come into the scenario. An «optional» stereotype simply describes a model element that exists under some circumstances and in others, not. This extension is simple and can be applied on all UML model elements - even on behavioural diagrams if desired.

**Figure 2.6: Use of Variation Points and Optional Elements**
2.3 Storage of Variant Information

This section focuses on where and how the variants can be stored. An interchange format is also discussed to see whether it supports the storage of variant information or not.

2.3.1 Requirement Interchange Format

After researching about how the variant information can be modelled, now it is important to know the formats that can store the variant information. Requirement Interchange Format (ReqIF) is a format based on the Extensible Markup Language (XML) file format that can be used to exchange requirements and its associated meta data between software tools that are developed by different vendors. It was developed by Herstellerinitiative Software (HIS) which is a consortium of German Automotive manufacturers. ReqIF was adopted as a formal specification by Object Management Group (OMG) in April, 2011. OMG basically approve standards that can be used internationally in different industries.

Ever since the development of software started, one of the main concerns was recording of requirements that describe the system to be built [5]. This worked reasonably well for a long time. But as the years passed and technology became more advanced, systems were getting significantly more complex, the production was mostly being outsourced to suppliers, electronic data processing triggered a move from a paper-based work flow to a digital one.

Figure 2.7 [6] shows a generic meta model of ReqIF. The meta model of the ReqIF is used to define the requirement types and requirement objects. Suppose there is a functional requirement having attributes ID, value and name. We can create a ReqIF Specification Type (SpecType) as functional requirement with matching attribute definitions for ID, value and name. The node that contains the requirements is called the Specification Object (SpecObject). Requirements have user defined attributes. Each attribute can be of either Boolean, Integer, Real, String, Enumeration (with user-defined values) and XHTML type. Based on defined specification type we can also create the specification objects with attribute values for all the attribute definitions.

It has many benefits like lossless exchange of data with suppliers, generate HTML documents of the specifications etc. but the reason that it plays an important role in this thesis is because it was necessary to determine whether this requirement format can support the storage of variants or not. Although ReqIF does not directly support the handling of variant information, it is generic and can be extended. So, there are extensions made for ReqIF that support the handling of variant information.

According to [7] product lines and handling of the complexity of variants is one of the current challenges in many industries. The current implementation for variant handling in the requirement allows the specification of the conditions responsible for the presence of a particular variant. So, they developed an extension for the ReqIF to support product line variant management for the requirements. This extension has a tailored version of the requirement that provide a specific view on the variant. Using the feature of ReqIF the hierarchy structure of requirements
is not directly stored under the requirement. Instead they store it under specification hierarchies. The element of these hierarchies is just to build up the structure as they do not contain any data, rather only point to the actual requirements. Then a second hierarchy that is a copy of the first hierarchy is created with conditions responsible for the presence of a particular variant applied to it. Due to this the original data is also preserved. See Figure 2.8 [7]

2.3.2 Variant handling in simulink

According to [8] model based languages and models are predominantly used in the automotive domain to define artifacts for software. Models and code result in high complexity due to large number of variants that are present. Variants can be formed due to elaborated functionality or different realizations of the functionality. This particularly focuses on Simulink models, which are essential in developing the automotive software. Although it is not directly related to the topic of this thesis, it gives an idea about how we can store variants so that they could be handled properly.

Approach

Software variants in the automotive domain are usually modelled incrementally by reusing the parts of the old software. These methods result in unclear designs, where abstractions are missing. Common parts of the model are hard to understand if the complexity increases. Hence, resulting in decreased quality in terms of re-usability, extensibility etc.
2 State of the Art 2.3 Storage of Variant Information

Re-factoring overcomes the above stated problems by two main activities:

- Differencing Simulink models to identify different as well as common parts
- Restructuring the models with variability mechanisms to improve quality

Differentiator function gets two Simulink models as input and gives three models as output, a commonality model and two difference models. Commonality model contains Simulink blocks and connections that are similar or common in both the input models. It also comprises the
points of variation. They are further elaborated by the difference models that are there. They include only variant specific details. Figure 2.9 [8] shows the approach that is presented above. This approach can also be used on different levels like software architectures, code and all the other relevant modelling artifacts used in the automotive software.

2.4 Extraction of variant information from code

Each of the artifacts including code consists of the variants. Variant information in the code is mainly the pre-processor statements. To implement variability, developers often use conditional compilation mechanisms of C pre-processor. Directives like #ifdef and #endif frame code fragments that are conditionally excluded during compilation process. Depending upon the feature selection in the configuration files the pre-processor generates different variants of the code.

Unfortunately, the C pre-processor has properties that make it difficult to analyse. Some of them are listed below:

- C processor is token based and uses lexical macros. While the pre-processor can be used independently but can also introduce syntax errors, for example, parenthesis mismatches. Pre-processor has no mechanism to describe potential problems in the underlying code
- Conditional compilations such as #define and #undef is not only used for implementing compile time variability but it also provides facility for the inclusion of guards. An include guard prevents multiple inclusions in the same file, which uses the same conditional compilations and is difficult to ascertain the variability by tools.

Approach

According to [9], analysis becomes difficult when dealing with the variability implemented by the C pre-processor as conditional compilation directives are mixed with file inclusion and macros. They developed a partial pre-processor that not only evaluates file inclusion and macros but also retains the variability for future analysis. It also deals with alternative expansion of macros and inclusion of guards in an automated way without heuristics.

This partial pre-processor was implemented in Java and Scala on top of jccp which is an actual Java implementation of the C pre-processor. They implemented a library of feature conditions for #if conditions so that the tautologies and contradictions (satisfiability) can be verified. They used the SAT4j solver for this.

A partial pre-processor tracks the current PCs when a file is under pre-processing. SAT4j was used to produce the satisfiability of that condition to evaluate a conditional compilation directive. Each time a #define or #undef directive is encountered, the alternative expansions for macros are saved and their PCs are revised. Also, when expanding tokens that have multiple satisfiable PCs, the tokens are replaced by all possible expansions where each expansion is wrapped under its own conditional compilation directive.
2 State of the Art 2.5 Tools and Technologies

The desired output that we get here is a token stream where each of the token has its PC i.e. under which feature selection token is included in the compilation. Macros are also expanded. Tokens that have alternative expansions are expanded to all possible solutions. For example type T in line 16 (see Figure 2.10a [9]) is expanded to both long and short with respective PCs in line 41 (see Figure 2.10b [9]).

Limitations

Most of the solutions that use the unsound heuristics for searching of the variant information from the code search only for the pre-processor directives not for the underlying code. If there are some conditional if statements that are the variants, they cannot be categorised as the configuration variant or a runtime variant.

2.5 Tools and Technologies

This section discusses various tools and technologies that were studied and might be helpful during the implementation phase for the accomplishment of the format developed in the conceptualization phase. Each tool and technology is understood from the point of the research work that has to be carried out. Understanding how each of them should be used is also an important aspect.

Visual studio Integrated Development Environment (IDE)
This IDE was used for the implementation in the thesis. A Satisfiability Modulo Theories (SMT) solver was later integrated in the tool and used afterwards.

**CDK**

Component Development Kit (CDK) is an internal tool developed at Vector Informatik GmbH. It is used for parsing, analysing code and static testing. It is basically used to integrate different scripts that are later run with the database in order to generate reports.

**Enterprise Architect**

One of the crucial steps when starting system development is understanding the requirements clearly. Enterprise Architect (EA) is a tool that is developed by Spark systems. It is a visual modelling tool based on UML that provides modelling support for software, real time or embedded systems. It also helps in discovering specifications to all the artifacts like analysis, design, implementation and test using open standards like UML. It is a visual modelling and a graphical design tool by OMG. It supports languages like C, C#, C++, Java, Python etc. Some of the features of EA are listed below [10]:

- Data Modelling
- System Engineering
- Requirements Management
- System Development

It is used by organizations to model the architecture of the system as well as process the implementation of these models across the application development life cycle. Figure 2.11 shows the EA interface.

![Figure 2.11: Enterprise Architect](image-url)
UML

UML was adopted as a standard by OMG in the year 1997 [11]. It contains 13 different types of diagrammatic representations to depict the software development process. Some of these graphical representations are used to explain the structural behaviour of the system while other depicts the functional behaviour. Following are the main categories of the types of diagrams present in UML [12]:

Structural Diagrams
- Deployment diagram
- Component diagram
- Class diagram
- Package diagram
- Object diagram
- Composite structure diagram

Behavioural Diagrams
- Activity diagram
- Use case diagram
- Sequence diagram
- Communication diagram
- Timing diagram
- State diagram
- Interaction overview diagram

Some of these diagrams were used to represent the implementation and conceptualization part of the research with the help of EA.

C#

C# is an object oriented and multi paradigm programming language developed by Microsoft within its .NET initiative [13]. Some of the features of C# are listed below [14]:

- Meta Programming
- Portability
- Namespace
- Polymorphism
- Functional programming
• Exceptions
• Memory access

C# is designed for Common Language Infrastructure (CLI) which contains runtime environment and executable code. CLI allows use of high level languages on different architectures and platforms. C# was used as the language for the implementation of the research in this thesis.

**XML**

XML was designed to be both human and machine readable. Its main purpose is to transport and store data by defining a set of rules for encoding documents in a particular format. It is a highly used language because of its simplicity in defining rules.
3 Concept

3.1 Fundamentals

Before proceeding to the extension of VSF format for the support of variant information we should know all the basic terms that have been used. In order to develop a new concept, the understanding of all these topics is necessary. This section covers the topics like:

- Abstraction levels
- Variant information in different artifacts

3.1.1 Abstraction levels

All the variant examples from the different artifacts were collected and the variability was categorised into two types:

**Structural Variability**

These are the types of variants that are either available or not available. If available, they can be used. But if not, the related functionality would not exist.

![Figure 3.1: Structural Variability](image)

An example to explain this type of variability is shown in 3.1. A function can have different behavior depending upon which feature is on but the function itself needs to be present in order to show that behavior. A feature is defined as a distinguishing characteristic of software, for
example, functionality or performance. This can be seen as variability in a higher abstraction level. This type of variability is called structural variability that can either be present or absent. So here, depending upon whether feature A is on or not the availability of function is affected. It is assumed that, if the function itself is not available then the objects and their variants inside this function are also not available.

```c
#ifdef FEATURE_A
    Function 1
#endif
#ifdef FEATURE_B
    Function 2
#endif
```

**Figure 3.2:** Code example depicting structural variability

A code example for the structural variability can be seen in Figure 3.2. The availability of the Function 1 and Function 2 depends upon the availability of the FEATURE_A and FEATURE_B attached to the function respectively. This function can be available or unavailable depending upon whether the PC is true or not. If a structural variability of a function is not true the further lower level functional variability for it won’t exist.

**Functional Variability**

This is a lower level variability. According to functional variability a behavior of an object might be available or unavailable i.e. the inside(s) of the object might vary.

**Figure 3.3:** Functional Variability

An example can be, a function having two different behaviors depending upon the type of configuration that is set. This function can either have behavior 1 or 2 depending upon whether
feature B is on or feature C is on instead. This can be seen as variability in a lower abstraction level. An example to depict this type of variability is shown in 3.3

```c
Function 1
#elif (FEATURE_A)
    behavior 1
#endif
#elif (FEATURE_B)
    behavior 2
#endif
```

**Figure 3.4:** Code example depicting functional variability

A code example can be seen in Figure 3.4. This Function has two behaviors depending upon the configuration variants that are true. As we can see in this example the behavior of the function is variable. If FEATURE_A is available then Function 1 will have the behavior 1 or if FEATURE_B is available then Function 1 will have behavior 2.

3.1.2 Variant information in different artifacts

This subsection deals with the variant information present in different artifacts. Analysis of PC in different artifacts is carried out in this part. Different forms in which the variant information is present are also shown.

**Requirement**

Variants in the requirement artifact are always depicted in an informal textual manner. Requirements can be structural and functional. Functional requirements tell what a software system should do while structural requirements place the constraints on how the system will do it. Structural variants in the requirements artifact are not available here at Vector Informatik GmbH i.e. the structure is not defined but the focus is on the functionality to be performed. Functional variants in the requirement artifact can be present wherever there are two or more than two possible solutions. Figure 3.5 is an example of a functional variant in the requirement.

There are two possible situations:
- Feature A is true : Behavior 1
- Feature A is false : Behavior 2

**Figure 3.5:** Example of a functional variant in requirement artifact

In contrast to the example stated above, a more specific example is stated in Figure 3.6. It tells the functionality of a module. A priority of a module can be enabled or disabled which in turn leads to different behaviors of that module. The requirement here is represented in a textual manner same as they are represented at Vector Informatik GmbH.
Module shall provide queuing mechanism
Priority is enabled: There are two queues, one is the priority queue and other is the normal queue
Priority is disabled: There is only one queue, queue shall work in FIFO
Additionally, priority configuration is only available with queuing

Figure 3.6: Concrete example of how requirements are depicted

Design
Design artifact contains more refined and detailed form of variants. Design can be either abstract or detailed. Abstract design is usually documented using UML, for example, activity diagrams, state machines and sequence charts. Abstract design can further be structural and functional. Detailed design is usually contained in implementation and is represented by the function declarations, definitions, prototypes, parameters, return types that define the structure of a program and some descriptions of how will it be implemented in the code. All the prototypes, parameters etc. that are mentioned in the Component Detailed Design (CDD) i.e. the function definitions in the form of code in a program are the structural variants in the code. Variants in the design artifact can be represented by, for instance an alternative block in the sequence diagram; activity diagrams showing behavior with enabled or disabled behavior or the application interfaces that more or less depend upon the functionalities that are available. It may be possible that if a particular functionality is not available the related interface may not exist. An example of a design artifact as an activity diagram is shown in Figure 3.7. This figure can also be considered
as a refinement of the requirement in Figure 3.6. Here, the feature in the diagram has no information about the binding times, so we cannot determine with the given conditions whether it is a configuration or a runtime variant. Figure 3.8 shows an example of a sequence diagram in

![Sequence diagram having alternative and optional blocks](image)

**Figure 3.8:** Sequence diagram having alternative and optional blocks

the CDD artifact showing the alternative blocks and the optional blocks representing functional and structural variability respectively. In the alternative block the prioritization can have two different behaviors depending upon whether it is enabled or disabled. Hence, it represents the functional variant. On the other hand if we look at the optional block, it represents whether it can be there or it cannot be there i.e. the priority configuration is only possible if the queuing mechanism is there otherwise it is not possible.

One of the other representations of configuration variants can be the state machines. See Figure 3.9. The state machine is again the representation of the same requirement stated in Figure 3.6.

**Implementation**

As we go to the further artifacts the variants become more detailed. Implementation artifacts can also be structural or functional. All the pre-processor statements in the code and all the underlying if statements that do not change during the runtime and are reconfigured are the configuration variants represented here. There are some underlying if statements that are also present but if they do not change during the runtime they are configuration variants. Figure 3.10 is an example of how variants can be present in the code. Some kinds of variants present in a particular artifact can be an structural as well as an functional variants, it totally depends upon the abstraction level we are looking at.
3 Concept 3.1 Fundamentals

Figure 3.9: State machine showing configuration variants

Function 1
#ifdef (FEATURE_B == STD_ON)
    Behavior 1
#endif
#ifdef (FEATURE_C == STD_ON)
    Behavior 2
#endif
If (Condition 2)
    Function 2

Figure 3.10: Example of a variant in implementation

#ifndef (QUEUING == ON)
    #ifdef (MODULE_PRIORITIZATION == ON)
        MODULE_Normalpriority == ON;
        MODULE_Highpriority == ON;
    #endif
    #ifdef (MODULE_PRIORITIZATION == OFF)
        MODULE_Normalpriority == ON;
        MODULE_Highpriority == OFF;
    #endif
#endif

Figure 3.11: Specific code example depicting variability
3.2 Concept Overview

In this section, the overview of the new concept can be seen in Figure 3.12. All the yellow color objects here represent the data sources that are given as an input to the VSF file. Currently, VSF can be used for many tooling purposes. The intermediate file is given to the developers which contains the definition for certain data, it can be used to generate user documents. This document contains details of product design. Apart from this, the format can also be used to generate test scripts which can be used for static checks of the component developed and can gives a possibility to verify the requirements if we know what are the results achieved and what should be the expected results.

The proposed approach would extend the intermediate format VSF here at Vector Informatik GmbH. This format would provide all the present variant information. Once we have all the variant information available in the generic format, it would give architect(s) and component developer(s) an automated way to verify all the variant information that is present, for example consistency checks, find specific information etc.

3.2.1 Concerns

Before thinking of how the format should look like there had several questions that were to be answered and analysed. These questions are listed below:

- In the starting, we had to look what are variants and which categories for variants should be described
- Where is the variant data present (what artifacts are involved) and what type of data is it?
- Are the variants part of an object or are the objects only available under specific variants?
- How is the variant information documented?
- How to document the missing variant information in the specifications?

By the end of the conceptualisation phase all these questions should be answered.
3.2.2 Use-cases

Use-cases describe the visible behavior of a system from user perspective. This methodology is used in systems for analysis and clarification of system requirements. It can also be considered as a collection of all possible scenarios related to a goal, but sometimes use-cases and goals are also interchangeably used. Below are some use cases that had to be implemented. See Figure 3.13.

**Checking the PCs in the same artifact**

Requirement of the stakeholders is the data that is usually present in the architecture level. This data can have number of variants depending upon different needs of stakeholders. Architects use this data to design the product architecture and the same data is then given to the component developers as their initial requirements. The variant data that is present there should be consistent and verifiable so that the system development becomes convenient.

Consider a CDD artifact, a scenario for this type of a use-case can be three different functions (Function A, B, C) having a call relation to each other. See Figure 3.14. Each of them have their PCs attached. If the PC is not true for any of the function the relation to that particular function will not exist.
Figure 3.13: Use Case 1, 2 and 3

Figure 3.14: PC in the same artifact

Checking the PCs in different artifacts

Consider two different artifacts i.e. requirement and design. Variant information present in the requirement artifact should also be reflected in the design artifact i.e. the variant information in both the artifacts should be consistent. Consider an example where a requirement artifact with some PC has a trace by relation to some CDD artifact. Requirement is present or not present based on the configuration variant. If the requirement is not present, the respective relation to the CDD will also not be present.
3.3 Proposed Approaches

Once all the requirements were clear, two approaches were defined. Focus there was upon PC and where can it be placed.

3.3.1 As global as possible

This is a naive approach. In this approach we wanted to keep all the PCs as global as possible i.e. the PC is only attached to the topmost node and we multiply the objects for the PC. See Figure 3.15. If the upper level object here is not present the lower level will not be there too. As shown in Figure 3.16 the worst case condition for this kind of approach could also be that we would make separate VSF for each of the objects present.

Advantages

- We are making PCs generic as it should be in the VSF format as the introduction of a new node is not needed
- Only minor modifications are necessary for existing analysis tools as the PC is only added on the top level i.e. not to each object separately

Disadvantages

- It will grow exponentially depending upon the PCs that are there. The more PCs, the bigger will be the file
3.3 Proposed Approaches

Figure 3.15: Keeping PC as global as possible

- Redundancy will occur as there will be duplication of information. As shown in Figure 3.16, the return type of the variant is same in both the cases but it is still displayed multiple times which results in more memory usage as redundant information is present although it is not needed.

- It will only work for a small number of variants. As the number of variants increase it is not really possible to apply this approach because of the exponential increase in size.

3.3.2 As local as possible

This is the type of approach where the PCs are kept as local as possible with respect to the objects. Each object has its own PC attached as shown in 3.17. Both the variants shown in the figure have the same return type but the parameter type for both of them are different. Presence of parameter a or b depends upon the PC attached to both of them.

Advantages

- It does not grow exponentially as we have separate conditions for each of the objects which results in less memory usage as compared to the first approach.

- Conversion of the PC from this format to the other is easier as compared to the other way round as we can just multiply the objects for the PC.

Disadvantages
• VSF is a generic format but we are not keeping PCs as general, they are specific to a particular object. For example, this requires the introduction of additional structures in VSF as the PC maybe nested under elements that do not have children by definition.

This approach was chosen as the final one for implementation because of its advantages over the naive approach i.e. as global as possible. This approach can also be converted into the first approach listed but the vice versa is not possible and it is possible to use it with a large number of variants without exponentially increasing the size of the VSF file.

3.4 Satisfiability Checking

After choosing the approach, a choice of tool had to be made for verification on variants present. Any Boolean Satisfiability Problem (BSP) which is also known as Propositional Satisfiability Problem (SAT) is the problem that determines if there is an explanation which exists that satisfies a given Boolean formula i.e. whether the values in a given Boolean formula can be replaced by the values of TRUE or FALSE in such a way that the formula evaluates to true. If this is the case the formula is called ‘Satisfiable’, on the other hand if no such assignment exists it is called as ‘Unsatisfiable’. After the variants are stored in a particular format and the VSF has been extended we use the variant information for the verification of the variant information that is located on different levels and in the different artifacts. Despite this drawback many scalable and efficient algorithms for this, exist. Some of the algorithms to detect the satisfiability of a boolean expression are:
Maximum Satisfiability (MAX-SAT) problem determines the maximum number of clauses of a boolean formula that can be made true by assigning the truth values to the variables of that particular formula [15]. There are many solvers that have been recently developed to use MAX-SAT problems and many of them were also presented in the Satisfiability (SAT) conference, for example, QMaxSat and SAT4j. MAX-SAT is one of an optimized extension of BSP. There are several extensions to MAX-SAT problems:

- Soft satisfiability problem asks for the maximum number of sets that can be satisfied by an assignment from a given set of SAT problems
- Partial satisfiability problem asks for the maximum number of clauses that can be satisfied by any assignment of a given subset.
• Weighted maximum satisfiability problem asks for the maximum weight that can be satisfied by a particular assignment

• Minimum satisfiability problem

**DPLL**

The satisfiability problem is a fundamental problem in fields like constraint satisfaction, computing theory etc. Methods to solve satisfiability problem play an important role in computing theory and development of systems. Davis-Putnam-Logemann-Loveland (DPLL) is a complete algorithm for finding an assignment that is satisfying or unsatisfying for the given formula. The idea behind this basic procedure was proposed by Davis and Putnam. This is a recursive algorithm that repeatedly selects an unassigned literal \( l \) in the input of formula \( F \) and recursively search for satisfiability or unsatisfiability [16]. Family of the most widely used complete algorithms are the variants of this algorithm which are implemented in an iterative manner that results in significantly reduced memory usage.

After looking at the available algorithms four tools were found out. They were analysed for the verification of variant information (satisfiability checking) which are listed as follows:

**BDD Satisfiability Solver**

Binary Decision Diagram (BDD) solver mainly provides solution to industrial and practical asynchronous circuit design problems [17]. BDD satisfiability solver mainly consists of the structural SAT formula pre-processor and a complete incremental SAT algorithm that helps in finding an optimal solution. The pre-processor helps in breaking down of a complex SAT formula into smaller SAT formulas. This is done so that there is no need to solve large and complex SAT formulas and each of the small SAT formula can be efficiently solved by the BDD SAT algorithm. Eventually, the solution of all these simplified sub problems are integrated together that gives the solution to the original problem.

Some of the important features of BDD are:

• It is a library for manipulating binary decision diagrams

• It is mostly used for model checking and formal verification

• It is implemented in Java, its Application Interface (API) is designed to be object oriented.

• Designed for high performance applications

**OR Tools**

When the set of possible solutions is large it is not practically possible to search through all of them and find the best solution [18]. There are various approaches that can be used to speed up the search for searching the best solution, for example, combinatorial optimization. OR tools software suite makes it easy to solve different types of combinatorial problems like constraint programming or linear programming. This satisfiability tool was developed by Google. Some of the important features of OR tools are:
• It is a constraint programming solver
• Follows knapsack and graph algorithm
• Has simple interface to several linear programming and mixed integer programming solvers
• It is available in C, C++, C, Python, Java

**SAT4j**

SAT4j project started in 2003 as an implementation of MiniSAT specification [19]. SAT4j is an open source Java library that is used to solve boolean satisfaction and optimization problems. Some features of SAT4j are:

• It can solve SAT, MAXSAT and Pseudo boolean problems
• Being in Java, it is not fast but is full featured, robust and user friendly
• Cannot solve knapsack problems at its best

**Z3 Prover**

This satisfiability tool was developed by Microsoft [20]. It is an open source SMT solver. Some of the important features of Z3 Prover are:

• It combines several decision procedures and can solve linear, real and integer arithmetic
• Mainly used in test case generation, program analysis and verification
• Several input formats and model generation is also available
• **API** available in C, C++, C#, Python, Java and OCaml

Z3 was finally chosen as the verification tool for variants as it has an **API** available in C# which can be integrated with DotNet framework to work with. It also allows type checking that is one of the future goals to achieve and is capable of solving many complex operations as compared to other solvers that were analysed. Z3 supports more theories like non linear integer arithmetic which is not decidable and incomplete theory by using some heuristics or by translating the statements into real arithmetic which are complete and decidable.
3.5 Detailed design

The chosen approach of keeping PCs as local as possible suggests each object should have its own PC directly under it, before knowing where to place the PC node exactly in the VSF we should know how it looks. A meta model of PC node can be seen in Figure 3.18. PC here is a boolean expression that can either be true or false depending upon the configuration that is set. It was decided to have an object for the PC named as \( pc \). PC object can contain one or more operator objects. Only one expression inside a particular PC node is allowed. Operator object can either be binary or unary. If an operator is unary it can be of type defined and not but if it is a binary operator it can be of type and, or, equals, implies.

As we go from requirement to implementation, we see that the requirement itself and the information regarding variants become detailed and refined. There is some behavior variability information in artifacts that is introduced in the later stages but is not externally visible when
seen from a higher level of abstraction i.e. Requirement artifact. In this case sometimes structural variability might lead to lower level functional variability as well. Dependencies can occur in the same artifact as well as the different artifacts.

Consider a function that calls another function and that call is variable i.e. there is a PC attached to it. For that call to happen the call of the first function and the second function itself should be present. This shows the dependency of variants in one artifact. For the dependency in different artifacts we can take a simple example of two different artifacts i.e. Component Abstract Design (CAD) and CDD. CAD is related to the CDD artifact by a trace relation and it gets detailed when we move to the lower level artifacts. Variability depicted in the lower level artifacts might be located at different places and may or may not have relationship with each other.

In order to prove that the higher level variability implies all the lower level variability we need to put all these refinement at a particular place so that it is much clearer and easier to understand it. One of the options to depict the refinement of conditions was the feature model which is a compact portrayal of all the variants in terms of features. This was not chosen as the representation afterwards. Finally, to show the refinement of PCs in the format, we modelled a new object named as constraint. The PCs there are shown by a property object named as expression. Although, this object has the same structure as a PC object it was not named the same to distinguish between the PCs that are true for a particular configuration and the refinement of those PCs which may or may not be true.

An overview of the VSF representation after adding the PC object is shown in Figure 3.19. PCs were added to the relations, properties and objects in the VSF. For a function, property object can have more than one parameter. A new node value was introduced inside a property node of a parameter which contained the actual value of the parameter object in the form of text. PCs was attached to the value inside the property object (parameter) rather than attaching it directly to the object itself because the PC was for a particular value of a parameter and not for the whole parameter itself. So, the PC object was attached to the value. This was done to make the attachment of a PC to an object more specific. Depending upon the attachment of PC to a particular value, parameter type can be present or absent.

Return type can only be of a single type for a particular function. A new value node for the object return was also introduced and there the PC was attached the same way as it was attached to the property object. If there are two different values inside a property node or any other element with the same PC, it would be confusing which value to choose. So, it was assumed that this scenario is not possible in this thesis.
Figure 3.19: File model of the VSF after adding PC object
4 Implementation

4.1 Extension of VSF

In the previous chapter we developed a design to store the variant information in VSF. Taking the chosen approach into account i.e. as local as possible, the VSF was extended by adding the presence condition nodes under objects, relations and properties. It was decided to have a tag for the presence condition named as 'PC' rather than making it a property named presence condition because properties can only be nested inside objects but cannot be nested inside a relation or property itself. Also, the properties were extended to have a node named value which contains the value of a property as text.

![Figure 4.1: Structure of a PC node](image)

Each node in the diagram 4.1 is explained below:

- A **PC** node contains exactly one operator node
- An operator node has further one or two children nodes that can be either some more operators or an identifier
• Operators are identified by two attributes namely 'kind' and 'name'. Both of the attributes are mandatory

• Operators are either binary or unary which is identified by the attribute operator kind

• Unary operators are categorised as not and defined operators while binary operators can be categorised as and, or, not and implies

• Identification to segregate the type of unary or binary operator can be done by the attribute operator name

An operator node can contain further operator nodes inside it. Depending upon whether the operator is binary or unary, identifier nodes vary, i.e. if the operator kind is binary it will have two identifier nodes that are lhs and rhs of a binary expression but if the operator kind is unary it only contains one identifier node that is an operand. An example of a PC node is shown in Figure 4.2

```xml
<pc>
    <operator kind="binary" name="and">
        <operator kind="unary" name="defined">
            <Identifier name="FEAT_W"/>
        </operator>
        <operator kind="binary" name="or">
            <Identifier name="FEAT_U"/>
            <Identifier name= "OFF"/>
        </operator>
    </operator>
</pc>
```

Figure 4.2: Example of a PC node

For the depiction of refinement of conditions a node for an object kind constraint was added. It contains a property named as expression. Properties are identified by their name. The structure of nodes below this property are the same as PC nodes. An example of the constraint node is shown in Figure 4.3.

Apart from properties, PC node was also added to the relation nodes. A relation would or would not exist based on whether the PC is true or not. In the original VSF, property node had a value directly attached to it as a text. Plain text was used as a child of property. If PC was added to the property text then it would mean that property itself is a variant whereas it was important to depict that the value of the property is a variant. So, the structure was further extended by adding a value node to the property node which contained text inside it. PC was attached to the text inside the value node depicting the variance of the value rather than the property itself. A value node was added to a property and return type node. Figure 4.4 shows the value node added to the parameter. The PC node was then attached to the text that was inside the value node rather
4.2 Handling of presence conditions

Before looking at how PCs are handled in VSF it is necessary to see how other objects are handled in there. VXmlNode is the base class for all the other objects that are present in VSF and the PC was directly added to it. The reason that the PC are directly added here is because it is a common base for all the other objects. Other objects can either be a relation, property or an object node. These nodes are relevant to understand the further extensions.
• An object node is identified by its attribute kind and has two more attributes named id and name that are optional.

• A relation has attributes kind, objKind, objName that are mandatory and one more attribute named objId that is optional. A relation can further have nested relations or a property.

• A property node is identified by its name and has one more optional attribute named kind. See Figure 4.5.

Figure 4.5: Nodes in VSF

The highlighted portions in the diagram 4.5 are the additions that were made in the VSF file for the extension. A boolean expression PC was made which can be attached to any node with a PC. GetPresenceCondition is a function that was used to get all the PCs located in any of the objects. If there is a PC attached to a particular node then it returns the whole PC otherwise it returns true as it has no variants and can never be false.

4.2.1 PC from VSF to internal model

Once the addition of PCs inside the VSF file is done according to the selected format, an internal model of all the expressions that are encountered in the PC node of the VSF file was created. An
abstract syntax tree of all kinds of expressions was made so that it could be traversed afterwards.

![Diagram showing classes and relationships between different types of boolean expressions.]

**Figure 4.6:** Expressions from VSF to internal model

As shown in Figure 4.6, every *Expression* in the VSF is basically a *Boolean expression*. Further, a *Boolean expression* can be an *Identifier*, *True expression*, *False expression*, *Unary expression*, *Binary expression* or an *Existential expression*. See Figure 4.7.

![Diagram showing a class diagram for boolean expressions with subclasses.]

**Figure 4.7:** Types of boolean expressions

Five types of *Binary operators* that are currently supported in the tool are *Or*, *And*, *Equals*, *Implies*, *Greater than*. Each of the *Binary expression* should have a left hand side and a right hand side and a *Binary operator* in between. A switch that is a name of a feature and its value are both *Identifiers*. See Figure 4.8.
Only 2 types of Unary expressions are supported namely Defined and Not. A Unary expression has only one Identifier and a Unary operator attached to it. See Figure 4.9.

4.2.2 PC from internal model to Z3

This subsection specifically deals with understanding the conversion from internal model to Z3 format. Having all the expressions in the form of an abstract syntax tree, it is important to visit
all those expressions and change them into the format which Z3 accepts. Some operations were
to be performed on the elements (expressions in our case) of an object structure and visitor
pattern was best suited for it. Visitor pattern helps in performing distinct operations on node
objects inside a data structure. This can be done separately without polluting the node classes
with different operations.

Visitor in our case also implements double dispatch which is a special form of multiple dispatch.
It dispatches a function call to the concrete functions depending upon the type of objects in a call
made during runtime. A double dispatch for all the types of expressions present in our Abstract
Syntax Tree (AST) is made. Execution of double dispatch depends mainly upon the type of
visitor and the type of element the visitor visits. Further implementation of the visitor requires
to create a visitor class that defines abstract visit() methods. This is basically the abstract base
class for all the other concrete classes that will be derived from it. Each visit method accepts an
element expression which is the reference to the original element. See Figure 4.10.

Each of the expressions that has to be supported in the base class(Expression visitor) is also in
the concrete derived class(Expression to Z3). The visit() method in the Expression visitor base
class is also defined in the Expression to Z3 class by overriding it along with the specified type
which is a boolean expression in our case. Then a virtual accept() method is implemented in
each element of the element hierarchy which takes a single argument that is a reference to the
Expression visitor (a reference to this keyword in C#). The accept() method helps to find the
correct expression. When the visit() method is invoked the flow of control reaches the correct
visitor sub class.

4.3 Working with Z3

Z3 has a context and a solver. Context manages all the Z3 objects, global configuration options
etc. It is also relevant for multi-threaded programs as each thread can have its own context and
can be accessed without any synchronization, for example, mutex. Each expression belongs to a
single context and cannot be used in different contexts. Solver contains the constraints that have
been added to it by the assert command i.e. it keeps track of all the constraints in the problem.
In the implementation an object of solver was created and was used to prove whether a boolean
expression is true or false by calling the check command of the solver.

4.4 Implementation of use cases

4.4.1 PC in the same artifact solution

A PC is always a boolean expression. If a boolean expression for all the possible assignments
of truth value to its variables is self-evident i.e. always true, it is called a tautology. On the
other hand, if a boolean expression for all possible assignments of truth value to its variables is
evidently false i.e. always false then it is called a contradiction. Contingency occurs in case a boolean expression is neither a Tautology nor a Contradiction, then it may be true or false i.e. It may or may not work in some configurations. Z3 was used to prove that the boolean expressions in the extended VSF are the tautologies or contradictions.

Once Z3 was integrated with the Visual studio C#, verification started with the easy cases where the satisfiability could be checked. Some PCs in the object function were added and the satisfiability of them was checked just to see whether a particular function is available or not. Check command in Z3 is used to verify the satisfiability of an expression. If the result is satisfiable then it results in a particular object being available otherwise the object is unavailable. Once an expression is returned as satisfiable, a representation that makes it true on the Z3 internal stack can be retrieved by the command get-model. Two boolean values true and false were created. True was returned whenever the function is satisfiable and false when it was not satisfiable.

This was extended when the checking of functions and their presence with respective relations was done. Here, the presence of a relation was dependent upon the presence of the respective
function. If a particular function is not satisfiable then it will not exist and if it does not exist the relations from that particular function to any other function will not exist too.

Then if a function exists, the PCs of relations inside it were considered. This was done by using the AND operator from the model. PC of a function and the relation were ANDED so that if one of them results in false the final result will be false. Tautologies and contradictions of these related expressions were seen and if an expression was neither a tautology nor a contradiction it was assumed that it may or may not work in some configurations.

As seen in Figure 4.11, along with the relation node the object also has a PC. The call of function A to B would only work if the PC of the call ANDED with the PC of the function is satisfiable. In the example shown, the PC is not consistent. So, in this case it is a contingency i.e. it may or may not work in some configurations.

Same example given in Figure 4.11 can be modified to show tautologies and contradictions. See Table 4.1. The first part that results in tautology has the PC of a function same as the PC of the call. If the PC for the function B is true then PC of the call to the function B will also be true for sure. In the second part the PC of the function B is opposite to the PC of call to function B, so this can never be true which results in a contradiction.

<table>
<thead>
<tr>
<th>Examples depicting occurrence of Tautology and Contradiction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PC of Function B</strong></td>
</tr>
<tr>
<td>defined(FEAT M)</td>
</tr>
<tr>
<td>defined(FEAT M)</td>
</tr>
<tr>
<td>defined(FEAT M)</td>
</tr>
</tbody>
</table>

Table 4.1: Examples depicting occurrence of Tautology and Contradiction

### 4.4.2 PC in the different artifacts solution

This subsection deals with the PCs in different artifacts. Say for instance, a requirement and a design artifact. There are some relation nodes in the requirement artifact that might be related to some functions in the design artifact. There are many different types of relations that exist in VSF, some of them are Include, IncludeBy, Call, CallBy but here the main concentration was only upon the TraceBy relations. If there is a TraceBy relation between two artifacts then the PC of both the relation node and the function node was considered. If both of them are true then it will always work (tautology), if both of them are false (contradiction) then it will never work, otherwise it may or may not work for some configurations (contingency). Also, if there is some PC in the relation node of the requirement artifact it should be the same as the PC of relation node in the design artifact.
Refinement of PC was the second part of this use case. As told earlier, it was handled by the object constraint in the VSF afterwards. Abstract information about PC in the requirement artifact should imply the detailed lower level information in the design or further artifacts. Consider a configuration variant in the requirement artifact in which a requirement is only available when \( A=1 \) but moving on to the design artifact this information gets more detailed and there the same configuration variant is represented as Function A and B are only available when \( A'=2 \) and \( A''=3 \) respectively. This in the VSF is depicted by the trace relations from one artifact to the other. See Figure 4.12. These traced Functions may or may not have relation to each other. To depict relation between them both the expressions were ANDED. The representation of refinement of conditions was done by an \textit{implies} operator from the requirement artifact to the design artifact. This is done in the constraint section which was included as an extension for the VSF. For this,
constraints were given to the context and were added to the stack. Implication of the constraint in the requirement artifact was checked against the constraint in the design artifact.

![Image of requirement tracing to design artifact]

**Figure 4.12**: Requirement tracing to design artifact

Tautology and contradiction for these refinements were also checked. Consider Figure 4.12 as an example of a requirement tracing to two functions, if function A has a PC (B \&\& A') and B is false then the function A will not be present and the whole implication from requirement to function will not be true. For this example the refinement would be represented as (A => A' \&\& A').

### 4.4.3 Find information in different artifacts solution

This subsection deals with searching of various types of information in different artifacts. In the beginning of this use case some easy examples were taken into account in which a function with a particular return type was to be searched, for example, only functions with return type void are needed. PC of both the void return type and the function were taken into account and a logical AND operation was performed between them. A similar approach was taken while searching for a parameter of a particular type. And then the satisfiability of the whole expression was checked.

The second part of this use case was to get a function with a particular signature. A signature consists of the parameter and return types. A parameter is itself an object that consists of a
property type node inside which there is a value node which contains the value of a parameter as text. **PC** inside a parameter and a value node both were checked as for the value inside a parameter to be available the parameter itself should be available. Logical AND was performed between the **PC** of the function, parameter, value of parameter and the return type. If all of them are true then only the particular signature exists. As seen in Figure 4.13, if there is a function with **PC** A, return type with B and a parameter object with two values uint16, uint8 respectively having their own **PC** and now it is possible to find a function in a particular configuration like (A && B && !C).
Figure 4.14: Function with two different parameter types
Extension of this use case was to find the functions with a particular parameter on a particular position as there can be more than one type of parameters for example, find a function with second parameter of type uint16. Due to the concept of keeping PC as local as possible this question cannot be answered just by iterating over all the parameters. So the idea was to flatten out the variants of each of the object and create every possible variant of it. This was repeated until no more variants were there. Finally, satisfiability of each of the variant was then checked and the variant that matched the condition was found.

An example of this can be seen in Figure 4.14 which depicts a function A with two parameters a and b. Here, b can be either of type short or int. Searching for a function with second parameter type short is not a trivial task. Hence, it cannot be solved just by iterating over all the parameters. If the parameters are not of more than two types it is possible to just iterate over them and find a particular parameter that has to be found. So, for the implementation of this part the context would be fed all the PCs that are encountered. By doing this the whole object in that PC is returned. If the context is fed the PC !D, then it will result in the XML shown in Figure 4.15. Similarly all the other PCs are evaluated.

```
<object kind="func" name="funcA">
  <object kind="Parameter" name="a">
    <prop name="Type"><value>int</value></prop>
  </object>
  <object kind="Parameter" name="b">
    <prop name="Type"><value>short</value></prop>
  </object>
  <object kind="Parameter" name="c">
    <prop name="Type">
      <value>int
      <pc>
        // E
      </pc>
    </value>
  </object>
</object>
```

**Figure 4.15:** Result when context is fed with PC equals !D

Depending upon what configuration of D and E is set results will be according to Table 4.2. While searching for all the possibilities with second parameter type short, it will result in last two configurations shown in the table.

A visitor implementation is written to visit a relation, property or an object node and to identify the PC inside it. See Figure 4.16. If a property or a relation node is visited by the visitor and
### Table 4.2: Parameter types in different configurations of D and E

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>(int a, int b)</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>(int a, int b)</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>(int a, short b, int c)</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>(int a, short b)</td>
</tr>
</tbody>
</table>

it has a PC that is a contradiction, nothing is returned but if it was a tautology, a whole new property is returned. For this, variants inside a particular object are multiplied. After that, the PC is fed to the context one by one. Once the context has a PC it is branched and all the other PCs are removed. Although, this part is not fully implemented, a proof that it can be implemented has been done in the implementation.

![Figure 4.16: Visitor for relation, property and object](image)
5 Verification

Testing is done to provide stakeholders the information about the quality of the system under test. It helps in detecting the difference between the achieved and expected implementation. Testing helps the developer to make sure that the system is built as per required standards. Verification is the process that is done to make sure the product or the component developed satisfies the conditions imposed at the start of the development phase. After the completion of implementation phase, verification for the implemented use cases was carried out to check the same.

5.1 Testing methods and levels

5.1.1 Testing methods

This subsection briefly describes different methods that can be used for software testing [21]. Some of these methods were used later for verification. Detailed description for these is given in further subsections.

- Black-Box Testing
  As the name suggests the application that is being tested with this method is a black box for the tester. This type of testing does not have any access to the source code and is done without any knowledge of the internal working. A tester may typically interact with just the user interface by giving inputs and inspecting outputs without having any knowledge about the system architecture or even how the inputs are worked upon. There are some advantages and disadvantages of this kind of a test:

  **Advantages**
  - It is suited for the systems where code access is not possible or where the code segment is too large
  - Separates the perspective of user from developer

  **Disadvantages**
  - Results in limited, blind coverage as only a number of test scenarios are tested and tester cannot target specific code segments because the code is not known
– Tester has a limited knowledge about an application hence leading too inefficient testing

• White-Box Testing

It is also known as structural, glass box, clear box or open box testing. This type of testing needs a knowledge of all the internal working as it is a comprehensive observation of the structure and internal logic of the source code. Some advantages and disadvantages of this type of testing are listed below:

**Advantages**

– Supports in code optimization

– High test coverage is reached as the tester has knowledge about the code and unnecessary lines from the code can be removed as they could produce defects

**Disadvantages**

– Costly as compared to black box testing

– It requires code analysers and debugging tools as it is not done efficiently if carried out manually

• Grey-Box Testing

It is also called translucent testing. This type of testing is in between white and black box testing as there is a limited knowledge about the source code here. In contrast with the black box where tester has no knowledge about the internal structure and working of the application here the tester has access to the database and design documents if there is any.

**Advantages**

– It incorporates the advantages of both black and white box testing

– Testing here is done from the user’s point of view

**Disadvantages**

– Time consuming and extensive approach

– Redundant test can be performed if the developer has already run a test case
5.1.2 Testing levels

These levels are also mentioned in the V-Model shown in Figure 1.2. There are four testing levels that we come across after developing any product:

- **Unit Testing**
  
  As the name suggests this type of testing is done by isolating each part of the program and show that all the separate parts are correct in terms of functionality and requirement. This type of testing is performed by developers. It is also known as module testing as each part is tested separately.

- **Integration Testing**
  
  Integration testing is where the combined parts of an application can be tested. It can be done in two ways namely bottom up and top down integration testing. Bottom up testing usually starts with unit testing at first followed by higher level combinations of units. Top down testing starts with higher level modules which are tested first and proceeds with lower level modules that are tested afterwards.

- **System Testing**
  
  This tests the system as a whole. Once all the components are integrated the system as a whole is tested thoroughly to see whether it meets all the stated quality standards or not.

- **Acceptance Testing**
  
  Carried out by the quality assurance team it is the most important type of testing. These type of tests are not only intended to point out simple spelling mistakes or interface gaps but also to reveal any software bugs that will lead to major errors or system crashes in the application.

5.2 Verified Use Cases

The use cases that were verified with respect to each of the goals are as follows:

**Use case 1**

Main goal of this use case was to have a possibility for verification of variants in the same artifact. We took a design artifact and a function inside it which calls another function in the same artifact which is depicted by some relation node where a relation can be `trace, use` etc. for the testing of this use case we included some presence condition in the relation itself i.e. a relation itself is available under a certain condition. See Figure 5.1. To verify that a relation is only available if a function itself is available we performed an AND operation between the PC of the function and relation. We also verified whether the PC of the relation in both the functions is the same or not.
Secondly, we tested a scenario where a function having a relation with one or more functions. See Figure 5.1. In both the cases it behaved the same, if it has a relation to more than one function then the AND operation is performed for all the relations and respective functions. All the verified checks mentioned above are done manually and are successful.

```
<object kind="Function" name="funcB" id="funcB">
  <prop name="Id" kind="Id">funcB</prop>
  <rel kind="CallBy" objKind="Function" objName="funcA">
    <pc>
      <operator kind="unary" name="defined">
        <Identifier name="FEAT_Z"/>
      </operator>
    </pc>
  </rel>
  <rel kind="CallBy" objKind="Function" objName="funcD">
    <pc>
      <operator kind="binary" name="equals">
        <Identifier name="FEAT_X"/>
        <Identifier name="OFF"/>
      </operator>
    </pc>
  </rel>
</object>
```

Figure 5.1: Relation under a condition and a function with relation to two other functions

Use case 2

Prime objective of this use case was to have consistency between different artifacts and a possibility of verification checks between them. Two artifacts were taken into account and it was verified whether the information between them is consistent or not. If a requirement and a function are related to each other and the relation itself is present under a certain condition but a trace is usually always present it does not have any PC attached to it. So, for this relation to be present the function itself should be present. Implies operator was used between the requirement PC and the function PC and it is proved whether both of the relations are a tautology or a contradiction to solve the above stated use case. In both the cases the results were successfully displayed. Some of the additional checks were made in order to see whether the implementation was functional. These checks are listed below:
• **A function traces to two requirements**

There are some instances when a given function is related to two requirements then the particular function with both the requirements was checked to see which one will work and which might not work by taking all the relations of a particular function into account and doing the same as above.

• **Two functions trace to the same requirement**

When two different functions trace to the same requirement then a requirement should contain the traces to both the functions as well. There was an example that was manually made and added to the VSF file to check how does the implementation react to this usage scenario. See Figure 5.2. It reacted normally as both of the functions presence conditions were checked with the same requirement.

```xml
<object kind="Function" name="funcB" id="funcB">
  <prop name="Id" kind="Id">funcB</prop>
  <rel kind="Trace" objKind="CREQ" objId="CREQ-103749"/>
  <pc>
    <operator kind="binary" name="and">
      <Identifier name="FEAT_F"/>
      <Identifier name="FEAT_W"/>
    </operator>
  </pc>
</object>

<object kind="Function" name="funcC" id="funcC">
  <prop name="Id" kind="Id">funcC</prop>
  <rel kind="Trace" objKind="CREQ" objId="CREQ-103749"/>
  <pc>
    <operator kind="binary" name="or">
      <Identifier name="FEAT_Z"/>
      <Identifier name="FEAT_V"/>
    </operator>
  </pc>
</object>
```

**Figure 5.2:** Two functions tracing to the same requirement example

• **Requirement affects the function call**

Consider a relation from requirement to function where the relation itself has variants according to the configuration that is set. Depending upon the PC whether its true or false the function calls from the requirements were made.

**Use case 3**
Primary intention of this use case was to automate the searching of information from various artifacts. This was possible after implementation. For example, if a function with a combination of a parameter type int and a return type void is to be searched, so that we can verify which variants result in which combinations of parameter types and return types. There were some small checks made to see whether the implementation is thorough:

- **Searching with more than one parameter type**
  More than one parameter type is manually added to a particular function in the VSF file to see how the program reacts to it. The program worked as expected and it took all the parameters that are listed there and found a parameter type that was searched. All those selected parameter types were taken into consideration and the AND of the function and parameter types and return types were done to see which particular signature is available under which configuration.

- **Searching the combination of a parameter and a return type**
  After it is possible to find the parameters and return types separately, the combination of a particular parameter and return type is searched. It is done manually and works as expected. It returns the functions that are in a particular configuration. Searching with parameter type Unit8 and return type void was carried out in this part of the use case.

One part of the use case is to find parameters of specific type at a particular position. As told earlier, it is not possible to iterate over the parameters directly by using foreach loop as the parameters can be multiple. The concept for this part of use case i.e. how can it be done was thought of but wasn’t implemented.

### 5.3 Evaluation

Evaluation phase helps us to check the results and evaluate the system that is under test, which further helps us to decide whether the software product or a component has passed the tests or not.

Previously VSF as a generic format was easily extendable but did not support the variant information in it. All the three use cases stated in the above section were completely done except the third use case (third part) which was not complete but an idea of how it should be done has been thought of. After extending the VSF to support variant information in different artifacts some goals were to be achieved. A brief evaluation of the thesis with respect to the objectives of the thesis are listed below:

**Format should provide analysis of code in one variant**

Verification between the configuration variants in a particular artifact is possible. To prove this artifact CDD was taken into consideration and some examples of this use case were implemented. One of the example is a function that has parameters and return type, but the parameters
and return type cannot be present if the function itself is not there. This is one of the scenario to prove the consistency checks within one artifact. In verification, some examples like the one stated above are used to prove this objective is achieved and it is possible to analyze the variants in one artifact.

**Format should provide analysis of code in multiple variants**

Consistency checks within different artifacts have been automated. Artifact CDD and requirement were taken into consideration for the proof of this use case. It was implemented using some examples, for instance, a requirement in the Component Requirement (CREQ) artifact has a TraceBy relation to a function in the CDD artifact. Function in the CDD artifact should also contain a trace relation to the requirement in CREQ and the presence condition of the requirement should imply the presence condition in the lower level design artifact which is done by constraints. Now, it is possible to analyze variants in different artifacts as some examples have been tested to prove this.

**Format should provide possibility for searching of variant information**

This use case has helped in automating the search of information in various artifacts. For example, take CDD artifact into consideration. If we want to find a variant of a particular function with return type ‘int’ and parameter type ‘void’ it is possible to do this automatically instead of doing it manually which is rather inefficient. By implementing this use case we are able to find a function in a particular configuration. This use case is not complete because the last part of it is still to be implemented although the concept that it is possible has been thought of.
6 Conclusion

The main objective of this thesis was to find a method for storing the variant information present in different artifacts. It deals with variants, where and in what form the variants are present, how can the variant information be handled and stored in a particular format. The focus of the research was just the configuration variants. Information about what are configuration variants, how to handle variants in general and store them at various places have been studied extensively. The V-Model has been considered at the start of this thesis as a development process. All the artifacts listed in the V-Model namely requirement, design and implementation are considered. The research here had two aspects: extension of format and the variant data present in it.

Main focus here are the problems that were not covered by the present approach. First part of the thesis included the understanding of the VSF format and how can it be extended. Conceptualization focussed on extending the intermediate format VSF to support the variant information present i.e. formalize the documentation of variant information in different artifacts at Vector Informatik GmbH. Once the intermediate format had been extended different types of checks had to be automated. Some of the checks were: Analysis of code in multiple variants, for example, if there is a function which has two variants, we need to mention their return types and what parameters that exist when a particular type of variant is present. Analysis of multiple variants, for instance, checking whether a variable is described before it is accessed or not.

The research results have been confirmed to be positive in terms of usefulness and usage by Vector Informatik GmbH. The important part here is that VSF being a generic format can be extended easily to support different needs and here it was extended to support all the variant information present in the different artifacts. This can further be extended to support variant information at different levels more widely.
7 Future Work

Research and development can always be extended to be more powerful and improve the current approaches. Following are the extensions that can be made to this research in future. This is a proof of concept of what can be done with the variant information present in different artifacts further implementation will be done as a part of future work. Some steps that should be applied to the given solution to make it available for Vector Informatik GmbH are listed below.

7.1 Extraction of variant information in different artifacts

Requirement

With the applied concept, we know how to store the variant information in every artifact but the extraction of data is still to be done. Currently, the requirement is present in the informal textual manner. The description of the variants should be formalized so that it is possible to parse the textual representation by definition of a formal language.

Design

Here, the extraction of variant information from the UML models is to be done as CAD is represented in that form, so a transformation from the models to an internal model shall be done in order to extract the variant information present there. The variant information contained in the CDD artifact, which is also represented in a textual manner is to be parsed by a simple textual parser. The difference between the representation of variant information in requirement and CDD is that CDD has formal representation of all the information so there is no need of making it formal in order to parse.

Implementation

For the extraction of variant information present in the C code. We need to parse the implementation. Some of the ways to do it are listed below:

- **SUPER C**

  In a C code there are basically two languages that need to be processed i.e The C code itself and the preprocessor statements. This particular future work deals with parsing of both the code and preprocessor statements efficiently by introducing a preprocessor which resolves the includes and macros but leaves the static conditionals for preserving variability. Secondly, a parser that forms an AST with choice nodes for static conditions.
This can be done by forking parsers when they encounter conditions and then merging them later after the condition is ended [22].

• **Typechef approach**

The second thing which can be done to extend the current approach is to implement a variability aware parser that will be able to parse the code that is not preprocessed in practical time. It should not only provide the possibility for detection of syntax errors but also should contribute to further analysis like variability aware type checking. It should also include a variability aware system that type checks the code and reports errors in specific feature combinations [23].

Lastly, to make it completely usable the integration of the implementation the current development environment must be done. To perform architecture verification the extraction of variant information should be possible, so that satisfiability and unsatisfiability of variants can be proved. Also, the possibility of static checks is available once this extension is done.
Bibliography


A Appendix

A.1 Examples of VSF and extended VSF

A.1.1 Example of VSF file format

```xml
<!--Intermediate format example based on real data:-->
<component>
    <object id="21" kind="SequenceDiagram">
        <prop name="name">SeqDiagram</prop>
    </object>
    <object id="31" kind="HeaderFile">
        <prop name="name">FileName</prop>
        <prop name="context"></prop>
    </object>
    <object id="1" kind="Routine">
        <prop name="id">1</prop>
        <prop name="name">A</prop>
        <prop name="desc">TEXT</prop>
        <prop name="callcontext"></prop>
        <prop name="details"></prop>
        <prop name="lockcontext"></prop>
        <prop name="reentrant"></prop>
        <prop name="returntype"></prop>
        <prop name="returntypedesc"></prop>
        <prop name="serviceid"></prop>
        <prop name="stereotype"></prop>
        <prop name="synchron"></prop>
        <prop name="preconditions"></prop>
        <rel id="rel1" kind="use" dir="o" relobj="2">
            <rel kind="contain" relatedId="ParamA1" dir="" id="rel123">
                <object id="123" kind="Parameter">
                    <prop name="name">ParamA</prop>
                </object>
            </rel>
        </rel>
    </object>
</component>
```
A.1 Examples of VSF and extended VSF

```
<prop name="desc">TEXT</prop>
<prop name="type">TEXT</prop>
<prop name="kind">TEXT</prop>
</object>
<object id="345" kind="Variable">
  <prop name="name">VarA</prop>
  <prop name="type">TEXT</prop>
  <rel kind="define" relatedId="3" dir="o" id="rel3">
    <rel kind="use" relatedId="1" dir="i" id="rel2">
      ...
    </rel>
  </rel>
</object>
<object id="2" kind="TypeDefinition">
  <prop name="type">TypeA</prop>
  <prop name="typedesc">TEXT</prop>
  <prop name="typemember">TEXT</prop>
  <prop name="typememberdesc">TEXT</prop>
  <rel kind="use" relatedId="1" dir="i" id="rell">
    ...
  </rel>
</object>
<object id="3" kind="Cfile">
  <prop name="desc">TEXT</prop>
  <prop name="includeheader">TEXT</prop>
  <prop name="variable">TEXT</prop>
  <rel kind="use" relatedId="1" dir="i" id="rell">
    ...
  </rel>
</object>
</component>
```
A.1.2 VSF file with extension

```xml
<?xml version="1.0" encoding="UTF-8"?>
<vsf>
  <prop name="Origin">UnderstandC2Vsf UC 770, Udb2Vsf 2016-03-14</prop>
  <prop name="CreateDate">2016-05-02</prop>
  <prop name="Author">vislsi</prop>
  <object kind="Section" name="CRS">
    <object kind="CREQ" name="CREQ-103749" id="CREQ-103749">
      <rel kind="TraceBy" objKind="Function" objName="funcC"/>
      <rel kind="TraceBy" objKind="Function" objName="funcB"/>
      <prop name="Type">Functional</prop>
      <prop name="SafetyLevel">ASIL_D</prop>
      <prop name="Desc"> <![CDATA[<p>there is an and operator between a switch x and a switch y</p>]]></prop>
    </object>
    <pc>
      <operator kind="unary" name="defined">
        <Identifier name="FEAT_M"/>
      </operator>
    </pc>
  </object>
  <object kind="CREQ" name="CREQ-10374" id="CREQ-10374">
    <rel kind="TraceBy" objKind="Function" objName="funcC"/>
    <rel kind="TraceBy" objKind="Function" objName="funcB"/>
    <prop name="Type">Structural</prop>
    <prop name="SafetyLevel">ASIL_A</prop>
    <prop name="Desc"> <![CDATA[<p>there is an equals operator between a switch A and 12</p>]]></prop>
  </object>
</object>
<object kind="Section" name="CDD">
  <object kind="Component" name="usecase1">
  </object>
</object>
```
<object kind="Section" name="Files">
  <object kind="Section" name="Constraint">
    <prop name="expression">
      <operator kind="binary" name="implies">
        <Identifier name="FEAT_D"/>
        <Identifier name="ON"/>
      </operator>
      <operator kind="binary" name="equals">
        <Identifier name="FEAT_E"/>
        <Identifier name="OFF"/>
      </operator>
    </prop>
  </object>
  <object kind="Section" name="Functions">
    <object kind="Function" name="funcB" id="funcB">
      <prop name="Id" kind="Id">funcB</prop>
      <prop name="Scope">Global</prop>
      <object kind="Parameter" name="LinTrcvIndex">
        <prop name="Type"><value>uint8</value></prop>
        <pc>
          <operator kind="binary" name="equals">
            <Identifier name="FEAT_T"/>
            <Identifier name="ON"/>
          </operator>
        </pc>
        <pc>
          <operator kind="unary" name="defined">
            <Identifier name="FEAT_M"/>
          </operator>
        </pc>
      </object>
      <object kind="Parameter" name="LinTrcvIndex2">
        <prop name="Type"><value>uint8</value></prop>
        <pc>
          <operator kind="binary" name="equals">
            <Identifier name="FEAT_T"/>
            <Identifier name="OFF"/>
          </operator>
        </pc>
      </object>
    </object>
  </object>
</object>
A.1 Examples of VSF and extended VSF

```xml
<pc>
  <operator kind="binary" name="and">
    <Identifier name="FEAT_A"/>
    <Identifier name= "FEAT_B" />
  </operator>
</pc>
<object kind="Return" name="ReturnType">
  <prop name="Type"><value>void</value></prop>
  <prop name="Desc"><![CDATA[]]></prop>
  <operator kind="binary" name="and">
    <operator kind="binary" name="and">
      <Identifier name="FEAT_A"/>
      <Identifier name= "FEAT_P" />
    </operator>
    <operator kind="binary" name="equals">
      <Identifier name="FEAT_T"/>
      <Identifier name= "ON" />
    </operator>
  </operator>
</object>
<rel kind="CallBy" objKind="Function" objName="funcA">
<pc>
  <operator kind="binary" name="and">
    <operator kind="unary" name="defined">
      <Identifier name="FEAT_X"/>
    </operator>
    <operator kind="unary" name="defined">
      <Identifier name="FEAT_Z"/>
    </operator>
  </operator>
</pc>
</rel>
<pc>
  <operator kind="unary" name="defined">
    <Identifier name="FEAT_M"/>
  </operator>
</pc>
</object>
```
A Appendix

A.1 Examples of VSF and extended VSF

```xml
<rel kind="Trace" objKind="CREQ" objName="CREQ-103749"/>

<object kind="Function" name="funcC" id="funcC">
  <prop name="Id" kind="Id">funcC</prop>
  <prop name="Scope">Global</prop>
  <object kind="Parameter" name="LinTrcv">
    <prop name="Type"><value>uint8</value></prop>
    <operator kind="binary" name="and">
      <Identifier name="FEAT_T"/>
      <Identifier name="FEAT_U"/>
    </operator>
  </object>
  <object kind="Parameter" name="LinTrcv2">
    <prop name="Type"><value>uint16</value></prop>
    <operator kind="binary" name="or">
      <Identifier name="FEAT_M"/>
      <Identifier name="FEAT_N"/>
    </operator>
  </object>
  <object kind="Parameter" name="LinTrcv">
    <prop name="Type"><value>void</value></prop>
    <operator kind="binary" name="and">
      <Identifier name="FEAT_X"/>
      <Identifier name="FEAT_U"/>
    </operator>
  </object>
</object>
```
<object kind="Return" name="ReturnType">
  <prop name="Type"><value>void</value></prop>
</object>
<pc>
  <operator kind="binary" name="and">
    <operator kind="unary" name="defined">
      <Identifier name="FEAT_X"/>
    </operator>
    <operator kind="unary" name="defined">
      <Identifier name="FEAT_Y"/>
    </operator>
  </operator>
</pc>
</vsf>
A.1.3 Schema of VSF file with PC

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2009 sp1 (http://www.altova.com) by Vector Employee (Vector Informatik GmbH) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="pc">
    <xs:annotation>
      <xs:documentation>Presence Condition Node</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="operator"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="operator">
    <xs:complexType>
      <xs:choice maxOccurs="2">
        <xs:element ref="operator"/>
        <xs:element name="Identifier">
          <xs:complexType>
            <xs:attribute name="name" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
      <xs:attribute name="kind" use="required"/>
      <xs:attribute name="name" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="vsf">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="object"/>
        <xs:element ref="prop"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="object">
    <xs:complexType>
      <xs:sequence>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="object"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:element name="metric">
  <xs:complexType>
    <xs:attribute name="name" use="required"/>
    <xs:attribute name="value" use="required"/>
  </xs:complexType>
</xs:element>
<xs:element ref="prop"/>
<xs:element ref="rel"/>
<xs:element ref="refFile"/>
<xs:element ref="object"/>
</xs:choice>
<xs:element ref="pc" minOccurs="0"/>
</xs:sequence>
<xs:attribute name="kind" use="required"/>
<xs:attribute name="id"/>
<xs:attribute name="name"/>
</xs:complexType>
</xs:element>
<xs:element name="prop">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0">
      <xs:element name="value">
        <xs:complexType>
          <xs:sequence minOccurs="0">
            <xs:element ref="pc"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element ref="operator"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
<xs:element name="rel">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="prop"/>
        <xs:element ref="rel"/>
      </xs:choice>
    </xs:sequence>
    <xs:element ref="pc" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="kind" use="required"/>
</xs:complexType>
</xs:element>
<xs:attribute name="objKind" use="required"/>
<xs:attribute name="objName" use="required"/>
<xs:attribute name="objId"/>
</xs:complexType>
</xs:element>
<xs:element name="refFile">
<xs:complexType>
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="prop"/>
</xs:choice>
<xs:attribute name="kind" use="required"/>
<xs:attribute name="name" use="required"/>
</xs:complexType>
</xs:element>
</xs:schema>