HERAUSRAGENDE MASTERARBEITEN AM DISC

FACHBEREICH Software Engineering for Embedded Systems

STUDIENGANG Master of Engineering

MASTERARBEIT

Recovery of Configuration Knowledge by Analysis of Variability Code Realizations

AUTOR/IN

Suparna Satheesh Nair

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

DISTANCE AND INDEPENDENT STUDIES CENTER

Declaration

Ich versichere, dass ich diese Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Bangalore, January 22, 2020 Suparna Satheesh Nair

_____ _____

Abstract

Industries use software product lines as a solution to the ever-increasing variety-rich customer requirements for the software products. In order to realize the variability in the product line, several variability realization techniques are used, of which, conditional compilation and execution are more frequently used in practice. This is not without its challenges.

As the product line evolves in space and time, several versions of products are released, thereby increasing the complexity of variability code in an uncontrolled manner. In most cases, there exists no explicit variability model to provide important configuration knowledge, or the variability model and variability code do not synchronize with each other, e.g. important dependencies from the code realizations are not reflected in the variability model.

When the domain experts leave the company, the product configuration knowledge will be lost. New employees will have to be trained on the domain knowledge and are left with the herculean task of tracking the code changes in the variability code for the different versions. They also have to understand the variability code to analyze the impact of code changes and how to adapt them. Overall, that lack of explicit and sound configuration knowledge results in higher efforts during the product configuration and quality assurance. Hence, industries are interested in recovering configuration knowledge via semi-automated analyses of the variability code and the existing product configurations. This Master's thesis investigates the various approaches that can be followed in order to recover existing configuration knowledge. It is an extension of the previous research works on the VITAL approach conducted at TU Kaiserslautern and Fraunhofer IESE. The focus of this research will be the solution space, i.e., the variability realization through variability code mechanisms like conditional compilation/execution. The goal is to analyze the preprocessor directives or respective constructs in programming languages, study respective state of the art advances in recent years and enhance the VITAL analysis method and tool. In particular, identification of configuration parameters, their values and ranges, the constraints and nesting between one parameter to the other are the primary objectives of the research. As secondary goals, visualization of the identified product configuration knowledge in the existing tool and optimization of the algorithms present in the tool will be implemented from the results of the primary goals. For the research, open source libraries and applications will be identified and used for analysis. The work will be guided by real world industrial settings.

Table of Contents

Acknowledgements

This thesis is the result of my distance learning masters' study from Technical University of Kaiserslautern. The thesis would not have been possible without the immense help and support from various people in my personal, academic as well as professional life.

First, I would like to extend my sincere gratitude to Dr.-Ing. Martin Becker, from Fraunhofer Institute of Experimental Software Engineering, Kaiserslautern, who gladly agreed to be my second supervisor for the thesis research. His disciplined way of tackling a problem and the well-organized manner in which the solution ideas are derived has influenced me and steered me in the right direction in my work. I am gratefully indebted for his valuable feedback on this thesis.

I would also like to thank my first supervisor Prof. Dr.-Ing. Peter Liggesmeyer, from Technical University of Kaiserslautern and the experts who were involved in the work reviews and presentations, whose feedback were of immense help to improve the thesis work.

Last but not the least, I express my profound gratitude to my family members, colleagues and friends for providing me with unfailing support and continuous encouragement throughout my study.

Abbreviations

VP	Variation Point
VPG	Variation Point Group
cpp	C Preprocessor
CV	Code Variant
AST	Abstract Syntax Tree
FST	Feature Structure Tree
PLE	Product Line Engineering
LLVM	Low Level Virtual Machine
SIS	Software Intensive System
SVD	Singular Value Decomposition
FCA	Formal Concept Analysis
VEL	Variability Exchange Language
ReqIf	Requirement Interchange Format
MOF	Meta-Object Facility

List of Tables

List of Figures

1 Introduction

With the ever-increasing complexity of software products due to increasing demand for customized solutions, there is a growing trend towards developing systems that cater to the myriad needs of consumers by strategically reusing software artefacts. Software product line engineering is a class of engineering approaches, which uses systematic reuse to solve this challenge. It is the development of a set of products from a reusable set of assets following a common architecture and a predefined plan [Clements and Northrop, 2001]. Since software product line engineering has proven successful for the mass production of software systems, the number of software variants for a software product line becomes unmanageably high. Variability management is a central component of every product line engineering approach. It is a concern that arises in product line engineering throughout all lifecycle phases.

There are many challenges to efficient variant management in large product lines. Firstly, if product derivation for such a huge product line is done manually, it will lead to human errors, consumes time and prove extremely difficult, compromising on the benefits of product line adoption. As software product evolves over space and time, maintenance becomes a challenging task. Furthermore, software systems may undergo variability erosion and the variability model becomes untraceable. Hence, disciplined and systematic approaches are needed to cope with the complexity of developing and maintaining sets of product variants.

In the next section, the main motivation of the thesis will be presented, followed by the problems that are addressed in the research. The specific research questions will be identified and defined in the subsequent sections. The scope of the research will be determined in the later section, followed by the outline of the thesis.

1.1 Motivation

In software product line engineering, the common and variable characteristics are specified and managed in domain engineering (also called family engineering). The domain engineering provides a set of core/reusable assets, called domain assets, which can be used to derive product line members, or specific applications. The differences among various product line members are introduced through the adaptation capabilities in domain assets, and these differences are called variabilities. The variability information is represented through variability models, which is used for deriving the product configurations. Variability in problem space is defined by variability models and is realized in solution

space through different mechanisms. These mechanisms range from modern paradigms like feature-oriented programming to traditional ways like conditional compilation, which have been increasingly adopted in the development of large and complex software systems.

One of the biggest challenges in variability realizations is software maintenance, primarily because of the amount of manual effort required to comprehend the different variability representations. The evolution of product lines in space and time results in an explosion of product variants, which makes it unmanageable. In such cases, the product configuration information may have been poorly defined, or incomplete, missing out on the important dependencies. Thus product configuration in problem space may not accurately represent the code realization in solution space. In addition, the transfer of domain knowledge to new resources in an industry could pose a challenge if the product configuration information is lacking in detail.

The magnitude by which code complexity increases due to variability realizations can be illustrated by exploring the different open source code bases available, like FreeRTOS, OpenCV, Linux kernel etc. Studies about the evolution of variability code in the FreeRTOS product line in terms of variability have shown that variability specification in problem space and its realization in solution space has increased considerably over years [Zhang, 2015]. In order to analyze the depths at which variability is realized in the code and how it is distributed across the source code files in a product variant and its dependencies, it is pertinent to have disciplined approaches. Several studies have been conducted to extract configuration knowledge from product variants [Shatnawi, Seriai, and Sahraoui, 2016][Zhang, 2015], which uses association mining, propositional formulae, to name a few.

The focus of this research is in extracting configuration and variability information from product variants which makes use of a specific mechanism for realizing variability in code, namely conditional compilation. The thesis is motivated by the previous works of research conducted at TU Kaiserslautern and Fraunhofer IESE [Zhang, 2015] and aims at improving the VITAL tooling approach formulated by the researchers for extracting important variability information. The next section of this chapter is dedicated to the problem statement and the different goals of this research.

1.2 Problem Statement

This section presents the research problems addressed in this thesis. There are problems that are identified in both problem space and solution space. However, the focus of this thesis will be on addressing the problems in the solution space.

- P1: Increasing complexity of variability code. Considering the variability realization mechanism using conditional compilation, increasing variable features would imply several different #if and #ifdef blocks, which are often nested and tangled. Additionally, there could also be conditional definitions and conditional inclusions in the complex code realization. This makes traceability of variability code realization back to the variability model very cumbersome and affects the maintainability of code.

- P2: Absence of explicit variability model in industries. Industrial experience shows that many industrial applications do not have a fully defined variability model, or it is limited to a list of features. These models do not reflect the inter-dependency between the features, or the hierarchies between parent and child features. This results in developers spending hours of effort in understanding the structure and features of the variability realizations while making adaptations to code, through reverse-engineering. This is a herculean task, often accompanied by human-prone errors.

- P3: Difficulty in tracking code changes in variability code for different variants. This problem is a direct consequence of P2. The limited variability model, if present, loses synchronization with the actual realization of variability due to the increasing complexity and number of variant code realizations. Often, it is an arduous task to manually document the dependencies and keep them up-to-date with every newly developed variant of the product line.

- P4: Lack of explicit and sound configuration knowledge. Configuration knowledge generally vests with domain experts and are often undocumented. This will prove challenging if the domain experts leave the industry and the task of adapting the code or deriving another variant of the product line is left to the developers who are newly introduced to the code. Adding or removing features would require in-depth knowledge of the domain and the software system under consideration, to assess the impact of change. The dependencies between features would lead to potentially error-prone configuration if done manually.

As a result of the above problems, the economic benefits of the software product line diminishes over time [Ganesan, Muthig, and Yoshimura, 2006] and the return of investment starts to decrease. Beyond this point, it does not make sense to further develop variants from the product line, and this results in a waste of investment, effort and resources.

1.3 Research Method

This section of the thesis presents the research methodology, namely the defined research questions, research procedure and solution ideas.

1.3.1 Research Questions

The following research questions were identified to address the problems mentioned in the previous section.

- RQ1. How can we semi-automatically extract feature dependencies from an existing product line with variability realizations?
- RQ2. How can we semi-automatically extract variability code elements from an existing product line with variability realizations?
- RQ3. How can we analyze the variability realizations to trace the feature dependencies in an insightful manner?

From the above research questions and the previously defined research problems, the following goals have been identified for this thesis:

- G1. Understand state of the practice and art on recovery of configuration knowledge from variability code realizations
- G2. Enhance Fraunhofer IESE's VITAL approach and tooling, especially with respect to:
  - G2.1. Support the identification of parameter ranges and default values
  - G2.2. Support the identification of hierarchical dependencies
  - G2.3. Support the identification of range constraints, e.g. if A AND B then C = 1
  - G2.4. Support also conditional execution as a variability mechanism
- G3. Show the feasibility of the approach along some open source systems

To achieve this goal, the following major activities were identified:

- A1. Conduct a literature review on recovery of configuration knowledge from variability code. Identify novel approaches that can enhance the VITAL approach.
- A2. Identify open source systems that can be used as analysis subjects
- A3. Conduct feasibility studies on novel analysis approaches

- A4. Enhance the VITAL approach and tooling to support the analyses mentioned above
- A5. Show the feasibility of the improved approaches along with some open source systems and document the approaches with them.

1.3.2 Research Procedure

From the previous research at TU Kaiserslautern and Fraunhofer IESE, the VITAL tooling has been defined and formulated [Zhang, 2015]. The solution idea for this thesis is based on the improved VITAL Process Workflow, as illustrated in Figure 1.

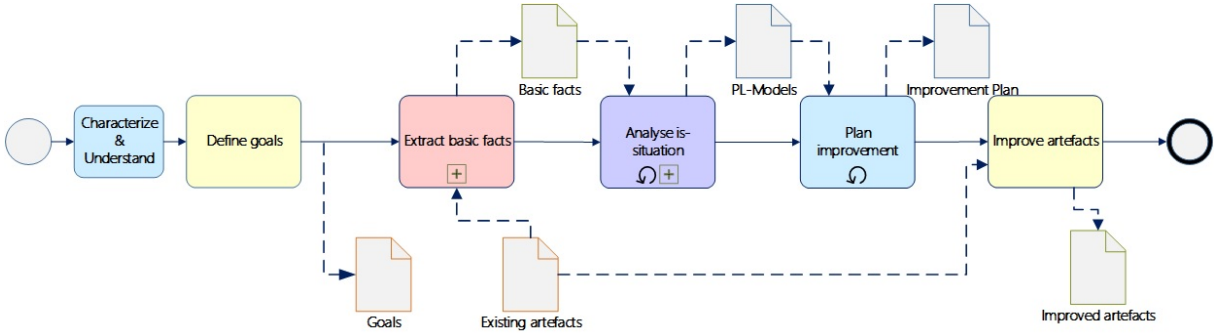

Figure 1: VITAL Process Workflow

We will be referring to this Process Workflow throughout the thesis for the different steps in solution implementation. The details on each step will be covered in the corresponding sections (subsection 4.2), which follow.

1.3.3 Solution Idea

The VITAL tooling was developed from earlier works in Fraunhofer IESE [Zhang, 2015]. These are two daughter tools developed as part of the VITAL tool-chain, **VITAL Cfg**, which is a complex feature correlation miner and **VITAL Src**, the source code analyzer. The solution ideas presented below is with respect to the improvements in *VITAL Src*.

In *VITAL Src*, a variability reflection model was formulated, which contains four types of variability elements, i.e., Variability (Var), Variation Point (VP), Code Variant (CV), Variation Point Group (VPG). The tool uses Python modules to parse the variability code realizations and extracts different metrics for the variability reflection model.

The variability realization mechanism under consideration in this research is Conditional Compilation. Later in the sections, ideas to apply the same techniques for conditional

execution is also discussed. In conditional compilation, a Var is implemented as a macro constant and a VP is implemented using #if directives, like #if, #ifdef, #else, #elif and #ifndef. Code variant is the code fragment that is enclosed within the VPs. Variation Point Group (VPG) is a group of VPs with logically equivalent #ifdef statements.

In addition to this, conditional definitions (#define statements inside of conditional compilation blocks like #if) and hierarchical dependencies between the VPs (nested conditional compilation expressions) also exist. One area of improvement in VITAL 1.0 tool is in modularizing the different scripts that perform the above-mentioned functionalities. If the monolithic code is broken down into manageable modules, it can be invoked independently based on the requirement. For example, if the user only needs to find the conditional compilation information, she/he needs to invoke only that specific module, and not the entire modules. The idea is to develop a tool-chain of simple utilities that has interfaces for easy plotting, analysis and storage.

Figure 2 and Figure 3 show the current VITAL 1.0 solution and the proposed solution. More details on each of the phases will be presented in the sections that follow.

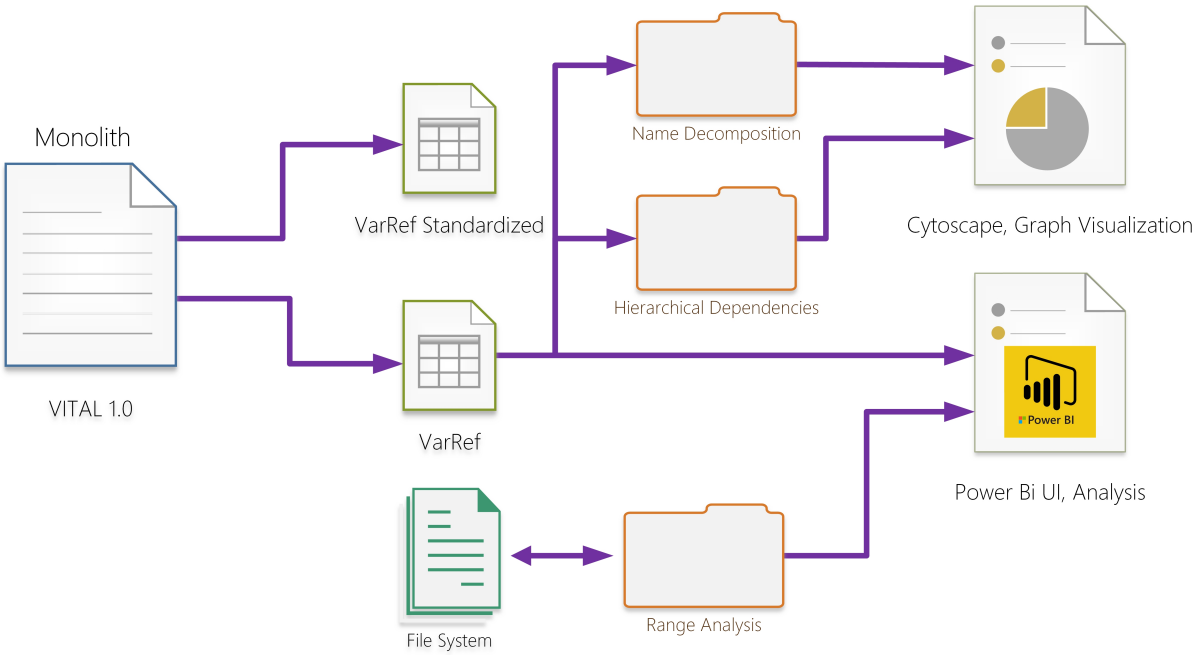

Figure 2: Block diagram of VITAL 1.0

Another area of improvement for VITAL 1.0 is to enhance the parsing algorithms. There are open-source libraries available that performs the C pre-processor parsing, which can be utilized and modified to suit the needs of the VITAL tool. This would ensure that the underlying algorithms are stable, under active development and can be easily replaced

when another enhanced algorithm is available from the open-source community. The interfaces of the modules in the proposed solution are developed such that they can be easily plugged into other extended tool-chains.

Several open-source libraries have been studied, analyzed and experimented in this thesis to formulate the second version of the VITAL tool. Thus the tool is expanded to use more libraries with enhanced algorithms. Enhancement and simplifications have been made in other functionalities offered by VITAL 1.0 and new functionalities have been incorporated.

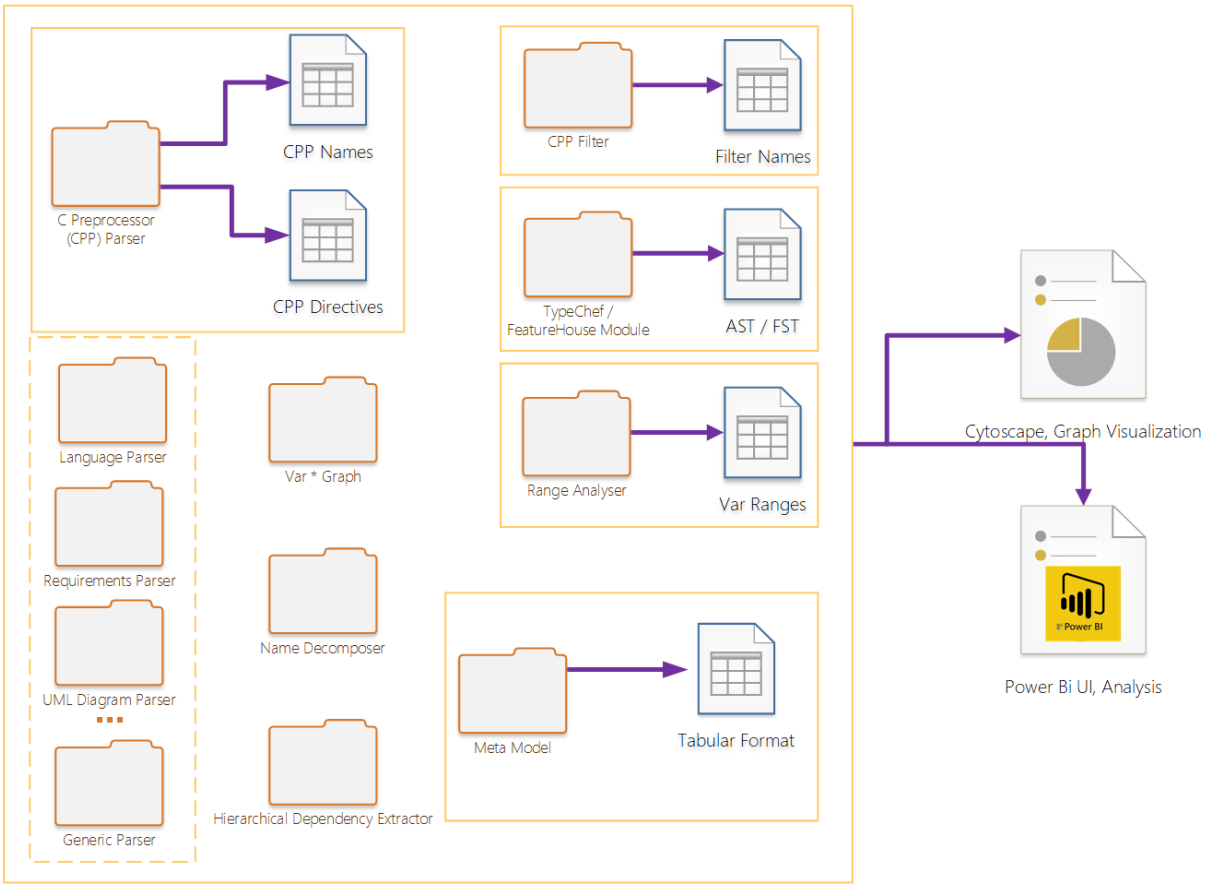

Figure 3: Proposal for VITAL 2.0

To summarize, the proposed solution is to develop independent, complete modules that can be invoked by the user over a terminal/command prompt or via a simple Graphical User Interface.

1.4 Contributions

The key contributions of the thesis include:

- **Literature review on recent achievements in the field**:
  Several state of the art practices has been explored in the area of feature extraction from variability models. Since conditional compilation is the key variability realization methodology in focus for this thesis, many open-source libraries and C pre-processor parser libraries have been experimented with, which provided insightful results and detailed understanding of the advantages and shortcomings of the different approaches. Some of the approaches that were explored are:
  - Study of TypeChef parser
  - Study of ANTLR parser generator
  - Study of FeatureHouse
  - Study of LLVM and Clang compiler
  - Study of CPIP library in Python
  - Study of PCPP library in Python
- **Improved VITAL approach:**
  The VITAL tool has been improved to incorporate the newer libraries and to reinforce the underlying parser algorithms. The tool has evolved to form a tool-chain with a set of modular utilities to perform the specific variability-aware analysis on the variability code realizations. These modules can be chained to form specific applications as per the need of the industry. The interfaces of these modules allow easy visualization, analysis and storage. The result of each module is a *pandas DataFrame*, which is a 2-dimensional labelled data structure in tabular form, with rows and columns, that can be used for a wide range of manipulations for further extension.
- **Example cases for the application of VITAL**:
  Results of the evaluation of improved VITAL tool on different open-source libraries like FreeRTOS, Linux Kernel and OpenCV are provided for future extension of this research.
- **Automatic extraction of feature Correlations in solution space:**
  The studies performed in previous work [Zhang, 2015] concerning the VITAL tool have identified a mechanism for automatic feature correlation extraction from problem space using data mining techniques, i.e., from existing product configurations.

An idea has been proposed which derives clusters of dependent features from the code realizations which can be used for automatic extraction of feature-dependencies in solution space, from the core code assets.

1.5 Research Scope and Limitations

The research methodologies presented in this thesis deals with specific problems in scope, in a specific context. Currently, conditional compilation is the main focus of this thesis and an attempt is made in generalizing this to other techniques of variability realization as well. The variation points and variation point groups are extracted with no or very limited domain knowledge on the application/variability realization code under study. Hence, there could be many VPs that are false positives, i.e., not exactly a feature from a variability perspective.

The solution idea is not one that completely removes the step of a domain expert; this thesis proposes a semi-automated way of extracting the variability information from the code realizations. However, an attempt has been made to automatically cluster the dependencies of the source code to form features, which could be used as a potential tool in filling this gap. The details of this experiment have been explained in later chapters.

This research is an attempt to standardize the interfaces for invoking various analysis commands for acquiring more insights on variability code realizations, feature interdependencies and extraction of product configuration information. Further fine-tuning and pruning of the results are needed to create specific applications tuned to the needs of the industry.

1.6 Thesis Structure

The remainder of this thesis is organized as follows.

Chapter 2 presents underlying concepts about product line engineering, introduces the necessary terminology and presents the conducted literature.

Chapter 3 describes the planning, preparation and execution of the experiments, including the addressed research questions and solution ideas. The various open-source libraries and inferences from them are detailed. Also, the VITAL Process Workflow is explained in detail, with reference to the studies conducted. The implementation of the improved VITAL 2 tool and the various aspects of its design and development are presented next.

Chapter 4 presents the results of the evaluation. Application of VITAL 2 toolchain and its performance profiling and results are detailed.

Finally, chapter 5 draws concluding remarks and presents the opportunities for further investigations.

2 Foundation

This chapter presents the foundation of the thesis. First, a general overview of product line engineering is given and related terms are introduced. This is followed by a discussion of variability specification and realization approaches. Finally, a detailed description of the literature studies conducted and the insights acquired from them are presented.

2.1 Background

2.1.1 Software Product Lines

Software reuse has been used as means to meet the needs of variety-rich product derivation, at the same time, providing strategic advantage and economic value. This is achieved through the mass customization of products. This reuse however, does come with a cost, and in order to make reuse approaches more efficient, the required adaptation support needs to be provided for necessary and foreseen changes.

Reuse Approaches
Several reuse approaches are followed in industrial practice. The natural but rather ad hoc reuse approach, *Clone and Own (or Copy and Modify)*, in which new product variants are based on previous product variants, can be found quite often. Even though this approach has the quick reuse benefits of reduced cost and time, challenges arise in its maintenance. Hence this approach is ideal if variance is small and where there is no necessity to manage and organize these artefacts for future use.

Another approach is to systematically and effectively exploit the reuse potential through specific disciplined techniques. One pivotal tactic to this end is to plan the future reuse and then to optimize the modularization of the reuse building blocks, e.g. via component-based development, service-oriented systems, etc. Domain engineering is one such reuse approach that pioneered the idea of planning and partially developing a similar system, in the same application domain, concurrently [Becker, 2017].

Product Line Engineering
Product line engineering (PLE) combines the principle of domain engineering with reuse-driven application engineering, i.e. it plans reuse strategically and takes care that the reusable assets are used in application engineering in an efficient and effective manner. Software Product Line Engineering is defined as a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of assets in a prescribed

way [Clements and Northrop, 2001].

Product Variant
Simply put, the software product derived from a software product line is called a *product variant*.

2.1.2 Domain and Application Engineering

The life cycle consisting of a set of processes for specifying and managing the commonality and variability of a software product line is called domain engineering or family engineering. The output of domain engineering is reusable artefacts, called domain assets.

Application engineering produces application assets, which are derived by the reuse of domain assets. These application assets are used to produce a specific product or variant in the product line, by using the application assets in conformance with the domain model, and by binding the variability of the specific platform. Figure 4 illustrates the life cycle of a product line.

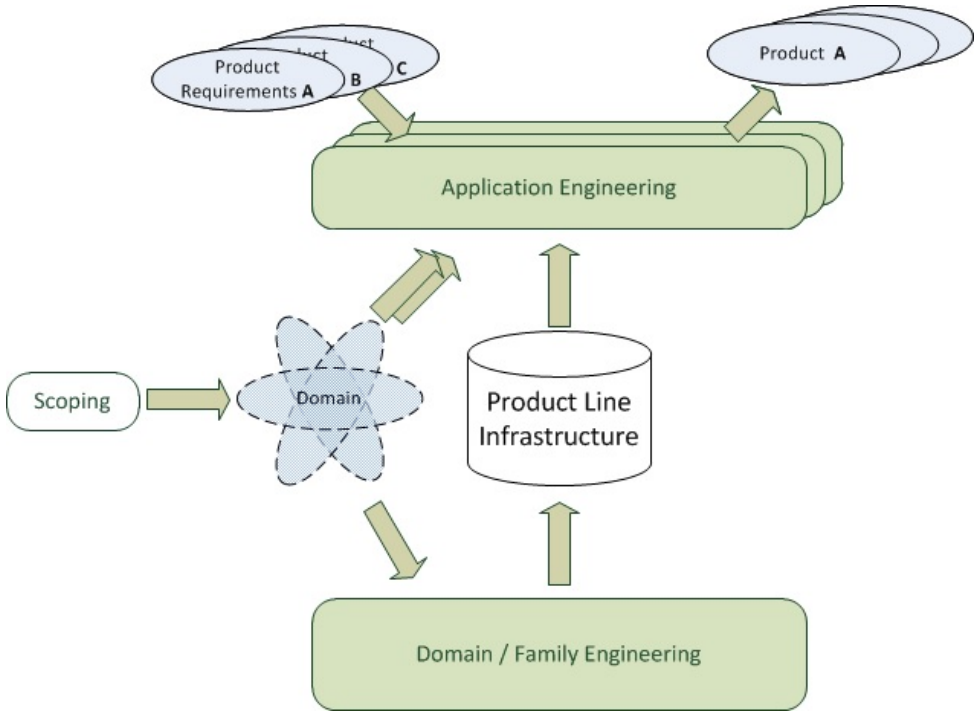

Figure 4: Product Line Engineering Life Cycle

Problem Space and Solution Space
The *problem space* consists of domain-specific abstractions, which describe the requirements of a software system and its intended behaviour. Domain analysis takes place in the

problem space, and results are documented as features. The *solution space*, on the other hand comprises implementation-specific abstractions, like code artifacts. The features in the problem space are mapped to a specific artefact in the solution space [Kästner, 2010].

2.1.3 Variability Modelling

Variability
Variability refers to the ability of the software product line development artefact to be configured, customized, extended or changed for use in a specific context [van Gurp, Bosch, and Svahnberg, 2001]. Variability characterizes software product line members.

External and Internal Variability
External variability is the variability that is visible to external stakeholders (customers, end-users etc.). Problem-space variabilities are external.

Internal variability is the variability that is hidden from external stakeholder, and is only visible inside the product line. Solution space variabilities are internal variabilities.

Evolution of Variability
Variability in time refers to the existence of different versions of an artifact that are valid at different times. These variants are called versions [Pohl, Böckle, and van der Linden, 2005].

Variability in space refers to the existence of an artefact in different shapes at the same time [Pohl et al., 2005].

Variation Point
Variation points represent the locations where a variation will occur in the different product line variants.

Binding Time
Binding time refers to the point of time in the product life-cycle at which the decision for a variability is made. This is where a variability is bound to a specific variant [Clements and Northrop, 2001]. It could be during preprocess time, compile time, link time, and deploy time or run-time.

Variability resolution and realization
Variation points support mechanisms to implement the variant elements which will re-

place these variation points at a later point of time, i.e., during binding, Thus, it provides a means to resolve the variability by replacing the variation point with one or more realization elements. For example, if C preprocessor is used as the variability realization mechanism, resolving variability means, for example, creating a header file with all the definitions. In a configuration file (*.ini file) on the other hand, certain configuration properties are set. Selection, generation, substitution and composition are the primitives that variability realization rely on [Becker, 2017].

Variability Support in Domain Assets

Different degrees of variability support can be followed in domain assets depending on the degree of common, shared and specific solutions. This is illustrated in Figure 5.

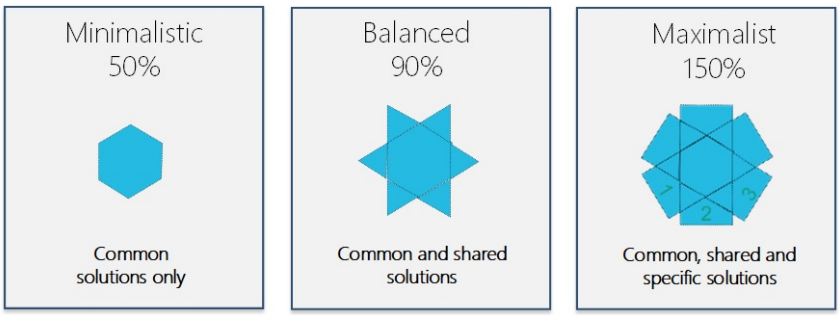

Figure 5: Degree of Variability Support in Domain Assets

In the minimalistic approach, only the common aspects of product line members are included in the domain assets. This is called platform approach.

In the balanced approach, the common, as well as shared aspects of the product line members, are supported in the domain assets. Variation points perform the necessary adaptations. This does not include application-specific artefacts, which will be later engineered as part of application engineering. This is called product line approach, the standard approach for product lines.

In the third, maximalist approach, domain assets include all the necessary adaptations required to completely derive the product line members. Thus, the common, shared and application-specific aspects are supported. There should be well-defined variability for this approach to work. Usually found in the automotive domain, this is called the production line approach (also called the 2^{nd} generation product line engineering) [Becker, 2017].

2.1.4 Variability Management and Separation of Concerns

Industrial products can easily comprise several thousands of variation points and configuration parameters, which makes management of this variability a very important aspect. Variability management comprises all variability-related activities in the life-cycle of variability, including its specification, realization, resolution, and evolution [Clements and Northrop, 2001]. This is a key feature in every product line approach. The activities of variability management can be separated into four areas of concern as illustrated in Figure 6.

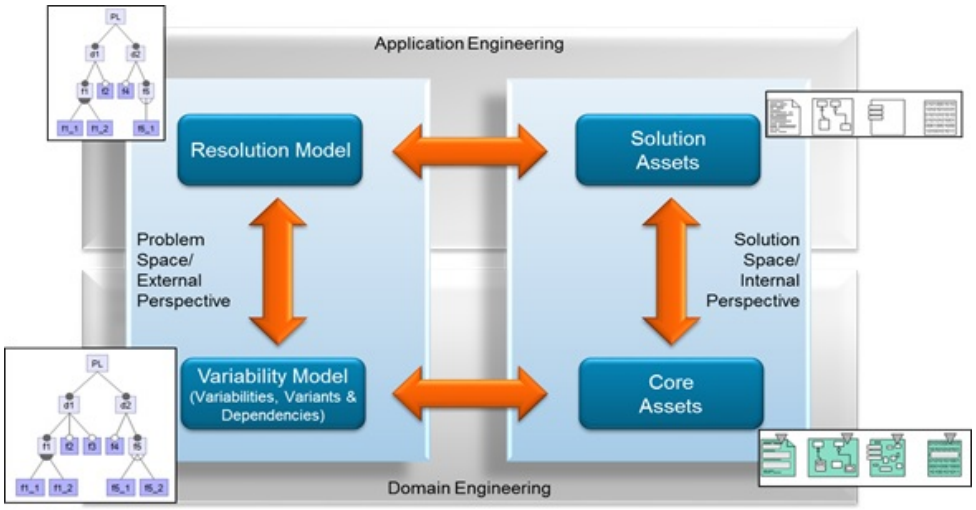

Figure 6: Separation of Concerns

- The problem space is concerned with the external perspective on variability, i.e., the product line member characteristics, dependencies etc.
- The solution space represents the internal perspective on the product line, where variability is realized and resolved. It consists of reusable artefacts like requirements, code, design, verification artefacts etc.
- The variability modelling phase generates the variability model, which specifies the variabilities and the supported variants and inter-dependencies.
- In resolution models, the customer's requirements are mapped to feature selections based on the variable features specified in the variability model.
- In domain assets development, reusable domain artefacts are developed, that contains the common and reusable artefacts.
- In solution assets development, reusable domain artefacts are combined as per the resolution model, producing the product line members.

2.2 State of the Practice

2.2.1 Variability Modelling

In variability modelling, the common and variable features of the product line members are represented in a structured and disciplined manner. This is done during domain analysis, by the domain experts. Variability models are an important aspect in software product line and variability management, as it helps document variabilities in problem space, and also analyze the various aspects of the variants. Variability models are central to several implementation approaches and automated extraction of variant information.

Different approaches exist, to model variability information, including feature models, decision models, UML-based notations, domain-specific languages, and other formal and non-formal approaches like spreadsheets. However, the most frequently used notation [Kästner, 2010] in variability modelling is the feature model. Decision model is another commonly used variability modelling mechanism. In the next section, an overview of feature modelling is presented.

2.2.2 Feature Modelling

The feature model specifies the commonality and variability of product line members in terms of features. *Features* are prominent and distinctive characteristics of a system that are visible to the user. Feature models are an effective way of communicating common and variable aspects of product line members to the stakeholders. It has a hierarchical tree structure, with each node representing a feature, comprising a parent-child relationship. The different groups of features represent a type of variability. Figure 7 is an example of feature model [Benavides, Segura, and Cortés, 2010].

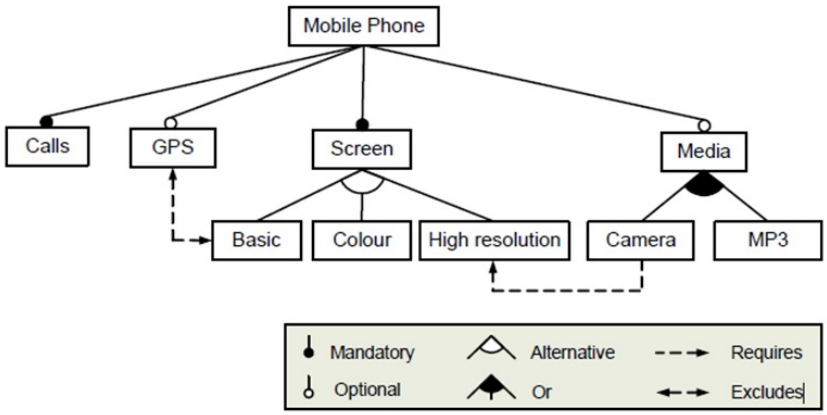

Figure 7: Feature Model

The following are the relationships allowable between the parent and child feature:

- **Mandatory**: Here, the child feature is included in all systems in which the parent feature is included.
- **Optional (0 or 1)**: Child feature can be optionally included in systems where parent feature is included. If the parent feature is not included, the child feature cannot be included either.
- **Alternative (1 out of n)**: A set of child features have alternate relation with the parent feature such that only one of the child features is included when the parent feature is included in the system.
- **OR (multiple co-existing, m out of n, m > 0)**: Here, one or more child features can be included when the parent feature is included in the system.

Mandatory feature is included only in the systems in which the parent feature is included, whereas common feature is included in all the systems. Figure 8 shows the different types of features from a variability perspective.

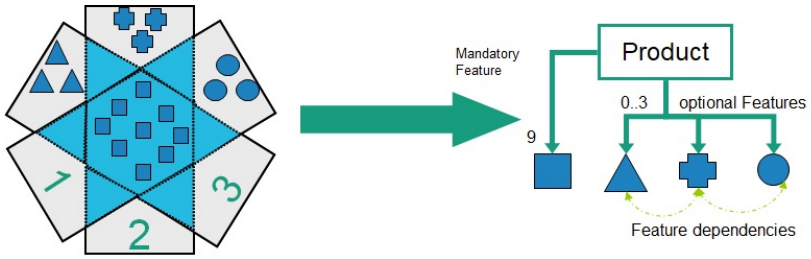

Figure 8: Variability and feature types

The different relationships between features in a feature model are illustrated in Figure 9.

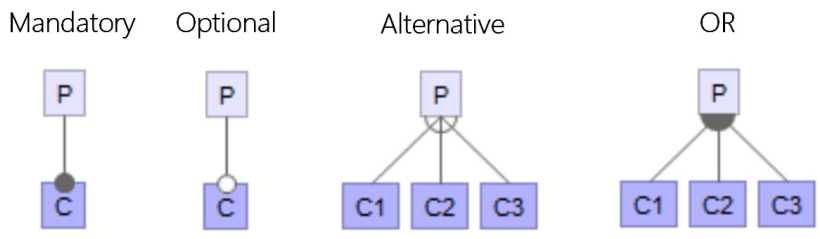

Figure 9: Relationships between features

The model and notation illustrated in Figure 9 are based on the FODA (Feature-Oriented Domain Analysis) model developed by Kang et al [Kang, Cohen, Hess, Nowak, and Pe-

terson, 1990]. There could be cross-tree dependencies as well between these features. For example, *requires* and *excludes* relations:

- If feature A is included, then feature B must also be included - Requires relation.
- If feature A is included, then feature B must not be included - Excludes relation.

Variability modelling does come with challenges. One of the main challenges is the increasing size and complexity of the software product line systems, resulting in the need for a huge variability model. The other challenge is the lack or no documentation of feature interdependencies in the complex product line systems. Thus most of the features need to be manually selected and configured, thereby affecting the efficiency and effectiveness of the product configuration process [Zhang, 2015].

2.2.3 Variability Realization Techniques

This section presents the variability realization techniques used in the industry, the advantages and their shortcomings.

Variability Code Elements

- **Var:** This represents a variable feature, which could be a parameter name, class name or module name based on the variability realization mechanism. A Var can be thought of as a feature in problem space [Zhang, 2015]. An example of a Var can be the parameter ENGINE_SELECTION which holds two values, ECU_ENGINE and MECHANICAL_ENGINE which represents the type of engine a specific transport refrigeration unit could have. Here, engine selection can be considered as a variable feature offered by the product line.
- **Variation Point:** This represents a specific realization of Var in product line member. A Var may be realized with multiple VPs scattered across different code locations [Zhang, 2015].
- **Code Variants:** Represent the block of code within the variation point. Thus, a variation point is adapted using different code variants [Zhang, 2015].
  A VP with multiple CVs can be considered as an alternative VP.
  A VP with only one CV is an optional VP.

Dependencies between Variability Code Elements

Different relationships exist between the variability code elements.

- **Var Interdependency**: Multiple Vars may be combined in a specific Variation Point. For example, in conditional compilation, the variation point `#if Var_X > 0 && VAR_Y > 3` has two Vars, X and Y respectively that are *tangled* together. This is called #ifdef tangling [Liebig, Apel, Lengauer, Kästner, and Schulze, 2010].
- **Hierarchical Dependencies between Vars and VPs**: If multiple VPs are nested, then the child VP is dependent on the parent VP. Similarly, the Vars in the parent and child VPs will be inter-related. Also, the Variation Points that represent the same Vars implement the same feature, or sub-features of the same feature.
- **CV Inter-dependency**: The code variants belonging to the same VPs are inter-related, since they either belong to the *Alternative* or the *Or* group.

Techniques for Realizing Variability

Variability realization techniques can be broadly grouped into two approaches, the *compositional approaches* and *annotative approaches* [Kästner, 2010]. In compositional approaches, features are implemented in modules like classes, files, packages etc. and are *composed* to derive a specific product variant. The *FeatureHouse* approach is one such example, which will be delved deep into, later in the thesis. In the annotative approach, code fragments are annotated to control their inclusion and exclusion during binding time. The C preprocessor is an example of annotative approach.

Both approaches have their own pros and cons. While compositional approaches are a disciplined approach, they have limitation with regards to granularity and multi-dimensional separation of concerns. This is mostly used in research and academia. The annotative approach, on the other hand, is flexible and easy to use. However, this approach is commonly criticized for its code obfuscation, error proneness, code tangling and so on.

The focus of this thesis is on Conditional Compilation, which is an annotative approach and hence, this approach is elaborated.

Below are the some of the general annotative mechanisms used in industry:

- Cloning
- Templating
- Preprocessing
- Module Replacement

In cloning, the available domain asset is copied and the copy is modified and evolved without affecting the original artefact. Its practical benefits include low creation effort, independence of cloned variants etc. This approach can be used when the number of variabilities is small and for experimental functionalities. High maintainability and reduction in artefact quality are some of the several practical challenges for cloning. This is illustrated in Figure 10.

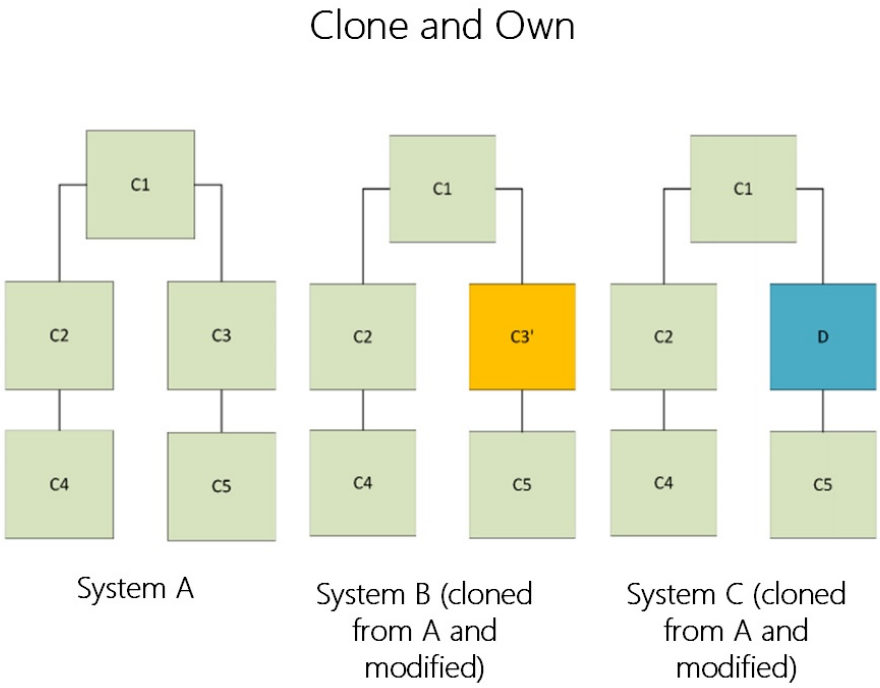

Figure 10: Clone and Own

In templating, variation points are identified through annotations in the domain assets. Along with identification of variation points, easy resolution is also provided by giving additional information in the code annotations. The benefits of templating are the same as in cloning, with additional benefits of easy identification and resolution of variation points. Figure 11 illustrates variability code annotated with variability information. Here, the `ADD SENSOR INIT HERE`, `ADD SENSOR UPDATE HERE` etc. are annotations representing variation points.

```
#include<string.h>
#include<stdio.h>
#include<stdbool.h>
#include<stdint.h>

// hardware initialisation
void init();

// wireless transmission
// string to send
extern char send_buffer[61];
// sends send_buffer
void send();

// actuator abstractions
// switches led 2 on or off
void set_led_2(bool);
// toggles led 2: on <-> off
void toggle_led_2();

// clock abstraction
// clock value
extern int32_t the_clock;
// periodically set by ISR every sec
extern volatile bool period_elapsed;
// converts clock value to string
char* timetoa(int32_t);

/* ADD SENSOR VALUES HERE */
extern int16_t x_position;
/* END */

/* ADD SENSOR OERATIONS HERE */
// init_.. call before first use
// update_.. refeshes sensor value
void init_x_position();
void update_x_position();
/* END */

bool event_happened=false;
int32_t event_time=0;
int16_t tilt_count=0;
int16_t tick=0;
```

```
void main() {
  init();
/* ADD SENSOR INIT HERE */
  init_x_position();
/* END */
  while(true) {
    if(period_elapsed) {
      period_elapsed=false;
/* ADD SENSOR UPDATE HERE */
      update_x_position();
/* END */
/* ADD DETECTION & TRANSMISSION HERE */
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot timer
        }
        else { // a tilt has ended
          // has the device been tilted between 1 and 5s?
          if(the_clock-event_time>0
             && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      tick=tick+1;
      if(tick%5==0) {
        tick=0;
        sprintf(send_buffer,"drink=%d",tilt_count*25);
/* END */
/* ADD PRE-TRANSMISSION BEHAVIOR HERE */
        if(event_happened) {
          strcat(send_buffer,",time=");
          strcat(send_buffer,
                 timetoa(the_clock-event_time));
        }
/* END */
        send();
      }
    }
  }
}
```

Figure 11: An example for templating

In preprocessing, the variation points are included and excluded during the preprocessing stage of the compiler. It is one of the most frequently used annotative approaches. It is easy to use and the variation points are explicitly identified. There are no efficiency overheads to this approach since variabilities are instantiated during preprocessing time and hence no longer exists in runtime. This approach, however, has challenges. Some of the practical challenges include increased artefact complexity, lack of separation of concerns (the common and variable parts are interleaved in the code), and maintenance, due to the growing complexity of the variabilities. Refer Figure 12 for more clarity.

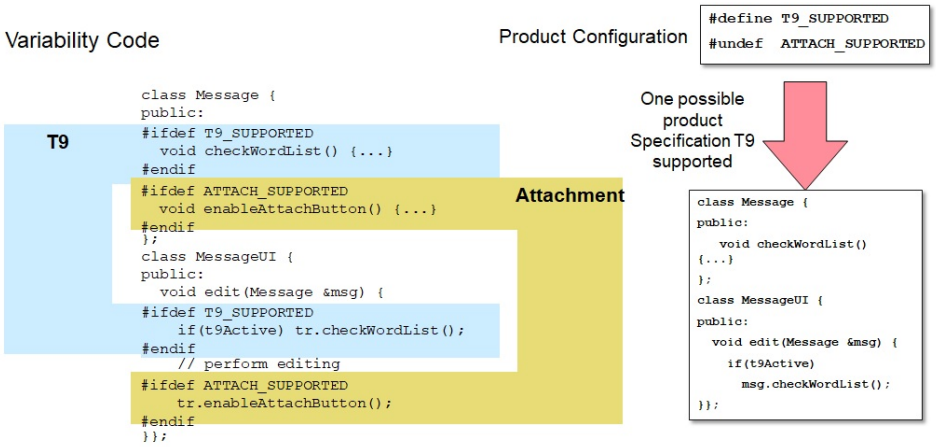

Figure 12: An example for Conditional Compilation (Preprocessing)

Here, the variability code example contains two variabilities or features, *T9* and *Attachment*. The code fragment belonging to each feature is enclosed under the #ifdef directive. The product configuration file has defined only `T9_SUPPORTED` and has undefined `ATTACH_SUPPORTED`. This means that after preprocessing stage, i.e., after product instantiation, only code fragment belonging to T9 feature will be present, as illustrated in Figure 12.

In module replacement, the variants of an artefact are provided in separate modules. Each module contains a logic that determines which of the modules is used to derive the variant [Becker, 2017]. This is illustrated in the below example. If for instance, the variant is to be derived for AVR32_UC3, then the contents in the folder with the name `AVR32_UC3[VAR_HAL==‘AVR32_UC3’]` will be copied under the HAL folder and all other folders will be removed. In this approach as well, there is no overhead on efficiency since the modules are replaced before runtime. Also, the common and variable parts are in separate files, thereby following the separation of concerns principle. However, it has limited visibility of variation points and only alternatives can be selected this way. Figure 13 illustrates an example in FreeRTOS where the hardware abstraction is realized through this technique.

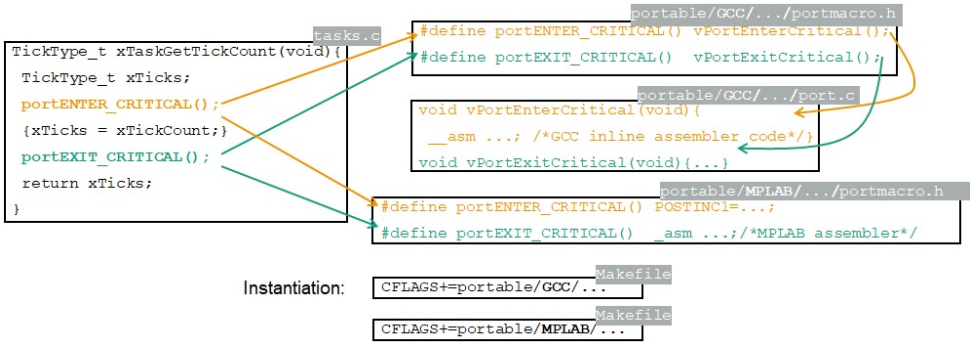

Figure 13: Module Replacement - Realization of hardware abstraction in FreeRTOS

2.3 State of the Art

The Software Product Line Paradigm is centered around a number of practices leading to systematic code reuse [Clements and Northrop, 2001]. Examples of large scale SPLs include Linux, FreeBSD and FreeRTOS, which contain several thousands of features and dependencies. Thus, industrial size product lines can easily grow large and complex, incorporating thousands of variation points and configuration parameters. These and other challenges have been already discussed earlier in the thesis. This section presents an overview of the solution approaches available in the industry and academia to solve the

challenges of variability management and analysis of complex variability code.

The following are the research questions defined for the problems mentioned in the thesis:
RQ1. How can we semi-automatically extract feature dependencies from an existing product line with variability realizations?
RQ2. How can we semi-automatically extract variability code elements from an existing product line with variability realizations?
RQ3. How can we analyze the variability realizations to trace the feature dependencies in an insightful manner?

Feature models will help to cope with the challenge of analyzing variability realizations to a great extent, as this will give a high-level understanding of the software system and also its intricate dependencies and relationships. For existing systems, the ability to semi-automatically extract these dependencies and code structures will improve efficiency and productivity and reduce human errors during product configuration.

Software applications are composed of source code, which contains inherent structures, design patterns and concepts. De-programming or reverse engineering of software systems is the process of reverting the source code back into concepts, patterns and designs [Coppel and Candea, 2018]. This allows us to create representations of the system at a higher level of abstraction. Reverse engineering software into feature models have evolved to become a major topic of interest in research as well as in industry due to its potential in software product line engineering and variant management.

There are two broad approaches to constructing feature models by a domain analyst. The first approach is to manually construct the feature model from user requirements for the system to be developed or from the feature description of existing software systems. This is the top-down approach. The second approach, which employs reverse engineering, identifies the features from a source code and other software artefacts of the software system to generate a feature model. This will be highly beneficial when the domain experts are not present, or when they leave the specific company, leading to loss of domain knowledge. This approach is a bottom-up approach and has the potential for partial or complete automation. Several studies have been conducted to this end and the results are promising [Paškevičius, Damaševičius, Karčiauskas, and Marcinkevičius, 2012].

Furthermore, the identification of features in a software system helps enhance program comprehension for developers, especially those who are new to the software system under

consideration. It is crucial to extract feature in legacy software systems to migrate the legacy systems into product line [Tang and Leung, 2015]. Extraction of features is known under several names in research such as feature mining, fact extraction, model extraction, concept analysis, feature location, concept location, dependency finding, concept assignment, semantic clustering and topic mining, pattern discovery etc. [Paškevičius et al., 2012].

Several studies have been conducted by academia and industry in the context of extracting features from software artefacts. She et al [She, Lotufo, Berger, Wasowski, and Czarnecki, 2008] uses association rule mining to retrieve propositional formulae to identify feature groups, mandatory features and implies/excludes edges. The research proposes a heuristic for selecting the parent feature for each feature while building a feature hierarchy. This is a semi-automated approach. Since domain experts may not be always present while building feature hierarchy, the authors propose lists of most likely parent candidates based on *Ranked Implied Features* and *Ranked all Features* technique using similarity matrices. Feature hierarchy is built using implication diagrams which are derived through propositional formulae. This procedure however assumes that feature names, descriptions and dependencies can be extracted from the software project under consideration.

Yang et al [Yang, Peng, and Zhao, 2009] use FCA, concept pruning/merging, structure reconstruction and variability analysis to recover domain feature models. In addition, information retrieval based techniques [Poshyvanyk and Marcus, 2007] are also used for improving the precision of feature location.

Botterweck et al [Botterweck, Lee, and Thiel, 2009] propose an approach to automatically derive executable products through model transformations and aspect-oriented techniques. Linsbauer et al [Linsbauer, Lopez-Herrejon, and Egyed, 1975] propose an approach to extract variability information from product variants by identifying traces from features and feature interrelations to implementation artefacts, and computing their dependencies. The authors use the concept of modules, to express relationships between features and implementation artefacts. A base module represents an artefact that implements a single feature, and no feature inter-relations whereas a derivative module represents an artefact that implements interaction between multiple features. Trace and dependency extraction are performed on the software artefacts from the sequence and dependency graphs derived from the product variants. The studies, however, was conducted in software artefacts written in Java. The generated data structure was very much similar to the Abstract Syntax Trees. The authors claim to be successful in extending the approach to UML diagrams

and are currently investigating the application on artefacts like CAD drawings and Excel sheets.

In the next section, the various tools used for product line extraction will be discussed.

2.3.1 Tools for Product Line Extraction

Much of the work presented here is adapted from the literature survey of REVaMP2 (Round-trip Engineering and Variability Management Platform and Process) [Martinez and Parsai, 2019], an ITEA3 project which aims to conceive, develop and evaluate the first comprehensive automation tool-chain to support round-trip engineering of SIS (Software Intensive Systems) product lines. A brief overview of some of the popular tools is presented and later, some of the tools are selected for further study. Also, libraries and tools that perform preprocessing of C/C++ source code in Python are evaluated due to its high potential in customization, in the context of variability.

- **Pure::Variants**
  The pure::variants include prototypical extracts, which extracts variability from legacy source code files. The language-specific artefacts like #ifdef preprocessor directives are analysed for variation points. The tool, however, extracts only those variation points which have identifiers with some specific patterns, say, for instance, those switches having a prefix `VP_`. This is done in order to separate the VPs from other switches that are not related to the actual product line member. This, however, need not be the case in industrial product lines always. The results of the extraction process are in the form of VEL models that contain the concrete location of variation points.

- **BUT4Reuse - Bottom-Up Technologies for Reuse**
  This tool-supported framework helps automate relevant tasks for feature identification, feature location, feature mining, extraction of reusable assets and visualization of feature models. The tool is generic and extensible since it supports different software artefact types, like source code in Java, C, MOF-based models, requirements in ReqIf etc. The tool is built on Eclipse and is open source.

- **FLiMEA: Feature Location in Models through Evolutionary Algorithms**
  This approach relies on Evolutionary Algorithms to locate features in product models and to formalize them as model fragments.

- **SciTools Understand**
  This is a static analysis tool that focuses on source code comprehension, metrics

and standards testing. It provides code navigation using detailed cross-referencing, syntax-colouring smart editor and a variety of graphical reverse engineering views. *SciTools Understand* is mainly designed to understand and maintain large legacy code or newly created source code.

- **TypeChef**
  TypeChef [Kenner, Kästner, Haase, and Leich, 2010] is a variability-aware C-code parser that parses the source code, performs macro expansion, includes all variability information and produces a variability-aware Abstract Syntax Tree (AST). The main goal of TypeChef is to identify variability-related bugs.

Of the above-mentioned tools, TypeChef and BUT4Reuse have been selected for further study as these are potential candidates for the solution ideas for this thesis. One key feature that can be identified in all the tools is that they use a dependency graph or tree structure that is very much similar to Abstract Syntax Trees. This lead to the idea of using AST generating open-source libraries and evaluating its potential in the context of this thesis. If the libraries can directly output ASTs and provide access to its data structure, then this data structure can be exploited in developing a variability-aware tree that highlights the different variability code elements and dependencies. For this, the following libraries have been selected after extensive study and exploration:

- FeatureHouse
- CPIP
- Clang & LLVM
- PCPP

In addition, an IDE called CIDE was also explored to understand the different ways in which variability information can be represented. The subsequent sections in this chapter detail these.

3 Feasibility Studies and Investigations

This section elaborates on the studies conducted during this thesis and the analysis of results. Specifically, the feasibility of various open-source libraries and tools are evaluated and recommendations are provided.

3.1 Study on TypeChef

This section presents an overview of TypeChef, the experiments conducted with TypeChef library, analyses and evaluation.

TypeChef is a type-checker for product lines written in C, in which variability is realized using #ifdef preprocessor directives. However, the C preprocessor, *cpp*, works on a token level and makes analysis of variability realizations a difficult task.

In a software product line, whether a specific code fragment is present after compilation or not depends on the preprocessor flags, or features selected for that product variant. The author thus defines a terminology called *presence condition*; a code fragment is only included if its presence condition evaluates to true for that specific product variant. The aim of type-checking here is to ensure that all variants of a product line are well-typed, without generating all variants. The main challenge the authors faced while type-checking C code is in parsing the code that contains cpp directives.

In order to understand various variability code dependencies, it is essential to parse pre-cpp code. For this purpose, a partial preprocessor has been implemented in the research. However, there are constraints/assumptions to this approach. The parser understands only C code which wraps entire functions or statements, and not arbitrary tokens. These are termed as *disciplined annotations*. The idea of the research is to generate ASTs with these disciplined annotations, and then assign presence conditions to subtrees. Unfortunately, most of the code in industrial product lines are not in a disciplined form. Additional challenges in cpp include lexical macro substitutions and file inclusion. The #include directives and macros need to be expanded while parsing pre-cpp code. Cpp not only allows propositional formulae after #ifdef directives but also integer constants which may be defined, re-defined and un-defined during preprocessing. Additionally, a macro can also have alternate macro expansions (conditional definitions) [Kenner et al., 2010].

Analysis of pre-cpp code using TypeChef

In TypeChef, analysis of pre-cpp code is done in four steps:

- Partial Preprocessor
- Expansion of disciplined annotations
- Parsing
- Reference Analysis
- Solving

Our focus is on the first three steps since the aim of this thesis is not to perform type-checking of product line variants.

In the partial preprocessor step, all the macro expansions and file inclusions are processed without affecting the variability of conditional compilation constructs. This is performed by a two-step process. Firstly, all the #ifdef directives are commented and the original preprocessor is run on this modified code. This will expand the macros and processes file inclusions and result in a code with these #defines and #includes preprocessed. The second step is to uncomment the commented conditional compilations (#ifdef directives). This results in a code that includes only variability code elements like #ifdef directives. In addition to this, include guards are omitted from the list of variation points using pattern matching. Include guards is a standard pattern in C to prevent multiple or recursive inclusions of a file. It uses the same #ifndef and #ifdef directives and is not part of variation points.

Once all the files are included, macros substituted and disciplined annotations enforced, parsing is performed using a parser generator. In the research, ANTLR was used with an existing C grammar as the parser generator. The existing C grammar was extended with definitions for #ifdef directives, resulting in a parser generator specific to TypeChef.

TypeChef is a work-in-progress, and an ongoing study is addressing the limitations of TypeChef discussed previously. TypeChef has been able to successfully check the light-weight open-source web server Boa, which has 6200 LOC and 38 files. However, TypeChef cannot be used directly on large-scale industrial C projects, without the manual expansion of disciplined annotations.

Key Points

From this study, the hypothesis of using an Abstract Syntax Tree based approach was

strengthened as TypeChef also used a parser generator after the initial preprocessing steps. In order to generate a complete solution that can address the shortcomings of TypeChef, we need a library that can give the intermediate results of preprocessing and has good control over the generated data structure. Also, since more flexibility and control is required for generating such a solution, the solution idea pointed towards ANTLR or any open source parser generator, which can create customized parsers based on the requirements of this research. As a first step towards this, the ANTLR parser generator itself was studied and experimented with.

Experiments with ANTLR
ANTLR (ANother Tool for Language Recognition) is a parser generator for reading, processing, executing or translating structured text or binary files ["ANTLR", 2020]. ANTLR takes a grammar as input, which specifies a language and generates a source code for a recognizer of that language as output. This recognizer can be used to generate an Abstract Syntax Tree of the source code artefact. The focus of this experiment is to understand how ANTLR generates AST and the flexibility of the generated recognizer code for extending it to suit the requirements of the research, i.e., to extract variability code elements and their inter-dependencies. To process of deriving AST from ANTLR is summarized in the steps below:

- Define a lexer and parser grammar for the language we need to analyze (here C and CPP code)
- Invoke ANTLR - which will generate the lexer and parser grammar in the target language (here, we have considered Python)
- Use the generated lexer and parser and invoke them by passing the code to recognize, which returns the AST of the input source code.

Disadvantages of using Regular Expression Regular expressions can also be used instead of using a dedicated parser generator like ANTLR. However, there are severe limitations to using regular expressions:

- **Lack of recursion**: There is no straightforward way to find a regular expression inside another one unless we code it by hand for each level. This leads to maintainability issues.
- Not scalable for large programs
- ANTLR can create multiple parsers in different languages (Java, Python, C#, JavaScript etc.) easily compared to regular expressions, where significant modifications will be needed specific to each programming language in the latter.

Creating AST using ANTLR - Workflow

For this study, the Linux kernel source code was used. ANTLR is made up of two parts, the tool, used to generate the lexer and parser and the runtime needed to run them. The following are the steps required to set up ANTLR and generate the parser generator. The ANTLR .jar file was downloaded and recognizers were created in Python for C and CPP grammar file. To use ANTLR, a grammar needs to be written or used, which is a file with extension *.g4* (for ANTLR v4). This grammar file contains the rules of the language that is being analyzed. The *antlr4* program is used to generate the lexer and parser file used by the application for analyzing the source code constructs.

```
antlr4 <options> <grammar-file-g4>
```

For the study, C and CPP grammar were used.

```
antlr4 -visitor C.g4
```

There are different options available for generating the lexer and parser files. One is the target language, to generate these files. It can be Python, Java or JavaScript or any other language supported by ANTLR. There are also options to specify the visitor and listener files for the grammar. Visitor helps to control how the nodes of an AST are entered or to gather information from the nodes. It uses the depth-first search approach while traversing the AST.

The utility called *grun* can be used to visualize the AST generated (this utility is part of ANTLR and is only for the purpose of feasibility study). For further analysis and extraction of variability code elements, the AST data structure generated from the parser recognizer modules in Python needs to be used.

```
grun <grammar_name> <rule_to_test> <input_file>
```

In the study, different rules were tried out, some of which are: *conditionalExpression*, *selectionStatement*, *compoundStatement* etc.

In order to use the *grun* utility, the parser needs to be generated in Java. Once the .java files are created, .class files need to be generated for the *grun* utility to recognize. Thus compilation was performed for all the generated .java files:

```
javac C*.java
```

Results

For this research, a simple C, CPP source code was used to check the usability and flexibility of ANTLR. It contained simple if statements, assignment operators, function definitions and function call. It also included the *include guards* and a macro. The sample C file is shown in Figure 14.

```
1  #ifndef _SAMPLE_DEFINITION_
2  #define _SAMPLE_DEFINITION_
3
4  #define COUNT 5
5
6
7  int main(int& argc, char** argv)
8  {
9       int s = 0;
10      int r = 0;
11      if(s == r)
12      {
13         r = s + 1;
14      }
15      else
16      {
17          r = s * 2;
18      }
19      sum(3,4);
20      sub(9,4);
21      return 0;
22 }
23
24 int sum(int a, int b)
25 {
26     return (a + b);
27 }
28
29 int sub(int a, int b)
30 {
31     return (a-b);
32 }
33 #endif
```

Figure 14: Sample C code for parsing using ANTLR

When the *grun* utility was executed for the rules mentioned above, we did not get promising results, as the generated AST had too much internal information which was not useful for our variability code analysis. In addition, the C and CPP grammar was not able properly to parse and create AST for the code which contains include guards and macros. However, it was able to generate a tree with the interdependencies between the various tokens generated as part of the parsing process. This was promising since it was possible to access the data structures of the generated AST in Python, using which a much refined and stripped-off version of the AST with variability specific information could be generated. Figure 15 and Figure 16 illustrate some of the ASTs generated using *grun* utility.

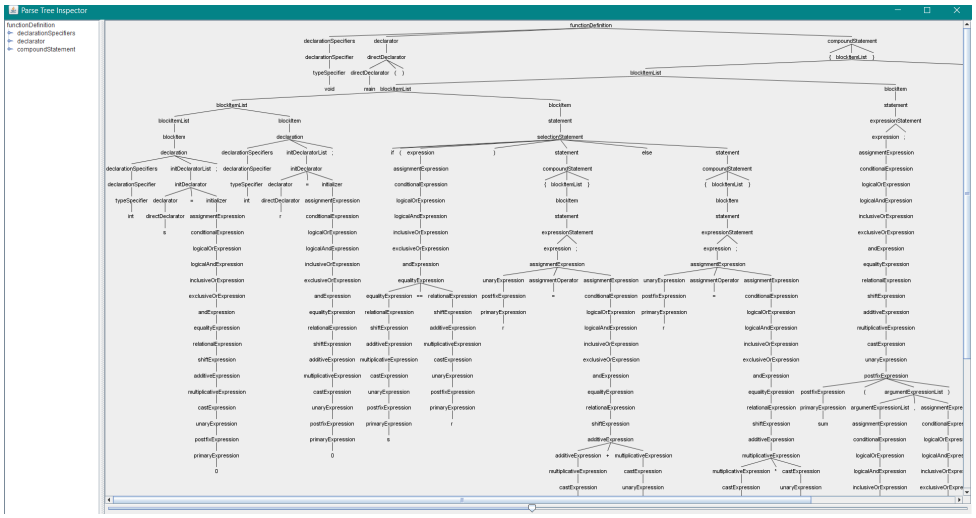

Figure 15: AST generated for function definitions using grun utility

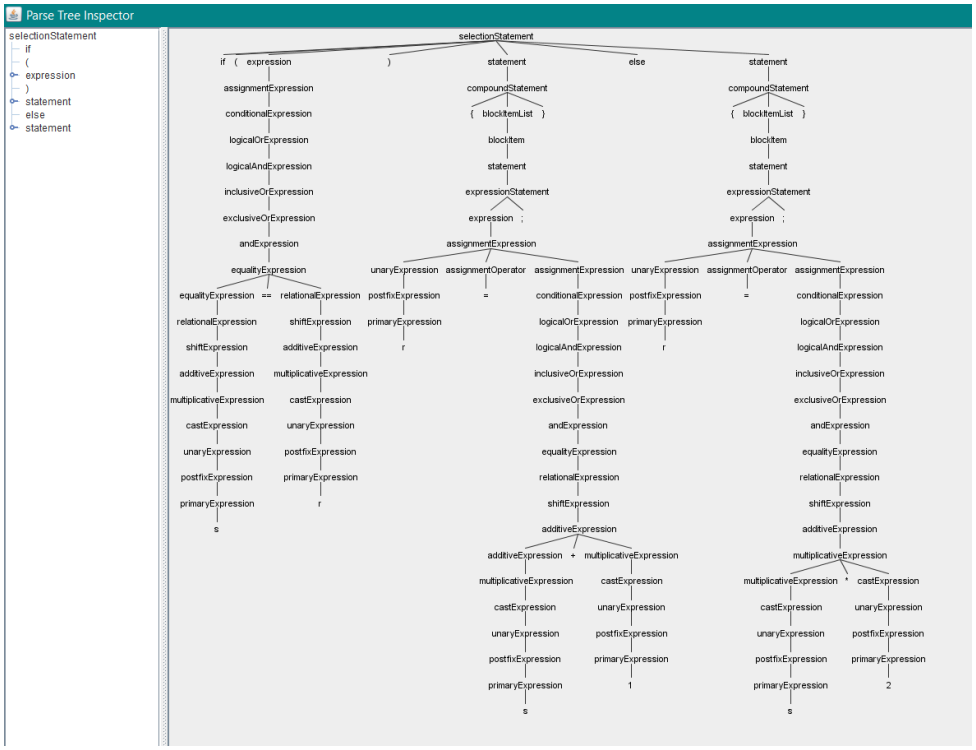

Figure 16: AST generated for if-else statement using grun utility

The next experiment was focused on modifying the AST generated using the parser generator derived in Python. Before delving into the details of the experiment, an overview of lexers and parsers are presented to the reader.

Lexers and Parsers

Lexers are also known as tokenizers. It takes the individual characters in a source code

and transforms them to *tokens*, which is used by *parser* to create the logical structure. This concept is best illustrated using Figure 17.

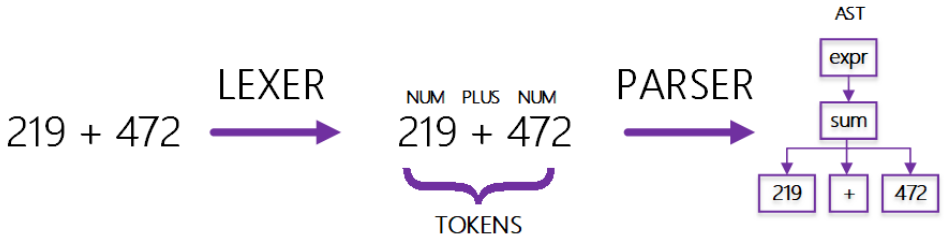

Figure 17: Lexers and Parsers

Suppose we are trying to parse a mathematical expression:

```
219 + 472
```

The lexer scans this line and identifies '2', '1', '9' and whitespace. The lexer recognizes this as a number. Then it identifies a '+' symbol, which it recognizes as an operator, and finally, in the same manner, it identifies the last number. The lexer identifies and recognizes these constructs from the provided grammar (here, C and C++). The lexer and parser rules will be written in the grammar file. The lexer rules are analyzed in the order that they appear.

As a first attempt, the generated tokens during the lexical parsing were visualized in *pandas Dataframe* format, as illustrated in Figure 18.

	Line No.	Text	Type
0	1	#ifndef _SAMPLE_DEFINITION_	Directive
1	2	#define _SAMPLE_DEFINITION_	Directive
2	4	#define COUNT 5	Directive
3	7	int	Int
4	7	main	Identifier
5	7	int	Int
6	7	argc	Identifier
7	7	char	Char
8	7	argv	Identifier
9	9	int	Int
10	9	s	Identifier
11	10	int	Int
12	10	r	Identifier
13	11	if	If
14	11	s	Identifier
15	11	==	Equal

Figure 18: Some of the tokens identified by ANTLR parser generator, represented as Pandas Dataframe

In this manner, it was possible for the parser generator to identify all the tokens present in the source code, including the preprocessor directives and macro definitions. However, upon inspecting the C and C++ grammar files, it became clear that the grammar does not contain the parse rule for preprocessor constructs. This is a shortcoming of the ANTLR C, C grammar in its vanilla state. Nevertheless, ANTLR provides very high flexibility in generating tailored grammar files, which can be utilized for generating a parser generator only for the variability code elements. This was exactly the step taken in TypeChef as well. Since the generation of variability-aware grammar requires a deep understanding of the ANTLR lexer and parser rules, the results from TypeChef were taken as the basis to conclude that this approach will not give a perfect solution for analyzing variability code elements (see section Study on TypeChef).

Another observation upon inspecting the ANTLR grammar files was that the previous version of ANTLR (version 3) included a grammar file for C Preprocessor as well. This could be used as a potential reference candidate for generating the preprocessor grammar file. However, no further attempts were made to pursue this approach as a better solution was found in parsing the preprocessor grammar (see section Study on CPIP).

One key takeaway from this study is that the ANTLR parser generator can be utilized for supporting conditional execution and parsing more generic types (say, for instance, requirements, UML diagrams etc.) due to its powerful flexibility in parsing any type of file, given its grammar. Competency needs to be developed for generating the grammar file for these specific types.

3.2 Study on REVaMP2

REVaMP2 stands for *Round-Trip Engineering and Variability Management Platform and Process*. It provides frameworks, automation tool-chain and processes to support round-trip engineering of SIS (Software Intensive Systems) Product Lines. The outcome of the project is to develop a prototype platform that seamlessly integrates the SIS Round-Trip PL Engineering automation services like [“REVaMP2 -Round-trip Engineering and Variability Management Platform and Process”, 2020]:

- Extraction of SIS PL variability model from legacy assets of implicitly related SIS sets.
- Multi-view visualization of legacy assets, extracted variability models and PL assets.
- Verification that a SIS PL satisfied hard constraints like safety.

- Refactor SIS PL, optimize the soft constraints through full exploitation of multi-core processor power, co-evolve related assets like software algorithms, hardware architectures, on which they run.

The project is organized into eight work packages which include asset extraction automation and visualization technologies, asset co-evaluation and visualization technologies, asset co-evolution automation technologies, asset verification automation technologies etc. to name a few ["REVaMP2 -Round-trip Engineering and Variability Management Platform and Process", 2020]. The work package 4 (asset extraction automation and visualization technologies) is of significant interest in this research. The asset extraction automation and asset visualization services are achieved through:

- Taking input legacy SIS assets with implicit commonalities and variabilities, resulting in a variability model, thereby making them explicit.
- Factorizing legacy assets into PL structure provided by the variability model, following an agile semi-automatic PL extraction process

An overview of their extensive set of toolchains has been published to the research and industrial community. BUT4Reuse, Eclipse Capra, EASy-Producer, KernelHaven etc. are some of them. The readers are encouraged to explore the reference material [Martinez and Parsai, 2019] for more information on these tools. Out of these tools, BUT4Reuse (Bottom-Up Technologies for Reuse) and Bosch's configuration mining tools have been studied in detail due to their potential alignment with the thesis' research goals. The next section presents an overview of these tools.

Configuration Mining Tool

Configuration mining tool was developed for Bosch to derive matching constraints from a product configuration, to generate feature models.

To identify implications in the product configurations, feature selections are encoded for each product and negations of these feature selections are also added. Once this is completed, apriori algorithms are used to find the implications between the different feature selections. The apriori algorithm finds correlations in input data by identifying *frequent itemsets* (feature selections which are frequently performed together). This result is used to derive association rules or implications. Once implications are derived, they can be transformed into feature constraints (for example, B requires A, B excludes D, A or C etc.).

To make the configuration mining algorithm more efficient, dead (always false) and always selected (always true) features are identified and removed before configuration mining

starts, as implications involving them would be trivial. The developer reviews the result of mining with the help of inputs from the domain expert. In addition, constraint filtering is performed to reduce the number of constraints for review. This is performed using a SAT solver, which checks for logical equivalence.

BUT4Reuse (Bottom-Up Technologies for Reuse)
BUT4Reuse is a framework that leverages existing software products that automate relevant tasks for extracting PL assets ["BUT4Reuse", 2020]. The processes supported by the tool-supported framework includes feature identification and location, mining feature constraints, extraction of reusable assets, feature model synthesis and visualizations to support domain experts ["REVaMP2 -Round-trip Engineering and Variability Management Platform and Process", 2020]. An extensive study on the Eclipse-based plugin was conducted.

The product line extraction from legacy variants is supported through parameterized variability identification. There are C/C++ adapters that are parameterized, i.e., it is possible to parse all or selected elements in C/C++ code, like methods, header files, include, source code, fields etc. It compares the selected parameters and gives results on similarities between different legacy code variants. User can modify the threshold for comparing the various parameters or put constraints to the parameters. An example constraint could be, 'Method names and modifiers should be similar', or 'Parameters of methods should be ignored' etc. A demonstration of the tool uses the soda vending machine example, with four variants made using clone and own technique for realizing variability. An operation to search for cloned code in the variants results in various blocks of cloned content.

Other functionalities/key takeaway of the plugin include:

- Function call hierarchy analysis for the functions in the variant product line members
- General metrics about the variant code, like constraint discovery, visualizer, word clouds (which visualizes the commonly used words in a source code according to its frequency), and statistics of generated blocks like percentage of common elements in different product variants
- It is possible to rename the cloned blocks with the most frequently used word generated from the word cloud.
- To generate feature model, this plugin uses FeatureIDE framework [Sven Apel, Kästner, and Lengauer, 2013, 1].

- For constructing core assets, pure::variants, #ifdef directives or custom annotations can be used.
- One constraint in identifying the #ifdef annotation is that the variability should start with FEATURE_ pattern. The plugin can show the beginning and end of #ifdef blocks based on this pattern.

Java and C, EMF Models, textual files, file structures, JSON and CSV files, images, requirements in ReqIf format and natural language text (using OpenNLP library) are some of the other artefact types supported by BUT4Reuse.

The Java and C adapter uses FeatureHouse source code visitor. Currently, it identifies similarity between the Feature Structure Tree (FST) positions and compares names of the different nodes from the tree generated for each variant. It can also identify node containment dependencies. Dependencies like call dependency graphs are ongoing work. This arose interest in FeatureHouse for its feasibility in generating a better variability-aware AST (or Feature Structure Trees as they are termed), which can be directly used for applications and has good control on manipulating the resultant data structure.
A deep dive was performed on the FeatureHouse library to evaluate its feasibility. The next section presents a detailed overview of FeatureHouse and the evaluation and results of this tool for the research.

Some of the other tools that were briefly studied were:

- Eclipse Capra: This is an open Source traceability management tool. One shortcoming is that it requires manual link creation from one artefact to the other, which often consumes a lot of effort
- RQS Requirements Quality Suite: This tool extracts requirements from requirement artefacts and generates feature models. This can be later used to generate requirements for specific product variants, from its feature model.
- Feature Dashboard: This tool supports different feature views for the documented features.

3.3 Study on FeatureHouse

Feature-Oriented Programming (FOP) and Conditional Compilation have become two major verticals for realizing variability in software systems. While the former one is language independent and popular mainly in academia and research, the latter could be

considered the most widely used variability representation technique [Santos, do Carmo Machado, de Almeida, Siegmund, and Apel, 2019]. Feature-Oriented Software Development (FOSD) is a paradigm for the construction, customization and synthesis of large-scale software systems [S. Apel and Kästner, 2009]. At the heart of FOSD is the concept of a feature, which is a unit of the functionality of a software system that satisfies a requirement, represents a design decision and provides a potential configuration option. The main idea of FOSD is to construct well-structured software by decomposing the software system into such features. This facilitates tailored software systems for the user based on his needs. The ability to handle features in FOSD is known as variability management and is accomplished at the implementation level by a variability representation [S. Apel, Batory, Kästner, and Saake, 2013].

FeatureHouse is one such tool that represents the group of techniques that physically separate the implemented features in the code base, whereas C-preprocessor virtually separates the implemented features [Santos et al., 2019].

3.3.1 Motivation of FeatureHouse

When it comes to variability, one can classify them broadly under two categories, positive/additive variability, where composition units are added on demand. This aims at keeping a traceable mapping between features and composition units [S. Apel et al., 2013]. FeatureHouse is a programming language-independent FOP technique to implement variability, which supports additive variability. It provides mechanisms to compose software artefacts to derive software products in a composition-based approach. FeatureHouse modules are represented by file-system directories, called containment hierarchies, in which the classes and their refinements (feature-specific lines of code in the class) are stored in files, inside the corresponding containment hierarchies [Sven Apel et al., 2013, 1].

Different code snippets concerning a specific feature are stored in different classes and depending on the product configuration, these code snippets/features will be composed, producing a fully functional software variant. FeatureHouse facilitates the composition of such code snippets by using the *original()* method call and Feature Structure Tree for composition. The *original()* method acts as a link between the existing refinements in the different feature implementations, which guides the execution of instruction sequence, depending on the features selected for binding and order of composition, which is specified in the product configuration.

3.3.2 Software Composition

Software composition is the process of constructing software systems from a set of software artefacts. These artefacts can be code units (like packages, classes, methods etc.), supporting documents (like models, documentation, Makefiles etc.) and so on. There are various approaches to software composition, mainly super-imposition and aspect weaving. FeatureHouse makes use of super-imposition approach to compose the software artifacts. Superimposition is the process of composing software artifacts by merging their corresponding sub-structures recursively (Figure 19).

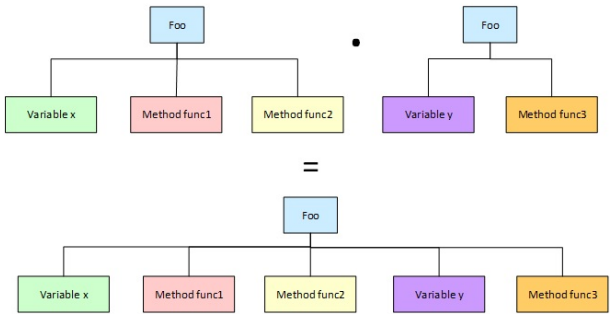

Figure 19: Superimposition of software artifacts

3.3.3 FeatureHouse

FeatureHouse is a structural approach to the composition of software artefacts written in different languages. It is a framework for software composition based on a language-independent model of software artefacts and an automatic plugin mechanism for the integration of new artefact language. FeatureHouse generalizes and subsumes FSTComposer, a previous software composition tool from the same research team [Sven Apel et al., 2013, 1].

FeatureHouse library comprises three ingredients:

- A language-independent model of software artifacts
- Superimposition as a language-independent composition paradigm
- An artifact language specification based on attribute grammars

Several languages are included in FeatureHouse, including Java, C, Haskell, JavaCC, C#, XMI/UML, Alloy, Ant XHTML etc.

Integration of a new language in FeatureHouse is based on the language's grammar, similar to the concept in ANLTR's parser generator. In the language's grammar, various attributes can be added as annotations. In addition, there are concise composition rules which govern the integration of a new language. As previously mentioned, FeatureHouse is derived from the FSTComposer tool, which relies on a general model of the structure of software artefact, called the Feature Structure Tree (FST) model.

3.3.4 Feature Structure Tree (FST)

FSTs represent an essential structure of a software artifact. It abstracts from language-specific details, i.e., irrespective of the language, the artifact can be represented as FST nodes. The FSTs are a stripped down version of Abstract Syntax Tree (AST), which contains only information necessary for specification of modular structure of artifact, and for its composition in other artifacts (Figure 20).

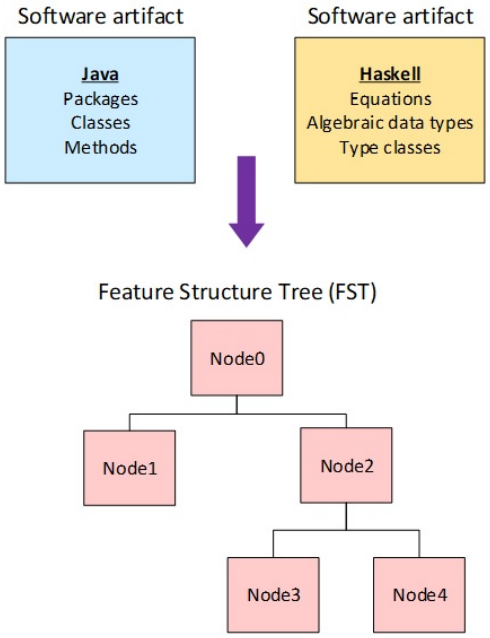

Figure 20: FST generated from multiple software artifacts

Each node of an FST has a name and type. Name of an FST node is same as the structural element, whereas type represents the syntactic category of the corresponding structural element. The *Inner nodes or non-terminals* denote modules (e.g., classes and packages) and *leaves/terminals* indicate module's content (Figure 21) (e.g., method bodies, field initializers etc.).

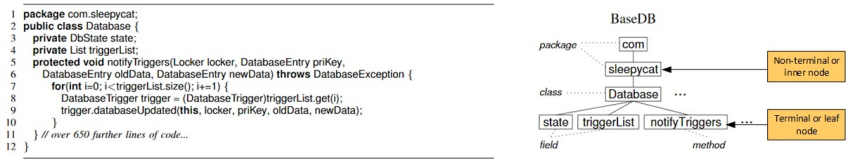

Figure 21: FST generated from a sample Java code["FeatureHouse: Language-Independent, Automated Software Composition", 2020]

The type of code elements that can be represented in FST depends on the language in which the software artefact is written and the level of granularity at which the software artefact needs to be composed. There are different levels of granularity defined based on different context. A *coarse granularity* will contain only packages and classes (for example, in Java) and not the methods or fields, in the FST nodes. Whereas in *fine granularity*, statements and expressions are also represented as FST nodes.

Software composition is performed in FeatureHouse library by merging the FSTs by their nodes, which are identified by their names, types and relative positions, starting from the root and descending recursively [Sven Apel et al., 2013, 1]. Figure 22 adapted from the original paper illustrates this.

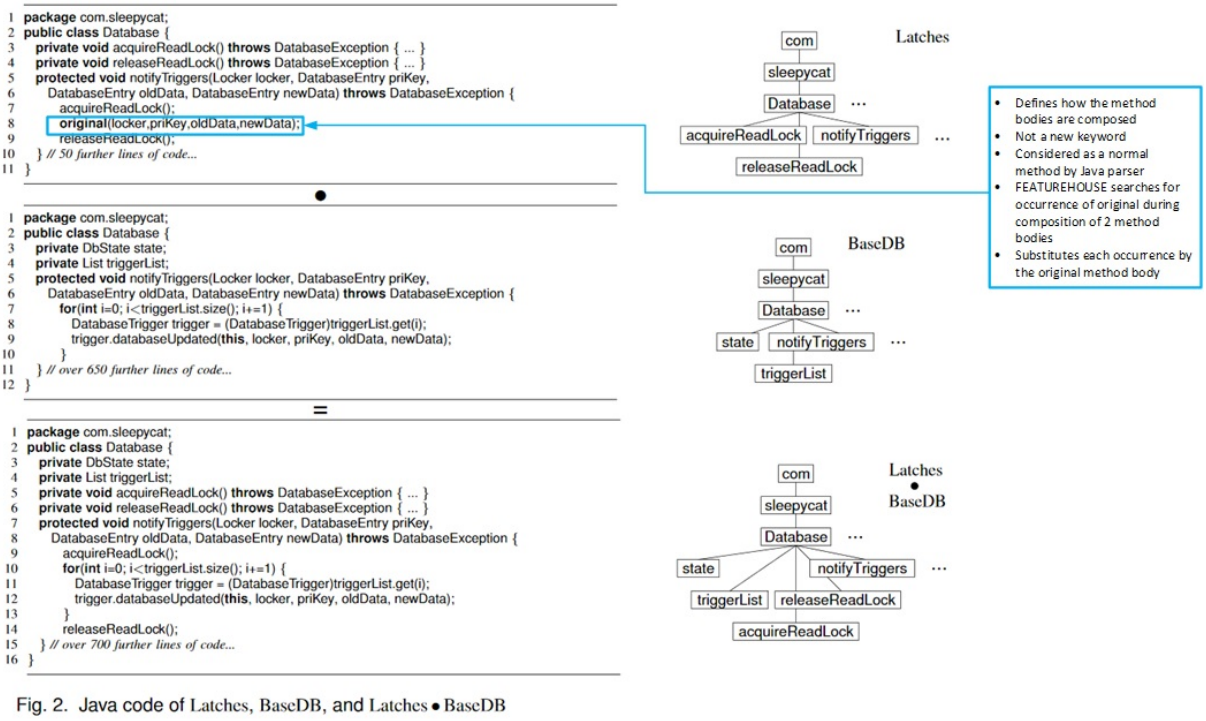

Figure 22: Superimposition of two FST nodes

3.3.5 Composition of software artefacts in FeatureHouse

The composition of two leaves of an FST that contain further content demands a special treatment. Depending on artefact language and node types, different composition rules need to be created for the composition of terminals. Often, simple rules like replacement, concatenation, specialization, overriding etc. suffice. This approach is open to more sophisticated rules.

How FSTComposer Works

Multiple software artefacts can be aggregated in a composition unit. The FSTComposer utility expects a list of units to be composed, which are organized in a subdirectory structure. The subdirectories are interpreted as non-terminal nodes and files located inside as terminal nodes. See Figure 23 for more clarity.

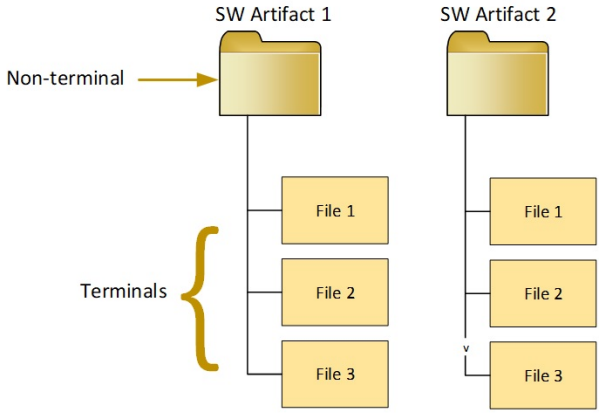

Figure 23: Hierarchical container for software artifacts

Integration of new languages using FSTGenerator

Integration of new languages can be easily performed in FeatureHouse. An automated generator tool, called FSTGenerator generates most of the code for integrating new languages. The FSTGenerator expects the grammar of the language to be integrated, in a specific format, called *FeatureBNF*. The FSTGenerator generates the following as output:

- **Parser**: to represent the parse tree for a specific language
- **Adapter**: to map the parse tree to the FST
- **Pretty Printer**: to write superimposed FSTs to disk (a tool known as un-parser, that takes a parse tree or an FST and generates source code)

The FeatureHouse Workflow

- FSTGenerator generates a parser for a specific artefact based on the grammar written in FeatureBNF for the language in which the artefact is written.
- The generated parser receives artefacts written in target languages and produces one FST per artefact and corresponding pretty printer.
- After generation, composition proceeds. FSTComposer performs composition.
- The generated pretty printer writes the composed artefacts to disk.

A library of composition rules has been developed and integrated for the composition of the content of terminal nodes. Figure 24 illustrates the architecture of FeatureHouse.

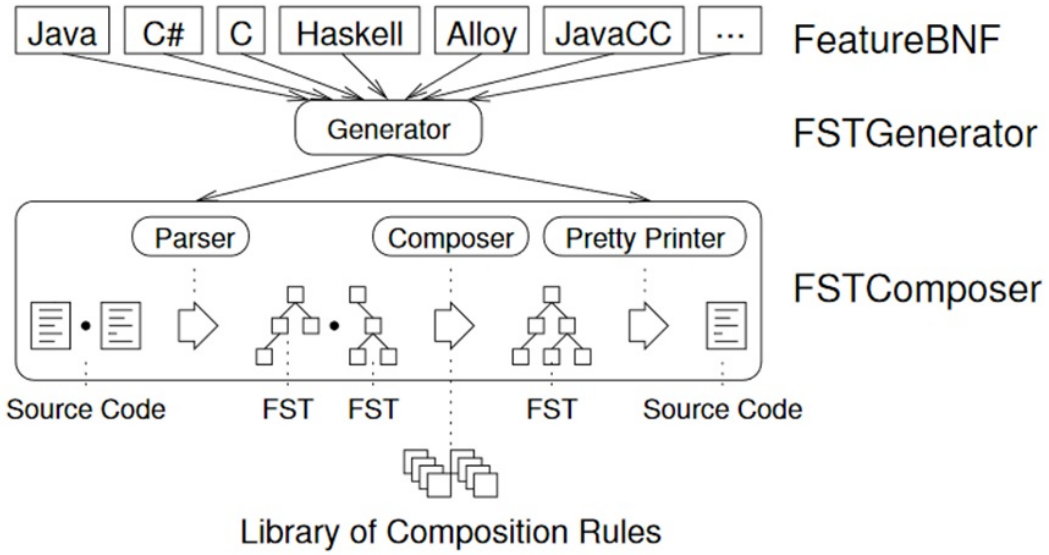

Figure 24: Architecture of FeatureHouse [Sven Apel, Kästner, and Lengauer, 2013, 1]

Table 1 summarizes the composition rules supported by FeatureHouse.

Table 1: Composition rules supported by FeatureHouse

Rule	Description
Method overriding	Merges two method bodies; *original* is used to inline one body onto the other
Grammar-rule overriding	Merges two grammar rules; *original* is used to inline the body of one rule into the body of the other
Constructor concatenation	Appends the statements of one constructor to the statements of the other
Field specialization	Assigns an initial value to a field in the case it did not have one before
Implements list union	Takes the union of the types of two implements lists, excluding duplicates
Modifier specialization	Specializes modifiers similar to Java's sub-typing rules
Replacement	Replaces one terminal node with the other
Text-content concatenation	Concatenates the text content of two terminal nodes

Exploring FeatureBNF Grammar

The language's grammar in FeatureBNF form is annotated with attributes to specify how the artefacts of a language are represented as FSTs. Without any attributes, FSTGenerator would create a single terminal node for each file. E.g., besides the non-terminals which denote the directories and files, there would be only a terminal node per class, and the class' members would appear as text in the terminal's content [Sven Apel et al., 2013, 1]. Refer Figure 25 for more clarity.

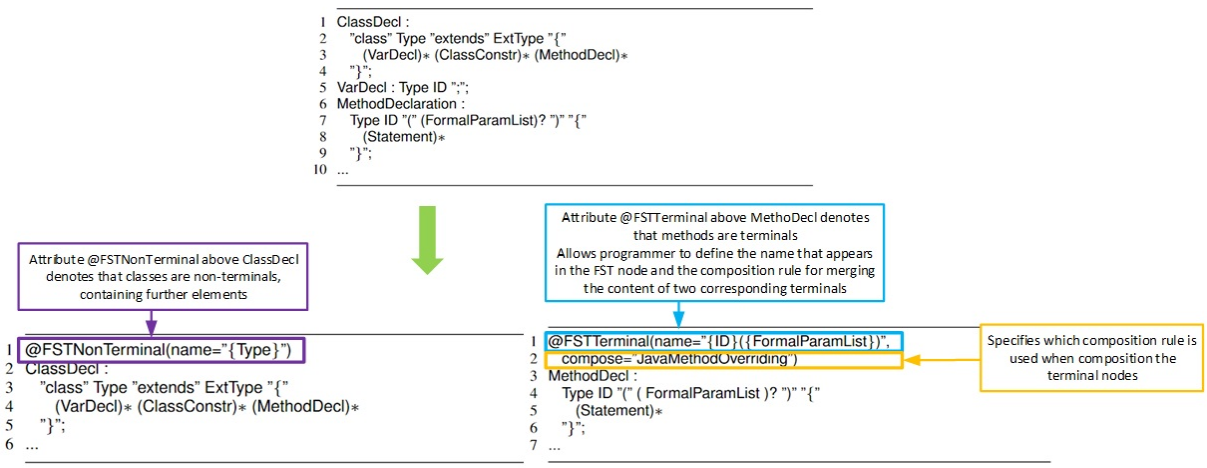

Figure 25: FeatureBNF Grammar

For superimposition to be feasible, the name of the FST node needs to be specified, i.e., two nodes are superimposed if and only if their names and types are identical.

Evaluation
Six languages were integrated into FeatureHouse in the beginning, imperative languages like Java, C#, and C, functional language like Haskell, specification and modelling language (Alloy) and domain-specific language for grammar specifications (JavaCC). The concept was tested in the composition of 50 software systems. These software systems were previously decomposed into different composition units for the purpose of the experiment. They were composed in different variants using FeatureHouse. The study defined some of the mandatory properties for a language to be plugged into FeatureHouse:

- Sub-structure of software artefact must be a tree
- Every element of an artefact must provide a name, which becomes the node's name.
- Every element of an artefact must belong to a syntactic category that becomes the node's type.
- An element must not contain two or more direct child elements with the same name and type
- Elements that do not have a hierarchical sub-structure represented in FST terminals must have compositional rules in order to be composable

Generality of superimposition approach
The superimposition approach is useful only in scenarios in which code of components is available and their structures are compatible for composition. The case studies in the research project were mostly product lines, whose features systematically refine the code of other features. Thus, one cannot generalize that superimposition is the most suitable composition technique.

Granularity and uniqueness of names
The more structure is exposed in an FST, the finer-grained the composition can be. This makes the composition more expressive and easier to implement. Unique names are central to composition with FeatureHouse. At coarse granularities, there are syntactic elements with unique names in all languages (e.g., Java classes). As granularity becomes finer, syntactic elements tend to have no or ambiguous names (e.g., Java statements). The syntactic structure of the language affects the granularity at which artefacts can be composed meaningfully.

Element order and granularity
At coarse granularity, the order of elements doesn't affect the program's or document's

semantics (e.g., Java methods). At finer granularity, order of elements become important in most languages (e.g., Java statements, C functions). Superimposition is useful at a level at which an element's order may vary, making it easier to add new elements, else it becomes difficult to insert elements between two existing elements.

Thus there exists a trade-off between granularity, compositional expressiveness and simplicity as illustrated in Figure 26.

Figure 26: Trade-off between granularity, compositional expressiveness and simplicity

Other important results

Composition using FeatureHouse is found to scale well with the number of composition units and lines of code. However, the composition granularity (FST depth) may influence composition time. The time taken for preparing and annotating grammars is moderate compared to implementing the parser generators and adapters from scratch. Also, varying the annotations in the grammar varies the granularity. In practice, only a few composition rules are needed.

Extensions to FeatureHouse

As it is difficult to write the grammar of an XML based language like XHTML in FeatureBNF, an XML schema was developed for describing the structure of XML-based languages. This way, FeatureHouse was extended to integrate new languages not only via FeatureBNF but also via annotated XML schema. In this schema, a combination of XML attributes to represent grammar annotations and XSLT (eXtensible Stylesheet Language Transformations) was used to generate FSTs. Using this schema, it was possible to integrate three more languages into FeatureHouse, namely XHTML, XMI/UML and Ant. The figure 27 shows the XML based schema developed for this extension.

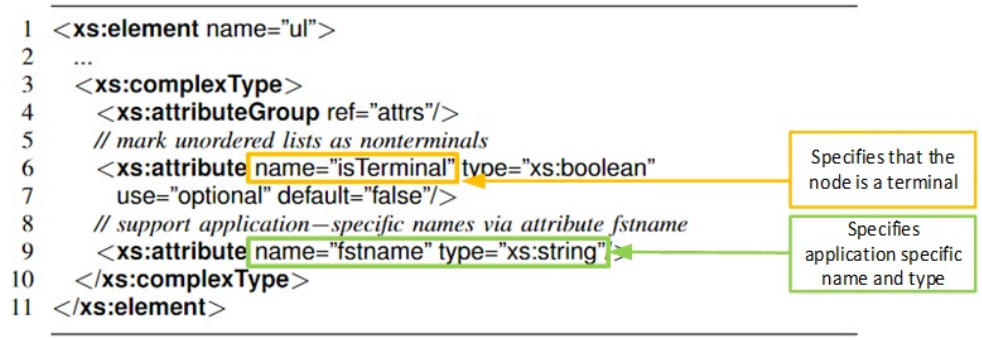

Figure 27: XML schema for extending FeatureHouse for XML-based languages [Sven Apel, Kästner, and Lengauer, 2013, 1

]

3.3.6 Composition by Quantification and Weaving

Another methodology for composition was also evaluated in FeatureHouse, the quantification and weaving process. Quantification is the ability to apply the same generic change in multiple places. Thus, when expressing changes, the points at which changes are applied were specified declaratively. The concept of modification includes two types of specification, traversal and rewrite specifications. *Traversal specification* characterizes the FST nodes that will be affected using composition, whereas *Rewrite Specification* specifies how these nodes will be affected. Modification is performed by an FST traversal. It first determines the nodes to be modified and then applies the necessary modifications to them. Thus, it takes an FST as input and produces a modified FST as output.

Advantages of Composition by Quantification and Weaving

In this approach, it is possible to locate the places of change, by a pattern on FST nodes that the structural elements of a program have to satisfy to be affected by a modification E.g., All methods in a package *util* whose names begin with *SET*. In superimposition, we have to specify each single target node, even though the change is made to all of them in the same way. Once we chose a program, we can find an equivalent FST for every modification that, when superimposed with the program, produces the same results as applying the modification [Sven Apel et al., 2013, 1]. Figure 28 illustrates quantification and weaving approach for FSTs.

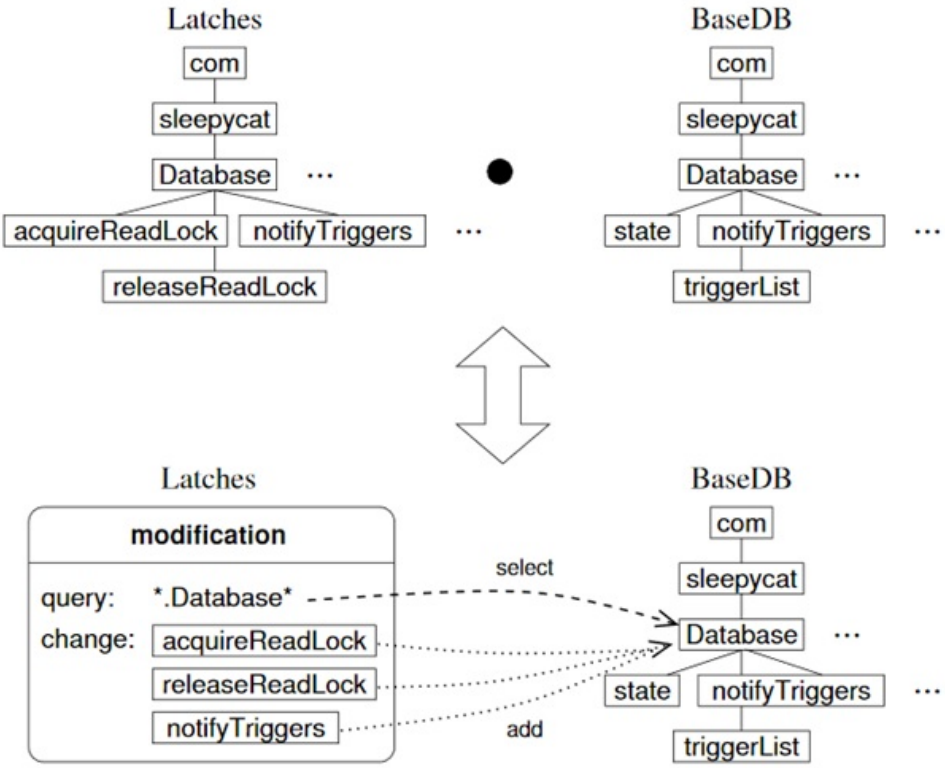

Figure 28: Composition by Quantification and Weaving [Sven Apel, Kästner, and Lengauer, 2013, 1]

This approach is also independent of a particular language. However, it cannot model all mechanisms of fully-fledged languages yet. The programmer can specify certain traversal patterns to select a set of target nodes. Coming to the re-write specification, there are two types of re-writes, a re-write that defines which new elements are added to the nodes selected by the corresponding traversal, and a re-write that defines which new elements are composed with the selected nodes via terminal composition. See Figure 29 for an illustration of this concept:

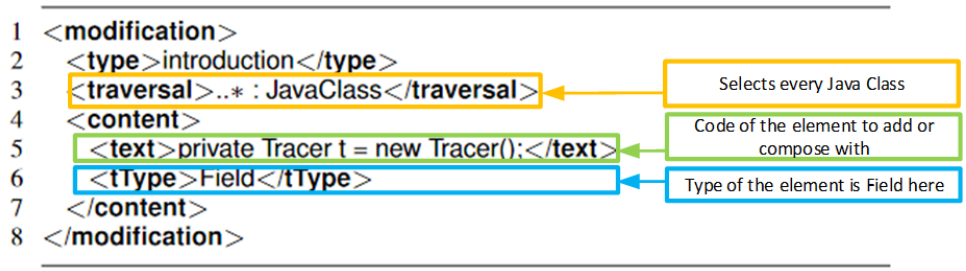

Figure 29: Re-writes in Quantification and Weaving [Sven Apel, Kästner, and Lengauer, 2013, 1]

The quantification and weaving approach is embedded in an XML document. Its implemented on top of FeatureHouse's FST classes using visitors, pattern matching and

terminal composition rules. It has been currently implemented for four software systems, written in two languages. Even though the quantification mechanism is language-independent, individual modifications are not. Thus, in the experiments, similar, but different traversal and rewrite specifications had to be created for both languages.

3.3.7 Results

FeatureHouse was chosen as a candidate for study to evaluate its potential in developing variability-aware feature structure trees. The source code hosted in the website, however, was not updated as the last version was released in March 2011 and hosted on GitHub. The source code has very limited documentation, and hence the understanding of Java language was necessary to interpret the source code of the library. It was found that the FSTGen module, which creates the FSTs is used within the FeatureHouse project, and doesn't expose a usable interface where we can generate the FSTs directly given a directory of software application source code.

Once the FeatureHouse jar archive is downloaded from the FeatureHouse website ["FeatureHouse: Language-Independent, Automated Software Composition", 2020], it is ready to use. Containment hierarchies can be created (file system directories), which contains multiple software artefacts. The FeatureHouse grammar file is in .gcide format, similar to the .g4 format for the ANTLR grammar file. The command below invokes FeatureHouse from the directory in which it was downloaded.

```
java -jar FeatureHouse.jar --expression <configuration file>
```

Here, configuration file has .feature extension, which includes the software artifacts (here, source code files) that need to be composed. The containment hierarchy and the .feature file should have the same name. An example is illustrated in Figure 30.

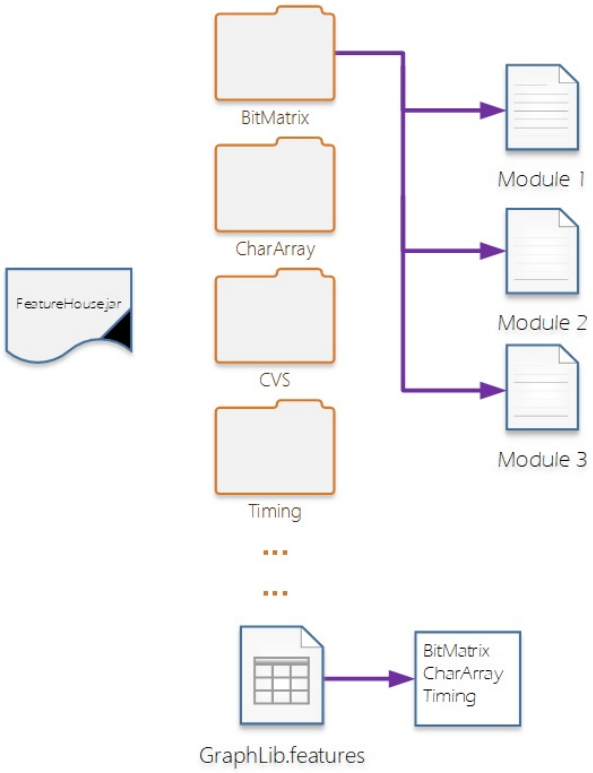

Figure 30: Containment Hierarchy for FeatureHouse library

This command-line execution, however, cannot be used for the thesis research since the objective is to understand the flexibility of FeatureHouse in generating FSTs. As the source code of FeatureHouse is written in Java, in order to port it into Python, a mediator or adapter was necessary. The *Py4J* library is such a bridge between Python and Java. It enabled Python programs running in a Python interpreter to dynamically access Java objects in a Java virtual machine. The Java methods are called as if the Java objects resided in Python interpreter and Java collections can be accessed through standard Python collection methods ["Py4J – Bridge between Python and Java", 2020]. From the gateway object, the `FSTGenComposer` class was invoked. This class generates FST for the given composition unit, using the `getFstNodes()` method, resulting in a FST data structure. This tree is visited using a visitor pattern, that performs depth-first search to generate a graph using *networkx* library in Python.

Different plots of the graph are illustrated below. The first graph (Figure 31) contains all the node types. The second graph (Figure 32) is a stripped-down version of the graph, which includes only #ifdef preprocessor directives, functions and statements. The third graph (Figure 33) is the first graph, with the labels removed, for easier visualization. Here, the *GraphLib* module provided in the sample project was used for the analysis.

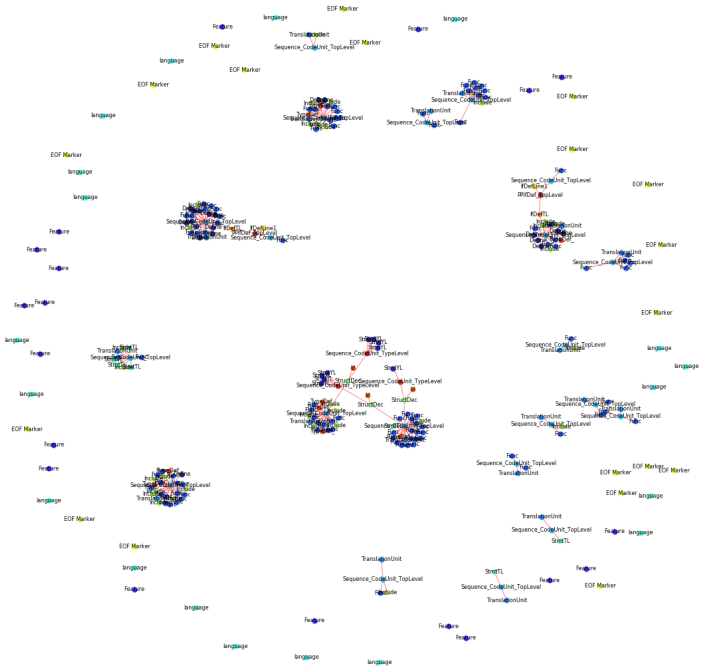

Figure 31: FST for GraphLib example, with all the node types

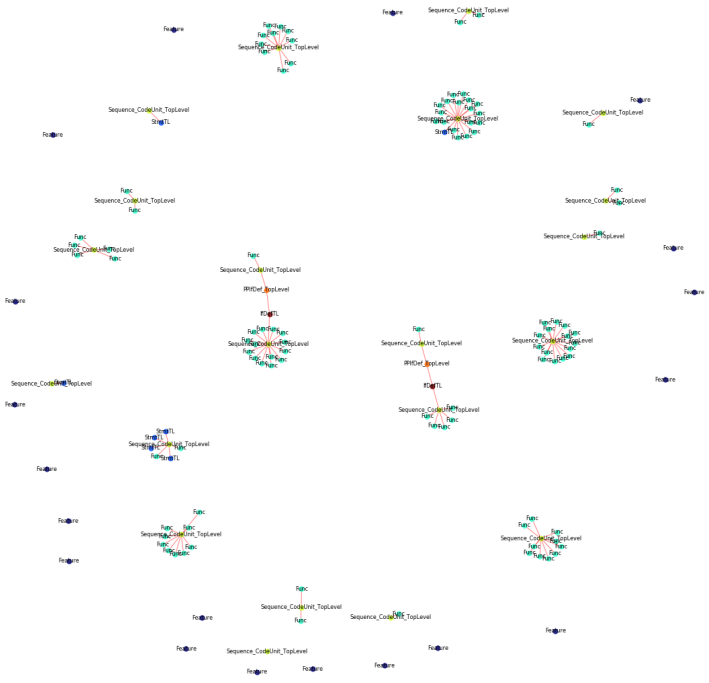

Figure 32: FST for GraphLib example, with only #if directives, statements and functions

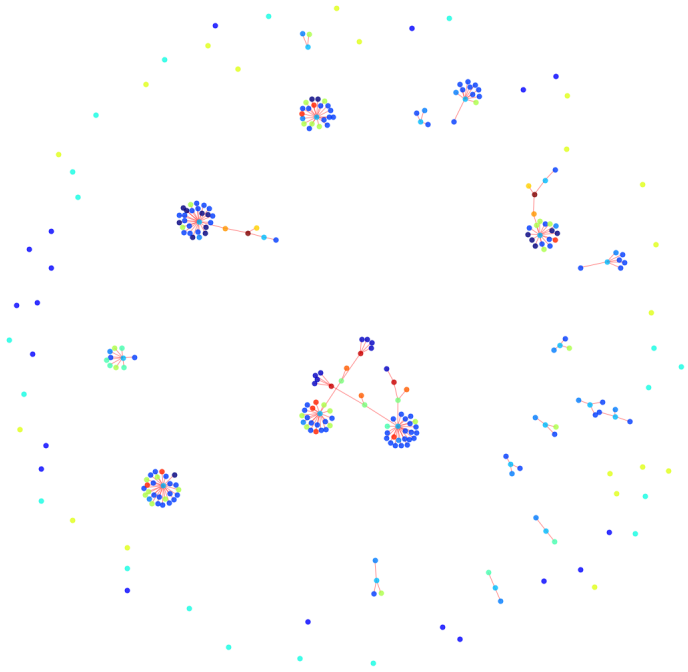

Figure 33: FST for GraphLib example, without labels

The results illustrated can be considered as one step better than the results obtained in the ANLTR example. Firstly, it is possible to generate Feature Structure Trees, which can be visualized and manipulated using the Python libraries. Secondly, the Feature Structure Tree can give dependencies between the different nodes, i.e., the parent-child relationships. However, identifying and extracting variability code elements is not a straightforward process here. FeatureHouse considers functions, methods, and classes as features, and not the actual variation points (#ifdef directives in C and C++ languages, for example). Also, the Feature Structure Tree analyses only the source code and parses it to give the dependencies between the different features described before. It does not give any preprocessing information, say, for instance, the file include graphs, conditional compilation states etc. This is one shortcoming of the FeatureHouse library. Thus, extracting preprocessing information requires a library that behaves similar to a compiler. Nevertheless, the FeatureHouse library can be parked aside as a suitable candidate for parsing grammar in other languages or software artefacts, like requirements, UML, Makefiles, .ini files etc.

3.4 Study on CIDE

CIDE, Colored Integrated Development Environment, is an open-source Eclipse plugin that is used for analyzing and decomposing product line members based on conditional compilation, by annotating code fragments using different colours [Kästner, 2020]. The tool highlights #ifdef directives, for better understanding for developers. Each feature

has a different colour, and the colour differs from the source code. All code fragments that are included when a specific feature (variant) is selected are shown with the same background colour [Feigenspan, Kästner, Frisch, Dachselt, and Apel, 2010].

Since code obfuscation is an inherent problem in variability realization using conditional compilation, special views are designed in CIDE to cope with this challenge. It also helps better understand scattered code in an application, where a specific variant/feature is implemented in several variation points. These views help developer view only those code fragments and files which belong to a specific feature or feature combination. CIDE also includes functionality that previews the code for a specific variant, i.e., it shows only the code of the generated variant by hiding all the irrelevant files and displaying only those files which contain the selected feature implementations.

One shortcoming in this tool is that before annotating a file, one needs to manually define the features and colours in a feature model. Feature models can be in form of a list, or a real feature model with the GUIDSL plugin, which provides a graphical feature model editor from FeatureIDE [Kästner, 2020]. However, it is not possible to specify the dependencies between features, except a parent/implies relationship. Once the feature list is created, one can switch to the Colored Editor view, select a code snippet and assign a feature to it from the feature list. Additionally, there is *ASTView*, which shows the underlying structure (AST) of the code. Once the code has been annotated, one can generate variants by selecting the features to include in that specific variant. When the plugin was used in Eclipse for the FreeRTOS source code, it did not parse successfully with the CIDE plugin as it was showing some compilation errors, though a simple sample C program was parsed successfully. However, it was not possible to colour specific parts of the code, instead, the entire source code was getting coloured. This could be a technical limitation due to the Java and Eclipse versions used.

This plugin was studied to gain an understanding of the different visual representations that could prove useful from a variability code context. Colouring the variability code elements and the code fragment included in a specific feature selection will help programmers identify the various features in the code and also in comprehending the legacy code, an improvement in the problem mentioned in section 1.2.

3.5 Study on PCPP

Considering the shortcomings of the approaches tried out in the previous sections, more studies were conducted along the lines of finding out a library similar to the C/C++ preprocessor, written in Python. PCPP was the first such candidate, a C99 preprocessor written in Python ["PCPP", 2020]. One unique and interesting functionality in this C preprocessor is partial preprocessing, which lets the programmer control how much preprocessing can be done by *pcpp*, programmatically. Thus it facilitates writing custom classes derived from the *Preprocessor* class, which can be used to decide what should be the action when, for instance, a #ifdef directive is encountered during the preprocessing stage, or when a macro is identified.

The library provides various methods which can be overridden in the custom derived class, using which one can identify the variants and variation points ["PCPP API Documentation", 2020]. Also, there are events that get triggered when the following occurs:

- When an include is not found - this can be used to check if a variant does not contain an included file, which would prevent it from compiling successfully.
- When a potential include guard is encountered - this can be used to filter out the include guards from the obtained macros. Include guards are not variabilities, and hence should not be mistakenly considered as one.
- When an unknown directive in an expression is found.
- When an unknown macro is found in a #if directive - useful to identify the errors that occurred during the generation of variants
- When a defined macro is operated on something unknown, is found in a #if directive - this also helps in detecting errors in variant generation.

These are some of the functionalities which are desirable while analyzing and type checking the variability code elements.

3.5.1 Results

The following section illustrates some of the results obtained by creating a custom Preprocessor class from the pcpp library to analyze some of the variability code information. For this study, the *tasks.c* file in FreeRTOS was used to analyze the number of include files, #if directives and macros. The implementation in pcpp was able to identify 71 include files, 557 #if directives and 1267 macros. Figure 34, Figure 35 and Figure 36 show the first 20 results.

	Include	Line No.	Source	Value
0	include	38	src/tasks.c	"FreeRTOS.h"
1	include	56	src/include/FreeRTOS.h	"FreeRTOSConfig.h"
2	include	59	src/include/FreeRTOS.h	"projdefs.h"
3	include	62	src/include/FreeRTOS.h	"portable.h"
4	include	45	src/include/portable.h	"deprecated_definitions.h"
5	include	42	src/include/deprecated_definitions.h	"..\..\Source\portable\owatcom\16bitdos\pc\por...
6	include	47	src/include/deprecated_definitions.h	"..\..\Source\portable\owatcom\16bitdos\flsh18...
7	include	52	src/include/deprecated_definitions.h	"../portable/GCC/ATMega323/portmacro.h"
8	include	56	src/include/deprecated_definitions.h	"../portable/IAR/ATMega323/portmacro.h"
9	include	60	src/include/deprecated_definitions.h	"../../Source/portable/MPLAB/PIC24_dsPIC/portm...
10	include	64	src/include/deprecated_definitions.h	"../../Source/portable/MPLAB/PIC24_dsPIC/portm...
11	include	68	src/include/deprecated_definitions.h	"../../Source/portable/MPLAB/PIC18F/portmacro.h"
12	include	72	src/include/deprecated_definitions.h	"../../Source/portable/MPLAB/PIC32MX/portmacro.h"
13	include	76	src/include/deprecated_definitions.h	"libFreeRTOS/Include/portmacro.h"
14	include	80	src/include/deprecated_definitions.h	"../../Source/portable/SDCC/Cygnal/portmacro.h"
15	include	84	src/include/deprecated_definitions.h	"../../Source/portable/GCC/ARM7_LPC2000/portma...
16	include	88	src/include/deprecated_definitions.h	"portmacro.h"
17	include	92	src/include/deprecated_definitions.h	"../../Source/portable/GCC/ARM7_LPC23xx/portma...
18	include	96	src/include/deprecated_definitions.h	"..\..\Source\portable\IAR\MSP430\portmacro.h"
19	include	100	src/include/deprecated_definitions.h	"../../Source/portable/GCC/MSP430F449/portmacr...

Figure 34: Analysis of FreeRTOS tasks.c file using pcpp - include files

	Directive	Line No.	Source	Value
0	ifndef	28	src/include/FreeRTOS.h	INC_FREERTOS_H
1	ifdef	51	src/include/FreeRTOS.h	__cplusplus
2	ifndef	28	src/include/projdefs.h	PROJDEFS_H
3	ifndef	40	src/include/projdefs.h	pdMS_TO_TICKS
4	ifndef	58	src/include/projdefs.h	configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES
5	if	62	src/include/projdefs.h	configUSE_16_BIT_TICKS
6	ifndef	32	src/include/portable.h	PORTABLE_H
7	ifndef	28	src/include/deprecated_definitions.h	DEPRECATED_DEFINITIONS_H
8	ifdef	41	src/include/deprecated_definitions.h	OPEN_WATCOM_INDUSTRIAL_PC_PORT
9	ifdef	46	src/include/deprecated_definitions.h	OPEN_WATCOM_FLASH_LITE_186_PORT
10	ifdef	51	src/include/deprecated_definitions.h	GCC_MEGA_AVR
11	ifdef	55	src/include/deprecated_definitions.h	IAR_MEGA_AVR
12	ifdef	59	src/include/deprecated_definitions.h	MPLAB_PIC24_PORT
13	ifdef	63	src/include/deprecated_definitions.h	MPLAB_DSPIC_PORT
14	ifdef	67	src/include/deprecated_definitions.h	MPLAB_PIC18F_PORT
15	ifdef	71	src/include/deprecated_definitions.h	MPLAB_PIC32MX_PORT
16	ifdef	75	src/include/deprecated_definitions.h	_FEDPICC
17	ifdef	79	src/include/deprecated_definitions.h	SDCC_CYGNAL
18	ifdef	83	src/include/deprecated_definitions.h	GCC_ARM7
19	ifdef	87	src/include/deprecated_definitions.h	GCC_ARM7_ECLIPSE

Figure 35: Analysis of FreeRTOS tasks.c file using pcpp - #if directives files

	Line No.	Macro	Source	Value
0	35	define	src/tasks.c	MPU_WRAPPERS_INCLUDED_FROM_API_FILE
1	29	define	src/include/FreeRTOS.h	INC_FREERTOS_H
2	29	define	src/include/projdefs.h	PROJDEFS_H
3	41	define	src/include/projdefs.h	pdMS_TO_TICKS
4	41	define	src/include/projdefs.h	xTimeInMs
5	41	define	src/include/projdefs.h	TickType_t
6	41	define	src/include/projdefs.h	TickType_t
7	41	define	src/include/projdefs.h	xTimeInMs
8	41	define	src/include/projdefs.h	TickType_t
9	41	define	src/include/projdefs.h	configTICK_RATE_HZ
10	41	define	src/include/projdefs.h	TickType_t
11	44	define	src/include/projdefs.h	pdFALSE
12	44	define	src/include/projdefs.h	BaseType_t
13	45	define	src/include/projdefs.h	pdTRUE
14	45	define	src/include/projdefs.h	BaseType_t
15	47	define	src/include/projdefs.h	pdPASS
16	47	define	src/include/projdefs.h	pdTRUE
17	48	define	src/include/projdefs.h	pdFAIL
18	48	define	src/include/projdefs.h	pdFALSE
19	49	define	src/include/projdefs.h	errQUEUE_EMPTY

Figure 36: Analysis of FreeRTOS tasks.c file using pcpp - macros

3.6 Study on Clang and LLVM

Clang is a compiler front-end for languages in the C language family like C, C++, Objective C/C++, OpenCL, CUDA and RenderScript [“Clang”, 2020]. It uses the LLVM compiler infrastructure as the backend and provides access to LLVM’s optimizer and code generator. Clang is not only a compiler but also a library for processing source code. It translates text into ASTs, resolves identifiers and symbols, expands macros and tracks source-level location information. It is an open-source project and currently backed by Apple. Even though the API of Clang is in C++, there are Python bindings available. The *libclang* shared library that comes along with Clang additionally supports source code parsing, indexing and cross-referencing, syntax-highlighting and code completion, which makes it a great platform for building a number of source-level tools. This library has been used by Apple in its Xcode development tools.

The LLVM compiler infrastructure project provides a collection of modular compiler and toolchain technologies, which can be used to develop the front-end for any programming language and backend for any instruction set architecture. As mentioned previously, Clang is such a C language family based front-end based on LLVM.

As this is a popular and stable compiler and used as an alternative to gcc, the potential of the libclang library of Clang in parsing source code, especially the preprocessor code

was studied. One big shortcoming for the Python bindings of libclang is that the documentation is dire. However, it was possible to set up and try out basic functionalities of libclang for the purpose of this research by going through the library's source code. In order to set up Clang, one needs to install LLVM binary and set the libclang DLL in `PYTHONPATH`. Once this is done, the C API of libclang can be invoked via the Python wrapper methods.

3.6.1 Results

The results obtained from the Clang experiment was, however, not promising. Using the libclang Python binding, it was possible to only fetch the translation units of the source code, which gives the includes used in the source code, the macros and the #ifdef and #ifndef directives. The library was not able to parse and identify the #if directives either.

A pre-order walk of the resulting AST was performed. However, the preprocessor code was already processed and hence, the AST did not include the variability information (the #if directives) and macros were expanded.

It was also possible to obtain the include graph from the source code. However, this include graph contained only the system includes, and not the user includes in the code. Thus, after analyzing and parsing the tokens from the translation unit, out of 113 #if directives, Clang could find only the #ifdef directives except for #if, which amounts to only 10 results. Out of 9 includes, Clang was able to identify all. Also, Clang could find all 31 macro definitions from the translation unit tokens. However, no further study was pursued in this direction. The results obtained are illustrated in Figure 37, Figure 38 and Figure 39.

	Line No.	Name	Location
0	29	<	src/tasks.c
1	30	<	src/tasks.c
2	38	"FreeRTOS.h"	src/tasks.c
3	39	"task.h"	src/tasks.c
4	40	"timers.h"	src/tasks.c
5	41	"stack_macros.h"	src/tasks.c
6	56	<	src/tasks.c
7	5197	"tasks_test_access_functions.h"	src/tasks.c
8	5203	"freertos_tasks_c_additions.h"	src/tasks.c

Figure 37: Analysis of FreeRTOS tasks.c file using Clang - includes

The includes represented as < are in fact the standard libraries which were not parsed correctly.

	Line No.	Name	Location
0	35	MPU_WRAPPERS_INCLUDED_FROM_API_FILE	src/tasks.c
1	62	taskYIELD_IF_USING_PREEMPTION	src/tasks.c
2	64	taskYIELD_IF_USING_PREEMPTION	src/tasks.c
3	68	taskNOT_WAITING_NOTIFICATION	src/tasks.c
4	69	taskWAITING_NOTIFICATION	src/tasks.c
5	70	taskNOTIFICATION_RECEIVED	src/tasks.c
6	76	tskSTACK_FILL_BYTE	src/tasks.c
7	79	tskDYNAMICALLY_ALLOCATED_STACK_AND_TCB	src/tasks.c
8	80	tskSTATICALLY_ALLOCATED_STACK_ONLY	src/tasks.c
9	81	tskSTATICALLY_ALLOCATED_STACK_AND_TCB	src/tasks.c
10	87	tskSET_NEW_STACKS_TO_KNOWN_VALUE	src/tasks.c
11	89	tskSET_NEW_STACKS_TO_KNOWN_VALUE	src/tasks.c
12	95	tskRUNNING_CHAR	src/tasks.c
13	96	tskBLOCKED_CHAR	src/tasks.c
14	97	tskREADY_CHAR	src/tasks.c
15	98	tskDELETED_CHAR	src/tasks.c
16	99	tskSUSPENDED_CHAR	src/tasks.c
17	106	static	src/tasks.c
18	112	configIDLE_TASK_NAME	src/tasks.c
19	123	taskRECORD_READY_PRIORITY	src/tasks.c

Figure 38: Analysis of FreeRTOS tasks.c file using Clang - macros

	Line No.	Name	Location
0	105	portREMOVE_STATIC_QUALIFIER	src/tasks.c
1	111	configIDLE_TASK_NAME	src/tasks.c
2	568	FREERTOS_TASKS_C_ADDITIONS_INIT	src/tasks.c
3	2028	FREERTOS_TASKS_C_ADDITIONS_INIT	src/tasks.c
4	2528	portALT_GET_RUN_TIME_COUNTER_VALUE	src/tasks.c
5	2962	portALT_GET_RUN_TIME_COUNTER_VALUE	src/tasks.c
6	3848	(	src/tasks.c
7	4483	portLU_PRINTF_SPECIFIER_REQUIRED	src/tasks.c
8	4499	portLU_PRINTF_SPECIFIER_REQUIRED	src/tasks.c
9	5196	FREERTOS_MODULE_TEST	src/tasks.c
10	5205	FREERTOS_TASKS_C_ADDITIONS_INIT	src/tasks.c

Figure 39: Analysis of FreeRTOS tasks.c file using Clang - #ifdef directives

3.6.2 pp-trace

As a final attempt at Clang's tools, the pp-trace tool was studied. *pp-trace* is a standalone tool to trace Clang's preprocessor activity, which is part of the LLVM project. User can

derive custom class from the *PPCallbacks* class and override the specific methods to display the relevant preprocessor information when a source code file runs through Clang's preprocessor. Unfortunately, the pp-trace utility is written in C++ and currently, no Python bindings as available. Though it is possible to generate Python bindings, this activity was not undertaken. Instead, the command-line utility was invoked in Python. The tool outputs result in a high-level *YAML* format.

The command-line utility provides a number of callbacks that can be otherwise overridden using a derived class of PPCallbacks, as mentioned previously. An exhaustive list of callbacks are available in the pp-trace user manual ["pp-trace", 2020].

From the pp-trace utility, the following callbacks were useful:

- InclusionDirective: This callback is invoked when an inclusion directive (#include) is called. This can be used to extract all the files that were included in a specific variant.
- MacroExpands: This callback is called when a macro is being invoked, i.e., defined and referenced. This can be used for extracting all the variabilities that are referenced in a specific variant. The unreferenced ones can be considered to be not used in that specific variant.
- MacroDefined: This callback is called when a macro definition is seen (#defines).
- MacroUndefined: This callback is called when a macro has been #undef'ed.
- If, Elif, Else, Ifdef, Ifndef and Endif: These callbacks are called when the corresponding preprocessor directives are seen.

Another point to emphasize is that for #Else and #Elif and #Endif callbacks, the corresponding #if directive locations can also be obtained. This is useful in tracking the alternate variation point for a specific variation point.

Results

The following figures illustrate the results obtained from the study using pp-trace. Table 2 summarizes the statistics extracted from pp-trace for the tasks.c file in FreeRTOS.

Table 2: Statistics extracted from pp-trace for tasks.c

Variability Code Information	No.
Macro definitions	364
Macro references	20
Include directives	6
Macro undefs	1
#if directive	130
#ifdef directive	4
#ifndef directive	1
#endif directive	135

	Callback	IncludeTok	FileName	IsAngled	SearchPath	RelativePath
340	InclusionDirective	include	stdlib.h	True	/usr/include	stdlib.h
341	InclusionDirective	include	string.h	True	/usr/include	string.h
343	InclusionDirective	include	FreeRTOS.h	False	/usr/include	FreeRTOS.h
344	InclusionDirective	include	task.h	False	/usr/include	task.h
345	InclusionDirective	include	timers.h	False	/usr/include	timers.h
346	InclusionDirective	include	stack_macros.h	False	/usr/include	stack_macros.h

Figure 40: Analysis of FreeRTOS tasks.c file using pp-trace - #includes

	Callback	MacroNameTok	MacroDirective
356	MacroDefined	taskNOTIFICATION_RECEIVED	MD_Define
357	MacroDefined	tskSTACK_FILL_BYTE	MD_Define
358	MacroDefined	tskDYNAMICALLY_ALLOCATED_STACK_AND_TCB	MD_Define
359	MacroDefined	tskSTATICALLY_ALLOCATED_STACK_ONLY	MD_Define
360	MacroDefined	tskSTATICALLY_ALLOCATED_STACK_AND_TCB	MD_Define
363	MacroDefined	tskSET_NEW_STACKS_TO_KNOWN_VALUE	MD_Define
365	MacroDefined	tskRUNNING_CHAR	MD_Define
366	MacroDefined	tskBLOCKED_CHAR	MD_Define
367	MacroDefined	tskREADY_CHAR	MD_Define
368	MacroDefined	tskDELETED_CHAR	MD_Define
369	MacroDefined	tskSUSPENDED_CHAR	MD_Define
373	MacroDefined	configIDLE_TASK_NAME	MD_Define
376	MacroDefined	taskRECORD_READY_PRIORITY	MD_Define
377	MacroDefined	taskSELECT_HIGHEST_PRIORITY_TASK	MD_Define
378	MacroDefined	taskRESET_READY_PRIORITY	MD_Define
379	MacroDefined	portRESET_READY_PRIORITY	MD_Define
382	MacroDefined	taskSWITCH_DELAYED_LISTS	MD_Define
383	MacroDefined	prvAddTaskToReadyList	MD_Define
384	MacroDefined	prvGetTCBFromHandle	MD_Define
387	MacroDefined	taskEVENT_LIST_ITEM_VALUE_IN_USE	MD_Define

Figure 41: Analysis of FreeRTOS tasks.c file using pp-trace - macro definitions

	Callback	MacroNameTok	Range	Args	file_name
455	MacroExpands	tskSET_NEW_STACKS_TO_KNOWN_VALUE	(tasks.c:848:7, tasks.c:848:7)	(null)	tasks.c
488	MacroExpands	prvAddTaskToReadyList	(tasks.c:1129:3, tasks.c:1129:35)	[pxNewTCB]	tasks.c
489	MacroExpands	taskRECORD_READY_PRIORITY	((nonfile), (nonfile))	[<l_paren> pxNewTCB <r_paren> <arrow> uxPriority]	tasks.c
490	MacroExpands	taskYIELD_IF_USING_PREEMPTION	(tasks.c:1141:4, tasks.c:1141:34)	[]	tasks.c
515	MacroExpands	configIDLE_TASK_NAME	(tasks.c:2002:9, tasks.c:2002:9)	(null)	tasks.c
525	MacroExpands	prvAddTaskToReadyList	(tasks.c:2203:6, tasks.c:2203:35)	[pxTCB]	tasks.c
526	MacroExpands	taskRECORD_READY_PRIORITY	((nonfile), (nonfile))	[<l_paren> pxTCB <r_paren> <arrow> uxPriority]	tasks.c
529	MacroExpands	taskYIELD_IF_USING_PREEMPTION	(tasks.c:2265:6, tasks.c:2265:36)	[]	tasks.c
530	MacroExpands	prvGetTCBFromHandle	(tasks.c:2344:10, tasks.c:2344:44)	[xTaskToQuery]	tasks.c
543	MacroExpands	taskSWITCH_DELAYED_LISTS	(tasks.c:2687:4, tasks.c:2687:29)	[]	tasks.c
544	MacroExpands	prvAddTaskToReadyList	(tasks.c:2752:6, tasks.c:2752:35)	[pxTCB]	tasks.c
545	MacroExpands	taskRECORD_READY_PRIORITY	((nonfile), (nonfile))	[<l_paren> pxTCB <r_paren> <arrow> uxPriority]	tasks.c
568	MacroExpands	taskSELECT_HIGHEST_PRIORITY_TASK	(tasks.c:2999:3, tasks.c:2999:36)	[]	tasks.c
573	MacroExpands	taskEVENT_LIST_ITEM_VALUE_IN_USE	(tasks.c:3048:75, tasks.c:3048:75)	(null)	tasks.c
576	MacroExpands	prvAddTaskToReadyList	(tasks.c:3119:3, tasks.c:3119:41)	[pxUnblockedTCB]	tasks.c
577	MacroExpands	taskRECORD_READY_PRIORITY	((nonfile), (nonfile))	[<l_paren> pxUnblockedTCB <r_paren> <arrow> ux...	tasks.c
580	MacroExpands	taskEVENT_LIST_ITEM_VALUE_IN_USE	(tasks.c:3171:57, tasks.c:3171:57)	(null)	tasks.c
581	MacroExpands	prvAddTaskToReadyList	(tasks.c:3183:2, tasks.c:3183:40)	[pxUnblockedTCB]	tasks.c
582	MacroExpands	taskRECORD_READY_PRIORITY	((nonfile), (nonfile))	[<l_paren> pxUnblockedTCB <r_paren> <arrow> ux...	tasks.c
663	MacroExpands	portRESET_READY_PRIORITY	(tasks.c:5101:3, tasks.c:5101:74)	[pxCurrentTCB <arrow> uxPriority, uxTopReadyPr...	tasks.c

Figure 42: Analysis of FreeRTOS tasks.c file using pp-trace - macro references

	Callback	Loc	ConditionRange	ConditionValue
348	If	tasks.c:51:2	tasks.c:51:5	CVK_False
350	If	tasks.c:59:2	tasks.c:59:4	CVK_True
361	If	tasks.c:86:2	tasks.c:86:4	CVK_False
375	If	tasks.c:115:2	tasks.c:115:5	CVK_True
385	If	tasks.c:241:2	tasks.c:241:4	CVK_False
389	If	tasks.c:256:3	tasks.c:256:6	CVK_False
391	If	tasks.c:266:3	tasks.c:266:6	CVK_False
393	If	tasks.c:270:3	tasks.c:270:6	CVK_False
395	If	tasks.c:274:3	tasks.c:274:6	CVK_False
397	If	tasks.c:279:3	tasks.c:279:6	CVK_False

Figure 43: Analysis of FreeRTOS tasks.c file using pp-trace - #if

	Callback	MacroNameTok	Loc
370	Ifdef	portREMOVE_STATIC_QUALIFIER	tasks.c:105:2
443	Ifdef	FREERTOS_TASKS_C_ADDITIONS_INIT	tasks.c:568:2
519	Ifdef	FREERTOS_TASKS_C_ADDITIONS_INIT	tasks.c:2028:4
667	Ifdef	FREERTOS_MODULE_TEST	tasks.c:5196:2

Figure 44: Analysis of FreeRTOS tasks.c file using pp-trace - #ifdef

	Callback	MacroNameTok	Loc
372	Ifndef	configIDLE_TASK_NAME	tasks.c:111:2

Figure 45: Analysis of FreeRTOS tasks.c file using pp-trace - #ifndef

	Callback	IncludeTok	FileName	IsAngled	SearchPath	RelativePath
340	InclusionDirective	include	stdlib.h	True	/usr/include	stdlib.h
341	InclusionDirective	include	string.h	True	/usr/include	string.h
343	InclusionDirective	include	FreeRTOS.h	False	/usr/include	FreeRTOS.h
344	InclusionDirective	include	task.h	False	/usr/include	task.h
345	InclusionDirective	include	timers.h	False	/usr/include	timers.h
346	InclusionDirective	include	stack_macros.h	False	/usr/include	stack_macros.h

Figure 46: Analysis of FreeRTOS tasks.c file using pp-trace - #undef

3.7 Study on CPIP

CPIP is the most promising library that was obtained in this thesis research, which has the maximum flexibility in creating custom classes for analyzing and extracting variability code information. The previous libraries mentioned did give some advantage, however, the flexibility in obtaining preprocessor information was limited. While pcpp provided information on the variation points and variabilities (#defines and #if directives), it did not give complete information on the parent-child relationship between the various nodes in the AST. The FeatureHouse library facilitated FSTs, but this also did not provide any information on the conditional compilation state of the specific nodes. CPIP library can be considered as a solution that plugs these gaps. There are however certain gaps that need to be plugged in this library as well. Nevertheless, it was possible to modify the CPIP library itself, to provide the variability information that was required for the product line members. For the remainder of the work, CPIP has been used as the base library for analyzing and extracting variability code information.

3.7.1 Overview of CPIP

CPIP is a C/C++ preprocessor implemented in Python [“CPIP”, 2020]. It records a number of details related to preprocessing for further inspection and tooling. This is an open-source project hosted in GitHub. CPIP is based on C99 as the standard for C-preprocessor. An overview of preprocessing C and C++ is presented next.

The main task of the preprocessor is to generate translation units for a compiler to work with. This is achieved by file inclusion, conditional compilation, macro definition and

replacement. These are the basic functionalities supported by CPIP. CPIP can generate include graphs and conditional compilation graphs which can be further processed and analyzed based on the needs of the user. In addition, CPIP also keeps track of where macros are defined, un-defined and where they are referenced or substituted. The main advantage of CPIP is that it retains all the preprocessing information that it has discovered along the way and makes it available to the user in different views. User can create custom interfaces with this information to develop further analysis tools.

3.7.2 Architecture of CPIP

Much of the information and graphics provided in this section is adapted from the CPIP documentation ["CPIP", 2020].

At the heart of CPIP architecture [Figure 47] is the *PpLexer* object. In order to construct a PpLexer, user has to provide the following:

- The initial translation unit (ITU), i.e., the file that needs to be preprocessed
- Pre-include files, if any
- An include handler object that handler the #includes
- An optional *CppDiagnostic* object to handle error conditions, and an optional *Pragma-Handler* to handle #pragma statements.

Once this is received, the PpLexer object processes the file token by token with the pp-Tokens() method. This method generates the preprocessing tokens, i.e., the PpToken objects, with the help of the PpTokenizer object. The PpTokenizer keeps track of logical to physical file location. There are various internal objects that keep track of file inclusion, conditional compilation and macro environment. A feature worth highlighting here is that PpLexer maintains all the internal data structures and provides the user with an interface to access them.

CPIP provides a useful command-line utility that preprocesses the given source file and outputs information on preprocessing in HTML and SVG format. However, for this thesis research, the Python library was used to implement different classes with the library modules as a base, to provide more flexibility and to modify for extracting variability code information.

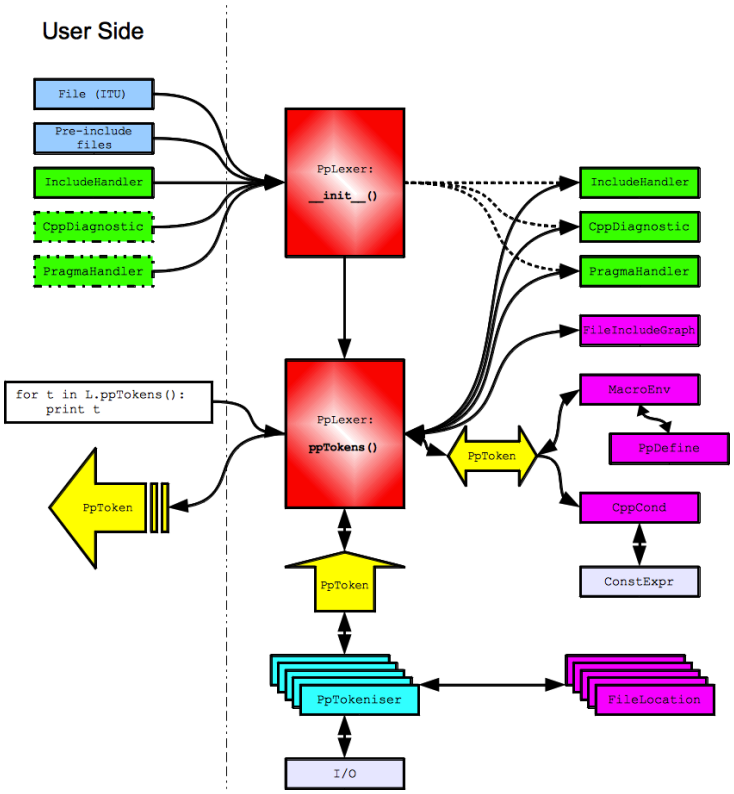

Figure 47: CPIP Architecture ["CPIP", 2020]

Changes were required to be made in the library itself due to the limitation of the library. However, CPIP proved to be the best candidate out of the different libraries and toolchains that were studied for implementing the VITAL 2.0 upgrade.

In the next chapter, an overview of VITAL 1.0 and its advantages and shortcomings will be discussed. Also, the design decisions and details on the implementation of VITAL 2.0 will be presented, which makes use of the CPIP library for implementing some of the functionalities. Finally, the results and evaluation are presented.

4 Enhancement of VITAL Tool

In this chapter, enhancement of the VITAL tool, the instantiation of variability re-engineering method, is proposed. VITAL tool is developed from the previous research work done in Fraunhofer IESE, based on the VITAL (**V**ariability **I**mprovemen**t** **A**nal**y**sis) method. It is an abstract product line improvement process that performs improvement on specific product line artefacts depending on the improvement goals.

4.1 VITAL v1.0 Overview

Variability Improvement analysis (VITAL) is based on an abstract process model, called the MAPE-PL [Zhang, 2015], inspired from the MAPE-K model for autonomic control loops, introduced by IBM [Horn, 2001]. The following summarizes the activities performed for variability improvement:

- **Monitor**: This activity takes existing product line artefacts as input and extracts an artefact model. The product line artefacts may be stored in various forms, like XML, macro definitions, .ini files etc.
- **Analyse**: An automatic analysis is performed on the extracted artefact model to investigate more in-depth knowledge of the product line, by identifying and synthesizing all the information possible, related to that product line artefact. This is then interpreted by domain experts, who formulate improvement ideas.
- **Plan**: A product line improvement plan is made to solve the gaps identified in the previous activity.
- **Execute**: This activity actually implements the improvement plan identified in the previous step.

The VITAL method proposed two solution ideas, one for the problem space, namely *Variability Specification Improvement* and the other, *Variability Realization Improvement*, in solution space. In Variability Specification Improvement, feature dependencies are extracted from the product line artefacts and integrated into a Variability Model, which provides feature recommendations for product configurations. In Variability Realization Improvement, the variability reflection model is extracted from variability specific code. The variability reflection model contains variability elements and their inter-dependencies, and the core assets developed for reuse in domain engineering. In addition to that, it also contains the product configurations realized in application assets, which is used to instantiate the variability code. When a new product variant needs to be instantiated, the domain assets along with the variability elements are configured in the corresponding

product configuration. Based on the values given to the different Vars, the variation point includes or excludes the specific code variant for that specific product variant instantiation.

The main focus of this thesis is in improving the implementation of the variability realization process, through the research goals G1 through G3 mentioned previously, to address the research questions, RQ1 through RQ3 [subsection 1.3].

As the product line members increase in number and the number of variabilities increase, the variability models become intricate and often undocumented. This results in maintainability issues and poses a challenge in comprehending and analyzing variability code realizations. This growing complexity in variability realization is termed variability erosion [Zhang, Becker, Patzke, Sierszecki, and Savolainen, 2013]. The VITAL method proposed the variability realization processes to investigate and propose countermeasures against the variability code erosion problem. To achieve this, the variability reflection model was extracted from variability code realizations and product configuration files. Variability (Var), Variation Point (VP), Code Variant (CV) and Variation Point Group (VPG) are the four types of variability elements that are developed in domain assets for reuse. In addition to extracting these, inter-dependencies between VPs and Vars have also been captured. These ideas are elaborated more in the next section.

4.1.1 Variability elements and inter-dependencies in product line variants

This section is a short refresher of the different variability elements, how they are realized in variability code and the inter-dependencies that can be found in large product lines.

In variability code realizations, which uses conditional compilation, a Variability, Var (feature in problem space) is modelled using macro definitions. A Variation Point (VP) is implemented using conditional definitions, with #if, #ifdef, #ifndef, #elif and #else blocks. The code fragment enclosed in each VP is a Code Variant (CV), which could be present in both positive and negative branches. Each VP could contain multiple macro definitions, or Vars, called variability tangling. The negative or alternate CVs are representative of alternate features.

In addition to the above-mentioned elements, a new concept, called Variation Point Group (VPG) has been introduced in the previous research on VITAL [Zhang, 2015], which contains a group of logically equivalent VPs. A Var can be used in multiple VPGs with different kind of logic for inclusion, but a VPG can have only VPs which implement

identical logic for inclusion. The VPG concept helps in providing a logical modularization for the VPs, which thereby improves the comprehension of variability code (Figure 48).

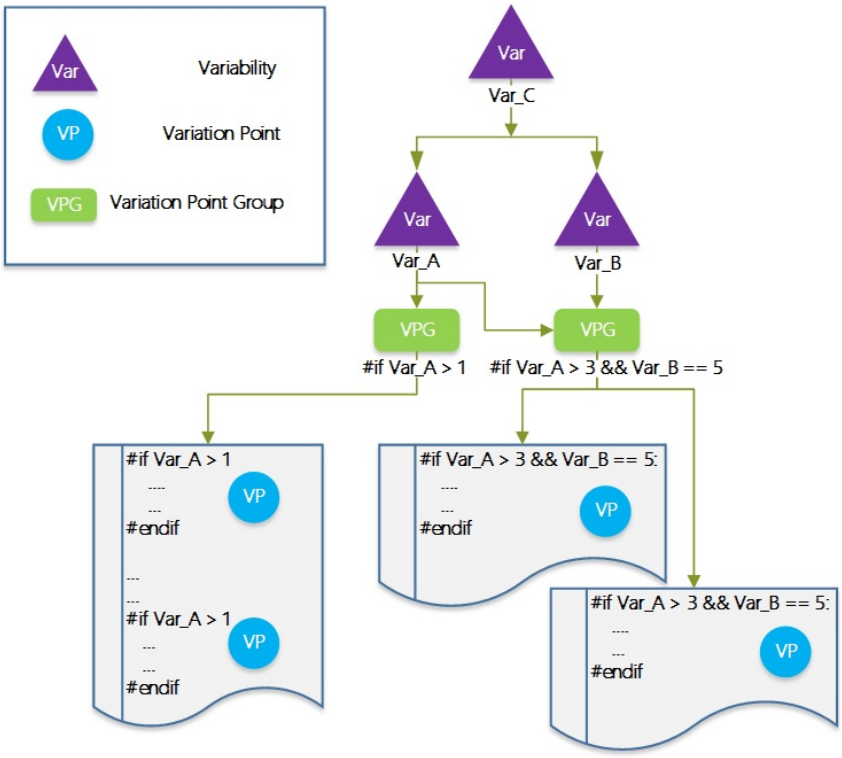

Figure 48: Var, VP and VPG

Furthermore, the variability code elements could have inter-dependencies in large and complex product lines, which is one reason for the incomplete feature model specification. Nested VPs, where the inclusion of a VP depends on its parent VP is one such scenario. There could be nested VPs to many depths and extraction of this information is vital to understanding variability code realization.

In addition to this, understanding product configuration files is also important since variability code realization is performed through variability (macro) definitions using #defines. Thus, the macro definitions need to be extracted from the product configuration realization files. In order to determine the macro constants of variabilities that are present in a specific product variant, the set of all identified macro constants need to be compared with the ones that are used in variation points, i.e., referenced using #if directives. The include guards, which are also defined using macro constants, need to be filtered out from the set of identified macros as these do not represent variabilities. Also, all the macros captured from the variation points need not be variabilities. Identification of this will be a manual effort performed by the domain expert unless the variabilities follow a specific naming convention. The thesis also proposes a novel method for clustering the features

from the source code, which could improve this identification process to a great extent. This will be discussed in the later sections.

The previous study has also proposed a GQM model for variability erosion detection and forecasting. This will not be in scope for this study. The VITAL method has also proposed methods for variability code refactoring which recommends the code fragments that could be replaced with *module replacement* and *parameterized inclusion* techniques instead of the traditional conditional compilation, to reduce the potential for variability code erosion.

4.2 Improved VITAL Process

This section presents an improved VITAL process workflow to aid the different activities in the upgraded VITAL 2.0 tool that is implemented as an outcome if this thesis study. Figure 49 illustrates the VITAL process workflow.

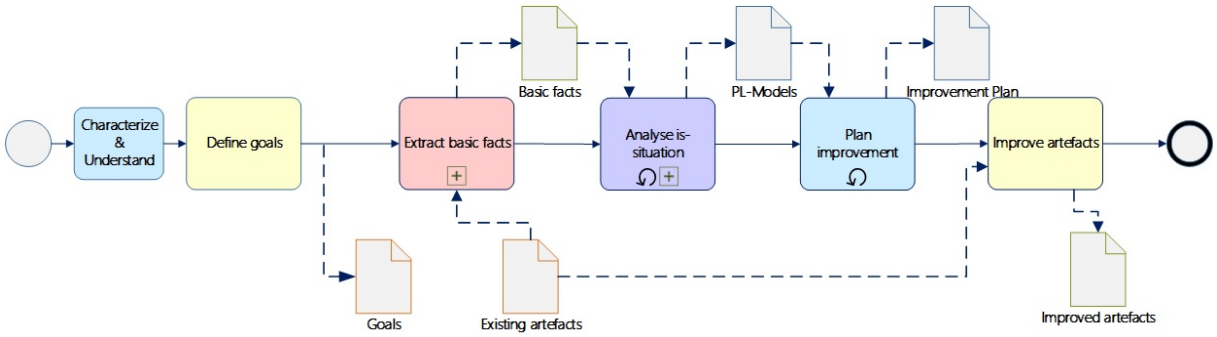

Figure 49: VITAL Process Workflow

Characterize and Understand

The first activity in the VITAL process workflow is to characterize and understand the problem at hand. The product line artefacts to be analyzed will be studied, to understand the new knowledge for the product line. This will be mainly identified by domain experts. The knowledge obtained from the product line artefacts will be used to then define the goals of the task and potential improvement options.

Define Goals

Once the product line artefacts are understood, the goals for the analysis task is formulated. In this study, the goals G1 through G3 are identified to address the research problems RQ1 through RQ3.

Extract Basic Facts

Next activity in the VITAL process work-flow is to extract the basic facts from the product line artifacts.

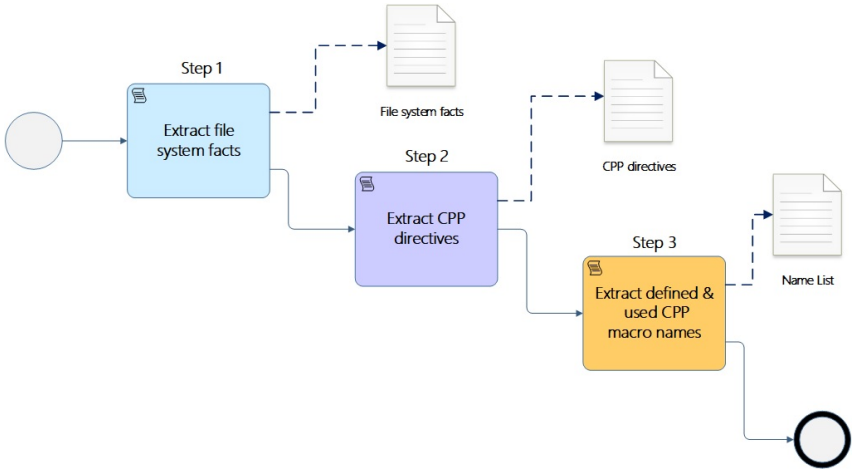

Figure 50: Extract Basic Facts

Each of the steps outlined in Figure 50 produces a tangible output, in the form of a file with an apt data structure that allows further processing and analysis of the output data. In the first step, information like the number of files, number of lines of code in each file, number of bytes etc. are calculated. In the second step, each of the .c, .cpp, .h and .hpp files need to be scanned for preprocessor directives. In the third step, macros are extracted using the #define preprocessor directive and from this list, the referenced or used macro names are identified. This can be done by comparing the set of identified macros with the macros found in the #if directives. From this information, one can identify the defined, but not referenced macros, which helps to detect the potentially unnecessary macros which add to the complexity. In addition, the referenced macros which are not defined can also be extracted, which is an indicator of lacking files or that these macros are defined elsewhere. Furthermore, it is possible to analyze if there are any naming conventions for the preprocessor macro names and the files that contain these macros. The distribution of these CPP directives across the files is another metric worth extracting for future analysis. Other information like, whether the CPP directives follow coding standards can also be used as inputs for further improvement ideas.

Analyse is-situation

In this activity, a detailed analysis of the variability code is performed. Here, the different variability code elements present in the domain assets are extracted, along with their inter-dependencies. In this activity, first, variabilities are identified. In variability code realizations with conditional compilation, variabilities are represented using macro

definitions. However, all the identified macro definitions need not be variabilities. There could be include guards, which need to be filtered out using pattern matching. Also, there could be macro expansions and other macros which do not exactly represent variabilities. Identification of relevant macro names or actual variabilities from this list can be only semi-automated unless the macro names corresponding to variability is named according to a specific naming convention, say, for instance, config_<FeatureName>. If this is not the case, a domain expert will be required to filter out the irrelevant macro names from the actual variabilities. An approach that would improve this filtering process can be achieved through feature clustering techniques. The result of the first step in this activity is a set of relevant macro names.

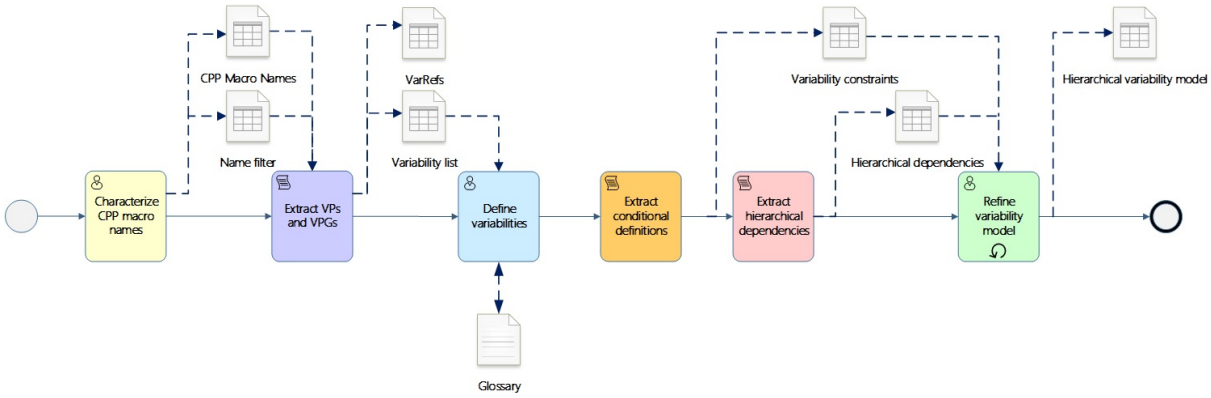

Figure 51: Analyse is-situation

In the next step as illustrated in Figure 51, Variation Points and Variation Point Groups are identified. In variability code realization where conditional compilation is used, Variation Points are realized using #if, #ifdef, #ifndef, #else and #elif directives. Each VP could have multiple Vars. Variation Point Group are Variation Points that contain the same Vars which implement identical inclusion logic. This can be obtained by analyzing the different VPs that are extracted. The result of this step is a set of Vars. A VarRef file is also generated as an output of this step, which includes information about the variability, an identifier for that variability, location of the variability in the file, Variation Point in which the variability is referenced, parent Variation Point if any, its depth etc. This VarRef file can be used for further processing and/or analysis.

In the next step, conditional definitions are extracted. Conditional definitions are #defines that are nested under #if directives. This means that a specific variability (macro definition) is included only if the parent variation point is selected. This can be obtained by identifying nested #if directives and by analyzing if a #define is present in its enclosing code variant (CV). The output of this stage is a list of variability constraints, as

conditional definitions act as a constraint for variabilities.

The next step is to extract the hierarchical dependencies. Hierarchical dependencies are represented by nested #if directives. The algorithm should be able to identify #if nesting at any depth. Variability tangling is another measurement that needs to be extracted. Variabilities are said to be tangled in there are multiple variabilities in a variation point, implementing the inclusion logic. The output of this stage is the set of hierarchical dependencies, with information about the parent VP, siblings, and their file locations etc.

The final step in this activity is to redefine the variability model. Often the variability model may be incomplete or even undocumented due to variability code erosion. The steps performed above help in plugging the gap between variability code realizations and its corresponding variability model. The information obtained from the above steps is used to formulate an updated version of the variability model which includes all the variabilities, Variation Points and inter-dependencies between them. This completes the traceability between the variability model and the realization artefacts. The output of this stage is an updated/redesigned hierarchical variability model.

Plan Improvement
In this activity, a product line improvement plan is made with the updated/redesigned variability model as the input, to cope with the issues or plug the gaps identified in the previous activities. This improvement plan should be supported by a solid improvement strategy and should contain all the necessary information for implementing the strategy. The cost-benefit analysis is made in this activity and the plan needs to be verified by a domain expert. The output of this activity is a documented improvement plan.

Improve Artifacts
In this final activity, the improvement plan is implemented in the existing product line. Once the product line artefacts are improved based on the plan, the impact of improvement is evaluated and verified by experts to see if the goals defined in the first activity have been met.

4.3 Improvement Ideas

The VITAL Src tool has been developed as an outcome of the previous research [Zhang, 2015] with an aim to develop a fully-fledged tool that supports variability code analysis. This section presents the improvement ideas proposed for the tool, which drove the requirements for the VITAL 2.0 upgrade.

The main idea behind VITAL Src is to extract the different variability code elements to generate a *VarRef* file, which can be further used as an input to different variability code analysis techniques. This is achieved by parsing the variability code realizations through various mechanisms and libraries.

To parse variability code realization developed through conditional compilation, a C-preprocessor (CPP) parser was developed in Python, using Pyparsing and SrcML libraries.

Figure 52 shows the block diagram of the VITAL Src.

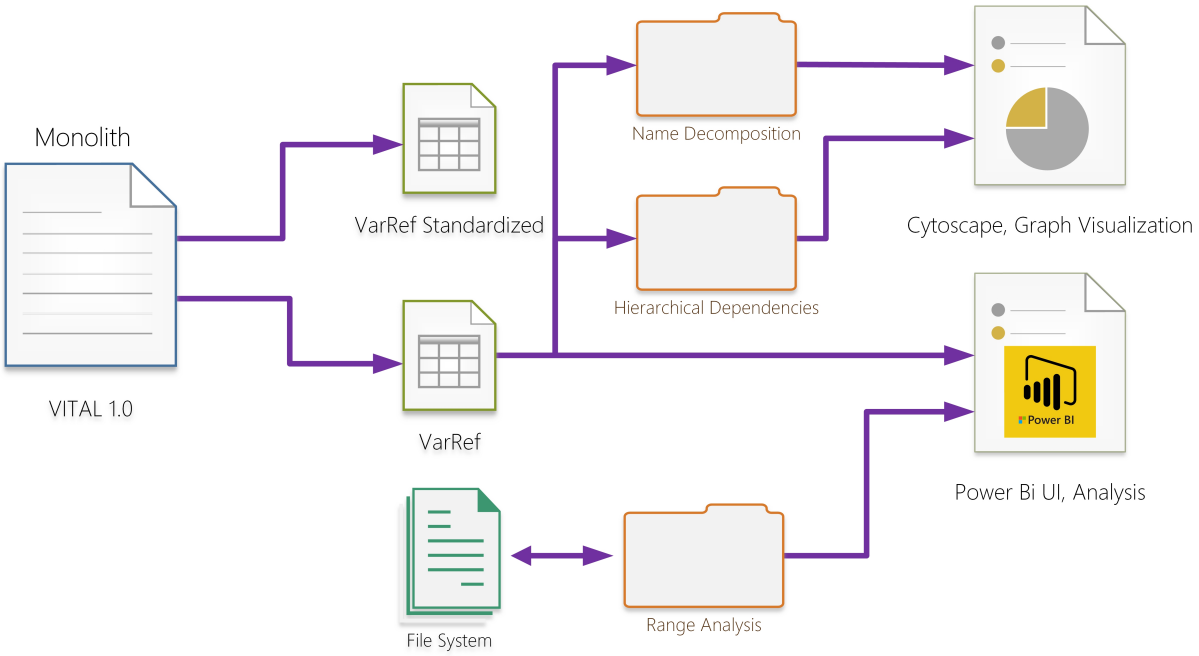

Figure 52: VITAL Src Block Diagram

I1: Modularization

One improvement aspect in the VITAL Src tool is to modularize the tool, as this is developed as a monolith. Modularization of the tools results in individual small utilities which can be invoked via command-line or a GUI. If the requirement needs only a specific functionality to be analyzed for the variability code realizations, it can be done easily by invoking that specific utility in the tool-set. This could be extended to formulate a VITAL toolchain, which consists of variability code analysis utilities, which are light-weight in nature.

I2: Standardizing the CPP parser

The CPP parser used in VITAL Src is developed using RegEx. RegEx becomes overly

complex for extracting VP and Var dependencies. Using a standard CPP parser can overcome this issue, and also make the process of parsing the variability code simpler. In addition, the standard CPP parsers use the ASTs to traverse through the preprocessor elements. Furthermore, it is possible to extract more information on the preprocessor elements from a C-Preprocessor library which contains callbacks when specific preprocessor elements are encountered during the preprocessing stage. Essentially, these libraries perform preprocessing of the source file, and hence it can keep track of the conditional compilation, file includes and macro expansions. Much of the parsing involved using RegEx can be delegated to the library and emphasis can be given in improving the quality of the toolchain, its analysis and variability code measurements. Standard open-source parsers also improve the reliability of the code, as these parsers will follow active development and frequent bug fixes, improved releases and patches.

I3: Automated Feature Clustering to improve filtering of Variabilities
An automated feature clustering technique has been proposed that helps in improving the Variability filtering process performed by a domain expert. This is based on graph-theoretical principles of Adjacency matrices, Eigenvalues and Singular Value Decomposition. This will be elaborated in later sections.

I4: Support conditional execution
Conditional Execution is another variability code realization mechanism in use in the industry. This technique uses if, else if and else conditional statements to implement variation point, similar to #if directives in conditional compilation. However, the difference is that in the former, the entire source code is compiled, irrespective of the conditional expression that is being satisfied, whereas in the latter, only the code fragment selected through #if directives are included in the compilation. Here, variability code is instantiated or bound during compilation, unlike in conditional compilation, where this happens during preprocessing stage. Providing support for conditional execution would add to the functionalities of the VITAL tool, and also helps in moving one step toward the vision of developing a generic parser. The results obtained from the studies conducted in this thesis on FeatureHouse and ANLTR can prove useful in achieving this.

I5: Propose the process to develop a generic parser
The vision of VITAL tooling is to develop a toolchain that can parse any generic product line artefact (requirements, UML diagrams, MakeFiles etc.) and derive the variability information as outlined in the previous sections. To develop this, a systematic software engineering approach needs to be followed. Concrete interfaces between the parser and

the different artefacts need to be formulated and developed. The parser should contain an adapter that can translate the software artefact into a common format that can be used by the parser directly to process it. Similarly, the output of the parser should be a common interchangeable format, which can be extended or used for further analysis, processing or measurements.

4.4 Enhancement Methodology - VITAL 2.0 Upgrade

4.4.1 Migration of VITAL Src 1.0 to Python 3

The current version of VITAL Src was developed in Python 2. However, Python 2 was the legacy language variant, whose official support ended on January 1, 2020, and it is required to port all the existing applications written in Python 2 to Python 3, going forward. With this in mind, the VITAL Src source code was also ported to the Python 3 variant.

This resulted in certain bugs in the code, mainly due to incompatibilities of both versions. After the bug fixes have been made, it was possible to successfully launch the tool and use all of its features.

4.4.2 High Level Requirements/Architecture Drivers

Software architecture is an important phase in the development life-cycle of a software-centric system. It defines the structure of the system, comprising the elements, their externally visible properties and the relationships among them. It is also a set of principal design decisions made about the system. Architecture bridges the gap between problem space and solution space.

The following are the key high level functional and quality requirements derived from the studies in this thesis and experiences from VITAL Src 1.0 tool. These serve as the architecture drivers for the tool.

- **M_FR1**: The tool shall have each major functionality provided as a utility that can be invoked independently
- **M_FR2**: The independent utilities within the tool shall produce outputs in a common format that allows further processing including visualization and data analysis.
- **M_FR3**: The tool shall be accessible via command-line
- **M_FR4**: The tool shall embody all the functionalities provided by VITAL Src

- **O_FR1**: The tool shall have a graphical user interface [Optional]
- **M_QR1**: The tool shall be modular

Here, the prefixes **M** and **O** signifies mandatory and optional requirements respectively.

4.4.3 Decision Rationale

The following are the major decision rationale taken for the development of the tool that is derived from the architectural drivers:

- **DR1**:
  Decision: The application shall be developed in Python
  Rationale: Python contains various active libraries that help in c preprocessor parsing, data storage and visualization. It is platform-independent and open-source.
- **DR2**:
  Decision: CPIP Parser library shall be used for processing pre-processor directives
  Rationale: CPIP Library has the most flexibility among the various options studied in this thesis. Refer to previous sections on studies for more clarity.
- **DR3**:
  Decision: Pandas DataFrame format shall be used for storing the output from each utility
  Rationale: The DataFrame structure offered by the Pandas library in Python is highly flexible and has adapters for many different views, for instance, plotting, data analysis etc. It is intuitive (tabular structure) and simple to manipulate data with this data structure.

4.4.4 Architectural Views

High-Level Design

The high-level design of VITAL 2.0 Upgrade is illustrated in Figure 53.

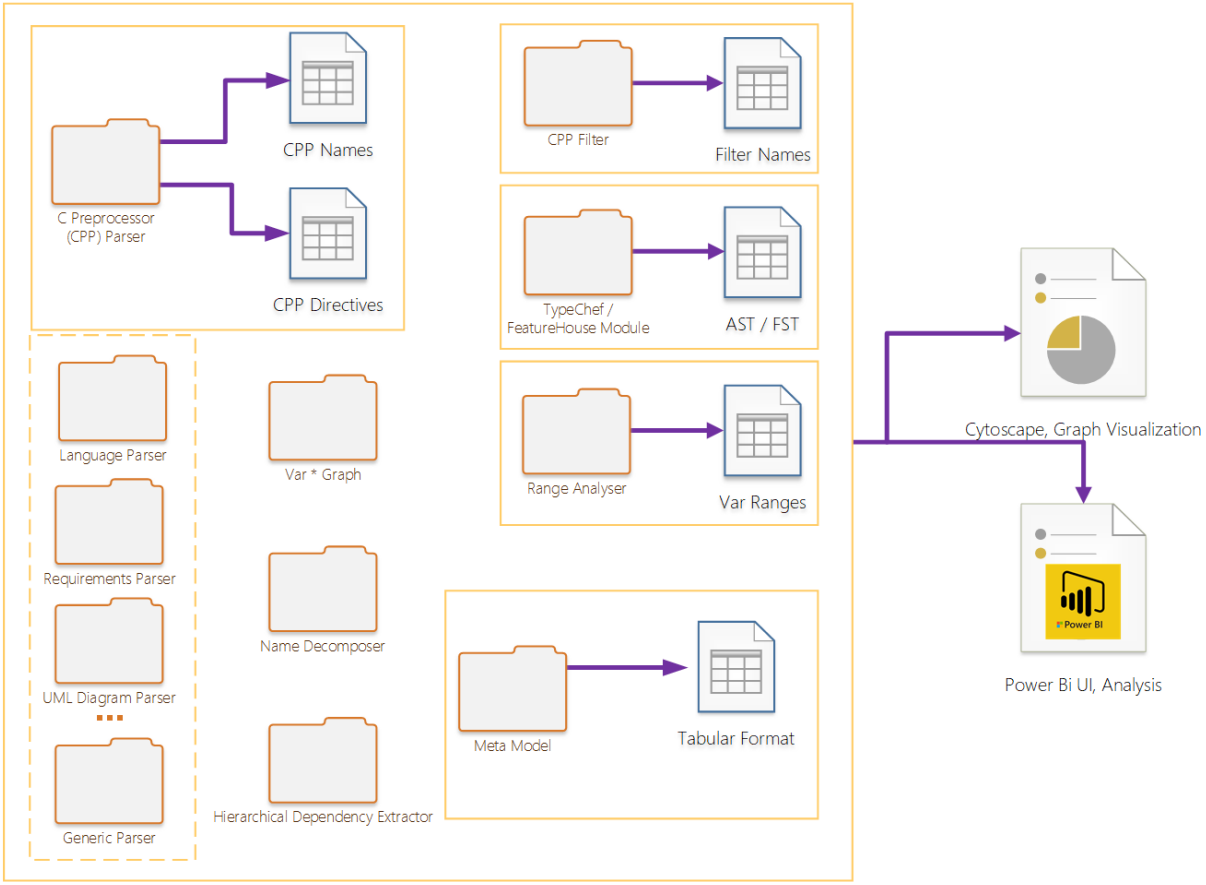

Figure 53: VITAL 2.0: High-Level Design

The following are the modules implemented in the tool:

- **M1: Fact Extractor**: Extracts basic facts about the given product line artifact
  Input: A directory containing product line source files
  Output: A Pandas DataFrame containing basic file system facts and location.

- **M2: CPP Directive Extractor**: Extracts all CPP directives.
  Input: A directory containing product line source files
  Output: A Pandas DataFrame consisting of all identified CPP macro definitions and its location.

- **M3: Referenced Macro Identifier**: Identifies the referenced macros from the list of macros.
  Input: A Pandas DataFrame containing all identified CPP macro definitions OR A directory containing product line source files.
  Output: A Pandas DataFrame containing all the referenced macros in the product line and its location.

- **M4: Used Macro Identifier**: Identifies all the CPP macros that are used, but not defined.
  Input: A Pandas DataFrame containing all identified CPP macro definitions OR
  A directory containing product line source files.
  Output: A Pandas DataFrame containing all the used macros in the product line and location.

- **M5: Variability Extractor**: Extracts all possible variabilities (macro definitions) in the product line, removing include guards
  Input: A Pandas DataFrame containing all identified CPP macro definitions OR
  A directory containing product line source files.
  Output: A Pandas DataFrame containing all the identified variabilities and location.

- **M6: Include-guard filter**: Identifies all include-guards from the given product line.
  Input: A Pandas DataFrame containing all identified CPP macro definitions OR
  A directory containing product line source files.
  Output: A Pandas DataFrame containing all the identified include guards and location.

- **M7: Variation Point Extractor**: Extracts all the Variation Points (statements with #if directives) in the product line
  Input: A directory containing product line source files.
  Output: A Pandas DataFrame containing all the identified Variation Points and location.

- **M8: Variation Point Group Extractor**: Extracts all the variation point groups from the variation points in the product line.
  Input: A directory containing product line source files OR
  A Pandas DataFrame containing all the variation points.
  Output: A Pandas DataFrame containing all the variation point groups and location.

- **M9: Include Graph Generator**: Generates the include path for all the file includes.
  Input: A directory containing product line source files
  Output: A Pandas DataFrame containing include file hierarchy which can be plotted as a graph.

- **M10: Conditional Definition Extractor**: Extracts conditional definitions from the product line source file
  Input: A directory containing product line source files.
  Output: A Pandas DataFrame containing the variability constraints with parent-child relationship and its location.

- **M11: Hierarchical Dependency Extractor**: Extracts hierarchical dependencies between variation points using #if directive nesting
  Input: A directory containing product line source files.
  Output: A Pandas DataFrame containing the hierarchical dependencies with parent-child relationships and its location.

- **M12: Variability Tangle Identifier**: Identifies variability tangling among the variabilities in variation points.
  Input: A directory containing product line source files OR
  A Pandas DataFrame containing all the identified variation points.
  Output: A Pandas DataFrame containing all identified variability tangles and its location.

- **M13: Name Decomposer**: Decomposes the variability names to identifiable clusters
  Input: A Pandas DataFrame containing all the identified variabilities OR
  A directive containing product line members.
  Output: A Pandas DataFrame containing the identified names and the related variabilities

- **M14: Variability Range Analyzer**: Identifies the range of all the variabilities in the product line.
  Input: A directory containing product line source files OR
  A Pandas DataFrame containing all the variabilities
  Output: A Pandas DataFrame containing variabilities and its valid ranges.

- **M15: Undef Macro Identifier**: Extracts all the macros that are undefined.
  Input: A directory containing the product line source files OR
  Output: A Pandas DataFrame containing all the undefined macros.

- **M16: Conditional Compilation Graph**: Identifies the conditional compilation state of the variation points in a product line member
  Input: A directory containing product line source files
  Output: A Pandas DataFrame containing the conditional compilation state of all the variation points.

- **M17: Static Macro Dependency Extractor**: Identifies the static dependencies between macros/variabilities defined in a product line member
  Input: A directory containing product line source files OR
  A Pandas DataFrame with the defined macros.
  Output: A Pandas DataFrame containing the static dependencies between the defined macros.

For this project, only the context and functional views have been created.

Context View

In context view, the system and its boundary are defined. In this view, the system is seen as a black box, and the humans and the external systems interacting with the system under analysis are identified, as well as the interaction between them.

In our case, the context system is the VITAL 2.0 application that is being developed. The external systems include the product line artefacts, which are fed to, and processed by the system. In addition, the third-party visualization and analysis tools act as external systems. The VITAL tool receives information from the product line artefacts in form of raw, unprocessed data in form of directories and files, and sends it to the external world, a *Pandas DataFrame* structure which can be used by analysis and visualization tool. This is illustrated in Figure 54.

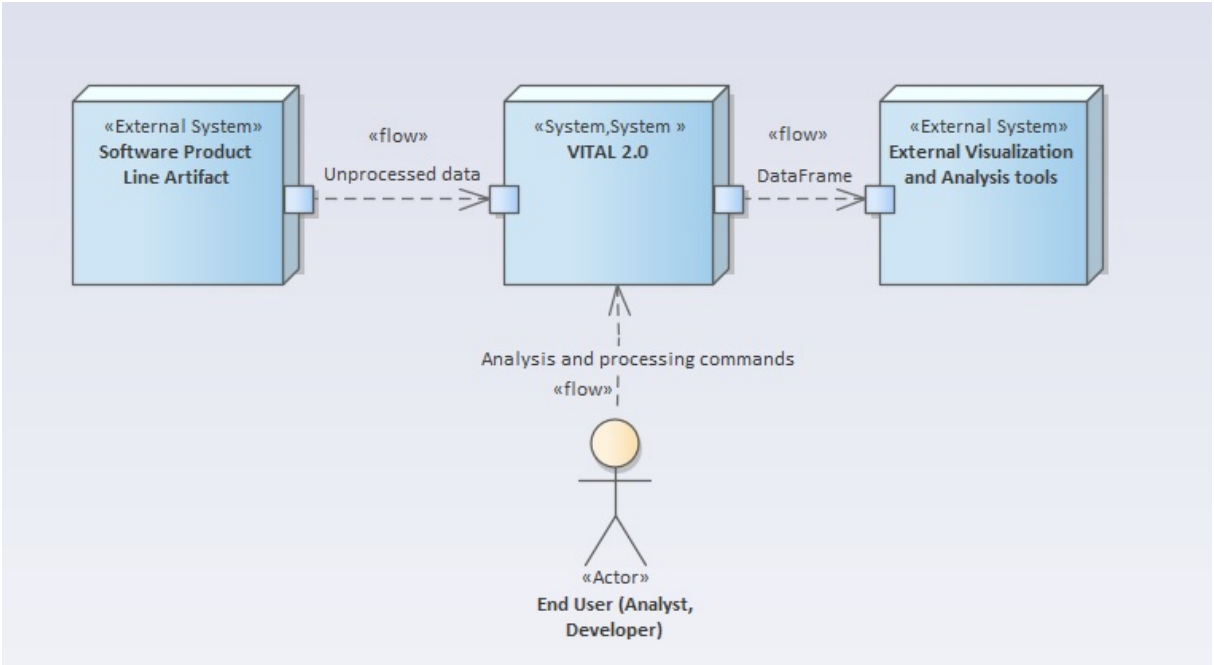

Figure 54: Context View of VITAL 2.0

4.4.5 Implementation and Results

The activities mentioned in subsection 1.2, A1 through A3 have been elaborated in subsection 2.3, which satisfies the research goal G1 (subsection 1.3). This section elaborates the implementation of activity A4, which satisfies goal G2. The next chapter describes activity A5, which satisfies goal G3. The functional views of the important modules are presented and elaborated in the next section.

Functional View of Preprocessor Parser

At the heart of VITAL 2.0 tool is the *Preprocessor Parser* module which creates the *Preprocessor Lexer* and *Include Handler* objects. The Include Handler object manages the #includes and in order to correctly process the file, the include directories need to be specified. This Include Handler object is then passed into the Preprocessor Lexer class, along with the source files. The resultant Preprocessor Lexer Object is the core of the parser module, using which all the processing will be performed. The functional view (Figure 55) illustrates this.

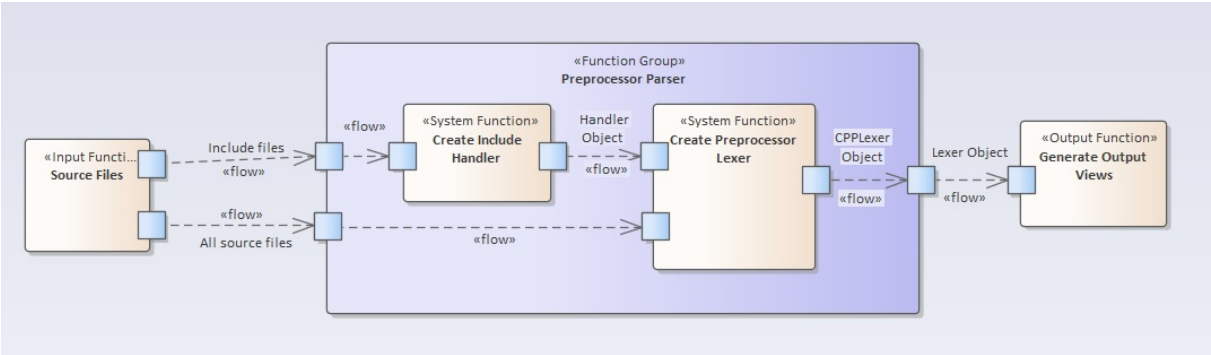

Figure 55: Functional View of CPP Parser

Functional View of Modules M10, M11 and M16

These modules generate the hierarchical dependencies between variation points and conditional definitions. A conditional compilation graph is first generated from the Preprocessor Lexer object. This graph is a generic graph data structure that can be walked or visited using a depth-first search algorithm. For this, a visitor class needs to be derived from the base visitor class of the CPIP library. This visitor will then visit different nodes in the graph and extract the dependencies between the #if directives and the conditional definitions, implemented through two different functions. These functions output Pandas DataFrame and a graph data structure corresponding to each functionality.
Figure 56 represents the functional view for this concept.

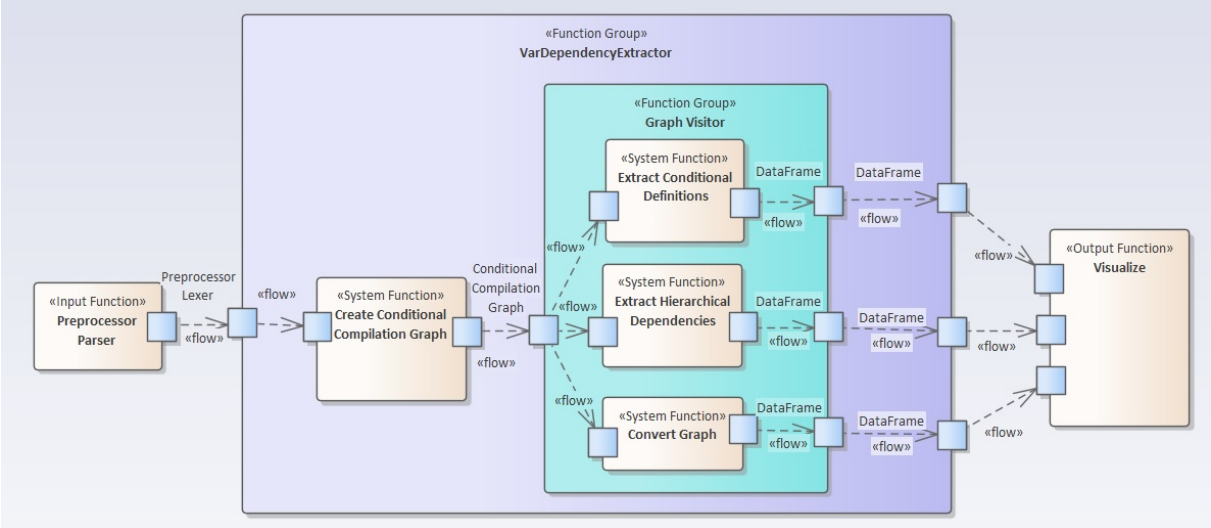

Figure 56: Functional View of Variability Dependency Extractor

Functional View of Modules M2, M3, M4, M13, M14, M15, M16

For the processing related to macros or variabilities, a Macro Environment Component has been implemented. This contains different methods that extract information from macro definitions. This module is the base for the implementation module to extract information related to variability, variation point and variation point groups. The macro environment component is illustrated in Figure 57. The data flow within the modules is not displayed for sake of simplicity.

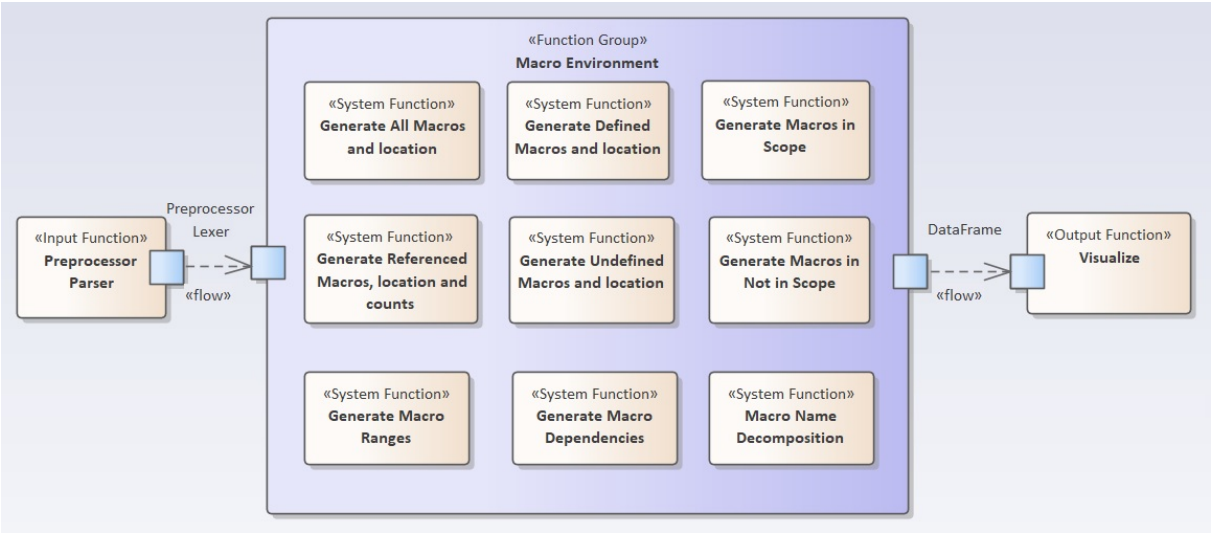

Figure 57: Functional View of Macro Environment Component

Functional View of Modules M5, M6, M7, M8, M12

The functional view represented in Figure 58 illustrates the functional decomposition of the modules that extract variability information. Note that this module uses the output

of the macro environment component.

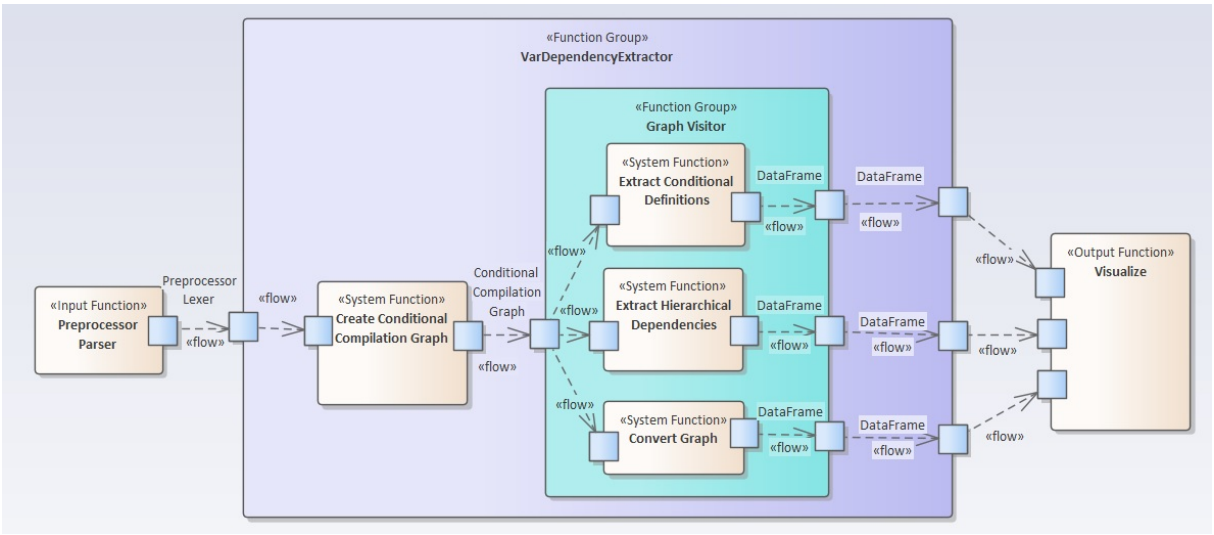

Figure 58: Functional View for Variability Parser

Functional View of Module M9

To generate file include graph, the Preprocessor Lexer Object is used. This object is passed into the method that traces all the file includes and the locations. This information will be generated in form of a graph and a depth-first search can be performed on this graph to extract the file include history in the desired form. For this, a visitor class is implemented that visits each node in the graph and generates a pandas DataFrame. This is illustrated in the functional view in Figure 59.

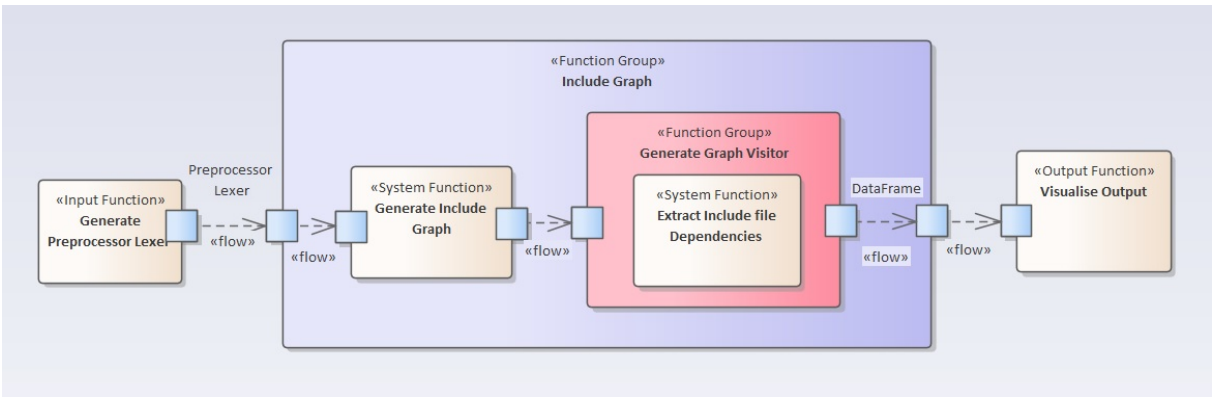

Figure 59: Functional View for File Include Graph

Functional View of Module M1

The fact extractor for files in a software product line is implemented by first performing a directory walk, i.e., recursively extracting the files contains in all the sub-directories of the directory. This generates a list of relevant files (.c, .cpp, .h, .hpp files) which is then

parsed to extract file information like the number of lines of code, number of bytes etc. The functional view in Figure 60 illustrates this.

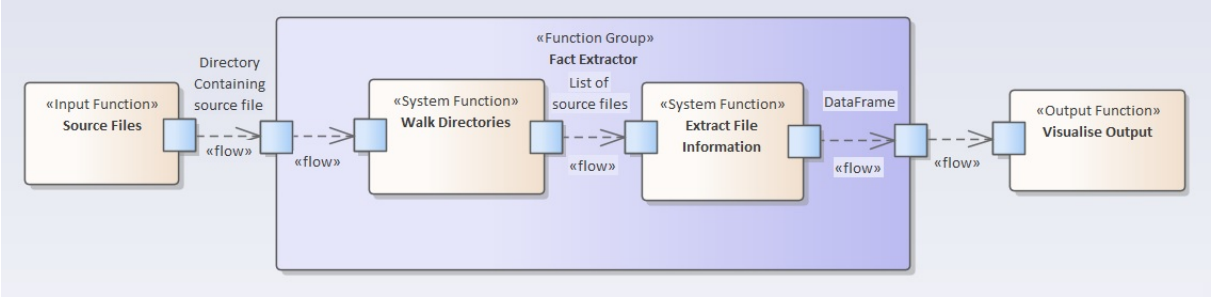

Figure 60: Functional View for File Fact Extractor

4.5 Automated Feature Clustering

As discussed in previous chapters, variability modelling is one of the key aspects of variability management, where the variabilities and commonalities of software product line members are documented. This is often achieved through feature models. The features in the feature model are developed as core assets for reuse (domain assets) and are instantiated while creating product line members in application engineering. This feature model is thus used in deriving product line members, through the product line configuration files derived from these models. However, as the product line features become intricate due to the growing complexity of the product, the product line variants will no more be traceable to the feature model, resulting in variability code erosion. If feature models are not present or are not in sync with the variability code realization, one has to derive the feature model or product configuration from the product variant through a bottom-up approach, in order to assist in the future product configuration process. This is currently a semi-automated task, which requires a domain expert. However, if the domain expert is not present, deriving product configuration becomes a herculean task since one has to understand the variability code realization and the underlying structure. The method proposed here is an attempt to plug this gap, i.e., by automatically identifying the prominent features from the variability code realization of product lines. When there are existing product configurations, deriving feature models from them are addressed in various researches [Zhang, 2015]. However, this approach tries to cluster the features from a solution space perspective, i.e., by analysis of variability code realizations.

4.5.1 Feature Diagram represented as a Directed Graph

Feature Diagrams share many aspects based on graph-theoretical concepts. This section describes the representation of feature diagram in the form of a graph and elaborates the relationships between features in a feature diagram in the language of graphs [Laguna, Marques, and guez-Cano, 2011].

A directed graph G is a set of ordered pairs (V, E), where V is a non-empty set of vertices (or nodes) and E is a set of directed edges.
G = (V, E)
A feature can be considered as a coloured vertex v_i^c, such that $v_i^c \in V^c$, where colour can be either black or white [S. Apel et al., 2013], in which white denotes that the feature is not selected/included, and black denotes that the feature is selected/included. A relationship between features is a sub-graph, $G_r(V_r^c, E_r^c)$, where $V_r^c = v_p^c \bigcup V_d^c$. v_p^c is the parent vertex or root node of the feature graph and V_d^c is the set of all descendant or child nodes of the parent node v_p^c, where $v_p^c \in V^c$ and $V_d^c \subseteq V^c$. Thus from a feature perspective, the feature model is a bi-coloured directed graph G(V^c, E^c), formed by the composition of bi-coloured directed Tree T(V^c, U^c) and set of edges B, representing constraints between the vertices, $V_B^c \subseteq V^c$, where U^c is a set of directed edges, which represents parent-child relationships between a pair of vertices or nodes. Thus, $E^c = U^c \bigcup B$.

Mandatory, Optional and Alternative Features
Feature selection for deriving a product variant can be considered as a feature graph colouring process. A white-colored edge implies that a child (descendant) vertex $v_s^c, v_d^c \in V_d^c$ may be *possibly* selected if its parent v_p^c is selected. A black-coloured edge, on the other hand, implies that the child vertex x_d^c must be *necessarily* selected if the parent vertex is selected. The child feature that is related to the parent feature by the black-coloured edge is a mandatory feature, whereas a child feature that is related to the parent feature by a white-coloured edge is an optional or alternative feature. In the optional feature, any number of child features may be selected independently from its feature group, whereas in the alternative feature, exactly one child feature must be selected from its feature group.

Variants and Variation Points
A feature with no parent is a root feature. A feature that is a parent of an alternative or optional feature group is considered as a *variation point*. The features which are leaf nodes, i.e., without any child nodes are *variants*.

Feature Constraints

Constraint between two variants, v_i^c and v_j^c is a predicate $b_t : (xv_i^c, v_j^c) \longrightarrow true, false$, where $b_t \in B$, the edges representing constraints. This predicate evaluates to true if the constraint exists, else, it evaluates to false.

Requires and Excludes Constraints

In a *Requires* constraint, the selection of one variant requires the selection of another variant. In graph theoretical notation, if $b_req(v_i^c, v_j^c) \longrightarrow true$,

$$(color(v_i^c) \longrightarrow black) \longrightarrow (color(v_j^c) \longrightarrow black)$$

In an *excludes* constraint, on the other hand, selection of one variant excludes the selection of another variant. In graph theoretical notation, if $b_excl(v_i^c, v_j^c) \longrightarrow true$,

$$(color(v_i^c) \longrightarrow black) \longrightarrow (color(v_j^c) \longrightarrow white)$$

Feature path, F^p is a sub-graph of G which contains only the features selected (coloured black) for a specific product variant. Thus, a configuration c is a multiset of all selected features in a feature path.

4.5.2 Methodology

Adjacency Matrix

The feature structure tree generated from the FeatureHouse library is converted to an adjacency matrix. For a simple graph G(V, E), the adjacency matrix is a square, $|V| \times |V|$ matrix A such that, $A_{ij} = 1$, if vertices i and j have a direct edge, and 0 otherwise. Instead of the values 0 and 1, the adjacency between vertices or nodes i and j can be weighted. Another representation worth exploring is the distance matrix, which gives the shortest distance between two vertices i and j.

Each row of adjacency matrix corresponds to a node in the FST/AST and the values in the columns indicate whether that specific node has adjacent nodes. Intuitively, one can see that the adjacency matrix indicates whether a specific node in the AST is connected or related to another node or node. For example, a set of function calls or statements enclosed within a conditional compilation block, or variation point, realized using #if directives, will be adjacent nodes to the variation point node.

Singular Value Decomposition

Once the adjacency matrix is generated, singular value decomposition (SVD) is performed

on the adjacency matrix. SVD is one of the most commonly used unsupervised learning algorithms today. Eigen decomposition is another technique for feature clustering, which follows the same principle as SVD, but it can decompose only square and symmetric matrices. SVD algorithm overcomes this issue and hence can be used for both rectangular as well as square matrices. The SVD of a square matrix is intuitively related to its eigen decomposition. A singular value decomposition of an mxn matrix A, with rank r, is an orthogonal decomposition of a matrix into three matrices, such that:
$A_{mxn} = U_{mxr} S_{rxr} V^T_{rxn}$
The columns of U are orthogonal to each other, and so are the columns of V_T. The matrix S is a diagonal matrix that contains the singular values of A, in descending order. The three matrices U, S and V can be multiplied together to get the original matrix A.

Truncated SVD
Furthermore, it is also possible to find an approximation of matrix A of rank k, where k $<$ r, by multiplying only the first k columns of U, the first k values in S and the first k rows of V^T. This is called the truncated SVD of a matrix. The truncated SVD gives the best possible rank k approximation of the matrix A. The value to be used for k can be determined by finding the index for which the magnitude of the singular values significantly drops.

After performing SVD on the adjacency matrix, the approximation matrix is found using the above method. This removes the nodes and dependencies which does not contribute much to the high-level features of the product line members.

Clustering using signs of the singular vectors
The singular vectors hold a wealth of information related to the connectivity of graphs similar to eigenvectors. SVD yields two singular matrices, the left singular matrix U and the right singular matrix V^T. The left singular matrix needs to be used to cluster along the rows whereas the right singular matrix clusters along the columns. In our scenario, clustering needs to be performed along the rows since the rows of the adjacency matrix denoted the different nodes of the feature graph.

In order to find clusters, the technique proposed in the *Extended Fiedler method* is used. Miroslav Fiedler proved that the eigenvector corresponding to the second smallest eigenvalue, Fiedler vector, indicates how a graph can be broken down into *maximally intra-connected components and minimally interconnected components* [Miroslav, 1973] [Miroslav, 1975]. In the Extended Fiedler method, rows that have the same sign patterns

in the first k singular vectors of the left singular matrix are grouped together, forming clusters. The value of k is chosen based on the granularity of features that need to be identified. However, it does not make sense in choosing k > 6 as this would not lead to meaningful clusters in our case.

4.5.3 Results

FreeRTOS library was used to generate the FST with the FeatureHouse API. Figure 61 gives the spring graph of the nodes identified in the feature structure tree.

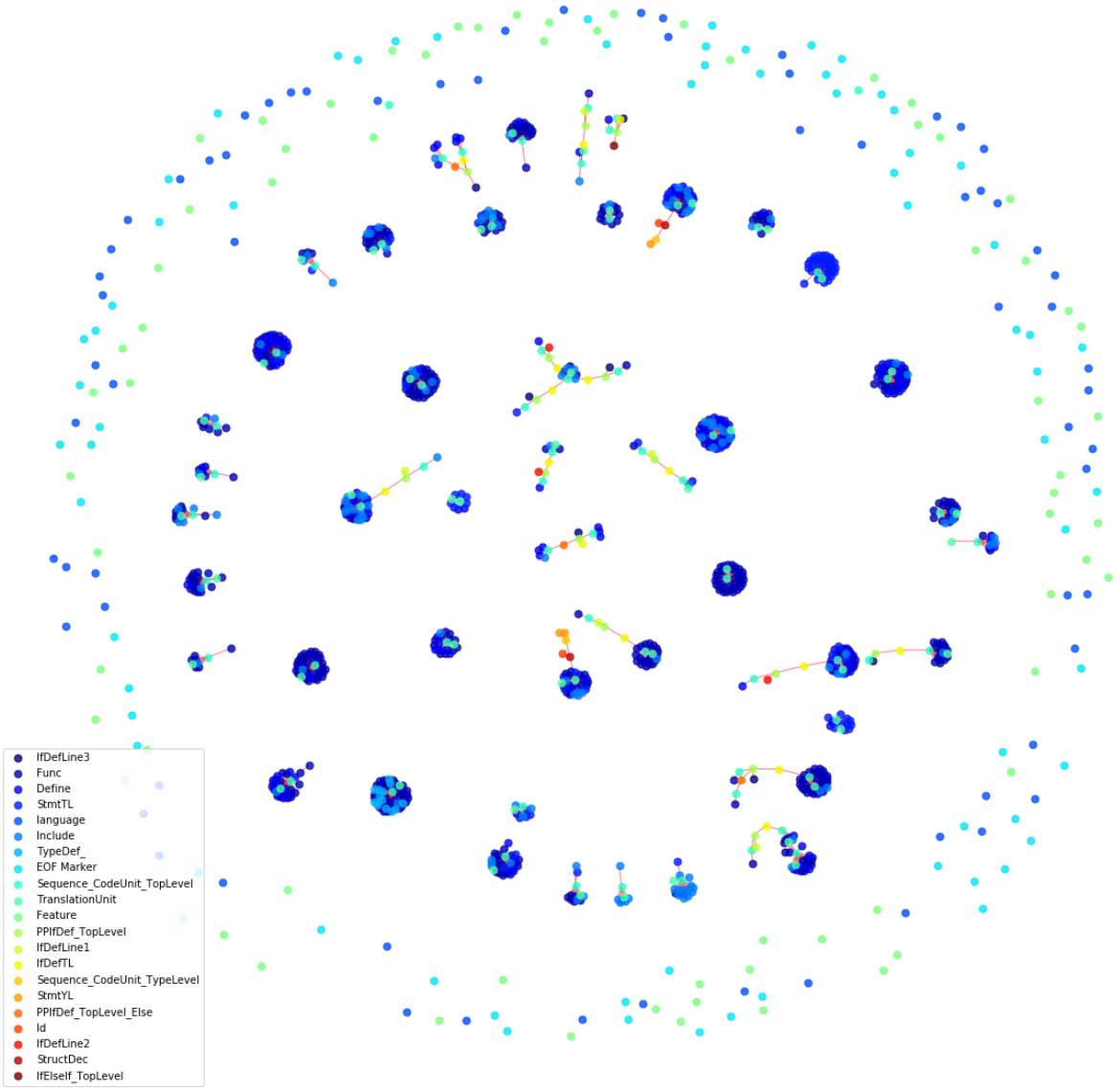

Figure 61: The FST generated for FreeRTOS

The adjacency matrix obtained from the FST is illustrated as a heatmap in Figure 62. 0 represents that the two nodes are not related, and 1 represents that they have a direct

edge.

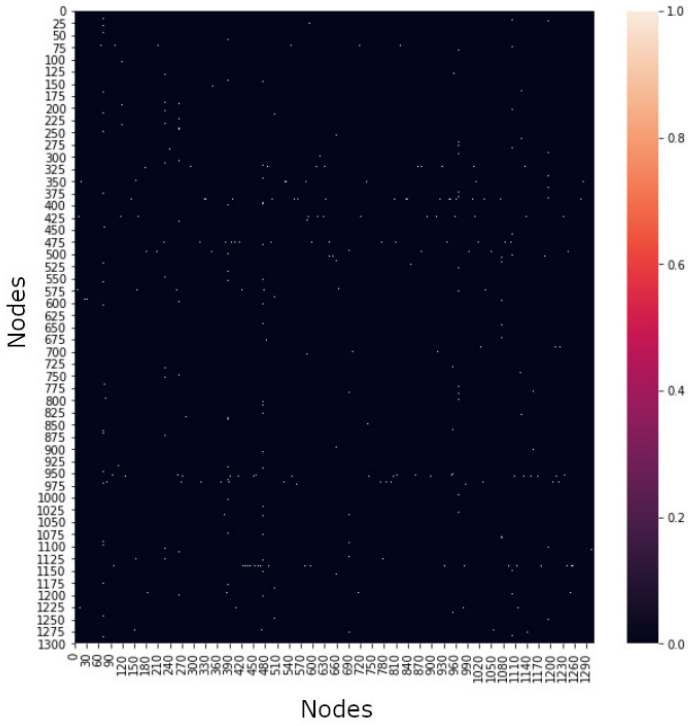

Figure 62: Adjacency Matrix for the FreeRTOS FST

The adjacency matrix is passed through the singular value decomposition using the *svd* method in Python's Numpy library. A plot of the singular values for each of the vectors in the left singular matrix are illustrated in Figure 63.

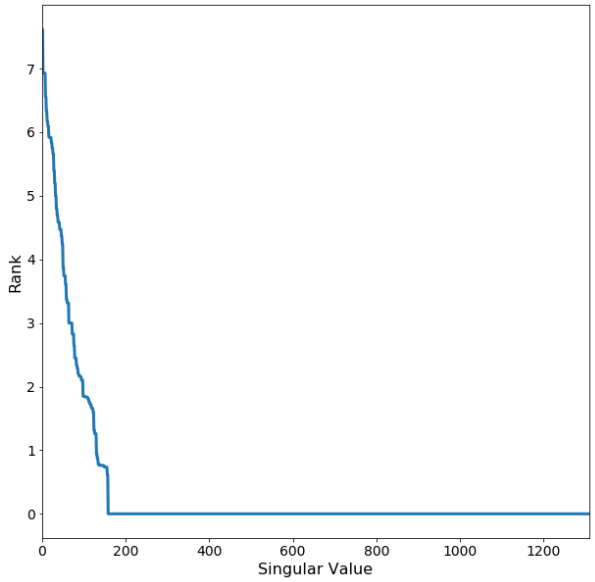

Figure 63: Singular Values from the SVD of the adjacency matrix

Upon inspection, it is evident that the value of k (numerical rank) of the left singular value matrix is around 180 since beyond this value, the contribution of the remaining singular values is close to zero. Hence the truncated SVD can be generated with the numerical rank of 180.

The next activity is to perform spectral clustering with this left singular matrix. This is fairly straightforward using the *SpectralClustering* method from the *sklearn* machine learning library of Python. The result of spectral clustering for a cluster count of 30, is illustrated in Figure 64.

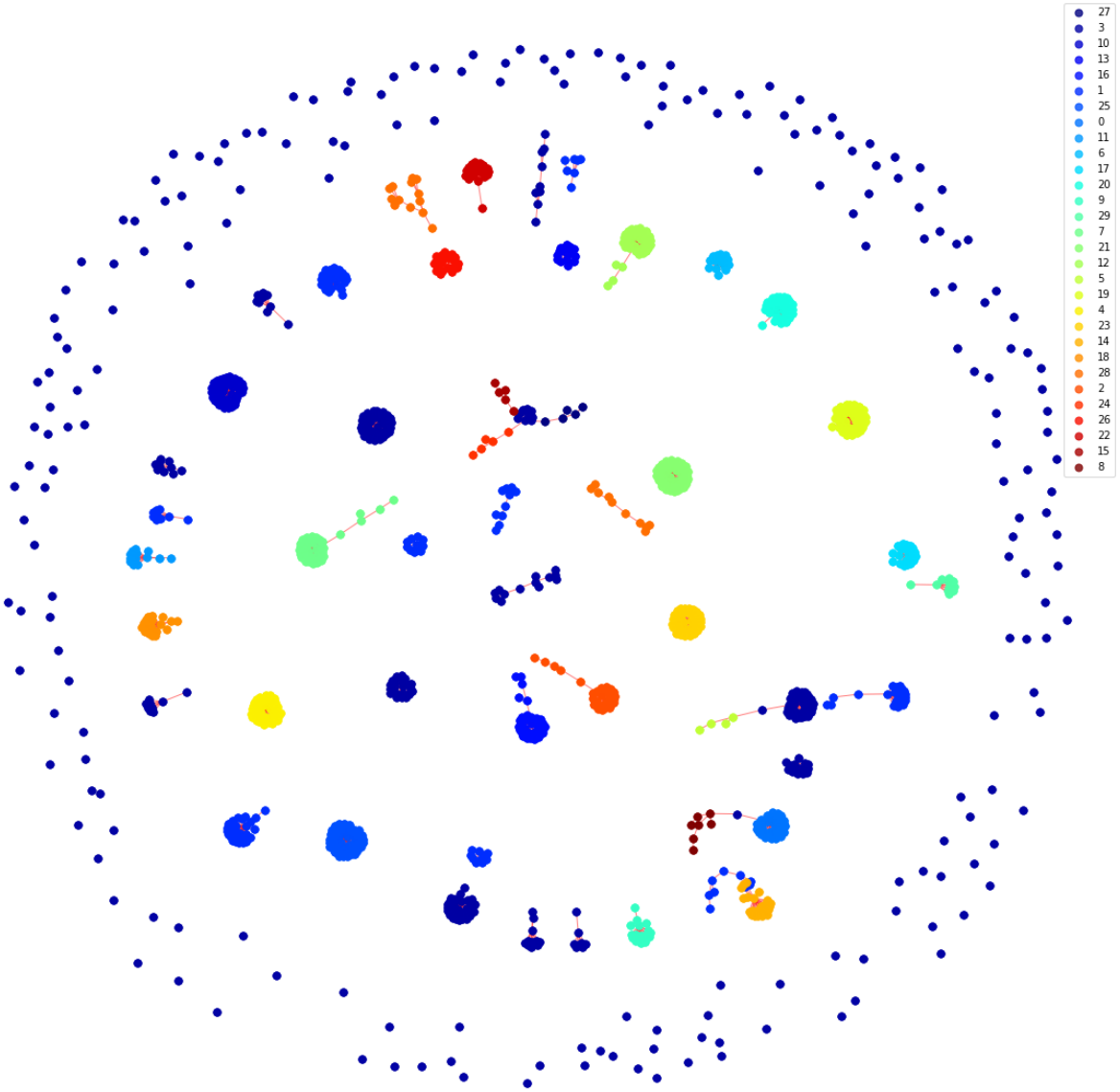

Figure 64: Clusters generated from the Spectral Clustering of FST

The treemap in Figure 65 illustrates the number of FST nodes (function calls, variants

etc.) per feature cluster identified through this method. The prefix C indicates *Cluster* and the value count is indicated below it.

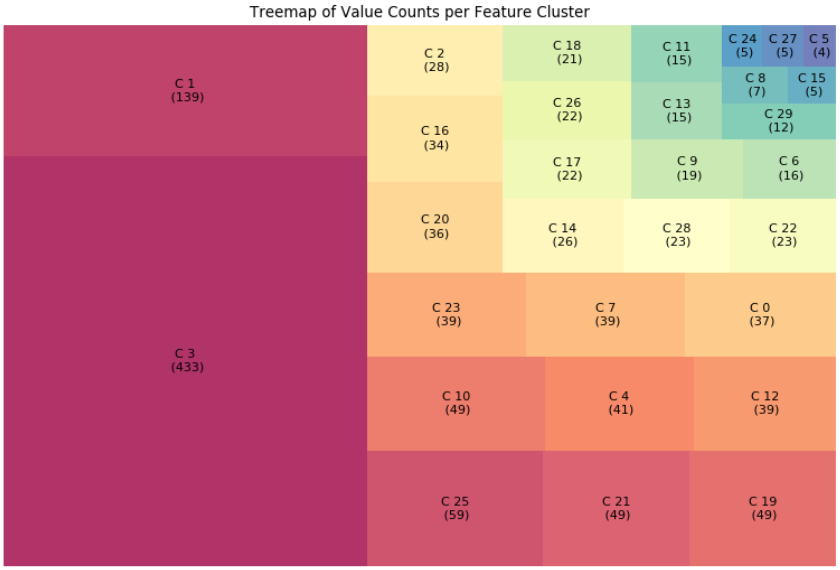

Figure 65: Treemap of Clusters and Node Value counts

An example of the nodes identified as belonging to a specific cluster is shown in Figure 66.

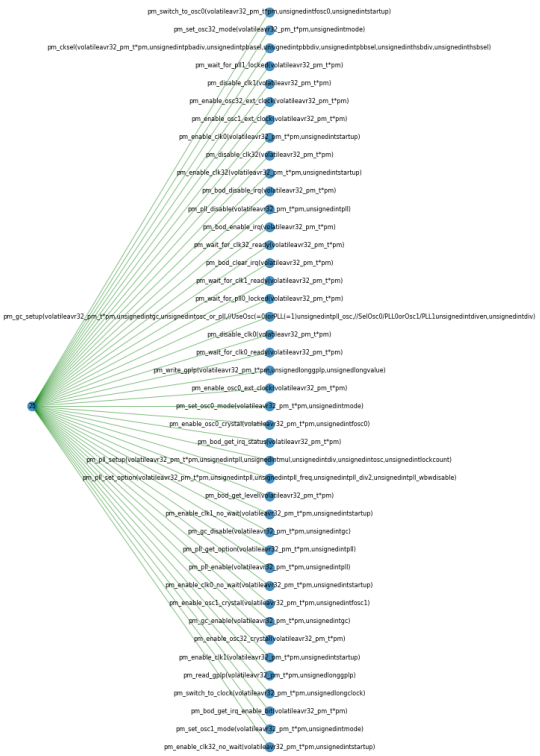

Figure 66: Nodes belonging to Feature Cluster 25

The feature clustering technique is one of the research contributions for this thesis. It has high potential in applications where features need to be automatically derived from the solution space (variability code realization). It is possible to generate a word cloud for the nodes in these clusters after pre-processing the text content in these nodes, to generate a representative name for the clusters.

Furthermore, several analysis can be conducted based on this cluster, one instance, the distribution of feature clusters across the different variability source files in a product line member, as illustrated in Figure 67.

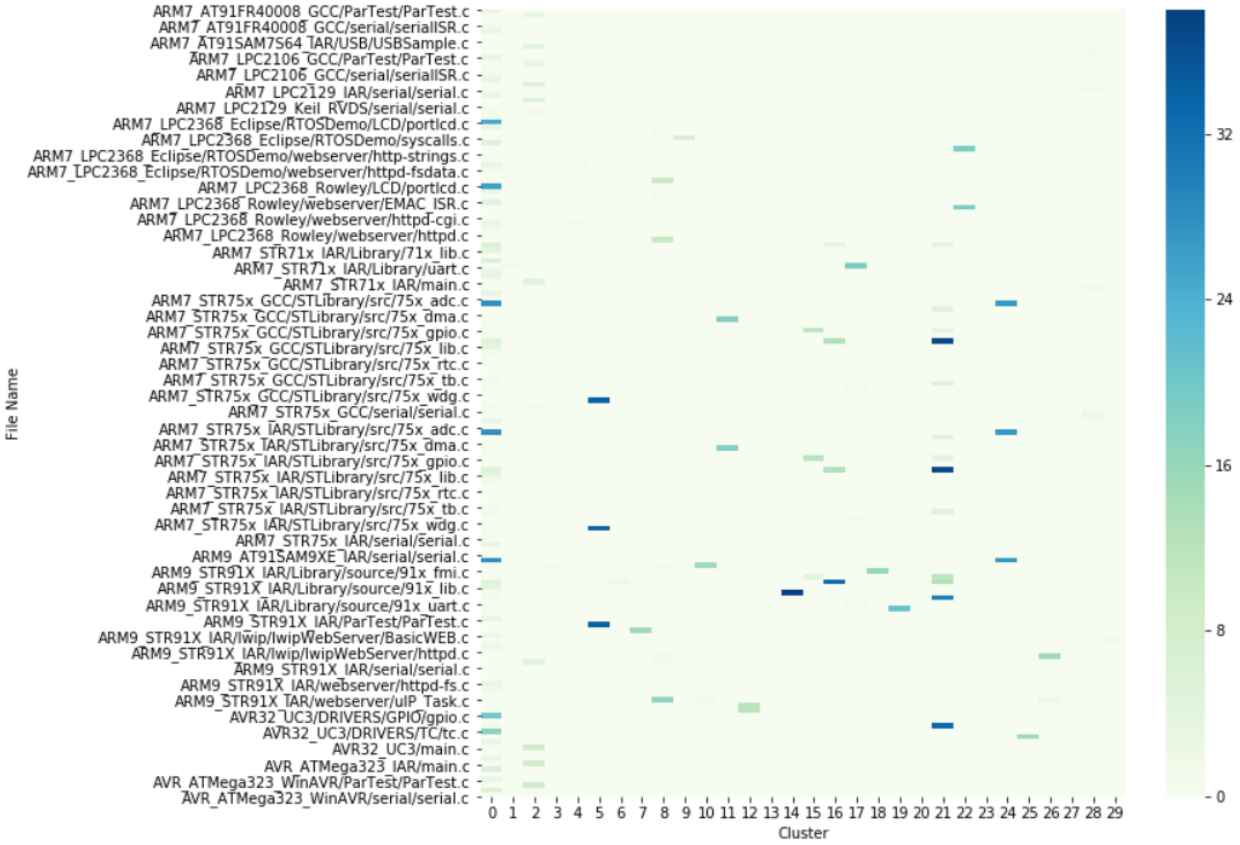

Figure 67: Cluster distribution across files in FreeRTOS

5 Evaluation

In Chapter 1, the research problems P1 through P4, regarding variability code analysis and product configuration recovery have been identified and research questions Q1 through Q3 have been defined to address these problems. Furthermore, specific goals were defined and activities to achieve these goals were determined. This chapter will present the evaluation of the proposed approach and the implemented solution in a real-life industrial setting. Specifically open source libraries have been used to validate the feasibility of the research approach. For this study, the FreeRTOS library has been chosen for analysis of variability code elements and their inter-dependencies.

5.1 Case Study of the FreeRTOS product line variability

FreeRTOS is a market-leading real-time operating system (RTOS) for micro-controllers and small microprocessors, distributed under the MIT open source license. It includes a kernel and a huge set of libraries, which keep growing over the years. As of January 2020, around 91 versions of the OS have been released, with increasing support for various industrial sectors and platforms. This library was chosen for the study due to its large number of platform support, which results in several product line members with intricate variabilities and switch-points. The FreeRTOS kernel can thus provide a setting equivalent to industrial projects. All the analysis, processing and visualization in this research is performed using the upgraded VITAL 2.0 tool-chain.

5.1.1 Data Preparation

Since FreeRTOS is under MIT open source license, its source code was available for download. FreeRTOS source is hosted in *SourceForge* and one version per year was selected for the past 15 years, i.e., from 2004 through 2019, the latest version being v10.2.1. Since there was no version released in 2016, a late version of 2015 was chosen for 2016. After removing the irrelevant files (for instance, demo folder, license files and other HTML help files), the directories for each release were numbered with the corresponding release year for ease of identification while processing the files in these versions. Figure 68 shows the directory structure of FreeRTOS. Note that the platform-specific files are stored under *portable* folder.

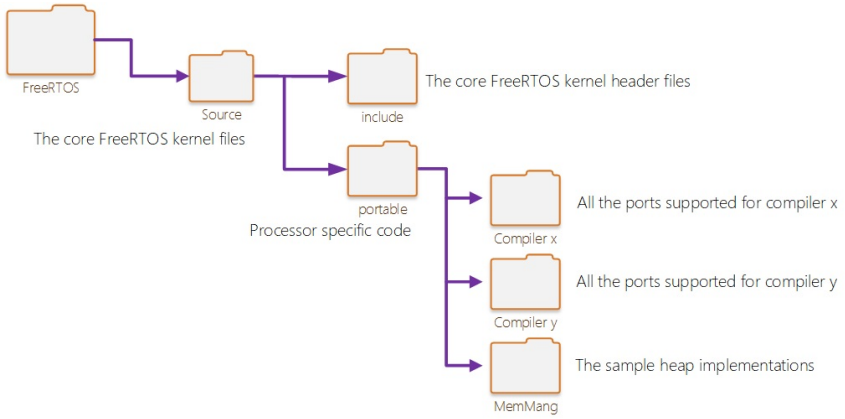

Figure 68: Directory structure of the FreeRTOS library

5.1.2 First Impressions

The first goal is to have an idea of the evolution of the FreeRTOS library itself, some analyses were performed for the selected versions of the FreeRTOS library using the VITAL tool-chain. This is done to get a fairly good understanding of the growing complexity of the FreeRTOS library. Figure 69 and Figure 70 give the number of source files and number of lines of code in each version versus the year.

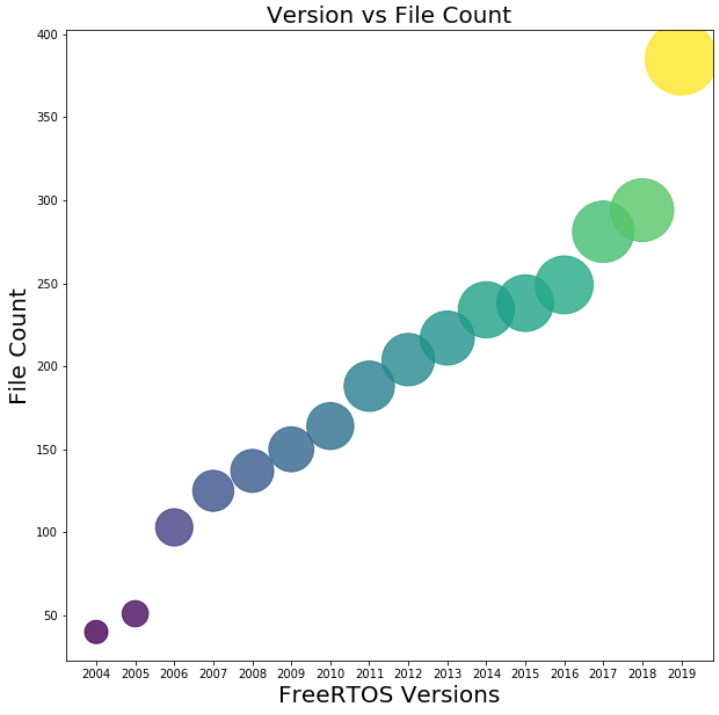

Figure 69: File Count for FreeRTOS from 2004-2019

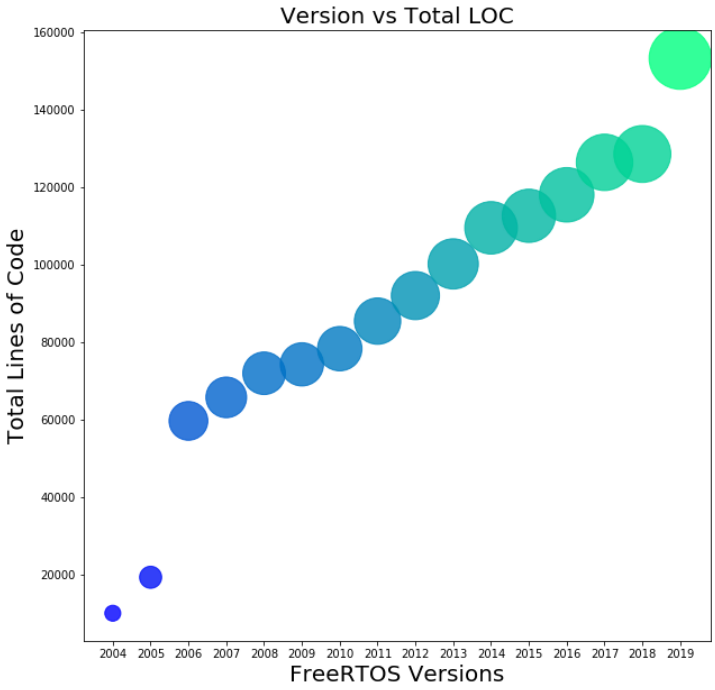

Figure 70: Total LOC for FreeRTOS from 2004-2019

Even with a cursory glance, it can be observed that the file count, as well as the number of lines of code, has increased almost linearly across the versions, over the years. The hypothesis is that a similar trend will be followed for the number of variabilities as well. To validate this, analysis was performed on each of the versions using VITAL 2.0.

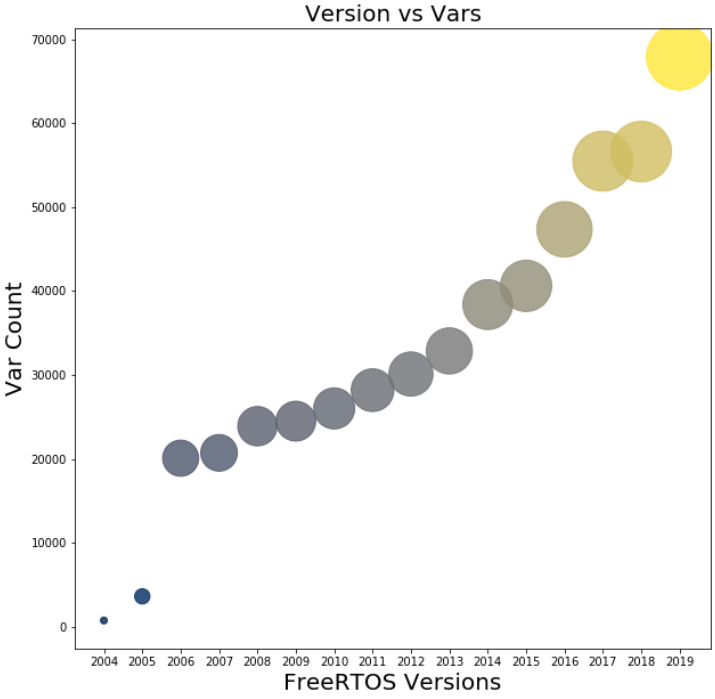

Figure 71: Total number of Vars for FreeRTOS from 2004-2019

The bubble plot in Figure 71 illustrates the growing number of variabilities with each version of FreeRTOS. It can be seen that this curve is more or less exponential.

Now that the hypothesis is confirmed, the latest version of FreeRTOS was used for all the remaining analyses.

5.1.3 Analysis & Results

Here, the modules developed for VITAL 2 has been invoked and results of various analysis have been presented. Firstly, preprocessor lexer objects were created for each of the files in the FreeRTOS library and stored for performing future operations. Each of the preprocessor lexers was further computed upon by the tokenizer, which created tokens for each file. These tokens were filtered to extract the variabilities. Variabilities are considered to be those macro definitions that are not include-guards. An include-guard filter filtered these from the extracted #defines. Once this is performed, various analysis was performed on the resultant vars.

Distribution of Vars across files

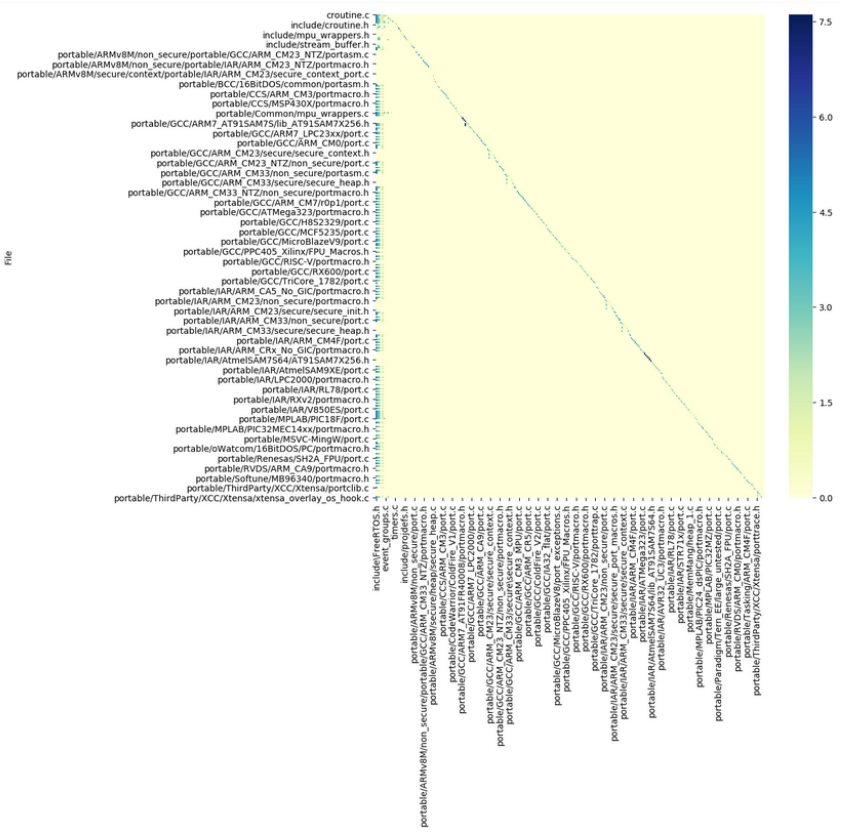

Figure 72: Distribution of Vars across Files

One such analysis was to find out the distribution of different Vars across different files. It was found that each file contains Vars that was defined in multiple included files and that the specific file contains only very few macro definitions on its own. The matrix of Vars across files was plotted as a heat map using the VITAL tool and the result is displayed in Figure 72. One important observation here is that the FreeRTOS.h file is being included and its macro defined in almost all the files. The linear curve in the graph signifies the macros that are present from that specific file alone.

Next, analysis was performed on the Vars identified in the previous step and the occurrence of these Vars across files were computed. From Figure 73, it can be inferred that the very high occurrence count of vars, more than 137, was present only in one file. Most of the files had less than variabilities. Around 850 files had a variability count of around 10.

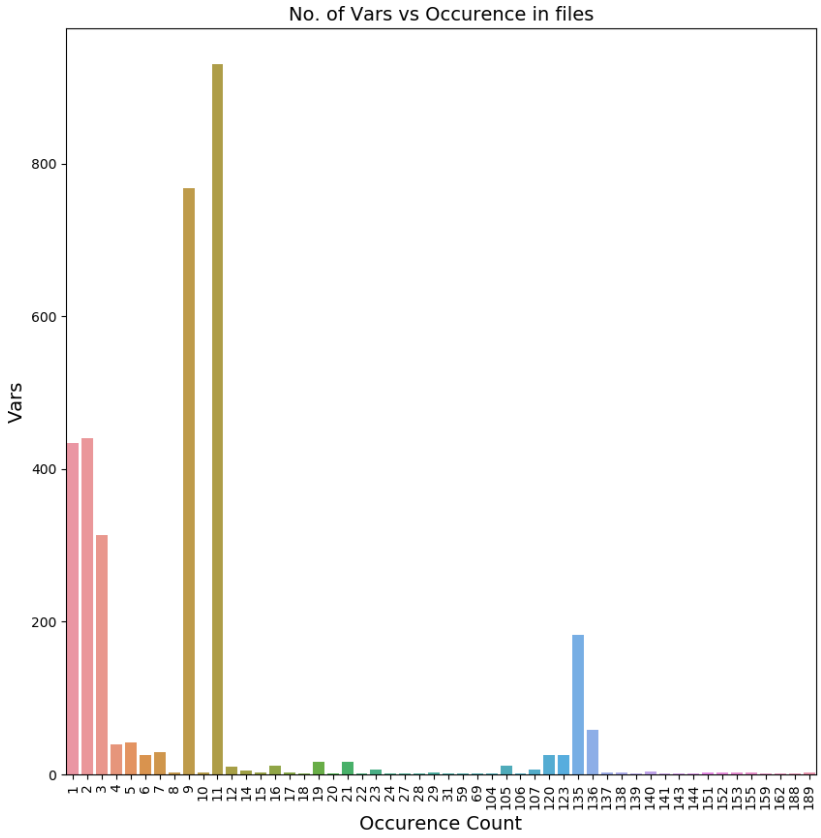

Figure 73: Vars vs its occurrence count across files

Extraction of Variation Points

As mentioned in the previous sections, pandas Dataframe is used to store the results of parsing, which is flexible in developing plots, storage and data analysis. Figure 74 is a snapshot of Dataframe for the different variation points that were obtained from the FreeRTOS source files:

	VariationPoint	Depth	CondState	PreprocessorDirective	FileID	FileIDPath	LineNum	FileName
0	INC_FREERTOS_H	0	True	ifndef	FreeRTOS.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	28	croutine.c
1	__cplusplus	0	False	ifdef	FreeRTOS.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	51	croutine.c
2	PROJDEFS_H	0	True	ifndef	projdefs.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	28	croutine.c
3	PORTABLE_H	0	True	ifndef	portable.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	32	croutine.c
4	INC_TASK_H	0	True	ifndef	task.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	29	croutine.c
5	CO_ROUTINE_H	0	True	ifndef	croutine.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	28	croutine.c
6	(configUSE_CO_ROUTINES != 0)	0	False	if	croutine.c	cs_freertos/2019_FreeRTOSv10.2.1/Source/crouti...	33	croutine.c
7	__cplusplus	1	False	ifdef	FreeRTOS.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	51	croutine.c
9	PROJDEFS_H	1	True	ifndef	projdefs.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	28	croutine.c
10	pdMS_TO_TICKS	1	True	ifndef	projdefs.h	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	40	croutine.c

Figure 74: Variation Points

Each variation point, along with its depth in the source file, location of the variation point in file (i.e., where the VP is referenced), file in which Vars are defined etc. are obtained. Also, the name of the #if directive which referenced the specific vars in a variation point and the conditional compilation state of that specific variation point are also extracted.

Extraction of Variation Point Groups

The variation point groups were obtained next. These are the variation points which contain Vars implementing the same inclusion logic. Essentially, these are the unique Variation Points obtained from the list of all the Variability references.

	VariationPointGroup	Count
0	__cplusplus	2553
1	(configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES ==...	1395
2	(configUSE_TRACE_FACILITY == 1)	1214
3	(portUSING_MPU_WRAPPERS == 1)	1075
4	INC_FREERTOS_H	700
5	portCONFIGURE_TIMER_FOR_RUN_TIME_STATS	567
6	((configSUPPORT_STATIC_ALLOCATION == 1) && ...	562
7	(portHAS_STACK_OVERFLOW_CHECKING == 1)	544
8	configUSE_APPLICATION_TASK_TAG	540
9	(configUSE_16_BIT_TICKS == 1)	466

Figure 75: Variation Point Groups

The results of the first 10 variation point groups and their occurrence count across all the

files are illustrated in Figure 75.

Distribution of VPGs across Files

The next analysis that was performed was on the obtained variation point groups. Similar to the distribution of Vars visualized previously, the extend of VPGs across different files were analyzed. The x-axis shows the top 50 variation point groups sorted by their occurrence in various files and the y-axis shows the different files in which they occur. It can be inferred that the VPG `configSUPPORT_DYNAMIC_ALLOCATION == 1` was used maximum, in a file, i.e., 32 times in mcu_wrappers.c file in the FreeRTOS file. The results are shown in Figure 76.

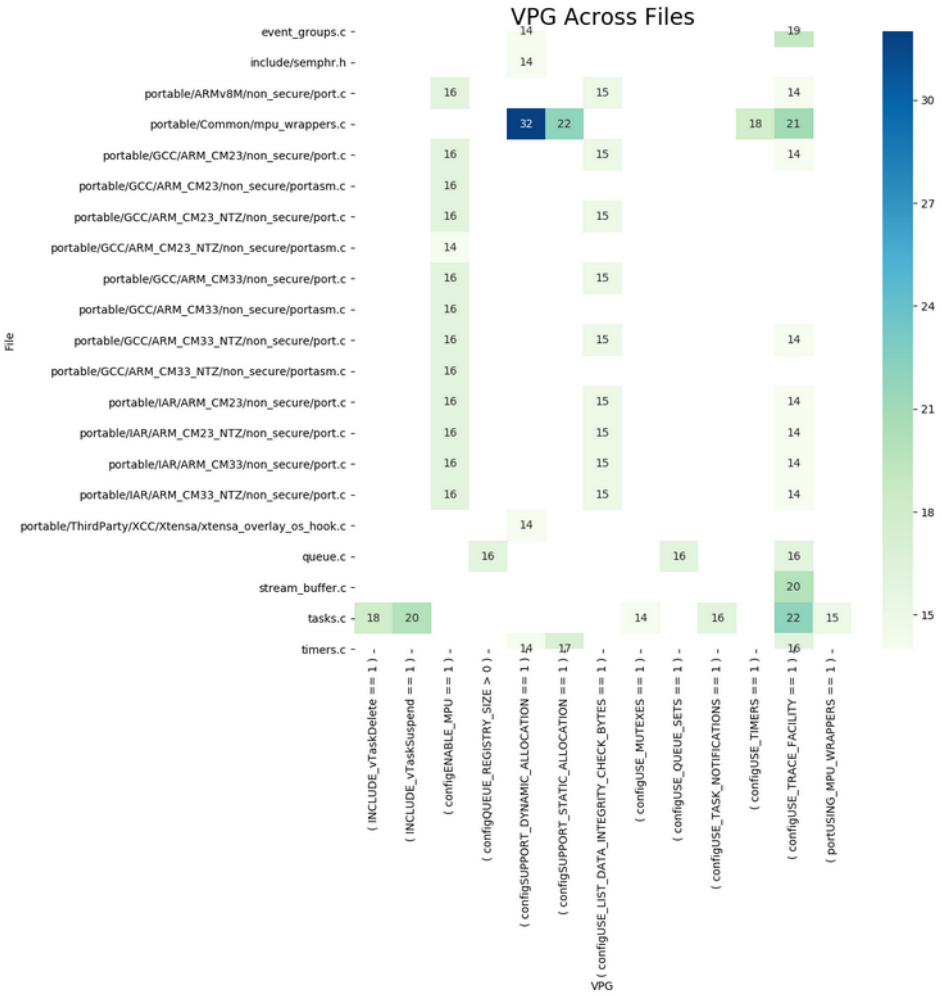

Figure 76: VPG Across Files

Nesting of the VPGs

The degree of nesting of each Variation Point Groups (VPG) is a key metric in understanding the complexity of the code. The more nested the VPGs are, the more complex and difficult it is, to comprehend the realization code. To analyse this, the nesting degree among the top 50 most occurring VPGs have been visualized using VITAL 2.0 for the different source files in FreeRTOS. The results are illustrated in Figure 77.

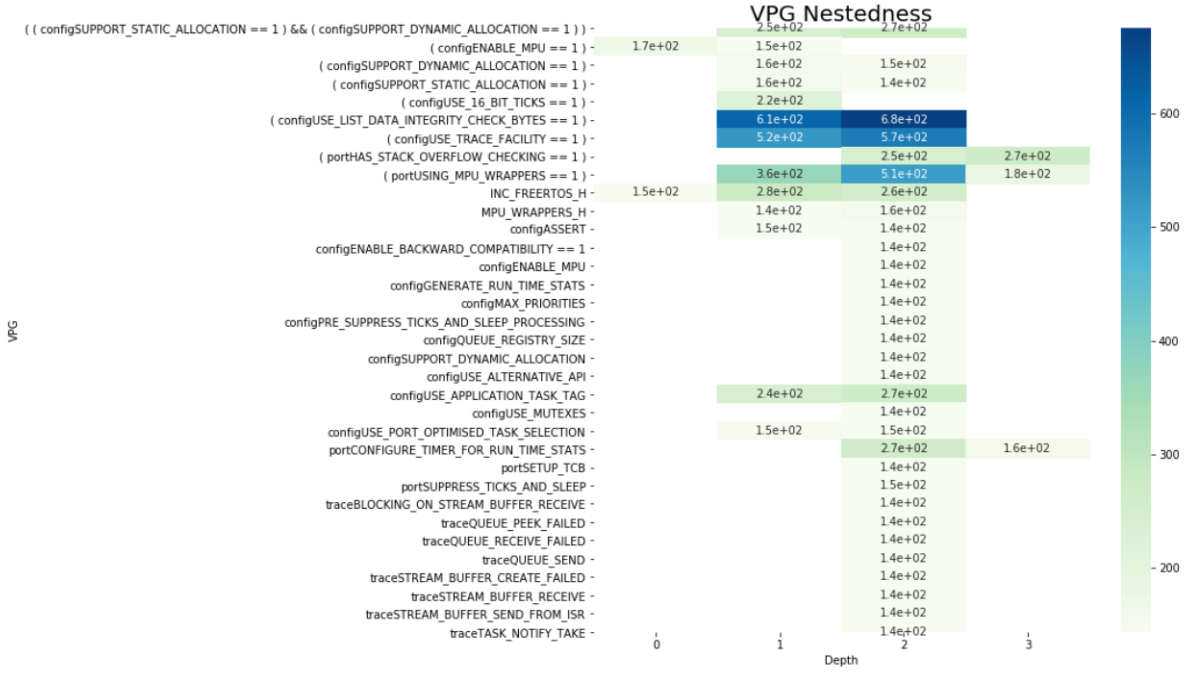

Figure 77: VPG Nesting Across Files

It can be inferred from Figure 77 that most of the Variation Point Groups are nested at 1-2 degrees, and not many VPGs have a higher degree of nesting of 3 or no VPG nesting at all.

Variability Tangling

Tangling between variabilities was analyzed next. To understand this, the tangling of variabilities in a variation point group was extracted. This is visualized in Figure 78.

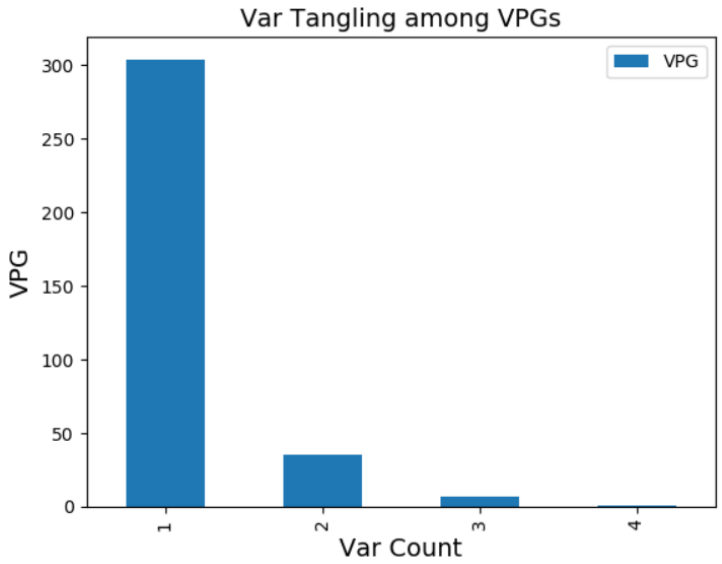

Figure 78: Var tangling across VPGs

It can be seen that the maximum degree of tangling, i.e. the maximum number of Vars used in a VPG is 4, and has happened only for very few VPGs. Most of the VPGs in the library contains only a single variation point and the number of VPGs having two variabilities are less than 50.

Variability Range Extraction

Using the VITAL 2.0 tool, the range or the values taken by different variabilities have been extracted. Figure 79 gives the values of some of the selected rows in the pandas Dataframe containing the value counts for variabilities.

	Identifier	Params
31	configEXPECTED_IDLE_TIME_BEFORE_SLEEP	2
39	configMAX_TASK_NAME_LEN	16
51	configSUPPORT_DYNAMIC_ALLOCATION	1
67	configUSE_TASK_NOTIFICATIONS	1
27	configENABLE_BACKWARD_COMPATIBILITY	1
18	INCLUDE_xTaskResumeFromISR	1
28	configENABLE_FPU	1
30	configENABLE_TRUSTZONE	1
66	configUSE_TASK_FPU_SUPPORT	1
33	configIDLE_SHOULD_YIELD	1

Figure 79: Variabilities and Values

There were two types of macros identified from the tool; Object type macros and function type macros. Object type macros defined data objects with specific integer or string values, whereas function type macros define function calls that get substituted during preprocessing. Furthermore, most of the Vars had a value of 0 or 1.

Extraction of Variability Inter-dependencies

A conditional compilation graph was extracted in this phase. Here, the custom graph visitor implemented in VITAL 2.0 walks through all the nodes in the conditional compilation graph and extracts the variability dependencies, like hierarchical dependencies between the Variation Points, Conditional Definitions. This is illustrated with many network graphs plotted using VITAL 2.0.

Hierarchical Dependencies between VPGs

The pandas DataFrame in Figure 80 captures the interdependencies between variation point groups across all the files.

	VariationPoint	Depth	CondState	PreprocessorDirective	LineNum	ChildVariationPoint	ChildLineNum	ChildDirective	ChildDepth
0	INC_FREERTOS_H	0	True	ifndef	28	__cplusplus	51	ifdef	1
1	INC_FREERTOS_H	0	True	ifndef	28	PROJDEFS_H	28	ifndef	1
2	INC_FREERTOS_H	0	True	ifndef	28	PORTABLE_H	32	ifndef	1
3	INC_FREERTOS_H	0	True	ifndef	28	INC_TASK_H	29	ifndef	1
4	INC_FREERTOS_H	0	True	ifndef	28	TIMERS_H	29	ifndef	1
5	INC_FREERTOS_H	0	True	ifndef	28	STACK_MACROS_H	28	ifndef	1
6	INC_FREERTOS_H	0	True	ifndef	28	(configUSE_STATS_FORMATTING_FUNCTIONS == 1)	51	if	1
7	INC_FREERTOS_H	0	True	ifndef	28	(configUSE_PREEMPTION == 0)	59	if	1
8	INC_FREERTOS_H	0	True	ifndef	28	None	63	else	1
9	INC_FREERTOS_H	0	True	ifndef	28	((configCHECK_FOR_STACK_OVERFLOW > 1) || (...	86	if	1

Figure 80: Hierarchical Dependencies across files

A host of information is presented in these DataFrames. Each parent Variation Point along with its child Variation Point are represented. The line number, preprocessor directive used in the Variation Point and the depth of the Variation Point for both parent and child VPs are extracted. Also, the conditional state of the parent VP is obtained. This DataFrame can be used for several further analyses which give several useful information about the variability code and its dependencies.

The network graph of this dependencies run across all the source files is shown in Figure 81. This is an interactive plot, and upon hovering over each node, it gives information on the variation point and the preprocessor directive belonging to that VP.

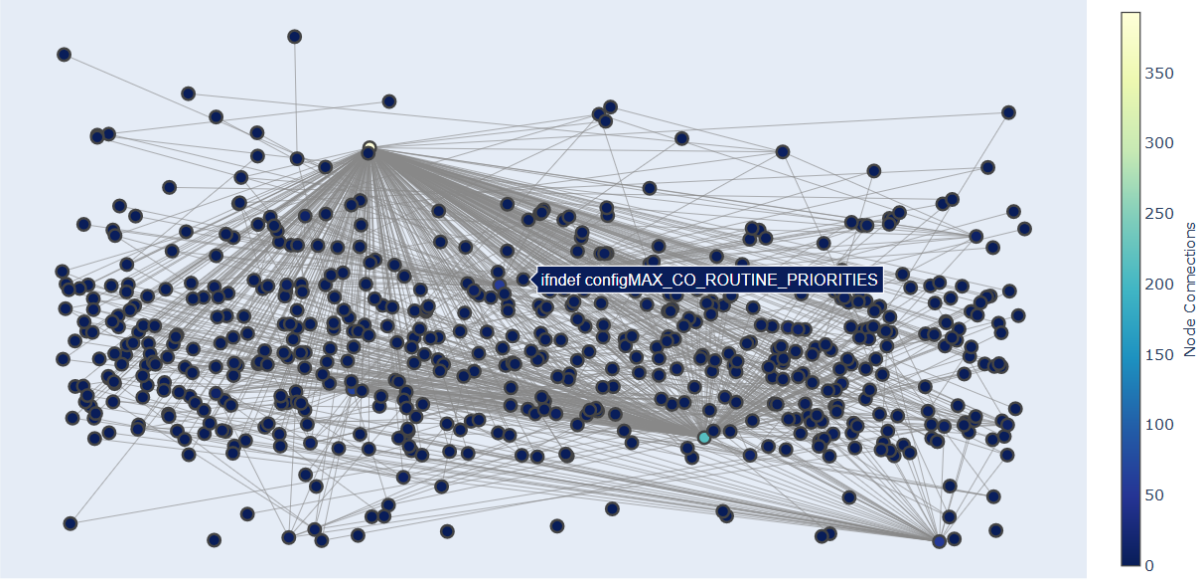

Figure 81: Hierarchical Dependencies across files - Plot

The darker the color, the higher the number of nodes connected from that specific VPG. It is an illustration for all the files in FreeRTOS put together. This view was generated for individual files as well. They are presented in Figure 82 and Figure 83:

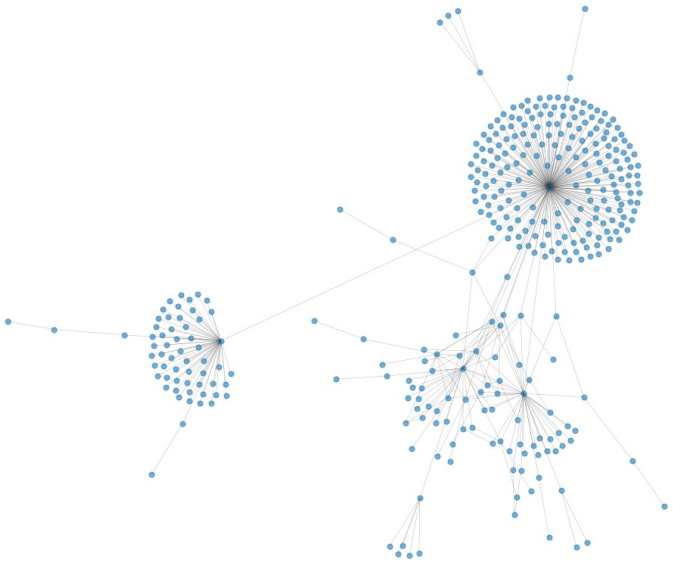

Figure 82: Hierarchical Dependencies for tasks.c file in FreeRTOS

The graph nodes in these Figures are colored based on the depth of each node.

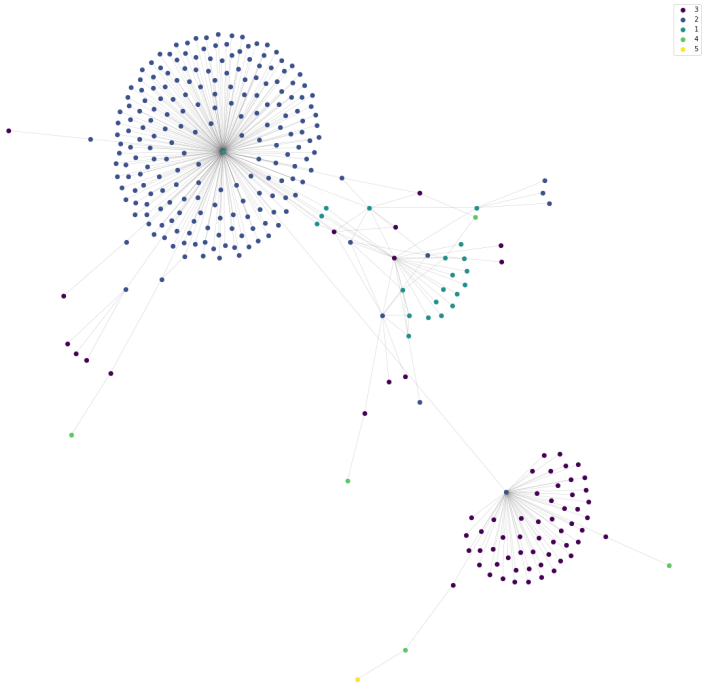

Figure 83: Hierarchical Dependencies for queue.c file in FreeRTOS

The information presented here is available in the DataFrame format as well, for each of the files.

Information about Macros

VITAL 2.0 tool also provides several other functionalities. Here, the DataFrame output of some of them is elaborated. Figure 84 gives a snapshot of all the macros available across the files. It gives information as to whether that particular macro is defined, referenced, line number in file, reference count, undefined line number if undefined later, macro parameters, and the type of macro. This DataFrame can be further filtered to create specific metrics for the user.

	Macro	IsDef	IsRef	Line	RefCount	Parameters	IsObjectTypeMacro	UndefFileId	UndefLine
1	MPU_WRAPPERS_INCLUDED_FROM_API_FILE	False	False	35	0	None	True	sample/freertos /tasks.c	47.0
2	INCLUDE_eTaskGetState	True	True	164	3	None	True	None	NaN
90	portPOINTER_SIZE_TYPE	True	True	296	5	None	True	None	NaN
91	portPRE_TASK_DELETE_HOOK	True	True	278	1	[pvTaskToDelete, pxYieldPending]	False	None	NaN
92	portPRIVILEGE_BIT	True	True	726	3	None	True	None	NaN
93	portRESET_READY_PRIORITY	True	True	156	2	[uxPriority, uxTopReadyPriority]	False	None	NaN
94	portSETUP_TCB	True	True	282	3	[pxTCB]	False	None	NaN
95	portSET_INTERRUPT_MASK_FROM_ISR	True	True	266	7	[]	False	None	NaN
96	portSUPPRESS_TICKS_AND_SLEEP	True	True	734	3	[xExpectedIdleTime]	False	None	NaN
97	portTASK_USES_FLOATING_POINT	True	True	766	1	[]	False	None	NaN
98	portTICK_RATE_MS	True	False	935	0	None	True	None	NaN
99	portTICK_TYPE_CLEAR_INTERRUPT_MASK_FROM_ISR	True	True	878	2	[x]	False	None	NaN

Figure 84: All the generated macros with different statistics

The Dataframe shown in Figure 85 is a snapshot of some of the macros that are referenced (i.e., accessed using #if directives) in the file.

	Referenced Macros
0	INC_FREERTOS_H
1	configUSE_NEWLIB_REENTRANT
2	configUSE_CO_ROUTINES
3	INCLUDE_vTaskPrioritySet
4	INCLUDE_uxTaskPriorityGet
5	INCLUDE_vTaskDelete
6	INCLUDE_vTaskSuspend
7	INCLUDE_vTaskDelayUntil
8	INCLUDE_vTaskDelay
9	INCLUDE_xTaskGetIdleTaskHandle

Figure 85: All referenced macros

Another example is the DataFrame of macros that are currently in scope, i.e., those which are not undefined. This is illustrated in Figure 86.

	FileID	IsDef	IsRef	Line	Macro	RefCount
0	sample/freertos\FreeRTOS.h	True	True	164	INCLUDE_eTaskGetState	3
1	sample/freertos\FreeRTOS.h	True	True	156	INCLUDE_uxTaskGetStackHighWaterMark	9
2	sample/freertos\FreeRTOS.h	True	True	160	INCLUDE_uxTaskGetStackHighWaterMark2	9
3	sample/freertos\FreeRTOS.h	True	True	116	INCLUDE_uxTaskPriorityGet	5
4	sample/freertos\FreeRTOS.h	True	True	132	INCLUDE_vTaskDelay	3
5	sample/freertos\FreeRTOS.h	True	True	128	INCLUDE_vTaskDelayUntil	3
6	sample/freertos\FreeRTOS.h	True	True	120	INCLUDE_vTaskDelete	19
7	sample/freertos\FreeRTOS.h	True	True	112	INCLUDE_vTaskPrioritySet	3
8	sample/freertos\FreeRTOS.h	True	True	124	INCLUDE_vTaskSuspend	29
9	sample/freertos\FreeRTOS.h	True	True	144	INCLUDE_xQueueGetMutexHolder	1

Figure 86: Macros in scope - not undefined

Figure 87 gives a Dataframe of the macros that are not in scope, i.e., those which are undefined.

	FileID	IsDef	IsRef	Line	Macro	RefCount
0	sample/freertos/tasks.c	False	False	35	MPU_WRAPPERS_INCLUDED_FROM_API_FILE	0
1	sample/freertos/tasks.c	False	False	35	MPU_WRAPPERS_INCLUDED_FROM_API_FILE	0

Figure 87: Macros Not in scope - undefined

Another analysis could be done on the extracted macros to find those which are referenced, but not defined anywhere. This would help in understanding the issues in a product variant where the variability is referenced, but not defined in any of the files, leading to errors. Figure 88 gives a snapshot of some of the results.

	Macro	FileID	Line
30	ROWLEY_LPC23xx	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	92
31	IAR_MSP430	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	96
32	GCC_MSP430	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	100
33	ROWLEY_MSP430	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	104
34	ARM7_LPC21xx_KEIL_RVDS	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	108
35	SAM7_GCC	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	112
36	SAM7_IAR	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	116
37	SAM9XE_IAR	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	120
38	LPC2000_IAR	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	124
39	STR71X_IAR	cs_freertos/2019_FreeRTOSv10.2.1/Source/includ...	128

Figure 88: Macros that are referenced, but not defined

Another useful information is the macros that has static dependencies with each other.

This is illustrated in Figure 89.

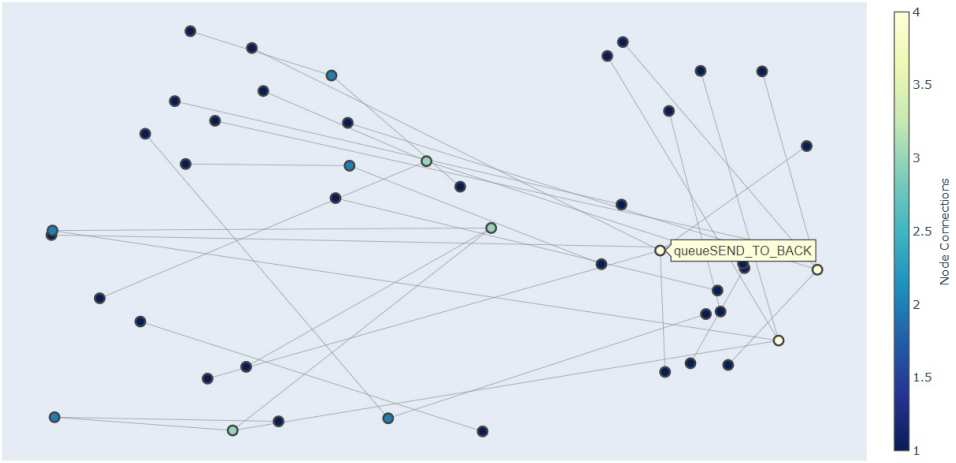

Figure 89: Macros with static dependencies with each other

5.1.4 Summary

The results presented above explores some of the possibilities that the improved VITAL 2.0 tool can unveil. Several similar analyses and visualization can be done with the variability code information extracted with the tool that is tailored for the needs of the user. All the relevant information about the variability code are stored as pandas DataFrame, which gives numerous possibilities for further extension to other tools and expansion.

6 Summary and Conclusions

The increasing complexity in the software product line, owing to a large number of features and their intricate dependencies, has become a challenge in the industry, which poses a threat to the benefits of software product line engineering. One of the concerning consequences of this complexity is variability code erosion, where the variability code realization cannot be traced back to its corresponding variability model. In most cases, there will be no explicit product configuration information for the product line or it will not be updated as the product line evolves over space and time. This thesis provides solution ideas and their corresponding implementations for extracting variability information from the code realizations to generate product configuration information. Specifically, it re-designs and develops the VITAL 2.0 tool-chain, which is composed of a host of independently invokable modules, which offers the user, ability to perform different variability code related analyses. The solution is implemented in such a way that it has a common interface format for external tools to plug in to, aiding in further analysis and visualization.

Another contribution of this thesis is a novel mechanism to generate clusters from the variability code realization through spectral clustering of the singular values of the AST, resulting in high-level feature clusters of the product line. This result has potential in automatic extraction of features from the product line, and this, together with the variability code analysis implemented in this research can aid the analyst or developer in generating product configuration from variability code.

In addition to the above two contributions, extensive study and experiments have been conducted on the state of the art techniques for variability code extraction, and new methodologies and libraries have been added to the literature. This will be beneficial for future research along with the same domain and the knowledge extracted from these studies can be used to further extend the VITAL tool-chain to develop a more generic variability code parser.

The different metrics that are presented in this study related to variability code elements will be beneficial in identifying the complex areas of the variability code, and in further re-structuring to combat the maintainability issues. Also, it gives good insights on the structural aspect of variability code realization and the inter-dependencies between different features over various dimensions.
Finally, the implementation of the solution ideas has been validated with open source projects that stand equivalent to the real-life industrial projects, to measure the scalability and robustness of the implementation.

Some of the key findings in the studies are listed below:

- For parsing variability code realization using conditional compilation in C/C++ language, the best option is to use a standard C preprocessor parser written in Python, which provides access to its data structures so that one can control and manipulate the data to the desired form.
- For parsing variability code realization using conditional execution (eg: C, Java etc.), the best option is to use a library that provides an AST, with the dependencies of conditional statements, with the same flexibility as mentioned above.
- For parsing any other software artefact, a library that allows users to write the grammar for that specific artefact is desirable. From the studies, FeatureHouse and ANLTR are the two recommended choices. ANTLR parser grammar has complete flexibility on what can be parsed, given that the user generates a grammar file for parsing the specific software artefact. On the other hand, FeatureHouse currently supports only a limited number of parsers, and studies are being conducted in the field to expand it to form a more generic parser. While the former is a standard parser generator for years in the industry, the latter is more focused on software product lines.
- To perform clustering of various software artefacts, Spectral Clustering is a very powerful technique. This technique is now widely used in various machine learning problems, as an unsupervised learning approach. In this research, clustering was performed on the variabilities and function calls in the software product line and has produced very promising results.

6.1 Open Issues and Future Work

This thesis provides many contributions to improve variability code analysis for product configuration extraction from software product lines. However, there is also additional scope for this study, which can be taken up as an extension or future scope for this work. They are listed below:

- It is possible to extend the existing implementation of hierarchical dependency extraction of variabilities and variation points to reconstruct a feature model. The VPs consisting of alternative branches can be mapped to alternate features and the variabilities enclosed inside conditional compilation blocks which have only #ifs can be considered as optional features. Similarly, the code fragments, features/Vars

that is not enclosed within a variation point can be considered a feature that is common to all product lines. In this way, working backwards, we can re-model the feature diagram for the variability code realization. Also, it is possible to convert this dependency graph into propositional formulae to automatically derive product configuration files.

- Different mechanisms using entropy-based calculations can be used to measure how well a cluster has been formed during the spectral clustering of features. This validation was not performed in this study and can be taken up as extended research. Furthermore, the cluster frequencies are comparable to term frequencies in the context of natural language processing and can be useful to find insightful metrics like the significance of a cluster in the product variant.

- In this study, different mechanisms have been explored for extracting variability code information, which yields equivalent outputs. The outputs generated from these mechanisms can be utilized to increase the confidence level of the predicted feature/cluster. This can be achieved through ensemble techniques like bagging, boosting and stacking.

- Another idea worth studying is the fact that the focus of this thesis is on reverse-engineering the variability code realizations, to get useful insights on variability and extract product configuration information. However, the recommendations and analysis functionalities will also greatly help in understanding the gaps in product line realization and improvement ideas for re-structuring the variability code realization, for *forward engineering*. Round trip engineering is an approach that considers both these aspects. Having a tool-chain that considers the software product line from an end-to-end perspective is the next big milestone this tool-chain and related research work can achieve.

References

ANTLR. (2020). Retrieved from https://www.antlr.org

Apel, S. [S.], Batory, D., Kästner, C., & Saake, G. (2013). Feature-oriented software product lines: Concepts and implementation. *Springer, Berlin*.

Apel, S. [S.], & Kästner, C. [C.]. (2009). An overview of feature-oriented software development. *Journal of Object Technology, Chair of Software Engineering, ETH Zurich*, *8*(5).

Apel, S. [Sven], Kästner, C., & Lengauer, C. (2013). Language-independent and automated software composition: The featurehouse experience. *IEEE Transactions on Software Engineering*, *39*, 63–79. doi:https://doi.org/10.1109/TSE.2011.120

Becker, M. (2017). *Software product line engineering*. Technical University of Kaiserslautern and Fraunhofer Institute of Experimental Software Engineering.

Benavides, D., Segura, S., & Cortés, A. R. (2010). Automated analysis of feature models 20 years later: A literature review information systems. *Elsevier*.

Berger, T., Rublack, R., Nair, D., Atlee, J., Becker, M., Czarnecki, K., & Wasowski, A. (2013). A survey of variability modeling in industrial practice. *7th Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS2013), Pisa, Italy*.

Botterweck, G., Lee, K., & Thiel, S. (2009). Automating product derivation in software product line engineering. *Software Engineering Conference 2009, Fachtagung des GI-Fachbereichs Softwaretechnik, Kaiserslautern*.

BUT4Reuse. (2020). Retrieved from https://but4reuse.github.io/

Clang. (2020). Retrieved from https://clang.llvm.org/

Clements, P., & Northrop, L. (2001). *Software product lines: Practices and patterns*. Addison-Wesleys.

Coppel, Y., & Candea, G. (2018). Deprogramming large software systems. *Proceedings of the 4th Workshop on Hot Topics in System Dependability (HotDep)*.

CPIP. (2020). Retrieved from https://clang.llvm.org/

FeatureHouse: Language-Independent, Automated Software Composition. (2020). Retrieved from https://www.infosun.fim.uni-passau.de/spl/apel/fh/

Feigenspan, J., Kästner, C., Frisch, M., Dachselt, R., & Apel, S. (2010). Visual support for understanding product lines. *IEEE Xplore*. doi:https://doi.org/10.1109/ICPC.2010.15

Ganesan, D., Muthig, D., & Yoshimura, K. (2006). Predicting return on-investment for product line generations. *Proceedings of the 10th International on Software Product Line Conference (SPLC '06), IEEE Computer Society, Washington, DC, USA*, 13–22.

Horn, P. (2001). Autonomic computing: Ibm's perspective on the state of information technology. *Technical Report. IBM Corporation*.

Kang, K., Cohen, S., Hess, J., Nowak, W., & Peterson, A. (1990). Feature–oriented domain analysis (foda) feasibility study (no. cmu/sei–90–tr–21). *Carnegie–Mellon Univiversity Pittsburgh, PA, Software Engineering Institute*.

Kästner, C. (2010). *Virtual separation of concerns: Toward preprocessors 2.0* (Doctoral dissertation).

Kästner, C. (2020). Cide: Virtual separation of concerns (preprocessor 2.0). Retrieved from https://ckaestne.github.io/CIDE/

Kenner, A., Kästner, C., Haase, S., & Leich, T. (2010). Typechef: Toward type checking #ifdef variability in c. *FOSD '10: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, 25–32. doi:https://doi.org/10.1145/1868688.1868693

Laguna, M. A., Marques, J. M., & guez-Cano, G. R. (2011). Feature diagram formalization based on directed hypergraphs. *Computer Science and Information Systems*, *8*, 611–633.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., & Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE '10. New York, NY, USA: ACM*, *1*, 105–114.

Linsbauer, L., Lopez-Herrejon, R. E., & Egyed, A. (1975). Variability extraction and modeling for product variants. *Czechoslovak Mathematical Journal*, *25*, 619–633.

Martinez, J., & Parsai, A. (2019). Round-trip engineering and variability management platform and process: D3.1 - identification of relevant state of the art.

Miroslav, F. (1973). Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, *23*, 298–305.

Miroslav, F. (1975). A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, *25*, 619–633.

Parr, T. (2007). *The definitive antlr4 reference*. The Pragmatic Programmers.

Paškevičius, P., Damaševičius, R., Karčiauskas, E., & Marcinkevičius, R. (2012). Automatic extraction of features and generation of feature models from java programs. *Information Technology and Control*, *41*(4).

PCPP. (2020). Retrieved from https://pypi.org/project/pcpp/

PCPP API Documentation. (2020). Retrieved from https://ned14.github.io/pcpp/

Pohl, K., Böckle, G., & van der Linden, F. (2005). Software product line engineering: Foundations, principles, and techniques. *Springer: Berlin, Heidelberg, New York*.

Poshyvanyk, D., & Marcus, A. (2007). Combining formal concept analysis with information retrieval for concept location in source code. *Proc. of the 15th IEEE Int. Conf. on Program Comprehension (ICPC '07), Washington, DC, USA*, 37–48.

pp-trace. (2020). Retrieved from https://clang.llvm.org/extra/pp-trace.html

Py4J – Bridge between Python and Java. (2020). Retrieved from https://www.py4j.org/

REVaMP2 -Round-trip Engineering and Variability Management Platform and Process. (2020). Retrieved from http://www.revamp2-project.eu/tool-chain

Santos, A. R., do Carmo Machado, I., de Almeida, E. S., Siegmund, J., & Apel, S. (2019). Comparing the influence of using feature-oriented programming and conditional compilation on comprehending feature-oriented software. *Empirical Software Engineering*, *24*, 1226–1258.

Shatnawi, A., Seriai, A.-D., & Sahraoui, H. (2016). Recovering software product line architecture of a family of object oriented product variants. *The Journal of Systems and Software*, 1–22.

She, S., Lotufo, R., Berger, T., Wasowski, A., & Czarnecki, K. (2008). Reverse engineering feature models. *Proc. of 33rd Int. Conf. on Software Engineering, Waikiki, Honolulu, Hawaii, USA*, 461–470.

Software and systems engineering – reference model for product line engineering and management. (2015). *ISO/IEC26550:2015*.

Tang, Y., & Leung, H. (2015). Feature mining for product line construction. *SoftEng 2015: The First International Conference on Advances and Trends in Software Engineering*.

Tomassetti, G. (2020). Antlr mega tutorial. Retrieved from https://tomassetti.me/antlr-mega-tutorial

van Gurp, J., Bosch, J., & Svahnberg, M. (2001). On the notion of variability in software product lines. *Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), Amsterdam, The Netherlands*.

Yang, Y., Peng, X., & Zhao, W. (2009). Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. *Proc. of 16th Working Conf. on Reverse Engineering, WCRE 2009, Lille, France*, 215–224.

Zhang, B. (2015). *Vital – reengineering variability specifications and realizations in software product lines* (Doctoral dissertation).

Zhang, B., Becker, M., Patzke, T., Sierszecki, K., & Savolainen, J. E. (2013). Variability evolution and erosion in industrial product lines: A case study. *Proceedings of the 17th International Software Product Line Conference, ser. SPLC '13, New York, NY, USA: ACM*, 168–177.