Software Product Line Adoption Process for Legacy Embedded Control Systems

Submitted to

Graduate School of Information Science and Technology

Osaka University

July 2009

Kentaro YOSHIMURA

ABSTRACT

The increasing functional requirements of embedded systems have led to an enormous increase in software complexity and size. Key factors in successful software development include short time to market, high product quality, and low development cost. Moreover, there are numerous software variations, since embedded control software is often optimized for each product to improve its real-time performance.

Software Product Lines (SPL) is a software development approach with systematic reuse throughout the development life cycle. The key idea of SPL is to exploit the reuse potential of a product line based on reusable assets, which support both common and variable product features in the product line. In recent years, much research has been conducted on introducing SPL for embedded control systems. Many organizations have reported success after SPL was adopted into their product lines. They have also, however, found serious problems in adopting SPL for legacy embedded control systems.

One problem is the definition of a strategy for adopting SPL for legacy systems. To adopt SPL, commonality and variability analysis must be conducted, and then platform and variation points must be defined. Generally, existing products include hundreds of functions specified in natural languages. Therefore, it takes a long time even for experts on the product line to analyze the specifications.

This dissertation introduces a novel method to assess the commonality and variability of existing systems introduced into a software product line. Refactoring an existing

implementation for future SPL is more reliable than introducing new SPL from scratch, especially for safety-critical systems, like in the automotive domain. The proposed approach identifies code clones between different systems in order to assess the commonality and variability across two products. In assessing commonality and variability, we classify code clones into four categories from the viewpoint of SPL variability. We also apply hierarchical decomposition assessment of systems. By using the proposed method, we can hierarchically assess the commonality and variability between existing systems from the viewpoint of implementation. The method is examined through a case study on engine management systems for vehicles.

Software componentization is also an issue to be solved. Some legacy software has been used for more than a decade. Its architecture was designed when the software's size was very small and is not suitable for current software that is enormous and complex. For effective reuse of software across product lines, a component-oriented software architecture must be applied.

Hence, this dissertation presents a development method for software components in embedded control systems. The development method integrates object-oriented software development and model-based development. The key feature of this method is that a wrapper wraps an automatically generated function, which is handled as an object, and the wrapper is also automatically generated. A software tool was developed to generate the wrapper from the automatically generated function. As a result, controller models can be embedded efficiently as software components, without knowledge of object-oriented design. The proposed development method was examined in terms of a control subsystem of an engine management system.

Another problem is the complexity of dependency constraints across variable features. Existing products include hundreds of "functionalities" and thousands of "features". Even if SPL is introduced, an effective configuration approach for variable features is required.

Therefore, a method is proposed to analyze crosscutting features in terms of logical coupling of product release histories, for migration into SPL. Crosscutting features help developers of large embedded systems to reduce the number of variable features. The times for analysis and quantitative evaluation, however, are problems to be solved. This dissertation focuses on the differences between existing products that can be extracted from a product release history. The method applies precision and recall as metrics and determines crosscutting feature candidates quantitatively and automatically. This proposed method was also applied to engine management systems, and it successfully extracted candidates with 97% precision and 31% recall.

The rest of the dissertation is organized as follows. In Chapter 2, we give an overview of SPL and explain the problem of legacy embedded control systems.

In Chapter 3, we present the strategy for introducing SPL for legacy embedded control systems. we first define inter-system code clones and then classify their variations. We then show the results of a case study conducted with legacy automotive control systems.

Chapter 4 presents the development method for reusable software components. We first define the software architecture of embedded control systems for SPL. Then, we describe the model-based development process for software components, before giving the results of a case study on the proposed method.

In Chapter 5, we describe the analysis method for crosscutting features. We first define a product release history and then define logical coupling sets of variable features. Again, we describe a case study on analyzing crosscutting features from existing products.

Chapter 6 gives the result of simulation experiments on introducing SPL for legacy embedded control systems by using both the conventional approach and the proposed approach, thus demonstrating the latter's effectiveness.

Finally, Chapter 7 concludes the dissertation with a summary and directions for future work.

LIST OF MAJOR PUBLICATIONS

(1) Kentaro Yoshimura, Taizo Miyazaki, Takanori Yokoyama, Toru Irie, and Shinya Fujimoto, A development method for object-oriented automotive control software embedded with automatically generated program from controller models, *SAE World Congress 2004: In-Vehicle Software Session*, Document Number: 2004-01-09, March 2004.

(2) Kentaro Yoshimura, Kohei Sakurai, Yuichiro Morita, Nobuyasu Kanekawa, Kenichi Kurosawa, Yoshiaki Takahashi, Shigetoshi Sameshima, and Akitoshi Shimura, A dependable and cost-effective vehicle control architecture for X-by-wire systems based on autonomous decentralized concept, In *Supplemental Volume of the 2005 International Conference on Dependable Systems and Networks (DSN-2005)*, pp. 130-138, June 2005.

(3) Kentaro Yoshimura, Taizo Miyazaki, and Takanori Yokoyama, A model-based development method for object-oriented embedded control systems, *IPSJ Journal*, vol. 46, no. 6, pp. 1436-1446, June 2005. (Japanese)

(4) Dharmalingam Ganesan, Dirk Muthig, and Kentaro Yoshimura, Predicting return-on-investment for product line generations, In *Proceedings of the 10th International Software Product Line Conference (SPLC 2006)*, pp. 13–22, August 2006.

(5) Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig, Defining a strategy

to introduce software product line using the existing embedded systems, In *Proceeding of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT '06)*, pp. 63–72, October 2006.

(6) Dharmalingam Ganesan, Dirk Muthig, Jens Knodel, and Kentaro Yoshimura, Discovering organizational aspects from the source code history log during the product line planning phase–A case study, In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pp. 211–220, October 2006.

(7) Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig, A method to assess commonality and variability of existing systems into a product line, *IPSJ Journal*, vol. 48, no. 8, pp. 2482–2491, August 2007. (Japanese)

(8) Kentaro Yoshimura, Fumio Narisawa, Koji Hashimoto, and Tohru Kikuno, Factor analysis based approach for detecting product line variability from change history, In *Proceedings of Working Conference on Mining Software Repositories (MSR '08)*, pp. 11-18, May 2008.

(9) Kentaro Yoshimura, Thomas Forster, Dirk Muthig, and Daniel Pech, Model-based design of product line components in the automotive domain, In *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*, pp. 170-179, September 2008.

(10) Kentaro Yoshimura, Fumio Narisawa, and Tohru Kikuno, A method to analyze crosscutting features based on logical coupling sets of product release history, *IPSJ Journal* (Submitted). (Japanese)

ACKNOWLEDGMENTS

During the course of this work, I have been fortunate to receive assistance from many individuals. I would especially like to thank my supervisor Professor Tohru Kikuno for his continuous support, encouragement, and guidance throughout this work.

I am also very grateful to the members of my thesis review committee, particularly Professor Katsuto Inoue, Professor Shinji Kusumoto and Professor Takao Onoye for their invaluable comments and helpful criticism of this thesis.

I would like to express my thanks to Associate Professor Kiyoshi Kiyokawa and Assistant Professor Yoshiki Higo for providing the valuable feedbacks, Associate Professor Tatsuhiro Tsuchiya and Assistant Professor Osamu Mizuno for their assistance for completing this thesis.

I want to thank some members of Fraunhofer Institute of Experimental Software Engineering, namely Mr. Thomas Forster, Mr. Dharmaligam Ganesan, Dr. Dirk Muthig and Mr. Daniel Pech for all the valuable discussions on the software product line engineering.

Some works I have engaged at Hitachi, Ltd. have been helpful in preparing this thesis. I would like to express my thanks to Mr. Yasushi Fukunaga, Dr. Takashi Hotta, Dr. Koyo Katsura, Mr. Shinobu Koizumi, Dr. Toshimichi Minowa, Mr. Mamoru Nemoto, Mr. Takaomi Nishigaido, Dr. Toshiharu Nogi and Mr. Takashi Shiraishi for giving me the opportunity of doing applied research in the interesting field of embedded software. I also would like to acknowledge the guidance of Professor Takanori Yokoyama at Tokyo City

University who used to work at Hitachi, Ltd.

Thanks are also due to many colleagues in Hitachi Research Laboratory, especially Dr. Masahiko Amano, Dr. Koji Hashimoto (currently George Mason University), Mr. Tasuku Ishigooka, Dr. Nobuyasu Kanekawa, Mr. Taizo Miyazaki, Dr. Yuichiro Morita, Mr. Fumio Narisawa, Mr. Kohei Sakurai and Mr. Kunihiko Tsunedomi for the fruitful discussions on embedded control systems and for providing valuable feedbacks from industrial perspectives.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Background

Embedded control systems are widely used in our society. They consist of microprocessor-based controllers built into mechanical or electrical equipment. Examples include automotive systems, train control systems, and flight control systems. Software is implemented and executed in these systems.

The increasing functional requirements of embedded systems have led to an enormous increase in software complexity and size. Key factors in successful software development are short time to market, high product quality, and low development cost. Moreover, embedded control software involves numerous software variations, since it is often optimized for each product to improve its real-time performance.

Software product line (SPL) is a software development approach with systematic reuse throughout the development lifecycle [5,7,22,29,37,39]. The key idea of SPL is to exploit the reuse potential of a product line through reusable assets, which support both common and variable features in the product line. In recent years, much research has focused on introducing SPL for embedded control systems [27, 31, 33, 34]. Many organizations have

reported success adopting SPL into their product lines. They have also found serious problems, however, in adopting SPL for legacy embedded control systems.

One problem is defining a strategy for adopting SPL for legacy systems. To adopt SPL, commonality and variability analysis must be conducted, and then platform and variation points must be defined. Generally, existing products include hundreds of functions, which are specified in natural languages. Therefore, it takes a long time to analyze the specifications, even for experts on the target product line.

Software componentization is also an issue to be solved. Some legacy software has been used for more than a decade. The architecture was designed when the software's size was very small, and such architecture is not suitable for current software, which is large and complex. For effective software reuse across a product line, a component-oriented architecture must be applied.

Another problem is the complexity of dependency constraints across variable features. Existing products can have hundreds of different functionalities and thousands of features. Even if SPL is introduced, an effective configuration approach for variable features is required.

Given these considerations, this dissertation addresses the problems of adopting SPL for legacy embedded control systems.

1.2 Main Results

1.2.1 SPL Adoption Strategy

In this dissertation, I introduce a novel method to assess the commonality and variability of existing systems introduced into a software product line. Refactoring an existing implementation into a future SPL is more reliable than creating a new SPL from scratch, especially for safety-critical systems like those in automotive domain.

Research in SPL has mostly focused on constructing product line infrastructures and activities by considering the requirements of future products: scoping, domain analysis, architecture creation, and variability management [27,31,34]. On the other hand, existing products contain much domain expertise and are reliable from an industry viewpoint. Commonality and variability analysis for existing software is one of the most important issues in defining a future product line while reusing existing software.

Kang et al. [16] proposed a method to adopt SPL for a legacy system through feature-oriented analysis. This technique recovers the implemented architecture from the dependencies of the system's functions and then improves the architecture by using the analyzed features. This approach, however, can only be applied to a single legacy system, yet even legacy systems can consist of several existing products. Knodel et al. [17] and Arciniegas et al. [1] proposed methods to compare an existing architecture to a reference architecture for SPL, but their approaches analyze only the dependencies of the component interfaces. Therefore, implemented source code cannot be migrated to an SPL.

In the approach introduced here, I identify code clones between different systems in order to assess the commonality and variability across two products. In this assessment, the code clones are classified into four categories from the viewpoint of SPL variability.

Higo et al. [14] and Kolb et al. [19] proposed methods to analyze code clones for refactoring a software system, but their methods can be applied only to code clones in a single system. Balazinska [3] et al. proposed a method to classify code clones into several types in order to refactor existing systems, but they applied this technique only for multiple product versions, not for product variations in parallel. Clements et al. [8] and Krueger et al. [21] presented approaches for improving software architecture by analyzing code clones at variation points in a reference architecture. Their focuses, however, were on improving SPL architecture, not on adopting SPL into an existing architecture. Yamamoto et al. [40] introduced metrics to evaluate the similarity between different systems, but these

metrics characterize only the system-level similarity.

In contrast, I apply hierarchical decomposition assessment of systems. By using this method, we can hierarchically assess the commonality and variability between existing systems, from the viewpoint of implementation. I also examine the method with a case study on engine management systems for vehicles.

1.2.2 Software Componentization

Next, I present a development method for software components for embedded control systems. This development method integrates object-oriented software development and model-based development.

In general, software componentization is a prerequisite for adopting SPL [29]. Component technology enables developers to package software as loosely coupled parts. Therefore, object-oriented software development, which excels in the reuse of software components, has been attracting attention in this domain [13].

In the field of control systems, model-based design is becoming more important. For example, an automotive engine controller is designed in a domain-specific language with a computer-aided design (CAD) tool, and the controller model is checked by simulation on the CAD tool. C code can also be generated from the controller model. Recently, the quality and efficiency of such code is reaching a level suitable for production [35], but development methods have not been established for generating embedded code automatically for production.

Methods for developing software components for embedded control systems by using block diagrams have been presented [11, 30]. These methods extend block diagrams to the design of objects on modeling tools. This requires not only software engineers but also control engineers to understand object-oriented design.

In contrast, one feature of my development method is that an automatically gener-

ated function, which is handled as an object, is contained in a wrapper, which is also automatically generated. I describe software developed to generate the wrapper from the automatically generated function. As a result, controller models can be efficiently embedded as software components, without knowledge of object-oriented design. I examine application of the proposed development method for a control sub-system of an engine management system for vehicles.

1.2.3 Crosscutting Feature Extraction

Lastly, I propose a method to analyze crosscutting features in terms of logical coupling of product release histories, for migration into SPL. Crosscutting features help developers of large embedded systems to reduce the number of variable features. The analysis time and quantitative evaluation of crosscutting features, however, are problems to be solved.

The concept of a logical coupling set has been applied for recovering software architecture [12] and guiding software changes [42] in the field of software maintenance. The approaches, however, are limited to analyzing linear change history and cannot be applied to software variations developed in parallel.

Loesch et al. [23] and Conejero et al. [9] proposed a method for reducing the number of variable features by introducing the concept of crosscutting features. Their approaches analyze SPL infrastructure, however, rather than legacy systems. Some embedded control systems, such as automotive control systems, are safety critical. These systems require dependability, and developers are thus eager to reuse legacy systems. Therefore, crosscutting features should be analyzed and extracted from product release histories.

Hence, I introduce a novel analysis method for crosscutting features, which uses the product release history. I focus on the differences between existing products that can be extracted from product release histories. This method also uses the precision and recall as metrics and determines candidate features quantitatively and automatically. I have

applied the proposed method to engine management systems and found that it successfully extracted candidates with 97% precision and 31% recall.

1.3 Overview of the Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, I give an overview of SPL and also explain the problems of adopting it for legacy embedded control systems.

In Chapter 3, I present the strategy for introducing SPL into such legacy systems. I first define inter-system code clones and classify variations of the clone types. I then show the results of a case study conducted with legacy automotive control systems.

In Chapter 4, I present the development method for reusable software components. I first define a software architecture for embedded control systems with SPL, and then describe a model-based development process for software components. Results are given for a case study on developing software components with the proposed method.

Chapter 5 describes the analysis method for crosscutting features. I first define the product release history, and then the logical coupling set of variable features. I also show the results of a case study on analyzing crosscutting features of existing products.

Chapter 6 gives the results of a simulation of introducing SPL into legacy embedded control systems, using both conventional approaches and the approach presented here, demonstrating the effectiveness of the proposed methods.

Finally, in Chapter 7, I conclude this dissertation with a summary and directions for future work.

CHAPTER 2

SOFTWARE PRODUCT LINES

2.1 Basic Concept of SPL

Software product line engineering is a paradigm for developing software systems by using platforms and customization. A software product line consists of a family of software systems that have both some common functionalities and some variable features. A functionality is an objective that a system should achieve, and a feature is a characteristic of the system. The following is a definition of a software product line by Clements and Northrop [7].

> A *software product line* is 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 core assets in a prescribed way.

Commonalities are characteristics that are common to all possible products of a product line. For example, an operating system (OS) can be a commonality when an organization decides to provide a solution for only one particular OS for a product line.

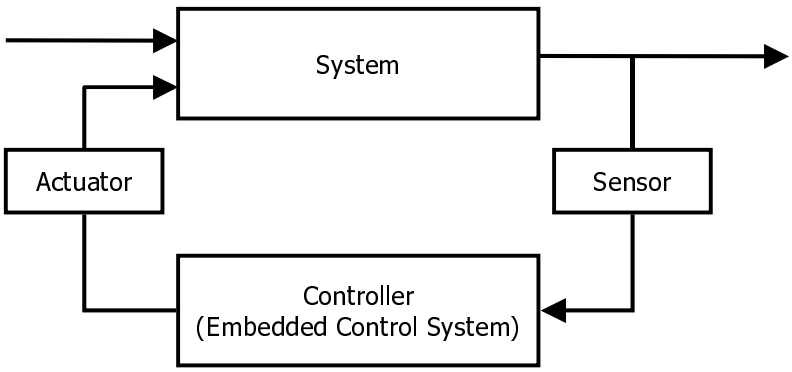

Figure 2.1: Embedded control systems

Variabilities are characteristics that can vary from one product to another in a product line. A variability represents a subset of all possible configurations of the product line. Variabilities are referred from all kinds of development artifacts: requirements, architecture, design, source code, and tests. Managing variabilities is a major goal of SPL.

2.2 Embedded Control Systems

In this dissertation, we define embedded control systems as microprocessor-based controllers built into mechanical or electrical equipment. Figure 2.1 shows an example of embedded control system. Examples of embedded control systems include automotive components, train control systems and flight control systems.

The embedded software is implemented and executed in a microprocessor. The automotive industry recognized that electronics and software developments would represent 90% of all new-vehicle innovations, with embedded software controlling a variety of mechanical functions. The increasing functional requirements of embedded control systems have led to an enormous increase in software complexity and size.

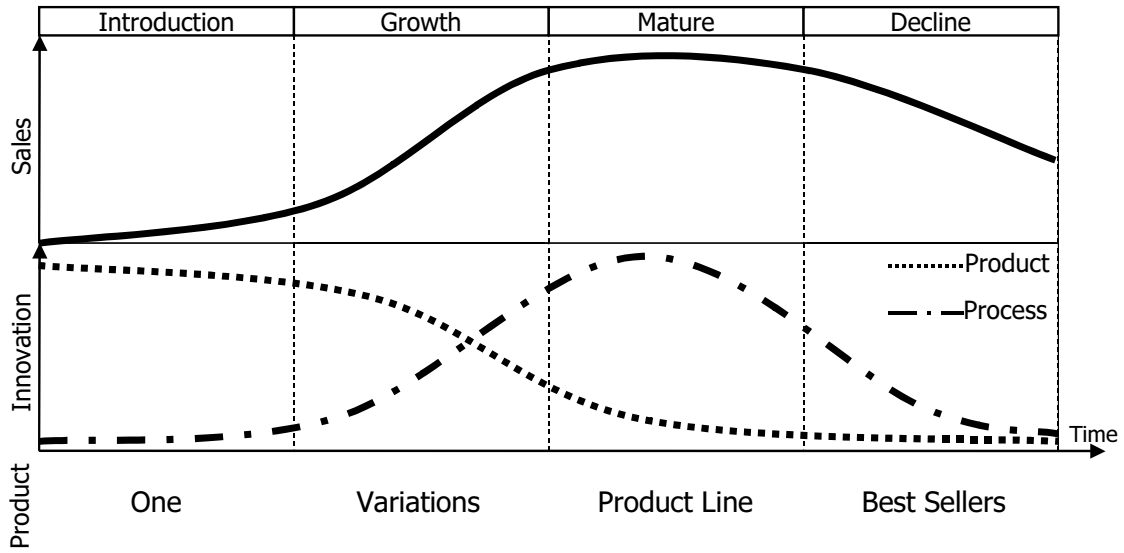

Figure 2.2: Product life cycle model

2.3 SPL Adoption for Legacy Systems

2.3.1 Product life cycle model

In general, a product life cycle goes through four phases, namely, the market introduction stage, the growth stage, the mature stage, and the decline stage. Figure 2.2 shows an overview of the product life cycle model [36].

The product life cycle starts from the introduction stage. The market is almost nonexistent, and sales volume is limited. At this stage, product innovation is essential for creating a new market. On the other hand, process innovation is less important, since there is only one or a few product variations.

Next, the cycle moves to the growth stage. Sales volume increases significantly, and profitability begins to rise. Competition begins and establishes the market players. Product innovation is still important for achieving requirements from the new customers accompanying market extension. Process innovations like cost reduction begin because of economies of scale. Product variations are introduced to satisfy the diverse requirements

of different customers.

Then the cycle goes to the mature stage. The sales volume peaks, and market saturation is reached. Costs are lowered as a result of production volumes increasing. Brand differentiation and feature diversification are emphasized to maintain market share. Product innovation becomes less important, since the basic design has been standardized by de facto market forces or an international standard. Product line infrastructure is introduced to achieve process innovation in the mature stage.

Finally, the cycle enters the decline stage. Generally, product innovation based on an emerging technology triggers this transition. The sales volume declines, prices drop, and profit decreases.

2.3.2 SPL Adoption

Although SPL is an effective approach in the mature stage, the product life cycle must first go through the introduction and growth stages. Especially in the growth stage, products evolve gradually to achieve the increasing requirements from new customers and to compete with other players in the growing market. As a result, many product variations have been developed as legacy systems when SPL is introduced.

Figure 2.3 shows a conceptual schematic of SPL adaption for legacy systems. First, there are legacy systems developed during the growth stage of the product life cycle. The SPL approach is introduced into the organization for these legacy systems. As a result of SPL adoption, SPL core assets are set up. These core assets include feature models, a reference architecture, reusable components, and so on. Then, the organization derives new products by reusing the core assets, i.e., by binding variable features, resolving the reference architecture, and implementing components for a specific product.

In this dissertation, we propose a novel SPL adoption process for legacy embedded systems. Figure 2.4 shows an overview of the proposed process. Legacy systems are the

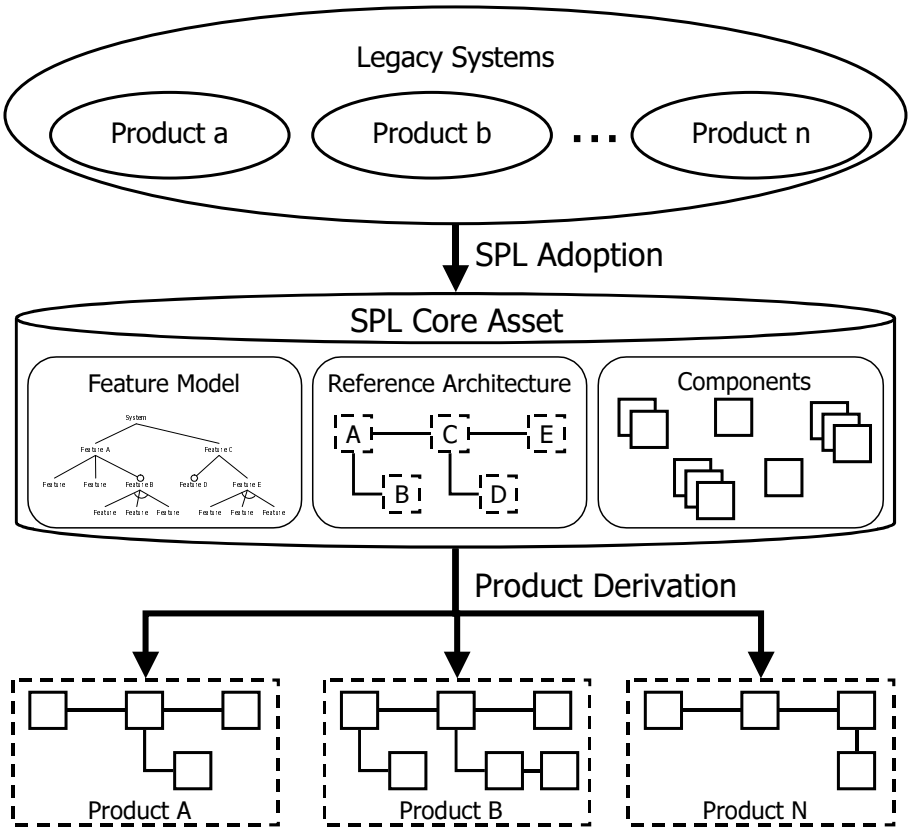

Figure 2.3: SPL adaption for legacy systems

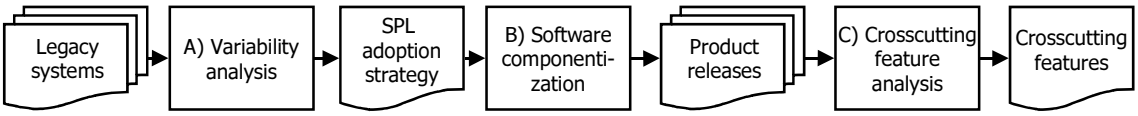

Figure 2.4: Overview of the SPL adoption process

inputs of the process.

First, we present a method to analyze the commonality and variability of legacy systems in order to define an adoption strategy. Through the strategy, the organization defines which parts of a system should be a platform for common requirements and which should be reusable components for variable requirements. Product-specific components should be also defined in order to match certain product-specific requirements.

Next, we describe a development method to componentize software, especially for embedded control systems. This method integrates object-oriented software development and automatic program generation from the domain-specific language of control systems.

Finally, we introduce a method to analyze crosscutting features in terms of logical coupling of product release histories, for migration into SPL. Crosscutting features help developers of large embedded systems to reduce the number of variable features.

CHAPTER 3

SPL ADOPTION STRATEGY

3.1 Introduction

The goal of the SPL adoption strategy is to determine how to introduce SPL into a set of existing embedded control systems. The basic approach in achieving the adaption strategy is to compare the implementations of existing legacy systems and assess their commonality and variability.

Variability analysis for existing systems must be automated. These systems consist of an enormous amount of software. For example, automotive systems can have 100 MB of software. Even a sub-system, like an engine management system, can have more than 500,000 lines of code. The specifications are also huge and consist of hundreds of pages. Moreover, specification formats differ among car manufacturers and are written in natural languages like English and Japanese. Automated variability analysis of specifications is not practical, even at the state of the art. Therefore, source code analysis should be automated for variability assessment.

In the domain of embedded control systems, reuse of legacy source code is essential, because safety-critical systems like automotive systems and flight control systems

must have reliability. Reused software is considered more reliable than newly developed software [4]. Therefore, SPL adoption should be based on variability analysis of legacy source code.

In this chapter, we propose a method to analyze the commonality and variability of existing systems for introduction into a product line. The proposed method hierarchically analyzes the commonality and variability of legacy software and classifies the variability according to code clone metrics.

First, we introduce an efficient analysis process for large-scale software variations. The process applies a reference architecture for the product line and hierarchically analyzes the variability among legacy products by referring to the reference architecture, e.g., at the system level, sub-system level, and component level. Through such hierarchical analysis, the common parts and varying parts of the software are sorted in a systematic way.

Next, we propose a metric to evaluate the similarity of code clones between existing products. Code clone analysis is used here as an approach for finding the common portions of source code, but previous approaches detect intra-system code clones to improve the maintainability of a system. The metric proposed here is for inter-system code clones, to evaluate the similarity between existing systems.

In addition, classification is required in order to understand whether code clone portions have commonality or variability in the context of SPL. In adopting SPL, inter-system code clone portions should be classified in terms of commonality or variability and refactored to common sub-systems or variants. We thus introduce a classification for code clones, which is based on the proposed metric and the interfaces of functions. This approach enables organizations to understand the potential for adopting SPL with legacy software.

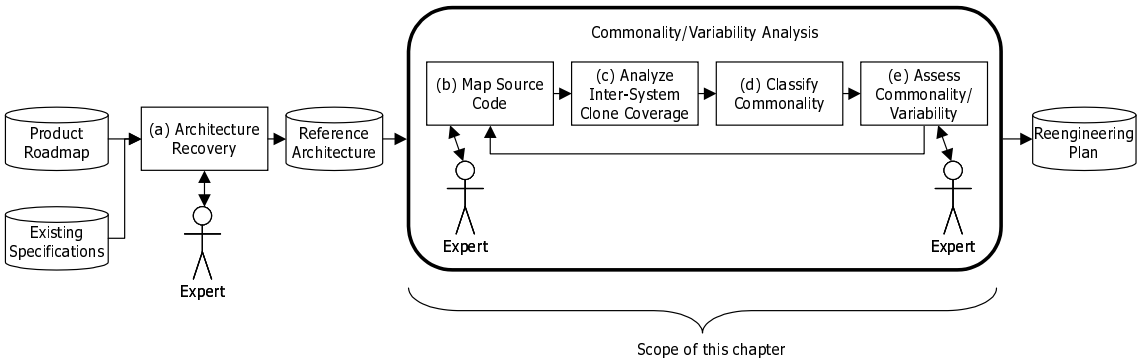

Figure 3.1: Overview of the commonality and variability assessment approach

3.2 Commonality and Variability Analysis

3.2.1 Concept of Proposed Method

An overview of the proposed approach is shown in 3.1. We assume that the target existing systems consist of sets of functions implemented as source code and developed by a single organization.

First, a reference architecture of the target systems is developed by domain experts. The reference architecture is a core architecture that captures the high-level design for the products of the product line and is commonly used by all products. The experts design the reference architecture according to the specifications of the existing products and a product roadmap that defines future products (3.1(a)). The experts must have technical understanding of the products. At this step, the reference architecture does not yet include variability.

Next, the commonality and variability of the legacy software are analyzed. This chapter focuses on the steps involved in analyzing commonality and variability. The functions of the legacy systems are mapped to sub-systems in the reference architecture, and sets of functions are formed (step (b)). Then, the functions are analyzed and evaluated in terms of a metric for inter-system code clones (step (c)). The functions are classified according

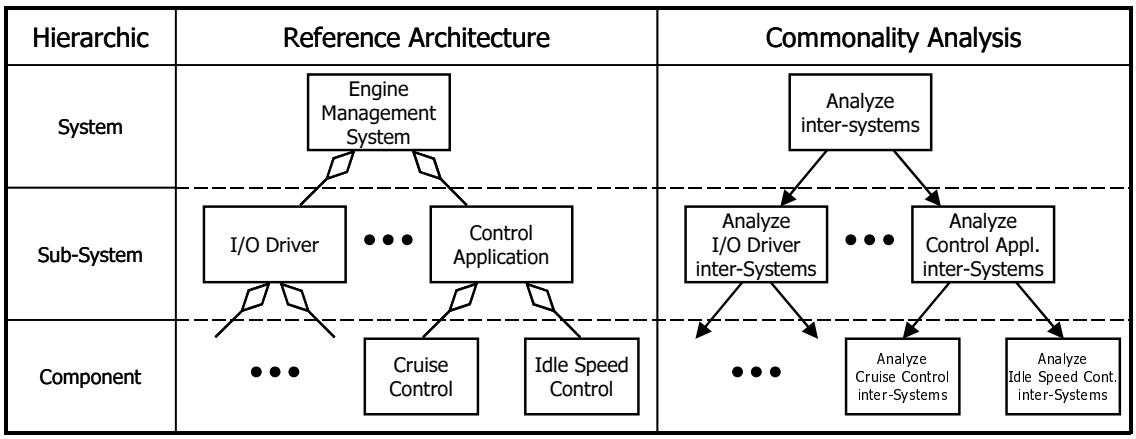

Figure 3.2: Commonality and variability analysis using decomposition hierarchy

to the metric into four types of code clones, as characterized from the viewpoint of SPL (step (d)). Finally, the domain experts determine whether the software sub-systems are common parts or variable parts, by referring to the distribution of the code clone classifications among the sub-systems (step (e)). If a sub-system consists of more finely grained sub-systems, it can be decomposed and analyzed again at a lower level in the reference architecture hierarchy.

The output of the proposed method is an SPL adoption strategy for the legacy systems. This strategy includes the decision to adopt SPL, lists of the common parts and variable parts of the reference architecture, and a reengineering plan for the legacy software.

3.2.2 Hierarchical Mapping

One goal of the proposed method is analysis of clone data from product A to product B. In order to interpret the collected clone data and assess merge potential, we propose a hierarchical clone analysis approach. Figure 3.2 shows an overview of this approach.

First, we assume that both products A and B have one monolithic sub-system, and we analyze the clone classification. Next, we analyze the clone classification of each sub-system from product A to product B, based on a reference. Then, we continue to analyze

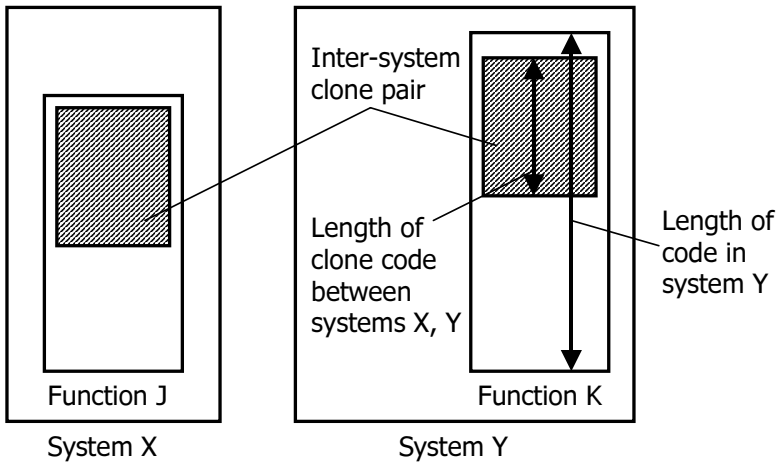

Figure 3.3: Inter-system code clone pair

the sub-sub-systems within a sub-system. In short, this clone analysis for merge potential assessment is carried out at different levels of abstraction by using the decomposition hierarchy shown in Figure 3.2.

3.2.3 Inter-system Code Clone

We first define code clones and clone coverage before explaining the details behind clone analysis to assess the merge potential of software product variants.

In the field of code clone analysis, there are different definitions of code clone [20]. In this approach, we define a code clone as an exact copy without modifications.

Generally, a code clone means a code fragment that is similar or identical to another code fragment [2]. Figure 3.3 shows an inter-system clone pair. Two code fragments form a clone pair if their program text is similar. In the proposed approach, we restrict this to clones of functions, that is, clones from a function in product A to a function in product B. The main reason for this restriction is that later, in order to resolve clones, we can replace existing cloned functions with generic functions that can be instantiated for each product. A commercial tool called CloneFinder [32] is used to find cloned functions.

CloneFinder can find clones that are either an exact copy from one product to another or a copy with some modifications (e.g., a renamed function). This tool does not, however, classify clones into the different types defined below. Hence, we wrote some wrappers around the tool to extract the different types of clones. It is worth clarifying that the level of granularity for reuse is not at the function level, but at the sub-system level. Using the clone coverage metric, we measure the commonality level among sub-systems of existing products.

Clone coverage means the similarity of two sub-systems in two different products. J and K are two sub-systems and K have branched from J. Then the clone coverage in K from J is defined as follows:

$$CloneCoverage(J, K)$$

$$= \frac{\text{Lines of Cloned Code from J to K}}{\text{Lines of Code in J}}$$

If CloneCoverage(J, K) is near 100%, then nearly all the lines of code in K are cloned from J, while if it is near 0%, then there is hardly any code similar to that in J. This clone coverage metric can be applied at any level of abstraction. That is, we can compute clone coverage from one product to another product, and then to the next level of the product decomposition hierarchy. From this point on, the number of lines of code (LOC) in a sub-system is defined as the sum of the numbers of non-commented lines in each function within the sub-system.

3.2.4 Clone Classification

To facilitate merge potential assessment, we propose classifying clones from product A to product B into the following different types. Note that we do not discuss clones within product A or product B but only consider analysis of clones from A to B.

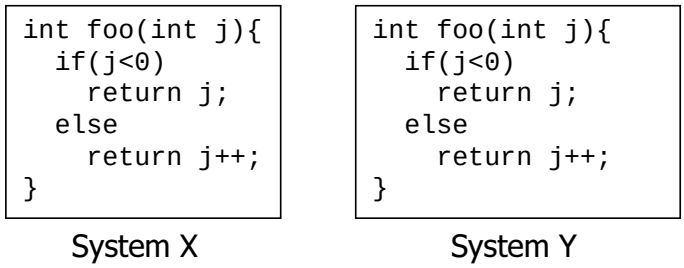

Figure 3.4: Example of a type 1 clone

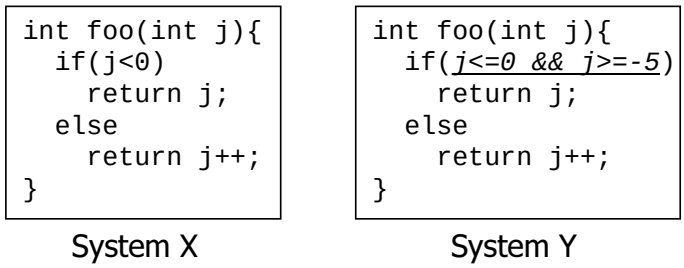

Figure 3.5: Example of a type 2 clone

Also note that we define the clone classifications for variability analysis, different from the types of clones classified by Koschke in a survey of software clones [20].

Type 1: Exact interface and implementation copy from product A to product B (Figure 3.4).

Type 2: Interface copy, but the implementation differ to satisfy product-specific requirements (Figure 3.5).

Type 3: Only the interface is copied, but the implementation differs sufficiently that common sense regards it as different code (Figure 3.6). The difference between type 2 and type 3 clones lies in the choice of the threshold for the clone coverage rate. The type 3 clone is introduced especially to identify variable parts in implementations.

Type 4: The interface is renamed, but the implementation is cloned (Figure 3.7).

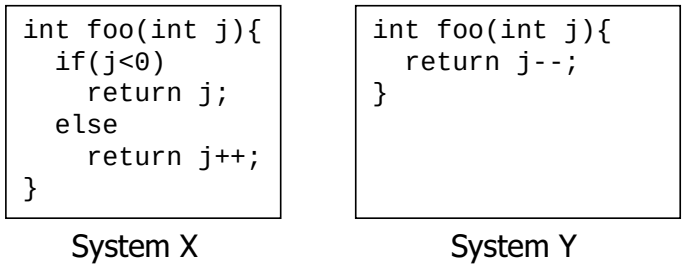

Figure 3.6: Example of a type 3 clone

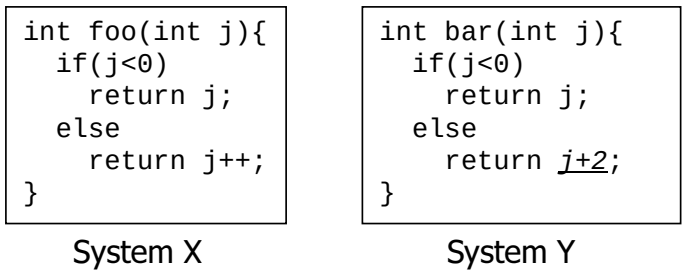

Figure 3.7: Example of a type 4 clone

Note that the above four types account for all possible function clones. The motivation for classifying clones into types 1, 2, and 3 was to quickly understand and identify the common and variable parts in the implementations of products A and B. Type 4 was defined for cases in which programmers rename interfaces but clone implementations from one product to another. To merge existing systems, we need to increase the number of type 1 clones, reduce the number of type 2 clones, keep type 3 clones only if a product needs the same interface but a different implementation, and modify type 4 clones to type 1. Figure 3.8 shows an example of the distribution of code clones of the four types.

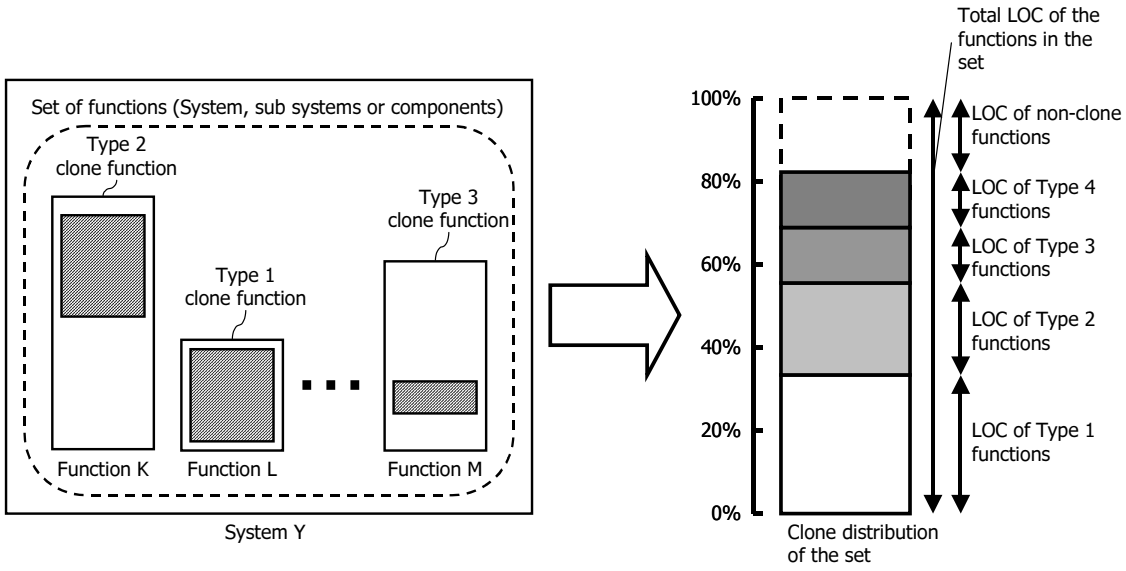

Figure 3.8: Example of clone distribution

3.3 Case Study

3.3.1 Overview

In this section, we apply the proposed process through a case study to assess the merge potential of two automotive products, specifically, two engine management systems (EMSs), for customers A and B. The current products were derived from an initial version, and different groups were formed to address the needs of the global market. Although these products share a common conceptual architecture, their implementation and maintenance are controlled by different groups. Hence, it was a wise decision to develop a merge strategy before introducing a product line.

To assess the merge potential of the EMS products, we use the software architecture as a reference point. The target EMS products are assumed to share the architecture shown in Figure 3.9 [10]. A sub-system in product A is compared with the same sub-system in product B, and the merge potential is assessed. To support this assessment, the product level is analyzed. Next, the sub-system and sub-sub-system level commonalities

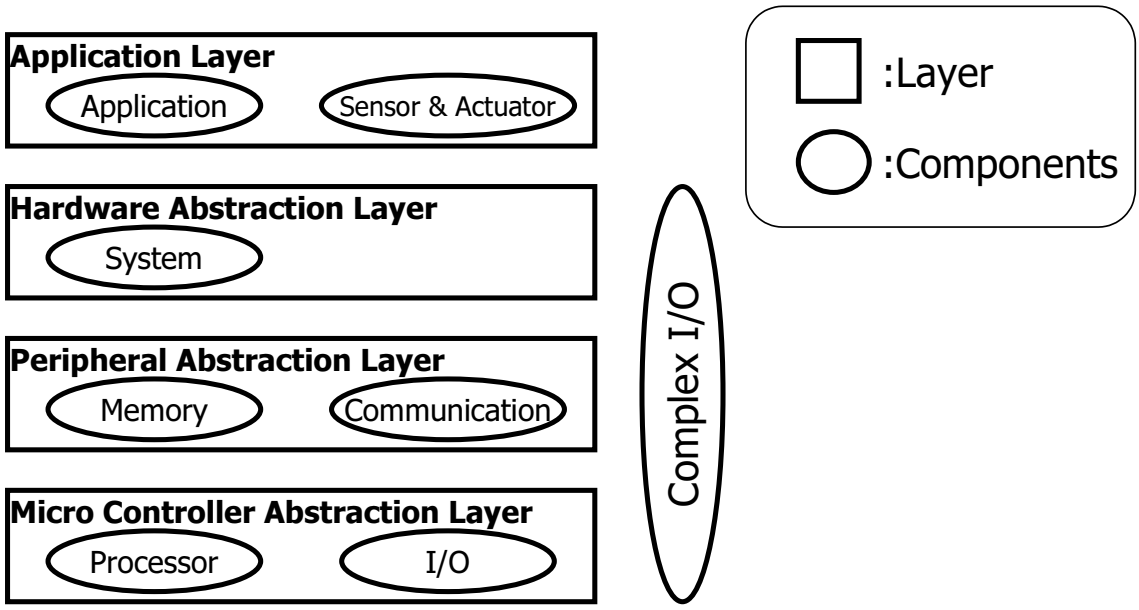

Figure 3.9: Reference architecture for the case study

are analyzed. At this point, a merge strategy can be developed for each sub-sub-system. Finally, we discuss the results of the clone analysis from the domain viewpoint, in terms of the proposed method.

In this case study, if the clone coverage rate of function f for product B from product A is less than 20%, we consider function f to be a type 3 clone.

3.3.2 Results

Clone Coverage: System View

Figure 3.10 shows the distribution of clone coverage at the system level. In the case of the analyzed EMS products, the lines of code for type 1 clones in product B from A cover around 9% of all function code in B. On the other hand, type 2 clones in product B from A cover around 19% of all function code in B. Ultimately, we would like to reduce the number of type 2 clones by separating common and variable parts, thereby reducing code duplication and introducing systematic reuse. Type 3 clones also exist in

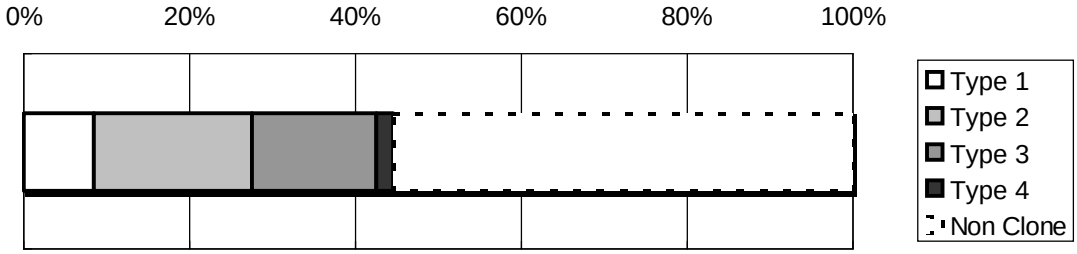

Figure 3.10: Clone distribution at the system level

these products, in this case, for two reasons: (a) certain portions of the EMS products were implemented by different groups, but the interface was reused from the initial root version; and (b) implementation of product-specific functionality was needed, but with the same interface for both products. For product line migration, to avoid code duplication, type 3 clones should be kept only if products require different implementations with the same interface. There are very few type 4 clones, which means that programmers have not changed function names from product A to B. For product B, 55% of the function code was not cloned at all. That is, 55% of the function code in product B has a different implementation than in product A.

We can observe from Figure 3.10 that the type 1 and type 2 clone coverage from product A to B is around 28%. This result confirms that parts of the EMS products can be merged and other parts cannot be merged. To understand this issue more clearly, we use the hierarchical clone coverage view introduced earlier. The next step is to analyze which sub-systems of the architecture are implemented in a different style in each product, and which sub-systems have high clone coverage from product A to B.

Clone Coverage: Sub-System View

we previously showed the system view of the clone coverage from product A to product B. This view is at a high level of abstraction and is only useful for understanding the merge

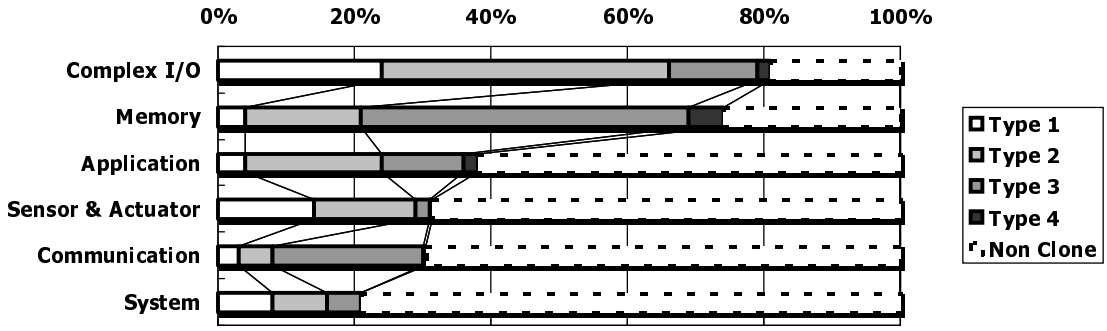

Figure 3.11: Clone distribution at the sub-system level

potential from the system level. That is, Figure 3.10 does not contain any information about the architectural sub-systems of the EMS products. Ideally, we would like to know the clone coverage for each sub-system so that the sub-system merge potential can be assessed. The difficulty with this, however, lies in the abstraction level: architectural sub-systems are not directly visible in source code, but the clone detection results are always at the code level and not at the sub-system level.

To solve this problem, we use mappings. That is, we map the abstract sub-systems to source code for both products from the domain viewpoint. For example, as shown in Figure 3.9, the reference architecture indicates that every file under the IO_Driver directory belongs to the IO_Driver sub-system. This mapping can be applied to lift the collected clone data to the sub-system level. Note that this reference architecture is based on AUTOSAR (Automotive Open System Architecture) [10].

Figure 3.11 shows the clone coverage for all sub-systems from product A to product B. Using this view and the domain knowledge of the architect, we can reason about the clone coverage for each EMS sub-system. Hence, we present the analysis of clone coverage at the sub-system level for all sub-systems of the EMS products.

The Memory Service, Sensor Actuator, and Communication Driver sub-systems implement product-specific functionalities, resulting in low type 1 clone coverage (below

15%).

The Memory Service sub-system has type 1 clone coverage of around 5%, because it implements a functionality related to flash memory operations, which are mainly supplier dependent. As a result, the implementation of Memory Service in product A is significantly different from that in product B. Also, around 50% of the Memory Service sub-system code consists of type 3 clones. This is because for both products, the external interfaces of Memory Service are the same, and hence, the interface was reused from the initial root version of the EMS.

For the Complex I/O Driver sub-system, the type 1 clone coverage is around 25%. This matches expectations, because this sub-system is "complex" and the developers tried to maintain commonality. Note, however, that the type 2 clone coverage is around 35%. We plan to work on resolving type 2 clones in the future.

The System Service sub-system implements system-level service routines and hence is mostly product specific. We can see from the clone coverage view that around 80% of System Service code is not clones.

There are also some unexpected surprises in the clone coverage results. For example, the Application sub-system has type 1 clone coverage of only 5%, whereas we expected around 30% to 40%. From the domain viewpoint, the Application sub-systems in both products contain common domain concepts, but the clone coverage metric does not show high commonality. The reason for this difference can be understood by analyzing the clone coverage for all sub-sub-systems within the Application sub-system.

Clone Coverage: Component View

Figure 3.12 shows the clone coverage for components in the Application sub-system. The figure thus indicates the clone distribution for the nine components in the sub-system.

At this point, we can proceed to develop a merge strategy for software components,

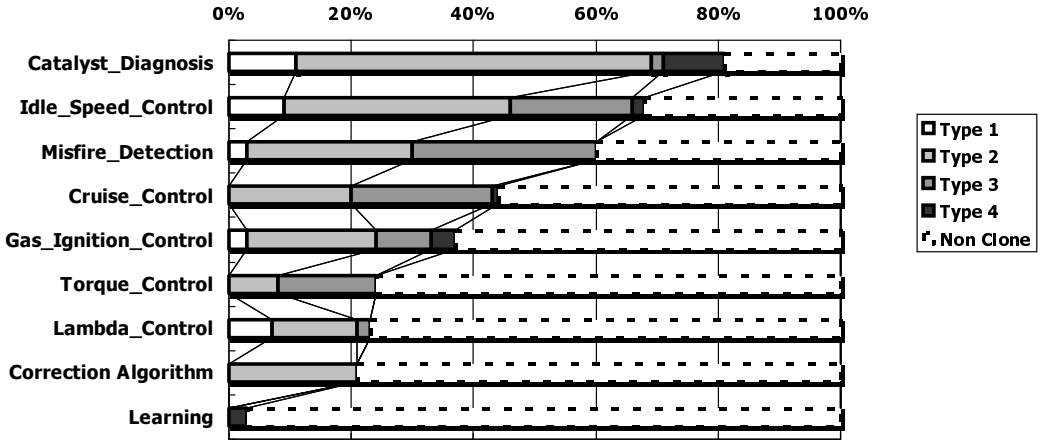

Figure 3.12: Clone distribution at the component level

according to these assessment results.

Reengineering Plan for Components

The Engine Gas Injection Control is a traditional component with stable requirements for engine control systems, but it also has differences or variations from one car model to another model. Nevertheless, this component should be merged and transformed into a generic component, with variation points. In Figure 3.12 we can notice that the type 1 and type 2 clone coverage for the Engine Gas Injection Control component are low: at least 50% was expected from the domain viewpoint. In this case, the merge plan is to transform the Engine Gas Injection Control component from the latest version, which is product A, into a generic component with variation points, which can be instantiated for product B and other future products.

Similarly, the requirements of the Idle Speed Control component are stable for engine control systems. The type 1 and type 2 clone coverage for this component from product A to product B is around 50%, which already indicates that this component can be transformed into a generic component. Here, the strategy is to merge this component from

product A and product B by first separating the common and variable parts from both implementations.

The functionality of the Torque Base Control component shares significant commonalities between products A and B. The clone coverage, however, is low (around 80% non-clone) because the root version of the EMS did not contain this component, and later, it was implemented in different styles by developers belonging to different groups. To merge this component, it is not rational to compare code because there is much more code difference than there is functionality difference. Therefore, the merge strategy is the same as for the Engine Gas Injection Control component.

The Cruise Control component has no type 1 clones and around 60% non-clone lines of code. The Cruise Control is an unstable component and not traditional with respect to engine control software; rather, it belongs to the vehicle control domain. Therefore, it is not a priority to merge the implementations of this component into a generic component.

For the Misfire Detection component, the type 3 clone coverage is around 35%. This means that the same application framework is used in both products A and B, but the implementations are different for specific customers. In this case, only the application framework will be integrated. The implementations of these components will not be merged into generic reusable components.

The Learning component does not have any clones from product A to B, because the learning behavior is different from one car model to another. Hence, this component also is not a candidate for merging into generic reusable components. In this case, the variability at the component level will be maintained (i.e., different learning components will be selected for different car models).

In summary, then, the merge strategy is to transform the Idle Speed Control, Torque Base Control, and Engine Gas Injection Control components into generic reusable components for the EMS products.

3.3.3 Discussion

We have shown that for the two EMS products, the type 1 and type 2 clone coverage from product A to product B was only 28%. Although these products have a significant degree of commonality, the clone coverage does not reflect the domain view. As mentioned earlier, products A and B have a common origin but started evolving separately to address different market segments. In addition, these products are controlled by developers who belong to different groups.

The clone analysis was performed on the two latest versions of products A and B, and analysis of the evolution history was not in the scope of the project for organizational reasons. We performed some additional analysis to understand the reasons for low clone coverage and found two characteristics with respect to product A: (a) around 30% of product A's code was generated automatically through model-driven development, and (b) some portion of the existing assembly code in product A was migrated to the C language. These two activities were not performed for product B. As a result, the code textually differs between products A and B, resulting in low clone coverage.

Another reason for the low clone coverage from product A to B is the nature of the EMS domain itself. An EMS is a mixture of multiple mechanical, electronic hardware, and software components. There are also market-specific regulations; for example, emission rules differ among Japan, Europe, and the United States. To handle all these issues, developers in different groups have tended to change existing code in various ways, and when more and more requirements have to be handled in a sequence of releases, the code commonality among similar products of the same origin tends to shrink.

Finally, we disscuss the validity of the case study. In this case study, we have applied the proposed method to products in one domain, and the products are developed by same organization, because one organization such as a company or a business division develops several products for a domain in general. The proposed approach does not depend on the

clone analysis tool, since we employ the general definition of “code clone” and most of clone analysis tool can extract this type of code clone. We assume that two products have same roots. In the case of two products are developed independently, the proposed approach is not applicable.

3.4 Summary

In this chapter, we have proposed an approach to assess the potential to merge existing systems into a product line. In order to adopt SPL for legacy embedded control systems, the adoption strategy must be planned through variability analysis of the implemented source code of different products.

Code clones between different systems are identified to assess the commonality and variability across products. In assessing commonality and variability, code clones are classified into four categories from the viewpoint of SPL variability. We also apply a hierarchical decomposition assessment of systems. With this method, we can hierarchically assess the commonality and variability between existing systems from the viewpoint of implementation.

We have also examine the proposed method in the context of a case study of engine management systems for vehicles, leading to an adoption strategy for the product line. In the case study, two legacy systems were analyzed hierarchically and sorted into common parts and variable parts. As a result, a reengineering plan was successfully derived.

CHAPTER 4

SOFTWARE COMPONENTIZATION

4.1 Introduction

In adopting SPL for legacy embedded control systems, software must be implemented as reusable software components. In this chapter, we describe a method to develop embedded control software that integrates object-oriented software development and model-based software development with automatic program generation.

In general, embedded control software is developed in two phases. The first is the control design phase, in which control engineers design specifications (control algorithms) for the software. The second is the software development phase, in which software engineers implement software according to the specifications.

Conventionally, specifications are described in natural language during the control design phase, and the software engineers program according to the specifications. Specifications written in natural language, however, are often vague and missing details. Therefore, it is not an easy task for software engineers to implement the control software designed by the control engineers during the control design phase. Moreover, the software engineers cannot reuse the implementation for a specific product.

In the field of control design, model-based design is becoming practical. For example, an engine controller model can be designed with a domain-specific language (DSL) tool, and the model is then verified through simulation on the DSL tool. Since the controller model is written as a formal model, model-based design helps developers to avoid misunderstanding between the control engineers and the software engineers. C code can also be generated from a controller model. Recently, the quality and efficiency of such code has reached a level suitable for production [35].

There are still issues, however, in applying model-based development for industrial production. Automatically generated code is not structured for implementing an embedded control system and not componentized for reuse across a product line. Therefore, model-based development is mainly applied for prototyping, not for production.

On the other hand, object-oriented modeling, which improves the reusability of software components, has also attracted attention [13]. Controller models, however, are designed with domain-specific languages, such as block diagrams for classical control theory and ladder logic for programmable logic controllers, so the object-oriented approach has not been applied for model-based designs for controllers.

The objective of our research is to formulate a development method that integrates object-oriented software design and model-based control design. This chapter describes a way to develop object-oriented embedded control software generated automatically from controller models.

We first propose a software architecture for embedded control systems. We define the granularity of a component as a state variable of a controller. Since a state variable is an element of a dynamical system, we can map a state variable to a software component that is an element of a software architecture. Then, we introduce an application framework in order to integrate components as a control system. Through this approach, a controller model designed using a domain-specific language for control theory can be transformed

to a software architecture designed by the object-oriented approach.

Second, we introduce a method to implement an automatically generated function as an encapsulated software component. A function for calculating a state variable is automatically generated from the controller model, and the function is wrapped by a wrapper interface for encapsulation as a component. The wrapper has update and get functions.

Finally, we have developed tools for generating the above framework and wrappers. Previous tools can generate a function for calculating a state variable from a controller model. To improve development efficiency, we should also automatically generate the framework and the wrapper for a function. The framework tool extracts the calculation order of state variables and generates the application framework. The wrapper tools analyzes a generated function and generates a wrapper for it. In the proposed approach, most application software can be automatically generated.

The rest of this chaper is organized as follows. In Section 4.2, we present the method to develop embedded control software as a set of reusable components. In Section 4.3, we examine the proposed method in the context of a a case study of an automotive system.

4.2 Component Design for Embedded Software

4.2.1 Concept of Proposed Method

Control Design Phase

In general, an embedded controller includes several control functions, so control engineers design controller models separately for the control functions. A controller model is designed using a DSL tool for control theory and for simulation of the model's behavior.

In Figure 4.1, the block shows a calculation, and the arrows show the direction of data flow. This example shows calculations for the target engine torque and the target throttle

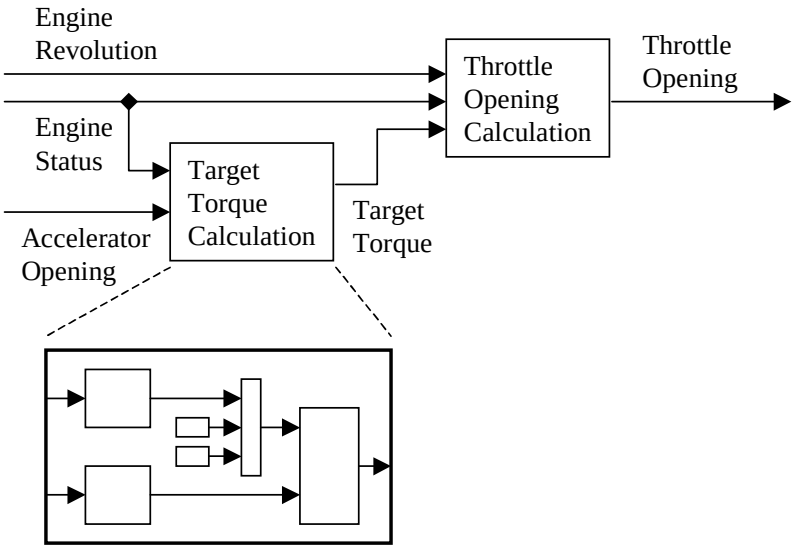

Figure 4.1: Example of block diagram

opening. The details of the calculations are described inside each block. The engine revolution, engine status, and throttle opening are input values, which are calculated by other control functions. Control is achieving by executing the control logic periodically, as described by the controller model.

A controller model is represented hierarchically so that control engineers can understand the control algorithm. In Figure 4.1, the detailed control algorithm for the target torque calculation is represented by the block diagram in the lower level of the hierarchy. The aims of hierarchical decomposition are to establish a common understanding for control engineers and to encapsulate controller models for reusable components.

Here, we introduce a data-centered method to decompose a controller model. In other words, we choose the controller system's state variables as the granularity of the components, such as the input/output values, system's observed state variable, and the target values of the system. These data are rarely deleted or added when the control logic is changed. Therefore, we can build a stable structure of components and modify the control logic by exchanging them.

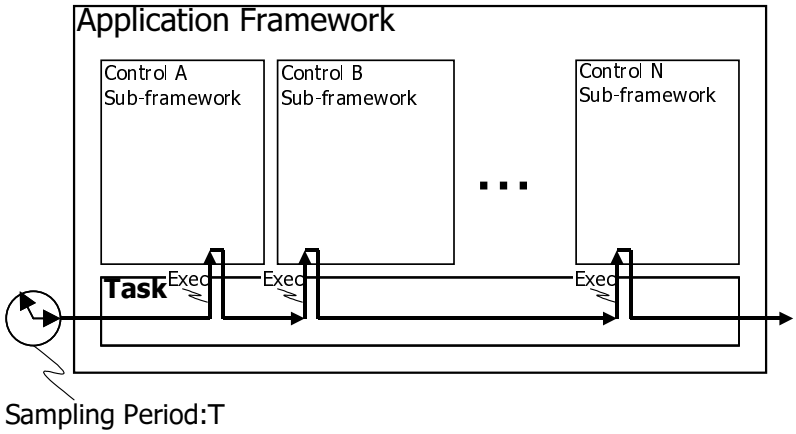

Figure 4.2: Structure of application software

Software Development Phase

In the software development phase, a software engineer implements the software components and application framework according to the controller model designed in the control design phase.

We use the concept of an "application framework" for constructing the control software architecture. An application framework is a standard pattern of objects to implement the functions of an application [15]. Our research group has proposed a model of time-triggered, object-oriented software for embedded control systems [26,41]. The framework presented in this chapter is based on this model.

Figure 4.2 shows a basic example of an application framework. A control function is a combination of control sub-systems that defines a detailed functionality. As the figure shows, a framework consists of several "sub-frameworks", and each sub-framework defines a control function in detail.

A sub-framework defines a composition of software components for a control function. We decompose a control function into software components according to the hierarchical decomposition of the controller model. The aim of the decomposition is to improve

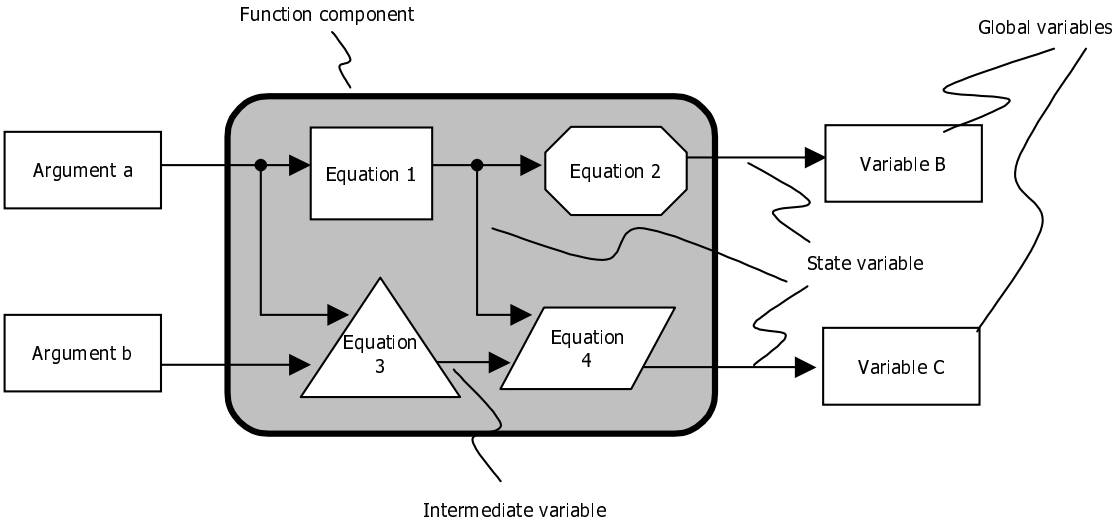

Figure 4.3: Conventional software component

the reusability and exchangeability of the software components.

The application software is usually decomposed into components by focusing on functionality from the viewpoint of control design. This is why several items of data processing and variables are combined in a component, as shown in Figure 4.3. The decomposition into control functionality is one reason why control software is complicated and not especially reusable.

In contrast, we propose a data-centered method to build an application from reusable components. As shown in Figure 4.4, a state variable of the control system is decomposed, and a component is defined in order to calculate and store the variable. Intermediate variables for calculating the state variable are encapsulated in the component, since they are often added and deleted in modifying a control algorithm.

In this approach, we can reuse an application framework for several products in a product line by exchanging software components. Moreover, we can seamlessly map the controller model to the software architecture, since we define the granularity of the software components as the state variables of the control systems.

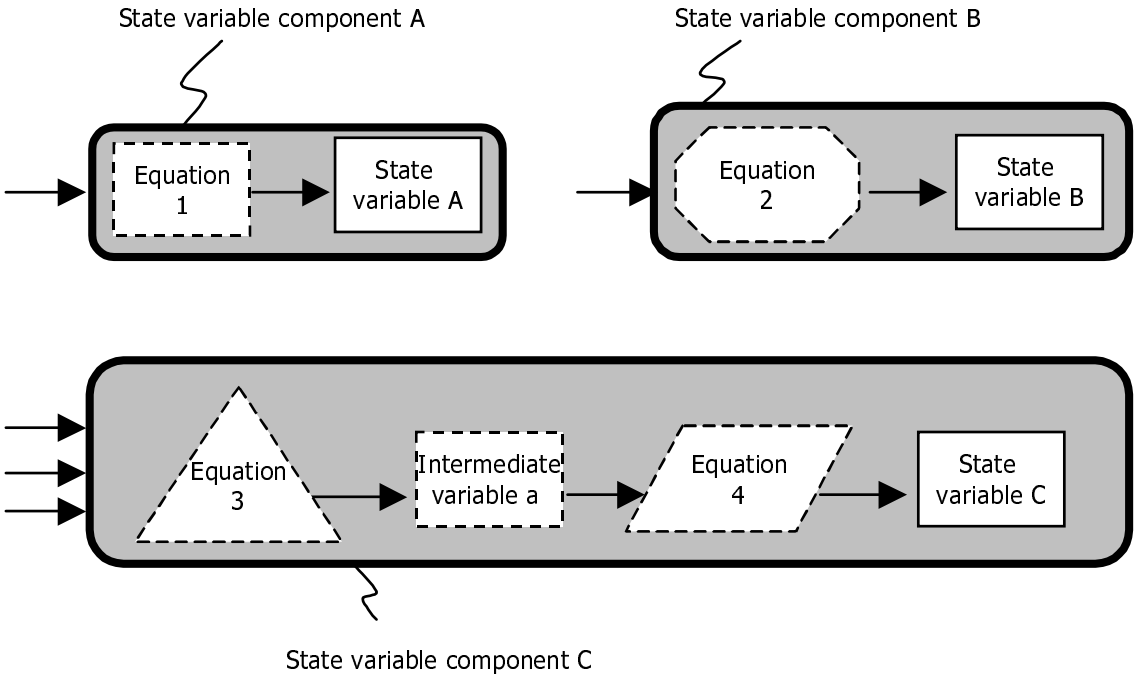

Figure 4.4: Proposed software component

4.2.2 Software Architecture

In the following sections, we describe our method with the example of an automotive engine control system. Figure 4.5 shows the software architecture for this example. The software consists of application software and platform software.

The platform software consists of a real-time OS and an input/output driver. We have adapted OSEK-OS [28] for the real-time OS, since it is one of the de facto standards for automotive systems. To enhance the portability of the application software, the interface of the I/O driver is independent of the hardware.

The application framework has periodic tasks (e.g., 10 ms, 4 ms) and event-triggered tasks (e.g., engine synchronous task). Control functions are implemented as sub-framework and executed from the tasks.

Figure 4.5 thus outlines the behavior of the proposed system. First, the real-time OS activates a periodic task when a timer interrupt occurs (e.g., 10 ms). At the beginning,

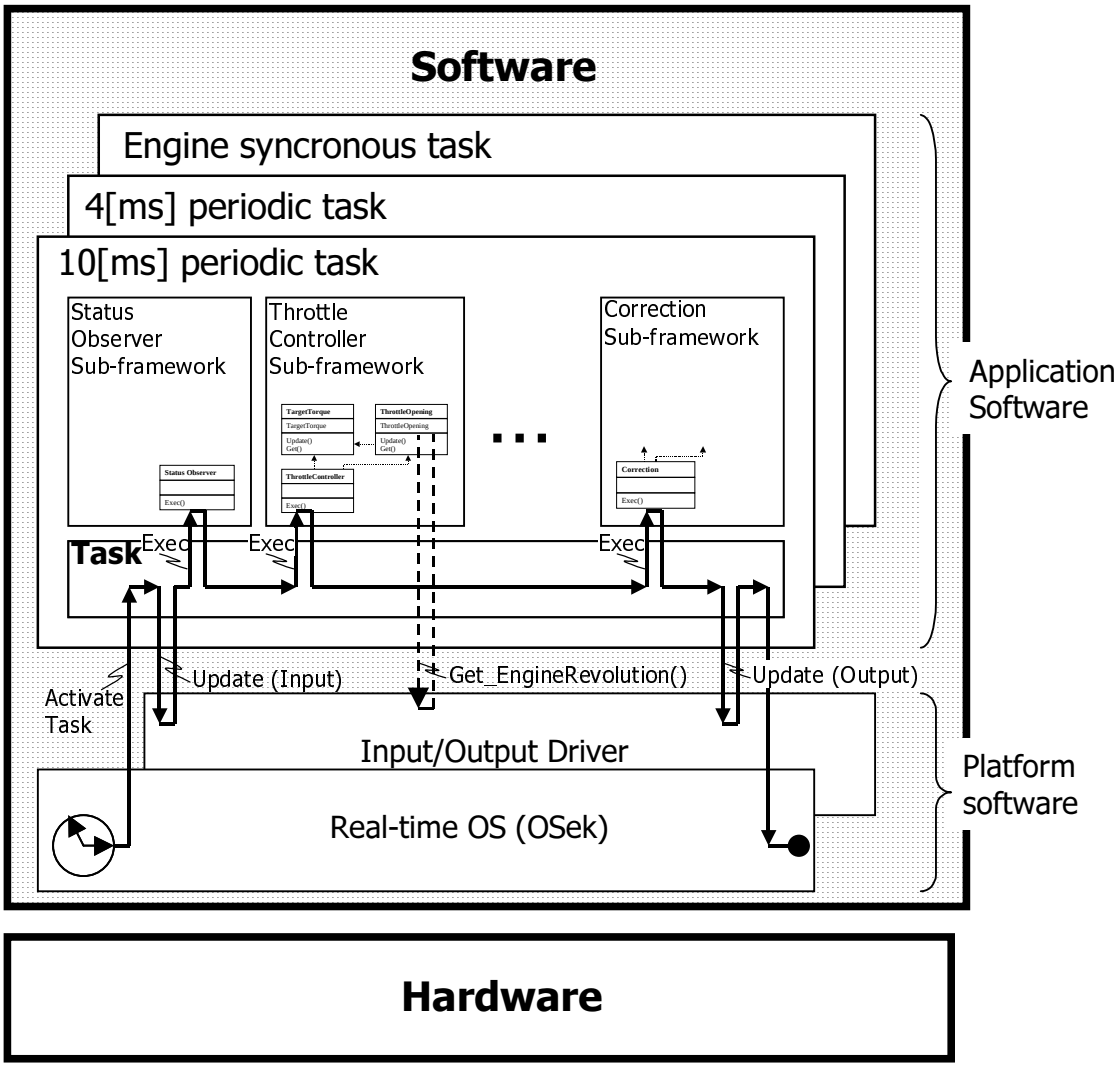

Figure 4.5: Example of proposed software architecture

the task requests the input driver to measure the external input data. Next, the task calls the "Exec()" methods of the sub-framework controller objects in this order: "Status Observer", "Throttle Controller", $\cdots$, "Compensation". Each sub-framework object calls the "Update()" methods of software components. Each software component then refers to input data, which are attributes of the other software components and external input data. The software components calculate and update their attributes. At the end, the task requests the output driver to control an actuator.

4.2.3 Software Component Design

Software components are written in the C language, which is not an object-oriented language, for efficiency of implementation size [25]. In this subsection, we describe the proposed software component design with an example of the component shown in Figure 4.6. This is a software component for the state variable TargetTorque. The TargetTorque component thus has an attribute "TargetTorque" to store the variable, a method "Update()" to calculate the variable, and a method "Get()" to access the variable.

A method for calculating the attributes of a software component is automatically generated as a C function by commercial code generation software [35], from the block diagram models of the DSL tool. The calculation methods are generated as functions with the syntax "Variable Calculate()". Input values are assigned as arguments of the C function. The output value's address is assigned the argument pointer of the C function. In the example of Figure 4.6, a C function "TargetTorque Calculate()" is generated.

To implement automatically generated functions as software components, we use wrappers. Each wrapper corresponds to a function. A wrapper declares a public attribute of the corresponding component and defines the data update and data access methods.

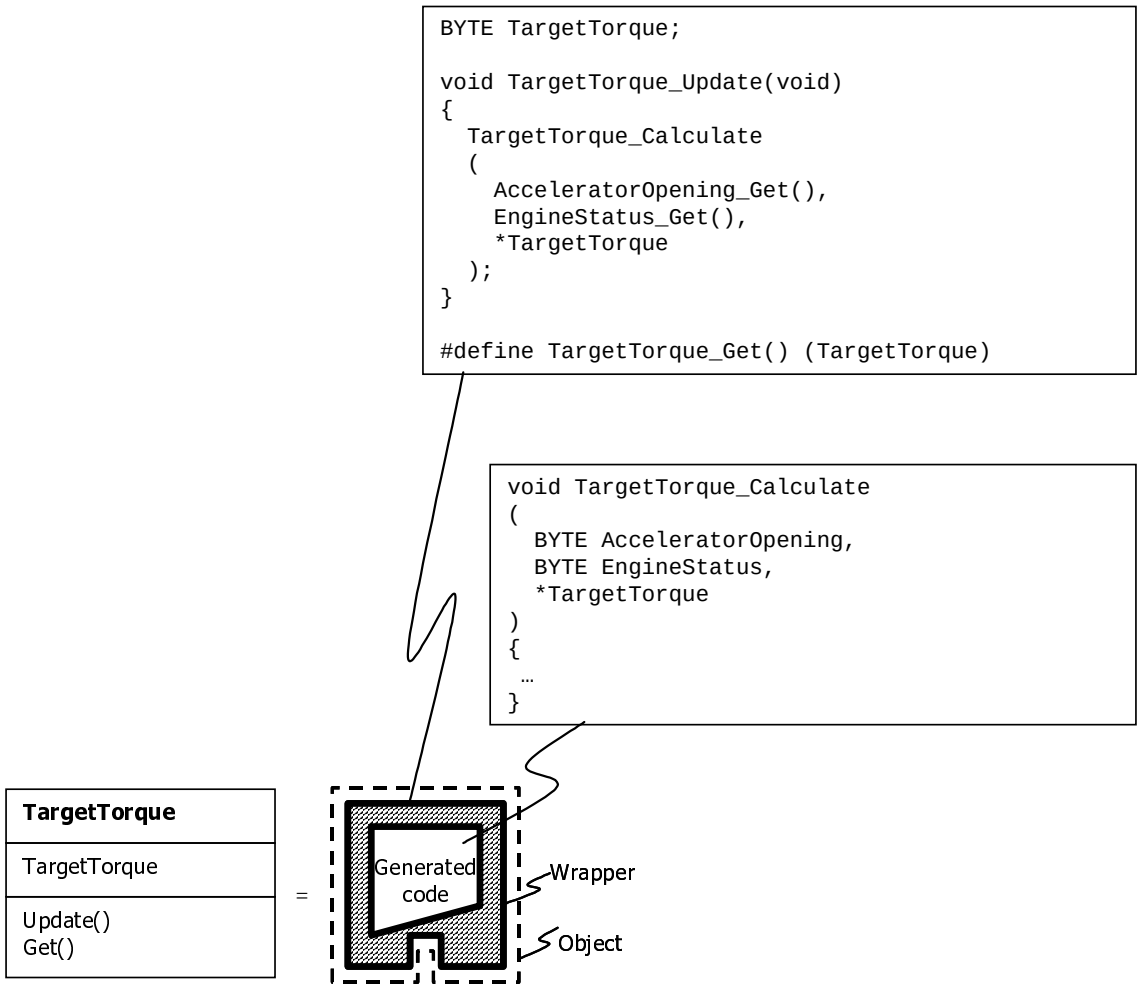

Figure 4.6: Example of a component

Declaring an attribute

Each wrapper declares a public attribute (data) of a component to store the result of a calculation. The attribute is a variable with the name of the value. In the example of Figure 4.6, an attribute "TargetTorque" is declared by the wrapper.

Data update method

The data update method executes a calculation for the attribute of a component. The data update method is a function named "Variable_Update()". This method gets input argument(s) from automatically generated functions by calling the data access methods of the software components that calculate the variables as arguments. This method also assigns the variables as input arguments of the function "Variable Calculate()", and it assigns the pointer of an attribute as the output argument of the function.

In the example of Figure 4.6, the method "TargetTorque Update" accesses the attributes "AccleratorOpening" and "EngineStatus" by calling the data access methods "AccleratorOpening_Get()" and "EngineStatus_Get()". It then assigns the variables as input arguments of the function "TargetTorque_Calculate()". The method also assigns the pointer of the attribute "TargetTorque" as the output argument of the function.

Data access method

The data access method is called when another software component refers to the result of the software component's calculation. This method is implemented as the macro "Variable_Get()" for efficiency of size and execution.

In the example of 4.6, the macro "TargetTorque_Get()" is defined as the data access method.

One of the features of the proposed approached is that the interface of the data update method has no arguments. When control logic is modified, a set of data is often added

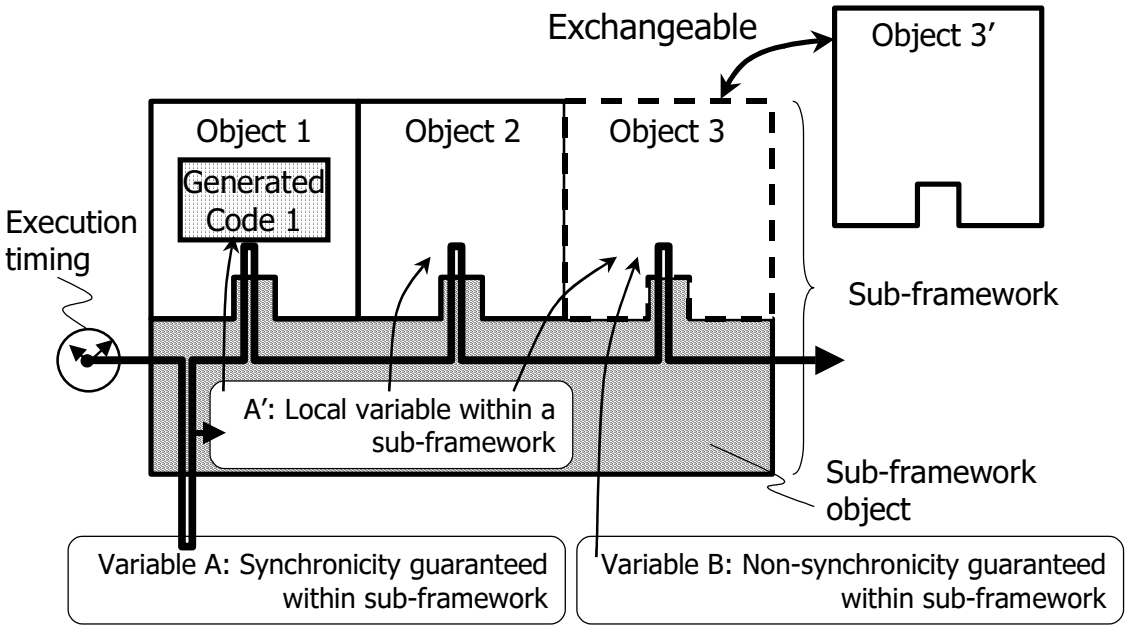

Figure 4.7: Sub-framework

or deleted, where such data can be referred to by each software component during calculations. This implementation method can hide the modification of the data set inside the wrapper's data update method, which is generated automatically. Therefore, it is not necessary to modify a sub-framework object that calls the data update methods of software components; that is, we can reuse the sub-framework object without any modification.

4.2.4 Sub-Framework Design

Figure 4.7 shows an overview of a sub-framework. It consists of software components, as described in the previous section, and a sub-framework object that executes the components. The aim of the sub-framework object is to perform a control function by executing software components in the order specified by a control engineer.

Figure 4.8 shows composition of a sub-framework object and software components. In this example, a sub-framework object "ThrottleController" calls the data update methods of the "TargetTorque" component and the "ThrottleOpening" component.

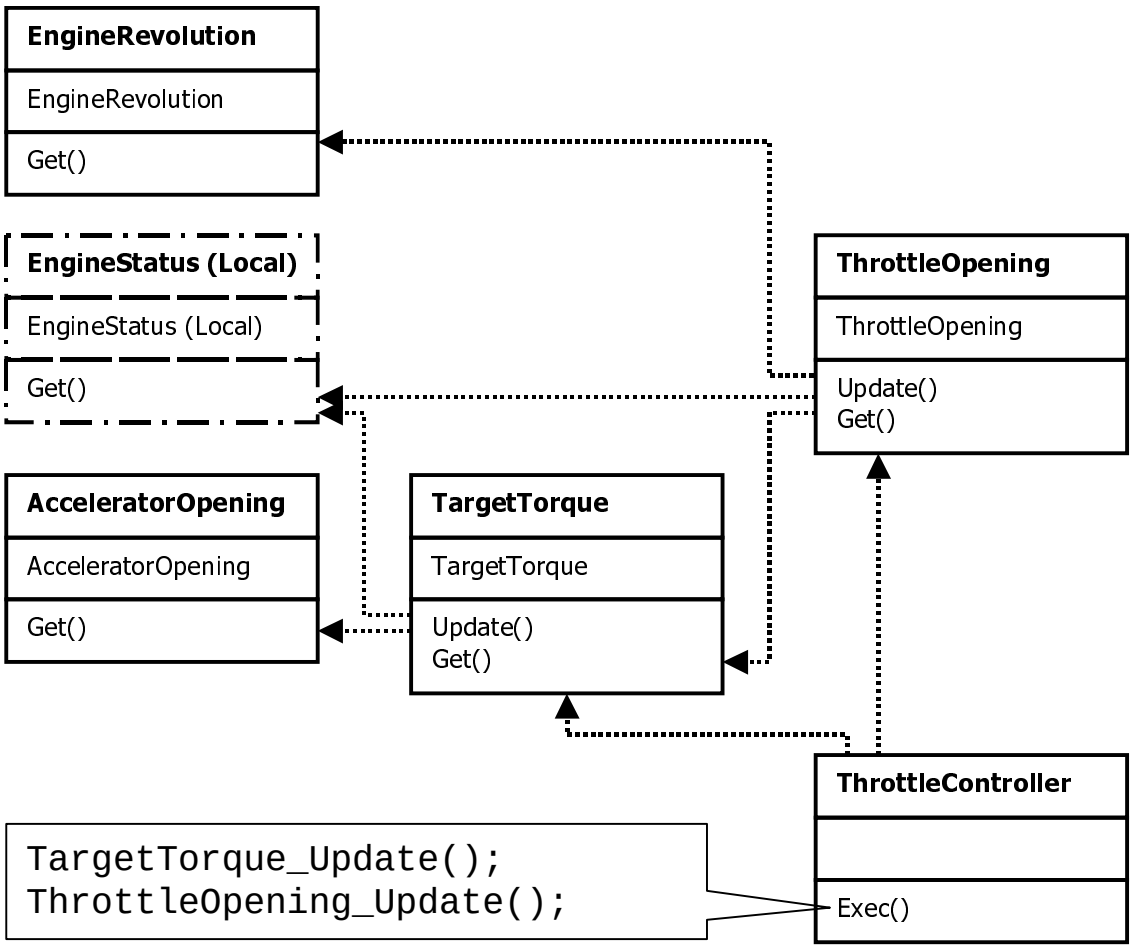

Figure 4.8: Example of sub-framework composition

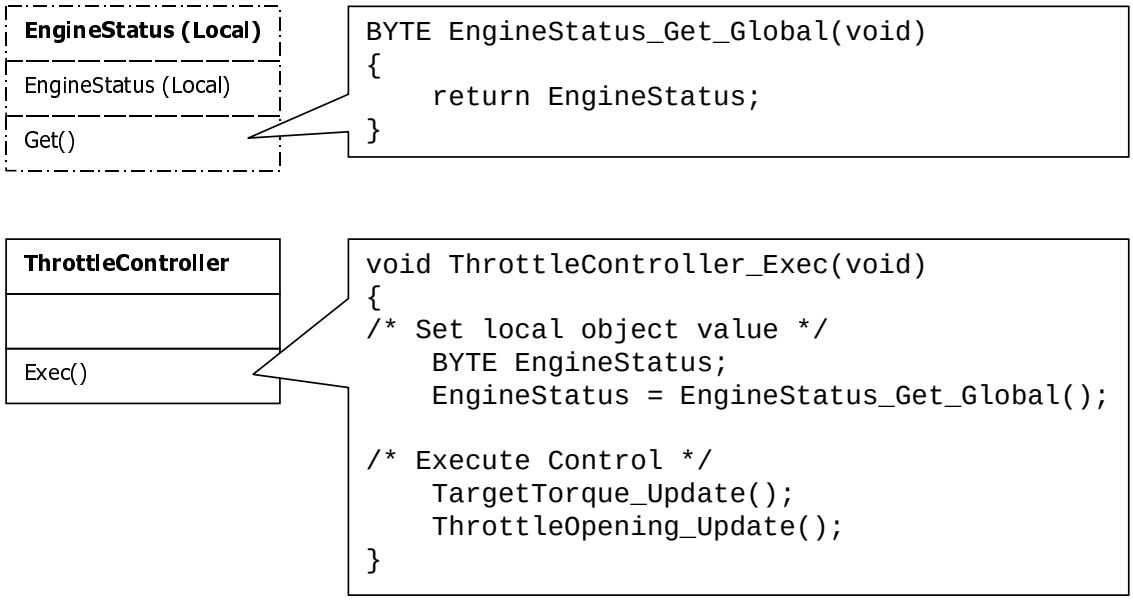

Figure 4.9: Example of a sub-framework object

Figure 4.9 shows an example consisting of the sub-framework object and a local component. The sub-framework object "ThrottleController" calls the update methods "TargetTorque_Update()" and "ThrottleOpening_Update()" in turn. "TargetTorque_Update()" calls the access methods "EngineStatus_Get()" (Local) and "AcceleratorOpening_Get()", and then calculates the value of "TargetTorque". "ThrottleOpening_Update()" calls the access methods "EngineRevolution_Get()", "EngineStatus_Get()" (Local), and "TargetTorque_Get()", and then calculates the value of "ThrottleOpening".

The order of execution is based on the data flow specified in the controller design, as shown in Figure 4.1. The sub-framework object has to activate the software components through a procedure that does not contradict the data flow specified in the block diagram of the controller model. The sub-framework object thus calls the data update methods from the head to the tail of the block diagram.

The sub-framework also guarantees the synchronicity of the variables. An embedded control system is generally a multi-tasking system consisting of several control functions.

Each function has sampling periods and priorities. The sub-framework object maintains the synchronicity of data values, in case multiple software components in the same sub-framework refer to the same value of a software component. Hence, the sub-framework can operate control functions under preemption control.

A local component is also generated for data that is managed to maintain the synchronicity of a variable. In Figure 4.8, although "EngineStatus" is updated in other sub-frameworks, it is referred to by two components in this sub-framework. If the priority of the sub-framework that updates "EngineStatus" is higher than that of the sub-framework shown in the figure, "EngineStatus" could be updated, interrupting the execution of the sub-framework. Therefore, a local component of "EngineStatus(Local)" is generated to manage the synchronicity of values. The local component declares a local variable "EngineStatus(Local)" and copies the attribute "EngineStatus" to the local variable. The access method "Get()" is a macro, so the scope of the access method "EngineStatus_Get()" is moved to the local variable. As a result, all components in this sub-framework can access the local variable "EngineStatus(Local)", and synchronicity is guaranteed.

4.2.5 Development Process

In this section, we describe a development process to implement the proposed software architecture.

Controller Model Design and Function Generation

Controller models are designed with DSL tools for control [24]. The resulting controller models can also be simulated with DSL tools, enabling control engineers to check the feasibility of the control logic. After validating the control logic, they design a quantization of fixed-point numbers.

Next, software engineers generate software components automatically with code gen-

eration tools. The granularity of the component is the state variables of the controller system, such as input/output values, the system's observed state variables, and the system's target values. In Figure 4.1, "Target_Torque" and "Throttle_Opening" are the variables mapped to software components.

Figure 4.6 shows the Target Torque component as an example of an automatically generated function. A code generation tool automatically generates a calculation function from a block in the controller model. The calculation function is an C function implemented as "TargetTorque_Calculate( )", with the input values "Accelerator_Opening" and "Engine_Status", and it outputs a calculated value to "TargetTorque", which is assigned as a pointer. Calculation functions can also be generated automatically for the other blocks of the controller model.

Wrapper Generation

Next, a software engineer encapsulates a component from an automatically generated function with a wrapper. We have developed a tool for generating wrappers, called "Wrapper Maker", which generates a wrapper by analyzing a function.

Sub-Framework Generation

Basically, a sub-framework object activates software components by calling their update methods in turn, according to the data flow specified in the block diagram. A sub-framework also, however, guarantees synchronicity among the software components in the sub-framework. For a sub-framework to guarantee synchronicity for all variables, it must have a large amount of stack memory for local variables. The memory sizes of processors for embedded control systems are relatively small, so the number of local variables must be limited.

We have also developed a tool for generating sub-framework objects that only define

order for execution. Local variables requiring synchronicity are selected and added to the sub-framework by a software engineer. In the example of Figure 4.9, the software engineer adds the declaration of the local variable for “EngineStatus” and the fragment for copying the data “EngineStatus” into the local variable.

In industrial applications, strict data synchronicity is rarely required (e.g., state transition management). Therefore, our approach is applicable for most embedded control systems.

Software Reuse

Until now, we have explained how to develop software as a set of software components. We now also describe how software engineers reuse components.

A new product is often developed on the basis of an existing product, and only the addition and modification parts of the control algorithm are changed. In our approach, a software engineer can develop new control software by reusing the application framework and exchanging software components. This is because most changes are encapsulated into components, and sub-framework objects can be reused as is.

In the case of adding new controls and variables, a sub-framework object might have to be changed. That is, the sub-framework object might be modified to define the order in which to call a new software component. In this case, only the sub-framework for the modified control function and a wrapper for the added variable must be generated.

4.3 Case Study

4.3.1 Overview

In this case study, we apply the proposed system to certain engine controls and evaluated the rate of automatic code generation and the reusability. The target controls are esti-

mation of the quantity of intake air, wall flow compensation control, and torque-based control.

4.3.2 Results

We generated the implementation code from a controller model and evaluated the rate of automatic code generation. Only the sub-framework object required manual coding for data synchronicity. In this case study, we can automatic generate more than 96% of source code for all target control functions.

Furthermore, to evaluate reusability, we changed the control logic of the torque-based control, whose development was already complete, since the software components were developed at the unit of the variable, which is meaningful in a control with few additional changes. Modifications are mainly needed inside components, so they are encapsulated from the architecture viewpoint. As a result of changing the torque-based control logic, we needed to modify only a few software components, and changing the composition of the software components was unnecessary. As a result, the sub-framework object and most software components could be reused.

4.3.3 Discussion

We disscuss the validity of the case study. In this case study, we have applied the proposed method only to functionality of the engine control systems that are designed with the block diagram. Embedded control systems has numerous functionality that are designed by the block diagram. For example, more than half of functionality for the engine control systems are designed by the block diagram. So the proposed method is effective for many embedded control systems.

4.4 Summary

In this chapter, we have proposed a development method for software components in embedded control systems. The development method integrates object-oriented software development and model-based development.

We introduced a method to decompose a control function into a set of software components based on the state variables of the control system. Each component is automatically generated from a DSL tool for control logic. We have also developed code generation tools for wrapping an automatically generated function as a component and for generating a sub-framework.

We conducted a case study for automotive engine control systems. More than 96% of the control function source code was successfully generated with the proposed approach. Additionally, most of the developed control software could be reused for deriving new products.

The proposed approach improves the development efficiency of software components for a core asset. This also improves the reusability of the software components, for developing a new product based on the core asset.

CHAPTER 5

CROSSCUTTING FEATURE ANALYSIS

5.1 Introduction

Combinatorial explosion due to variable features is a serious problem for SPL in real applications. For example, the automotive industry reported that automotive control systems consist of thousands of variable features [31]. This combinatorial explosion has led to increasing configuration workloads. Verifying the dependency constraints among variable features is also a problem to be solved. Reducing the number of variable features is one of the simplest, most effective solutions.

A crosscutting feature is a high-level conceptual feature that affects (crosscuts) multiple variable features. Loesch et al. [23] and Conejero et al. [9] proposed a method for reducing the number of variable features by introducing the concept of crosscutting features. Figure 5.1 shows an example of a crosscutting feature for automotive systems. A *Distance radar* f_i and a *Braking control* f_j are defined as variable features. Since f_i and f_j are independent variable features, they are separately configurable, but for products A, B, C, D, and E, f_i and f_j are simultaneously either selected or not selected . This means that the variable features f_i and f_j are combined by introducing a crosscutting feature,

Product	Variable features	
	f_i (Distance radar)	f_j (Braking control)
A	X	X
B	-	-
C	X	X
D	X	X
E	-	-
Pattern 1	X	X
Pattern 2	-	-

X: Selected
-: Not selected

Crosscutting feature X_k (Adaptive cruise control)
X
-

Figure 5.1: Crosscutting feature

Adaptive cruise control X_k. As a result, the number of options is reduced, and then the productivity of product variations can be improved.

Figure 5.2 illustrates the scope of the SPL approach through adopting crosscutting features. Previous reuse of software was the reuse of implementation, e.g. software components and modules. Unfortunately, such reusability has not been fully effective, since there is a gap between requirement specifications and implementation specifications, with a lack of traceability. The SPL approach bridges this gap by introducing variable features that relate requirements and software components. A developer can then select software components by selecting variable features. Moreover, crosscutting features improve reusability by reducing the number of variable features for industrial product lines with thousands of variable features.

The above approach using crosscutting features cannot be applied to SPL adoption for legacy systems, because it analyzes the SPL infrastructure. Embedded control systems such as automotive control systems are safety critical. These systems require dependability, and developers are thus eager to reuse legacy systems with actual results. Therefore, for legacy systems, crosscutting features should be analyzed and extracted from the product release history.

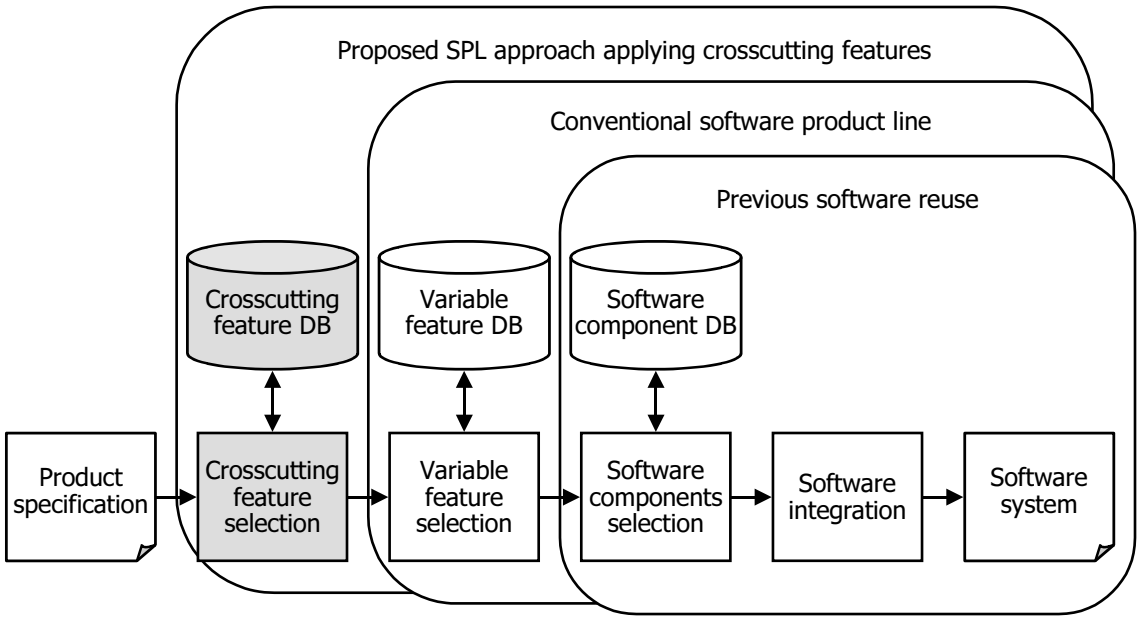

Figure 5.2: Scope of the proposed method

On the other hand, the concept of a logical coupling set has been applied for recovering software architectures [12] and guiding software changes [42] in the field of software maintenance. A logical coupling set is a set of development artifacts, such as files, functions, variables, and components, that are changed simultaneously during software development. These approaches are limited, however, to analyzing linear change histories and cannot be applied to software variations developed in parallel.

In this chapter, we introduce a novel analysis method for crosscutting features, which is based on the product release history. Figure 5.3 shows an overview of the proposed approach. First, (a) the product repository is preprocessed and change sets are extracted. Then, logical coupling sets are (b) extracted from the change sets and (c) evaluated as candidate crosscutting features. Crosscutting features are defined according to the logical coupling features, and finally, (e) the crosscutting features are reused as SPL core assets.

First, we present a preprocessing method for a product release history with a diverse range of variations. Related works have also proposed methods for preprocessing change

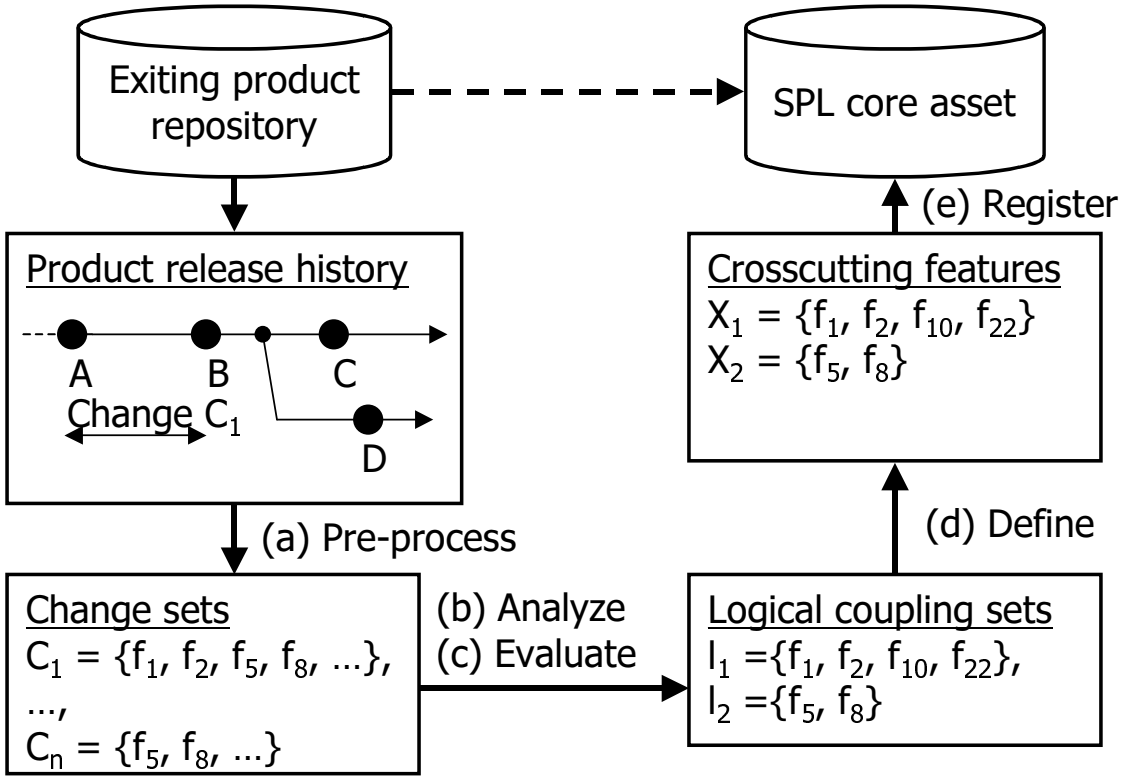

Figure 5.3: Extracting crosscutting features from product release history

histories by using repository data, but these approaches are of limited application for parallel development of product variations. In contrast, in the case of a branching product release history, our approach extracts the modifications between product release timings. This enables analysis on product domains developed in parallel.

Next, we describe threshold metrics for extracting candidate crosscutting features from a product release history. To analyze a large-scale repository of a product release history, such threshold metrics must be defined for automatically calculating and extracting candidates. The proposed metrics consist are the minimum frequency and the minimum confidence of co-change patterns. With these metrics, candidate crosscutting features can be extracted automatically.

Finally, we propose evaluation metrics for the extracted candidates. After the candidates are extracted using the proposed threshold metrics, they must be evaluated quantitatively from the viewpoint of SPL adoption. Therefore, we extend precision and recall to evaluate SPL. These are measures for evaluating the performance of information retrieval systems. Precision is defined as the number of relevant logical coupling sets divided by the total number of logical coupling sets extracted by the proposed method. Recall is defined as the number of extracted logical coupling sets divided by the total number of varied software components. The logical coupling sets can thus be evaluated quantitatively with the proposed evaluation metrics.

Through the proposed approach, candidate crosscutting features can be extracted automatically and quantitatively from an existing product repository. Note that the input of the proposed approach is limited to a product release history consisting of products that have evolved incrementally with the same software architecture. Linden et al. [38] mentioned, however, that the SPL approach is generally adopted for products in the mature stage of the product life cycle model. Therefore, the proposed approach is sufficiently useful even with this limitation.

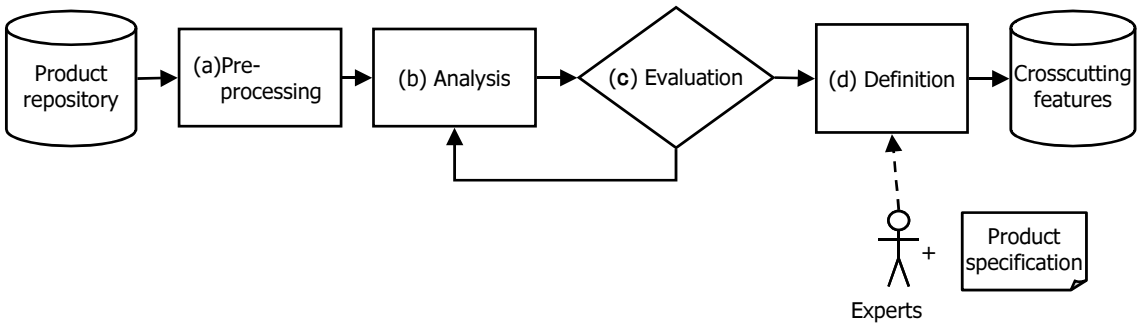

Figure 5.4: Overview of the proposed method

5.2 Crosscutting Feature Analysis

5.2.1 Concept of Proposed Method

In this section, we describe the details of the method for analyzing crosscutting features by extracting the logical coupling sets of the software components in the product release history. Figure 5.4 shows an overview of the proposed method.

First, in step (a), a product repository is preprocessed, and change sets are extracted. In our approach to extracting the crosscutting features, the variability between existing products is analyzed. Branching in software development is also considered, since branches occur frequently in parallel development of product variations. The input of the preprocessing is the version information of the software components composing the embedded control systems. The output is the sets of changes (modify, add, remove) of the software components between product releases. Each change set is divided into a learning set and an evaluation set.

Then, in step (b), logical coupling sets are extracted from the change sets for learning. We define two types of threshold metrics for extraction. The first is the minimum frequency of co-changes, while the second is the minimum confidence of co-changes between software components. The threshold metrics are varied gradually to extract the

logical coupling sets of crosscutting features.

The extracted logical coupling sets are evaluated next, in step (c), as the validity of the extracted candidates is evaluated. Here, we extend performance measures used in information retrieval systems, namely, precision and recall. Precision is defined as the number of relevant logical coupling sets divided by the total number of logical coupling sets extracted by the proposed method. Recall is defined as the number of extracted logical coupling sets divided by the total number of varied software components. The logical coupling sets can thus be evaluated quantitatively with the proposed evaluation metrics.

Finally, in step (d), the extracted candidates are defined as crosscutting features by domain experts. The experts relate software components consisting of a logical coupling set and a feature in the product line, by analyzing product specifications.

5.2.2 Product Release History

The product release history is used as the input of the proposed method. Figure 5.5 shows an example of a product release history. It consists of new product IDs, corresponding base product IDs, and the software components used for each new product. From the product release history, change sets of software components are determined from the differences between the base product and each new product.

Here, we introduce a notation for software components f_i and a set of software components F:

$$F = \{f_1, f_2, \cdots, f_n\} \tag{5.1}$$

A change set C_i is a set of software components changed in product release history i

Product	New	A	B	C	D	E	F	G
	Base	-	A	B	B	C	E	D
Software components	f1	1.0	1.2	1.3	2.0	1.3	1.3	2.1
	f2	1.0	1.1	1.1	1.1	1.2	1.2	1.1
	f3	1.0	1.1	1.1	1.2	1.2	1.3	2.0
	f4	1.0	1.0	-	-	-	-	-
	f5	1.0	1.2	1.2	1.3	1.3	1.3	2.0
	f6	-	-	1.1	-	1.1	1.1	1.1

Figure 5.5: Product release history

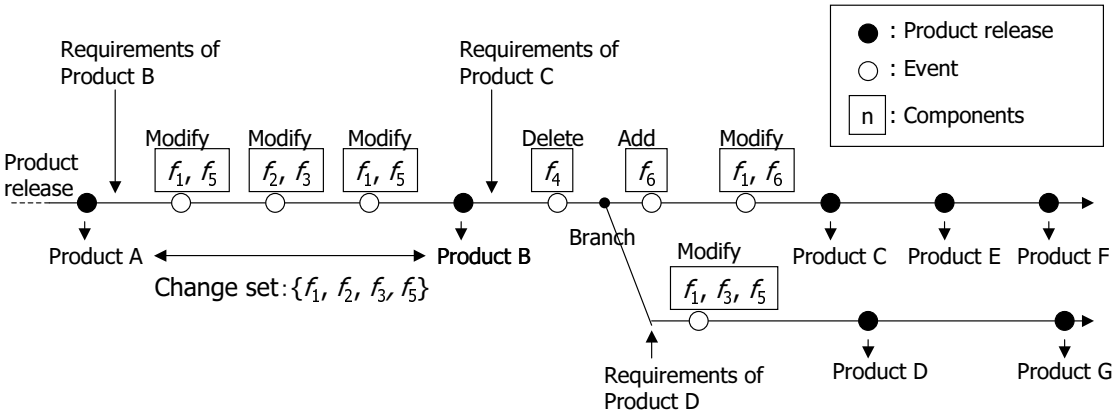

Figure 5.6: Product development flow

(i.e. for new product i), and it belongs to the power set of F (Power(F)):

$$C_i = \{f_j, f_k, \cdots, f_l\}, C_i \in \text{Power}(F) \tag{5.2}$$

In the example of Figure.5.5, the change set C_1 is as follows:

$$C_1 = \{f_1, f_2, f_3, f_5\}$$

Figure 5.6 shows an overview of the product development flow of parallel development for product variations. In general, a new product is developed through many modifications of multiple software components. For example, in the case of adding a new

software component, it must be updated more than once to fix bugs. Note that the goal of the proposed method is adoption of SPL, so we should concentrate on the differences between released products. Therefore, we deal equally with a single modification and multiple modifications.

In developing new products in parallel, several new products will have the same base product. This is known as a branch in software development. In Figure 5.6, when new product C was developed from product B, another new product D was branched. In this case, the change set is generated from the differences between the new product and the product released just before the branch.

The approach can be explained through the example shown in figure 5.6. We assume that SPL has not yet been introduced, so a variable feature of the product line is the functionality of each software component. When SPL has already been introduced, a variable feature can consist of several software components, in which case a change in a software component causes a change in the variable feature.

For the example of Figure 5.6, the change set C_3 between products B and D is the following:

$$C_3 = \{f_1, f_3, f_4, f_5\}$$

The following definition specifies the product line change set C, which belongs to power set of the power set of F:

$$\begin{aligned} C &= \{C_1, C_2, \cdots, C_n\}, \\ C &\in \text{Power(Power}(F)) \end{aligned} \tag{5.3}$$

The proposed approach applies the k-fold cross validation [18] to evaluate the logical coupling sets extracted from the change set. The change set is divided into k subsets.

One of these k subsets is used as an evaluation set, while the other $k-1$ subsets are used together as a learning set. The learning and evaluation processes are repeated k times, as every subset eventually becomes the evaluation set.

Here, we denote the evaluation set as C_E and put the other sets together to form the learning set, C_L:

$$C_L \subset C \tag{5.4}$$

$$C_E = C - C_L \tag{5.5}$$

We use the notations $C_{Li} \in C_L$ to indicate an element of C_L and $C_{Ej} \in C_E$ to indicate an element of C_E.

Let the product line change set C consist of the following:

$$\begin{aligned} C &= \{C_1, C_2, C_3, C_4, C_5, C_6\} \\ C_1 &= \{f_1, f_2, f_3, f_5\} \\ C_2 &= \{f_1, f_4, f_6\} \\ C_3 &= \{f_1, f_3, f_4, f_5\} \\ C_4 &= \{f_2, f_3, f_5\} \\ C_5 &= \{f_3\} \\ C_6 &= \{f_1, f_3, f_5, f_6\} \end{aligned}$$

Then, for the k-fold cross validation, let k equals 2 and define the learning change set C_L and the evaluation change set C_E as follows:

$$\begin{aligned} C_L &= \{C_{L1}, C_{L2}, C_{L3}\}, C_{L1} = C_1, C_{L2} = C_2, C_{L3} = C_3 \\ C_E &= \{C_{E1}, C_{E2}, C_{E3}\}, C_{E1} = C_4, C_{E2} = C_5, C_{E3} = C_6 \end{aligned}$$

5.2.3 Logical Coupling Sets

A logical coupling set is a set of development artifacts such as files, functions, variables, and components that are changed simultaneously during software development. We introduce the co-change frequency $freq$ and co-change confidence $conf$ as metrics for extracting logical coupling sets from the learning change set C_L.

The co-change frequency $freq$ is the number of occurrences of co-changes with respect to every software component in a set of components, x.:

$$x = \{f_j, f_k, \cdots, f_l\}, x \in \text{Power}(F) \tag{5.6}$$

$$\text{freq}(x) = |\{C_{Li} \mid C_{Li} \in C_L, x \subseteq C_{Li}\}| \tag{5.7}$$

The co-change confidence $conf$ is the number of occurrences of co-changes with respect to every software component in x, divided by the number of occurrences of changes with respect to the software components in the set:

$$\text{conf}(x) = \frac{\text{freq}(x)}{|\{C_{Li} \mid C_{Li} \in C_L, x \cap C_{Li} \neq \phi\}|} \tag{5.8}$$

We then introduce an extraction metric λ combining the co-change frequency $freq$ and the co-change confidence $conf$:

$$\lambda(x) = [\text{freq}(x), \text{conf}(x)] \tag{5.9}$$

Consider the following example set of software components:

$$x_{ex} = \{f_1, f_4\}$$

The components f_1 and f_4 were changed together twice in the learning change set C_2, C_3. Therefore, we have the following:

$$\begin{aligned} \text{freq}(x_{ex}) &= |\{C_2, C_3\}| \\ &= 2 \end{aligned}$$

In the learning change set C_1, C_2, C_3, at least one of the software components in x_{ex} was changed. Therefore, we have this:

$$\begin{aligned} \text{conf}(x_{ex}) &= \frac{\text{freq}(x_{ex})}{|\{C_1, C_2, C_3\}|} \\ &= \frac{2}{3} \\ &= 0.66 \end{aligned}$$

As a result, the extraction metrics λ for the set of software components is obtained as follows:

$$\lambda(x_{ex}) = [2, 0.66]$$

Threshold Metrics

Next, we introduce a threshold metric θ for extracting candidates for the logical coupling set. The minimum co-change frequency freq_{min} is the minimum co-change frequency among the candidates for the logical coupling set. Similarly, the minimum co-change confidence conf_{min} is the minimum co-change confidence among the candidates. The threshold metric θ is thus based on freq_{min} and conf_{min}, as follows:

$$\theta = [\text{freq}_{min}, \text{conf}_{min}] \tag{5.10}$$

The logical coupling set $L(\theta)$ is the set of software components x such that the extraction metricss λ is greater equal than the threshold metric θ:

$$\begin{aligned} L(\theta) &= \{x \mid \text{freq}(x) \geq \text{freq}_{min}, \text{conf}(x) \geq \text{conf}_{min}\} \\ &= \{l_1(\theta), l_2(\theta), \cdots, l_k(\theta)\}, \\ L(\theta) &\in \text{Power(Power}(F)) \end{aligned} \tag{5.11}$$

Consider the following example of the threshold metric θ_{ex} where the minimum co-change frequency freq_{min} is 2 and the minimum co-change confidence conf_{min} is 0.75:

$$\theta_{ex} = [2, 0.75]$$

First, the sets of software components $\{f_1, f_4\}, \{f_1, f_3, f_5\}, \{f_3, f_5\}$ are extracted, since their co-change frequencies freq are greater than or equal to the minimum co-change frequency:

$$\lambda(\{f_1, f_4\}) = [2, 0.66]$$

$$\lambda(\{f_3, f_5\}) = [2, 1.0]$$

$$\lambda(\{f_1, f_3, f_5\}) = [2, 0.66]$$

Then, the set $\{f_3, f_5\}$ is extracted, since its co-change confidence is greater than or equal to the minimum co-change confidence:

$$\begin{aligned} L(\theta_{ex}) &= \{x \mid \text{freq}(x) \geq 2, \text{conf}(x) \geq 0.75\} \\ &= \{l_1(\theta_{ex})\} \\ &= \{\{f_3, f_5\}\} \end{aligned}$$

Evaluation of Logical Coupling Set

Next, we introduce a metric to evaluate the logical coupling set $L(\theta)$ extracted from the learning change set C_L. The evaluation metric uses *precision* and *recall*.

The precision P is the probability that an extracted logical coupling set is in the evaluation change set. For a product release C_{Ej}, the precision P_j is defined formally as follows:

$$P_j(\theta) = \frac{|\{l_i(\theta) \in L(\theta) \mid C_{Ej} \in C_E, l_i(\theta) \subseteq C_{Ej}\}|}{|\{l_i(\theta) \in L(\theta) \mid C_{Ej} \in C_E, l_i(\theta) \cap C_{Ej} \neq \phi\}|} \tag{5.12}$$

The total precision of the evaluation change set C_E is defined as the average of the P_j:

$$P(\theta) = \frac{1}{|C_E|} \sum_j P_j(\theta) \tag{5.13}$$

The recall R is the probability that the evaluation change set is covered by an extracted logical coupling set. For product release C_{Ej}, the recall R_j is defined formally as follows:

$$R_j(\theta) = \frac{|\{l_i(\theta) \in L(\theta) \mid C_{Ej} \in C_E, l_i(\theta) \subseteq C_{Ej}\}|}{|C_{Ej}|} \tag{5.14}$$

The total recall R of the evaluation change set C_E is defined as the average of the R_j:

$$R(\theta) = \frac{1}{|C_E|} \sum_j R_j(\theta) \tag{5.15}$$

The harmonic mean H of the precision and recall is used to average of the evaluation metrics:

$$H(\theta) = \frac{2}{\frac{1}{P(\theta)} + \frac{1}{R(\theta)}} \tag{5.16}$$

The precision of the evaluation change set C_4 is denoted as P_4. In this case, if there is one logical coupling set ($\{f_3, f_5\}$), then P_4 is defined as follows:

$$\begin{aligned} P_4(\theta_{ex}) &= \frac{|\{f_3, f_5\}|}{|\{f_3, f_5\}|} \\ &= \frac{2}{2} \\ &= 1.0 \end{aligned}$$

$P(\theta_{ex})$ is then defined as follows, in terms of the arithmetic mean of C_4, C_5, C_6:

$$\begin{aligned} P(\theta_{ex}) &= \frac{1}{|C_E|} \sum_j P_j(\theta_{ex}) \\ &= \frac{P_4(\theta_{ex}) + P_5(\theta_{ex}) + P_6(\theta_{ex})}{|\{C_4, C_5, C_6\}|} \\ &= \frac{1.0 + 0 + 1.0}{3} \\ &= 0.66 \end{aligned}$$

For the change set C_4, the logical coupling set $L(\theta_{ex})$ has the following recall $R_4(\theta_{ex})$:

$$\begin{aligned} R_4(\theta_{ex}) &= \frac{|\{f_3, f_5\}|}{|\{f_2, f_3, f_5\}|} \\ &= \frac{2}{3} \\ &= 0.66 \end{aligned}$$

Then $R(\theta_{ex})$ is the mean of C_4, C_5, C_6:

$$\begin{aligned} R(\theta_{ex}) &= \frac{1}{|C_E|} \sum_j R_j(\theta_{ex}) \\ &= \frac{R_4(\theta_{ex}) + R_5(\theta_{ex}) + R_6(\theta_{ex})}{|\{C_4, C_5, C_6\}|} \\ &= \frac{0.66 + 0 + 0.5}{3} \\ &= 0.39 \end{aligned}$$

Finally, the harmonic mean H (θ_{ex}) is obtained as follows:

$$\begin{aligned} H(\theta_{ex}) &= \frac{2}{\frac{1}{P(\theta_{ex})} + \frac{1}{R(\theta_{ex})}} \\ &= 0.49 \end{aligned}$$

Sampling of threshold metrics

Different logical coupling sets are extracted by using different threshold metric values. The extraction process for the logical coupling sets is thus repeated and evaluated. One set of threshold metric values is then selected so as to obtain the best evaluation metric values for the extracted logical coupling sets.

Figure 5.7 shows an example of sampling the threshold metrics. In this example, the minimum co-change frequency is 2, 3, 4, or 5, and the minimum

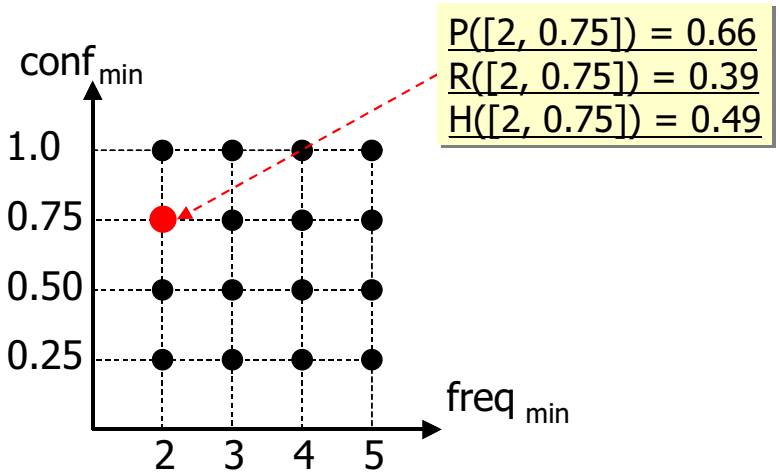

Figure 5.7: Sampling of threshold metrics

co-change confidence is 0.25, 0.50, 0.75, and 1.0. The extraction process for the logical coupling sets is repeated 16 times, with the evaluation metrics calculated each time. Next, the set of threshold metric values with the highest harmonic mean of the recall and precision is selected. Finally, the extracted logical coupling set with the selected threshold metric values $l_i(\theta_{max})$ is taken to define the crosscutting features X_i.

$$X_i = l_i(\theta_{max}) \tag{5.17}$$

Definition of Crosscutting Features

Finally, the extracted logical coupling set is used to define the crosscutting features. This step must be done by a domain expert who has knowledge of the product line from the requirement viewpoint.

The domain expert analyzes the specifications of the software components in the logical coupling set. The expert then assumes common functional and non-functional features of the components. An example is given in the next section.

Table 5.1: Case study example

Product line	Electronic Gasoline Injection sub-system
Number of products	37
Number of software components	63

Next, the expert confirms whether each supposition is true by checking the product specifications when the software components of the logical coupling set were changed. If the supposition is true, the supposed feature is defined as a crosscutting feature. If not, the expert analyzes the component specifications again.

5.3 Case Study

5.3.1 Overview

In this section, we evaluate the proposed method by examining its application to embedded control software for an engine control system.

Table5.1 gives an overview of the case study. We applied the proposed method to software with more than 500,000 lines of code. We selected one particular sub-system, the Electronic Gasoline Injection sub-system. This sub-system is modified frequently, since its specification must be modified to support variable features, such as the number of cylinders and the emission control regulations of markets. The sub-system consists of 63 software components. Each component can be related to a fine-grained variable feature, ranging from f_1 to f_{63}. The product release history consists of 37 individual products.

First, change sets are calculated from the product release history, obtaining a product line change set C with 36 change sets from the 37 product releases. Then, we apply leave-one-out cross validation to divide the change set C into a learning change set C_L

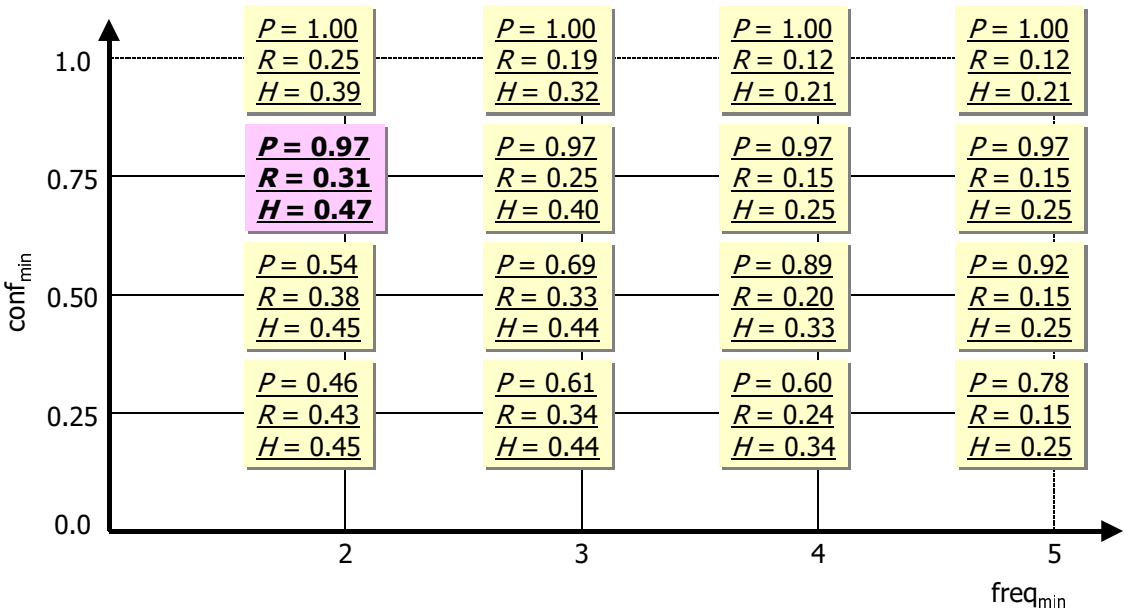

Figure 5.8: Evaluation metrics results

and an evaluation change set C_E. The leave-one-out cross validation is performed as k-fold cross validation with k equal to 1. Next, the logical coupling sets are extracted with the threshold metrics and evaluated with the evaluation metrics. This step is repeated for sampling of the threshold metrics. Finally, the evaluation metrics are calculated for each set of threshold metric values.

5.3.2 Results

The evaluation metrics are shown in Figure 5.8. For example, when the minimum co-change frequency is 2 and the minimum co-change confidence is 0.75, the precision is 0.97 and the recall is 0.31.

Next, the crosscutting features with the highest harmonic means of the precision and recall are selected. In this case study, the highest harmonic mean of 0.47 is obtained when the minimum co-change frequency is 2 and the minimum co-change confidence is 0.75. Finally, the crosscutting features are defined by domain experts. The results are

Table 5.2: Results of crosscutting feature definition

Crosscutting features	Variable features
X_1	f_{21}, f_{31}
X_2	f_{25}, f_{26}, f_{27}
X_3	f_{28}, f_{29}, f_{30}
X_4 (Fuel type)	f_{32} (Ignition timing), f_{39} (Fuel factor)
X_5	f_{50}, f_{51}
X_6	f_{52}, f_{53}

listed in 5.2. The crosscutting features are defined according to the variable features of the software components in the logical coupling sets.

For example, the crosscutting feature X_4 affects two variable features, *Ignition Timing* f_{32} and *Fuel factor* f_{39}. We can suppose that the crosscutting feature is the fuel type, since a difference in fuel type affects both the ignition timing (f_{32}) and the correction variable for the fuel (f_{39}). We can then confirm that the fuel type was changed in the product specification when the logical coupling set is the subset of the change sets of the product releases.

We thus define six crosscutting features $X_1, X_2, X_3, X_4, X_5, X_6$, representing fuel types and variable valve timing, for example.

5.3.3 Discussion

Since automotive engine control systems are developed in parallel, there are branches in their product development history. In this case study, the branches were preprocessed to form change sets by the proposed method.

The logical coupling sets were extracted automatically by this method with the threshold metrics and evaluation metrics. Therefore, the proposed approach can be applied to a

large-scale product release history.

We interviewed domain experts about the validity of the definition of the crosscutting features. They agreed that five of the six definitions are appropriate, while the other differs from their empirical knowledge. We examined this exception carefully and found that it should be recognized as a new crosscutting feature in the product line.

In this case study, we extracted six crosscutting features consisting of two or three variable features each. Since the sub-system consists of 63 software components, there are 1,953 possible combinations of two components and 39,711 possible combinations of three components. If domain experts were to analyze all 41,644 possible combinations, the workload would be too heavy for any real application. On the other hand, the proposed method successfully extracted 6 combinations from among the 41,644 possible combination. Therefore, the workload of the domain experts could be reduced significantly by applying this method.

We have applied the proposed method to the product release history consisting 37 products. If only a couple of products are released, crosscutting features can not be extracted by the proposed method. Defining the limitation of the proposed approach is a future topic.

CHAPTER 6

EVALUATION OF THE PROPOSED PROCESS

6.1 Introduction

In this chapter, we evaluate the proposed approach through simulation experiments. To evaluate the resulting improvements, we compare the costs of adopting SPL for legacy systems by the conventional approach and by the proposed approach. The total cost of adopting SPL and releasing several products is evaluated.

Figures 6.1 and 6.2 illustrate the SPL adoption process. In the conventional approach shown in Figure 6.1, the commonality and variability of the legacy systems are considered in terms of the specifications. If a product manager judges that SPL is applicable to the products, then the crosscutting features are analyzed to determine combinations of variable features. Then, the variable features and crosscutting features are implemented as software components through hand coding. Once the core asset base is established, software engineers develop new products by reusing the software components.

In the proposed approach shown in Figure 6.2, variability is analyzed in terms of the

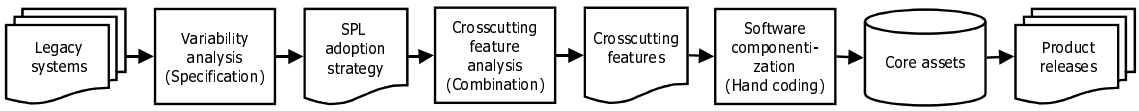

Figure 6.1: SPL adoption process (conventional)

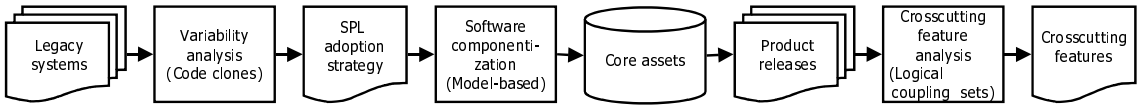

Figure 6.2: SPL adoption process (proposed)

code clone ratio of the implementation. If the product manager judges that SPL is applicable for the products, then the variable features are implemented through model-based development and automatic code generation. Once the core asset base is established, software engineers develop new products by reusing software components. After the products are released from the core asset base, the crosscutting features are extracted according to logical coupling sets.

6.2 Cost Model

6.2.1 Overview

We use the general cost model proposed by Bockle et al. [6], to compare the costs of introducing SPL with the conventional approach and the proposed approach. This cost model determines the general cost C of introducing SPL for n products p_i according to the following formula:

$$C = C_{org} + C_{cab} + \sum_{i=1}^{n}(C_{reuse}(p_i) + C_{unique}(p_i)) \tag{6.1}$$

C_{org} is the cost for the organization to adopt SPL, including the costs of reorganization, training, and so on. C_{org} is much the same for the conventional approach and the

proposed approach, so we eliminated this factor in the simulation experiments.

C_{cab} is the cost to develop the platform of a product line. It includes the costs of development activities, such as commonality and variability analysis, development of software components, and so on.

C_{reuse} is the cost of reusing core assets in the platform. This includes the costs of locating, checking out, binding variability, and configuring software components.

C_{unique} is the cost of developing unique pieces of software that are not based on the platform. This concerns the costs of developing product-specific software components and developing new software components that were not planned before introducing the SPL.

6.2.2 Conventional SPL Adoption Process

Figure 6.1 illustrates the SPL adoption process with the conventional approach. C_{cab} includes activities for analyzing variability and developing software components. In the conventional approach, the crosscutting features are included in C_{cab}, since they are analyzed upfront.

We define C_{cab} as follows:

$$C_{cab} = A_{var} \times N_{var} + A_{cross} \times N_{combination} + D_{new} \times N_{comp} \tag{6.2}$$

For analyzing variable features, A_{var} is the work effort for analysis and N_{var} is the number of variable features. For analyzing crosscutting features, A_{cross} is the work effort for analysis and $N_{combination}$ is the number of combinations of software components. For developing software components, D_{new} is the work effort for development and N_{comp} is the number of software components.

C_{reuse} is the cost of reusing software components. It is obtained as follows, where D_{reuse} is the work effort for locating and configuring a reusable software component, and

N_{reuse} is the number of software components for reuse:

$$C_{reuse} = D_{reuse} \times N_{reuse} \tag{6.3}$$

C_{unique} is the cost of developing software components for specific products and is obtained as follows, where N_{unique} is the number of new software components:

$$C_{unique} = D_{new} \times N_{unique} \tag{6.4}$$

Thus, the total work effort C_{conv} for the conventional approach to release $N_{product}$ products is the following:

$$\begin{aligned} C_{conv} &= C_{cab} + \sum_{i=1}^{n}(C_{reuse}(p_i) + C_{unique}(p_i)) \\ &= A_{var} \times N_{var} + A_{cross} \times N_{combination} + D_{new} \times N_{comp} + \\ &\quad (D_{reuse} \times N_{reuse} + D_{new} \times N_{unique}) \times N_{product} \end{aligned} \tag{6.5}$$

6.2.3 Proposed SPL Adoption Process

Figure 6.2 shows the SPL adoption process proposed in this dissertation. In the proposed approach, the crosscutting features are defined after $N_{product}$ products have been released from the platform.

Hence, we define C_{cab} as follows:

$$C_{cab} = A_{var} \times N_{var} + D_{new} \times N_{comp} \tag{6.6}$$

The definitions of C_{reuse} and C_{unique} are the same as for the conventional approach.

We also introduce C_{evo} for the cost of evolving a core asset. Specifically, C_{evo} is the cost of extracting crosscutting features from a product release history:

$$C_{evo} = A_{cross} \times N_{cross} + D_{new} \times N_{cross} \tag{6.7}$$

Thus, the total work effort $C_{proposed}$ for the proposed approach to releasing $N_{product}$ products is the following:

$$\begin{aligned} C_{proposed} &= C_{cab} + \sum_{i=1}^{n}(C_{reuse}(p_i) + C_{unique}(p_i)) + C_{evo} \\ &= A_{var} \times N_{var} + D_{new} \times N_{comp} \\ &\quad +(D_{reuse} \times N_{reuse} + D_{new} \times N_{unique}) \times N_{product} \\ &\quad +A_{cross} \times N_{cross} + D_{new} \times N_{cross} \end{aligned} \tag{6.8}$$

6.3 Experiments

6.3.1 Conditions

We conducted three sets of simulations to evaluate the conventional and proposed SPL adoption approaches. We also evaluated the conventional SPL adoption approach with the software componentization method (Model-Based Development: MBD) proposed in Chapter 4.

Table 6.1 lists the parameters of the simulation experiments. We defined $N_{product}$ and N_{cross} according to the case study results given in Chapter 5. N_{reuse}, N_{unique} were defined by applying expert knowledge of automotive engine management systems.

We normalized the work effort by defining the unit of effort for analyzing a variable feature with specifications as having a value of one. A domain expert estimated the work efforts A_{cross}, D_{new} and D_{reuse} through comparison with A_{var}.

Note that the esitimated work effort for the conventional approach is the worst case for analyzing the crosscutting features. We assume that $N_{combination}$ is the number of all possible combination for two or three software components.

Table 6.1: Simulation parameters

Parameter	Conventional	Conventional (MBD)	Proposed
N_{var}	5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60		
$N_{combination}$	${}_{N_{var}}C_2 +_{N_{var}} C_3$		-
N_{cross}	$0.1 \times N_{var}$		
N_{comp}	$N_{var} - N_{cross} \times 2.5$		N_{var}
N_{reuse}	$0.9 \times N_{comp}$		
N_{unique}	$0.1 \times N_{comp}$		
$N_{product}$	37		
A_{var}	1		0.5
A_{cross}	0.5		
D_{new}	40	8	
D_{reuse}	0.5		

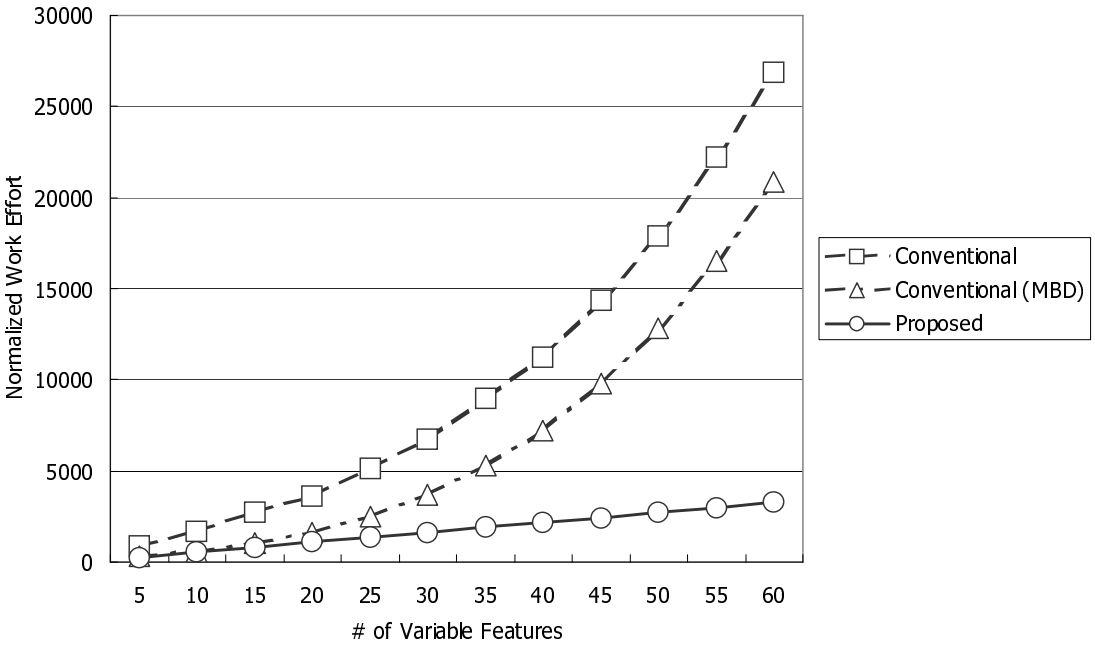

Figure 6.3: Experimental results

6.3.2 Results

Figure 6.3 shows the results of the simulation experiments. The horizontal axis represents the number of software components, and the vertical axis represents the normalized work effort.

When the number of variable features is small, the difference in total work effort between the conventional and proposed approaches is not significant. As the number of variable features increases, however, the work effort for the conventional approach increases exponentially. In contrast, the work effort for the proposed approach increases linearly.

Figure 6.4 shows the detailed distribution of work effort in the case of 30 variable features. With the conventional approach, the cost C_{cab} for analyzing the variable features and crosscutting features is a large part of the total work effort, since the number of possible combinations of variable features increases exponentially with the number of

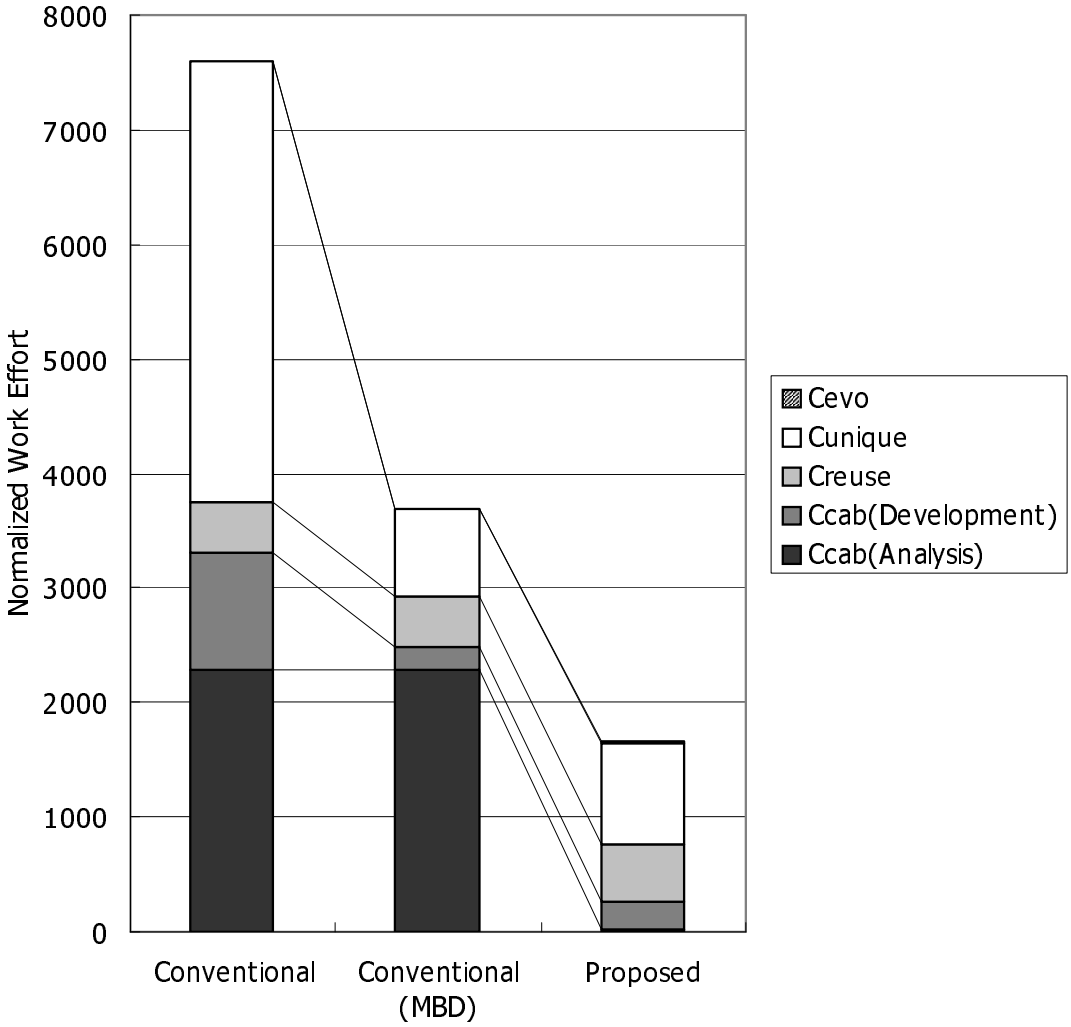

Figure 6.4: Work effort distribution (30 variable features)

variable features in the worst case. Moreover, the cost C_{unique} for developing product-specific or new software components is almost half the total work effort.

If we introduce the model-based approach with the conventional approach, we can improve C_{unique} and C_{cab} for developing software components. Even so, C_{cab} still has an enormous overhead for introducing crosscutting features upfront.

With the proposed approach, C_{cab} decreases significantly, since the crosscutting feature analysis is postponed until several products have been released. As compared with the conventional approach with MBD, C_{reuse} and C_{unique} increase, since the crosscutting concern has not yet been introduced. C_{evo} is also included for extracting the crosscutting features and refactoring them as new software components. For a product line with 30 variable features, the total work effort for the proposed method is estimated as almost a quarter of that for the conventional approach and half that for the conventional approach with MBD.

6.4 Summary

In this chapter, we have evaluated the conventional and proposed SPL adoption approaches through simulation experiments. We used a general cost model and estimated the cost of adopting SPL for legacy systems. The experiments demonstrated that the gap between the conventional and proposed approaches widens as the number of variable features increases. In general, legacy embedded control systems have large numbers of features. My proposed approach is thus effective for adopting SPL with large-scale legacy embedded control systems.

CHAPTER 7

CONCLUSION

7.1 Achievements

In this dissertation, I have addressed the problems of adopting SPL for legacy embedded control systems. First, I focused on the strategy for introducing SPL, and then I discussed software componentization and crosscutting feature analysis.

Regarding the strategy for introducing SPL, I first proposed a method to assess the commonality and variability of existing systems introduced into a software product line. In developing core assets from existing systems, analyzing and reusing existing implemented behavior are effective.

To assess commonality, I identify code clones between different systems. In assessing commonality and variability, I classify the clones into categories from the viewpoint of SPL variability. I also apply hierarchical decomposition assessment of systems. By using this method, we can assess commonality and variability between existing systems from the viewpoint of implementation. I also examined the proposed method through a case study on engine management systems for vehicles.

Next, I discussed a development method for reusable software components. This

method integrates object-oriented software development and automatic program generation.

The amount of embedded software is increasing dramatically, and there is an increasing demand for improving development efficiency. Object-oriented software development, which excels in the reuse of software components, has been gaining a great deal of attention. Recently, the quality and efficiency of automatic program generation from a controller model has reached production level. Therefore, the objective of the proposed development method is to establish a model-based development method for object-oriented embedded control systems adopting automatically generated software. The key feature of this method is that a wrapper wraps an automatically generated function that is handled as an object, and the wrapper is also automatically generated. I have developed software for such automatic wrapper generation, enabling an automatically generated function to be embedded efficiently. I also examined the proposed development method in terms of a control sub-system in an engine management systems.

Finally, I proposed a method to analyze crosscutting features in terms of logical coupling of product release histories, for migration into SPL. Crosscutting features help developers of large embedded systems to reduce the number of variable features. The times needed for analysis and quantitative evaluation, however, are problems to be solved.

The proposed method mines candidates for crosscutting features from a product release history, according to the logical coupling of software components. The method applies precision and recall as metrics and determines candidates quantitatively and automatically. I applied the proposed method to engine management systems and found that it successfully extracted candidates with 97% precision and 31% recall.

7.2 Future Research

In this dissertation, I have introduced an adoption strategy for SPL by analyzing inter-system code clones between two product implementations. The results of a case study showed that this approach was effective, but it is not applicable for many products. Further extension of the proposed approach is an important research topic.

I also discussed a model-based development approach for developing software components in SPL. Model-based development is of key importance for domain specialists to increase their productivity. I have not introduced this approach for an SPL-specific phase, however, such as variability decision. Integrating model-based development and SPL is another important research topic.

In this dissertation, I have focused on crosscutting feature analysis in terms of product release histories. To improve the development efficiency of SPL, traceability links between the variability of requirements and the variability of implementations are needed. By selecting a variant of a requirement variation point, a connecting variant of an implementation variation point could be automatically selected. From the results of preliminary experiments, however, I have found that most related works focus on a single development phase, i.e. the requirement or implementation phase. Therefore, analysis methods for traceability represent a third important future research topic.

BIBLIOGRAPHY

[1] J. L. Arciniegas, J. C. Duenas, J. L. Ruiz, R. Ceron, J. Bermejo, and M. Oltra. Architecture reasoning for supporting product line evolution: An example on security. In *Software Product Lines: Research Issues in Engineering and Management*, chapter 9, pp. 327–372. Springer Verlag, 2006.

[2] B. S. Baker. A program for identifying duplicated code. In *Proceedings of Computing Science and Statistics: 24th Symposium on the Interface*, pp. 49–57, 1992.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the Sixth International Software Metrics Symposium*, pp. 292–303, 1999.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, vol. 39, no. 10, pp. 104–116, 1996.

[5] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A methodology to develop software product line. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, pp. 122–131, 1999.

[6] G. Bockle, P. Clements, J. McGregor, D. Muthig, and K. Schmid. Calculating ROI for software product lines. *IEEE Software*, vol. 21, no. 3, pp. 23–31, 2004.

[7] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[8] P. Clements and L. M. Northrop. Sailon, Inc. - A software product line case study. *Technical Report CMU/SEI-2002-TR-038*, 2002.

[9] J. M. Conejero and J. Hernandez. Analysis of crosscutting features in software product lines. In *Proceeding of Early Aspects at ICSE: Aspect-Oriented Requirements Engineering and Architecture Design (EA 2008)*, pp. 3–10, 2008.

[10] H. Fennel, S. Bunzel, H. Heinecke, J. Bielefeld, S. Fuerst, K.-P. Schnelle, W. Grote, N. Maldener, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sanden, P. Heitkaemper, R. Rimkus, J. Leflour, A. Gilberg, U. Virnich, S. Voget, K. Nishikawa, K. Kajio, K. Lange, T. Scharnhorst, and B. Kunkel. Achievements and exploitation of the AUTOSAR development partnership. In *SAE Convergence 2006*, no. 2006-21-0019, 2006.

[11] M. Fuchs, D. Nazareth, D. Daniel, and B. Rumpe. BMW_ROOM - an object-oriented method for ASCET. In *1998 SAE International Congress & Exposition*, no. 981014, 1998.

[12] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*, pp. 190–198, 1998.

[13] W. Hermsen and K. J. Neumann. Application of the object-oriented modeling concept OMOS for signal conditioning of vehicle control units. *SAE technical paper series*, no. 2000-01-0717, 2000.

[14] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On software maintenance process improvement based on code clone analysis. *IPSJ Journal*, vol. 45, no. 5, pp. 1357–1366, 2004. (Japanese).

[15] R. E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, vol. 40, no. 10, pp. 39–42, 1997.

[16] K. C. Kang, M. Kim, J. Lee, and B. Kim. Feature-oriented re-engineering of legacy systems into product line assets - A case study. In *Proceedings of 9th International-Software Product Line Conference (SPLC 2005)*, pp. 45–56, 2005.

[17] J. Knodel, I. John, D. Ganesan, M. Pinzger, F. Usero, J. L. Arciniegas, and C. Riva. Asset recovery and their incorporation into product lines. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE '05)*, pp. 120–129, 2005.

[18] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, pp. 1137–1143, 1995.

[19] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 369–378, 2005.

[20] R. Koschke. Survey of Research on Software Clones. In *Proceedings of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software*, 2007.

[21] C. W. Krueger and D. Churchett. Eliciting abstractions from a software product line. In *Proceedings of International Workshop on Product Line Engineering The Early Steps (PLEES)*, pp. 43–48, 2002.

[22] J. Lee and D. Muthig. Feature-oriented variability management in product line engineering. *Communications of the ACM*, vol. 49, no. 12, pp. 55–59, 2006.

[23] F. Loesch and E. Ploedereder. Optimization of variability in software product lines. In *Proceedings of 11th International Software Product Line Conference (SPLC 2007)*, pp. 151–162, 2007.

[24] J. Moscinski. *Advanced Control with MATLAB and Simulink*. Ellis Horwood, Ltd., 1995.

[25] F. Narisawa, H. Naya, and T. Yokoyama. A code generator with application-oriented size optimization for object-oriented embedded control software. In *Object-Oriented Technology: ECOOP'98 Workshop Reader, LNCS-1543*, pp. 507–510. Springer, 1998.

[26] H. Naya, F. Narisawa, T. Yokoyama, K. Ohkawa, and M. Amano. Object-oriented development based on polymorphism patterns and optimization to reduce executable code size. In *Technology of Object-Oriented Languages and Systems - Tools-25*, November 1997.

[27] R. V. Ommering and J. Bosch. Widening the scope of software product lines - from variation to composition. In *Proceeding of the 2nd International Conference on Software Product Lines (SPLC2)*, pp. 328–347, 2002.

[28] OSEK-VDX. *OSEK-VDX Operating System Ver 2.2.1*, January 2002.

[29] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.

[30] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[31] M. Steger, C. Tischer, B. Boss, A. Mueller, O. Pertler, W. Stolz, and S. Ferber. Introducing PLA at Bosch Gasoline Systems : Experiences and practices. In *Proceedings of the Third Software Product Line Conference (SPLC 2004)*, pp. 34–50, 2004.

[32] Studio501. CloneFinder Webpage. http://www.studio501.com/products.htm (visited on June 2009).

[33] S. Thiel, S. Ferber, T. Fischer, A. Hein, and M. Schlick. A case study in applying a product line approach for car periphery supervision systems. In *SAE World Congress 2001: In-Vehicle Software Session*, no. 2001-01-0025, 2001.

[34] S. Thiel and A. Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, vol. 19, no. 4, pp. 66–72, 2002.

[35] T. Thomsen. Integration of international standards for production code generation. *SAE technical paper series*, no. 2003-01-0855, 2003.

[36] J. Utterback. *Mastering the Dynamics of Innovation*. Harvard Business School Press, 1994.

[37] F. van der Linden. Software product families in Europe: The Esaps & Cafe projects. *IEEE Software*, vol. 19, no. 4, pp. 41–49 , 2002.

[38] F. van der Linden, J. Bosch, E. Kamsties, K. Kansala, and H. Obbink. Software product family evaluation. In *Proceedings of the 5th International Workshop on Software Product Family Engineering (PFE 2003)*, pp. 352–369, 2004.

[39] D. M. Weiss and C. T. R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[40] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Similarity of software system and its measurement tool smmt. *Transactions of the IEICE(D-I)*, vol. 85, no. 6, pp. 503–511, 2002. (Japanese).

[41] K. Yoshimura, T. Miyazaki, T. Yokoyama, T. Irie, and S. Fujimoto. A development method for object-oriented automotive control software embedded with automatically generated program from controller models. In *2004 SAE World Congress*, no. 2004-01-0709, 2004.

[42] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.