ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20230216195620

Original scientific paper
Received: 2023-02-16, Accepted: 2023-03-20

# Variability Management Mechanism for Domain Engineering and Case Study in SunRoof Control Domain

Jeong Ah Kim

**Abstract:** This study aims to suggest variability mechanisms for software product line development and to explain the results of case study. Software product line engineering is an extension of software engineering and many organizations constantly engage in reengineering and refactoring to adopt the software product line engineering. Software product line engineering has two engineering processes: domain engineering process and application engineering process. Feature Identification and feature model are key success factor to construct variability model. Feature model describes the variable parts to be extended or replaced and common part to be reuse by themselves. Feature model gives the directions to the following architecture design and component implementation. However, feature model is not the design strategy and variability mechanism for product line implementation. Therefore, these are the obstacles for organizations that are unfamiliar with feature model and variability mechanism to adopt software product line engineering. Several variability mechanisms for software intensive software are suggested but these are not applicable for embedded software since it has different development process and structure. In this paper, variability elements in architecture design and component design of embedded software are identified as state variable, state transition information, and algorithm. Variability management mechanisms are defined for these elements. To provide the detail strategy and to evaluate the suggested variability management methods, process and results of real case study are described.

**Keywords:** domain engineering; embedded software product line; feature model, sunroof product line; variability management

## 1 INTRODUCTION

The software product line has been a software development paradigm that arose from an effort to maximize the quality attributes of software such as reusability and maintainability. The result of combining this paradigm with the existing field of software engineering is Software Product Line Engineering (SPLE). Software reuse, which began to gain attention in the 1990s, aims at reducing the effort required to develop or maintain a new system similar to a legacy system, and eventually reducing software development costs [1]. It has since departed from reuse at the level of the library or source code, originating from the reuse of subroutines in the 1960s, and the scope of reuse has been expanded to include domain knowledge, development experience, and design decisions. In line with the conceptual extension and expansion of software reuse, software re-engineering technology has advanced to create better software by analysing and reconstructing a legacy system, and to reduce the maintenance cost of the software. The gradually expanded reuse technology has recently evolved into a software product line engineering technology that takes into account both business and technical interests. Since the "product family engineering (PFE)" approach and address the commonalities and variabilities of software products with a focus on architecture in the 1980s [5, 16, 20]. The programs of an SPL are distinguished in terms of features, which are end-user visible characteristics of programs [3]. Based on a selection of features, stakeholders can derive customer specific programs that satisfy functional requirements [14, 19].

Software product line engineering has two engineering processes: domain engineering process and application engineering process [9]. Domain engineering process produces the core assets that will be assembled based on feature selection. Application engineering processes generate the product instance from core assets based on product requirements. For helping the organization to adopt the software product line engineering, a reference model is a good guide to define the process and strategies. In this research, domain implementation method and case study results of sunroof software product line for variability management in domain engineering are presented.

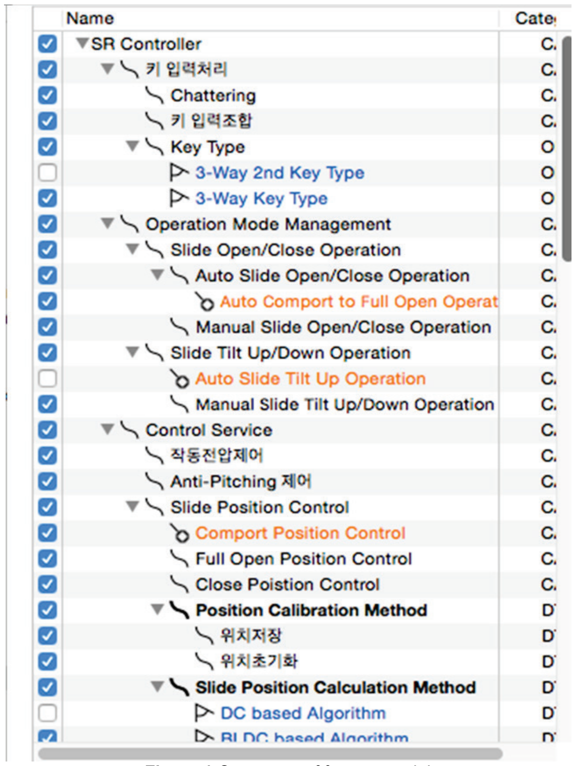
## 2 BUSINESS DOMAIN FOR SOFTWARE PRODUCT LINE ENGINEERING

This chapter introduces the company that has applied SPL as well as the background of such an application of SPL technology, and briefly introduces the domain to which SPL technology is applied. The company OOO Holdings is participating in this case study. It is a company that produces various parts used in the automotive body domain, and its representative product includes sunroof controller software. The adoption of SPL technology has been carried out by the Electrical System Department of the Future Technology Research Center.

These days, automobiles are evolving into electronic devices. As a result, the nature of the industry is changing from a hardware-oriented industry to a software-oriented industry. Automobile assembly OEMs are demanding various strict standards for software quality improvement from electrical system component developers in keeping with this trend [7, 6]. In terms of process, certifications such as CMMI [4] or A-SPICE [21] are required, and in terms of safety, the functional safety grades proposed by ISO-26262 [8] are required depending on the importance of parts.

This company hoped to shorten the development period of products for various customers and efficiently reuse the products that were developed by improving the engineering skills of developers through appropriate development methods. Domestic automobile parts development firms have only one or two developers, with a shortage of human resources. In order to overcome this, this company decided to adopt software product line engineering technology as a solution to increase the reusability and maintainability of software.

1. SR Controller [CA]
1.1. Ket Processing [CA]
1.1.1. Chattering [CA]
1.1.2. Key Combing [CA]
1.2. Operation Mode Management [CA]
(some more)
1.3. Control Service [CA]
1.3.1. Operating Power Control [CA]
1.3.2. Anti-Pinch [CA]
(some more)
1.3.6. Mode Management [CA]
    1.3.6.1. Wake Up Mode [CA]
    1.3.6.2. Sleep Mode [CA]
1.3.7. Motor Control [CA]
(some more)
1.4. Fail Safe [CA]
(some more)
1.4.5. Hall Sensor Diagnosis [CA]
    1.4.5.1. Hall Sensor Algorithm1 [CA][A]
    1.4.5.2. Hall Sensor Algorithm2 [CA][A]
(some more)
1.7. Parameter Configuration [CA]
1.8. Key Type [OE]
1.8.1. 3-Way Type [OE][A]
1.8.2. 3-Way 2nd Type [OE][A]
1.9. Blind Control [CA][O]
(some more)
1.14. EEPROM [OE]



| | Name | Cate |
|---|---|---|
| ✓ | SR Controller | C. |
| ✓ | 키 입력처리 | C. |
| ✓ | Chattering | C. |
| ✓ | 키 입력조합 | C. |
| ✓ | Key Type | O |
| ☐ | 3-Way 2nd Key Type | O |
| ✓ | 3-Way Key Type | O |
| ✓ | Operation Mode Management | C. |
| ✓ | Slide Open/Close Operation | C. |
| ✓ | Auto Slide Open/Close Operation | C. |
| ✓ | Auto Comport to Full Open Operat | C. |
| ✓ | Manual Slide Open/Close Operation | C. |
| ✓ | Slide Tilt Up/Down Operation | C. |
| ☐ | Auto Slide Tilt Up Operation | C. |
| ✓ | Manual Slide Tilt Up/Down Operation | C. |
| ✓ | Control Service | C. |
| ✓ | 작동전압제어 | C. |
| ✓ | Anti-Pitching 제어 | C. |
| ✓ | Slide Position Control | C. |
| ✓ | Comport Position Control | C. |
| ✓ | Full Open Position Control | C. |
| ✓ | Close Poistion Control | C. |
| ✓ | Position Calibration Method | D |
| ✓ | 위치저장 | D |
| ✓ | 위치초기화 | D |
| ✓ | Slide Position Calculation Method | D |
| ☐ | DC based Algorithm | D |
| ✓ | BLDC based Algorithm | D |

**Figure 1** Some part of feature model

The sunroof is one of the components constituting the body of an automobile. Although it is an optional part, it is gradually being applied to an increasing number of vehicles. Recently, not only general sunroofs but also panoramic sunroofs have been developed and used. As a device for user convenience, it is also a product that requires emotional functions such as low noise as well as safety.

## 3 RESEARCH METHODOLOGY

### 3.1 Domain Modelling of Sunroof Control SPL

SPL domain analysis activities consist of term definition, feature modelling, legacy system analysis, and domain requirement specification. Term definition and feature modelling are generally carried out in parallel. Legacy system analysis is an additional activity as an extractive approach is taken in this case study.

Feature model is the most important model for managing the variabilities in domain. Product lines for information systems present variabilities both in non-functional and functional features [5]. This task aims to clearly identify the commonalities and variabilities of the products included in the product line. Feature models are arguably one of the most intuitive and successful notations for modelling the features of a variant-rich software system. A feature is a characteristic or end-user-visible behaviour of a software system [18]. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle [17, 22].

Fig. 1 shows the partial feature model and feature diagram of Sunroof Domain. Eighty features are identified. Three optional features and nine alternative feature groups (20 features) are identified as variable features. FORM methodology defined the semantic of feature model [10, 11], Feature model explains the variability information of domain with feature and feature attributes. Feature attributes define the feature category with CA (Capability), OE (Operational Environment, and IT(Implementation Techniques). CA means that feature belongs to capability category such as functional requirements. OE means that feature belongs to Operating Environment category such as operation systems and network systems. IT means that feature belongs to implementation techniques such as processes and rules Also, feature attributes define the feature variability with M, A, O. Variability is modelled with M (Mandatory), A (alternative) and O (optional). M means that feature should be included in product instance. O means that feature can be included selectively based on feature selection at product configuration time. A mean that one of feature set should be included in product instance. The domain requirement specification is usually performed at the end of the domain analysis activities.

### 3.2 Variability Mechanism for Sunroof Control SPL

Domain implementation aims to make modules recognized through architectural design into reusable modules as much as possible. Making it into a reusable module means turning the module into a verified library, which can save 30 to 40% of the effort of testing the module when developing a product. In order to achieve this effect, it is necessary to find a way to quickly and easily reflect the modified contents in the module while avoiding the influence of other modules based on a thorough consideration of what variable information the module has to process when there is a change in the variable information. It is similar to a general software module design activity except for the task of finding variable information and applying it to the design. The domain implementation work is described for the activity of creating a document based on the source code [13].

## 3.3 Module Design for Processing State Variable Management Information

To separate control and operation, we need a special component for managing the control of state information. We have a "XYZ" module constituting the software is one of the modules that perform state management and plays a role in managing the operating state of the slide. This module has the variable attribute <<VOP>>. <<VOP>> stereotype is variability in implementation of function so that there are optional elements inside the module. The feature with the variable attribute <<VOP>> is as follows.

1.8.3. Slide Position Control [CA]

1.8.3.1. Comport Position Control [CA][O]

**Figure 2** Feature model with optional feature

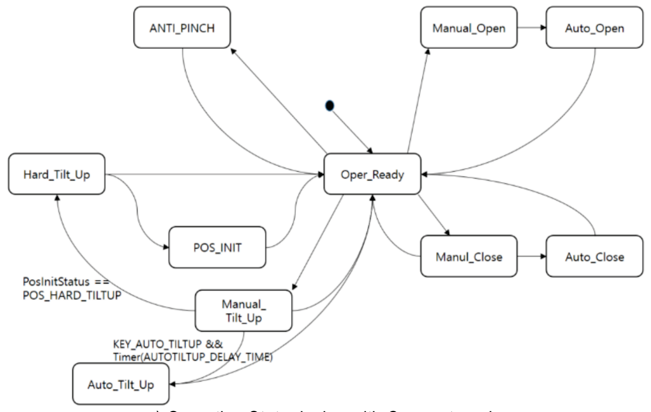
The control of the "Comport" position is optional depending on the slide controller. As a result, the action for managing the slide operation state for the operation command transmitted by the user input is affected. The results of modelling the state management behaviour with and without the Comport position control are as Fig. 3. At the request of the company, detailed items such as transition conditions are excluded.

The operation state behaviour model without the COM port mode is as shown in (b) of Fig. 3. There are a total of 10 states that comprise the state diagram. The modes are divided for each operation state in which the slide can be operated. The state returns to "Oper_Ready" when each operation is completed.

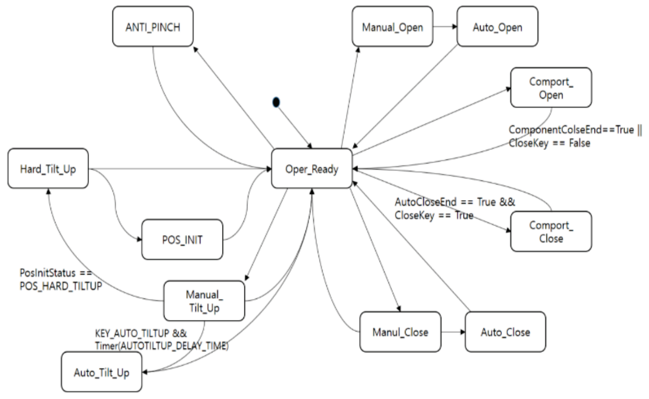
The operation state behaviour model with the COM port mode is as shown in (a) of Fig. 3. With the addition of the Comport mode, the "Comport_Open" and "Comport_Close" states have been added. The product line behaviour model created by combining the two operation state behaviour models is as follows.

The appearance of the product line driving state behaviour model is the same as the component mode driving state behaviour model as the behaviour model with the Comport mode includes the behaviour model without the COM port mode.

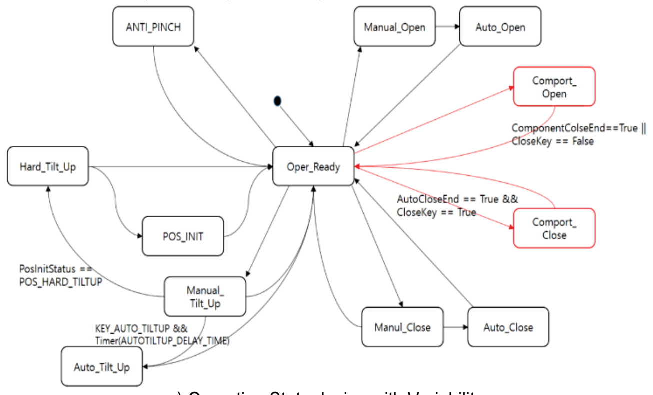
In the product line operational behaviour model, variable information is added to the "Comport_Open" and "Comport_Close" states. Variable information is indicated in the state by the stereotype as shown (c) of Fig. 3. The marked information, the "Comport Position Control" feature that determines whether to select the two states is written along with the meaning that the two states are optional states. The product line behaviour model of the module created in this way is used to create a behaviour model instance for a product by selecting a feature to develop a product in the subsequent application stage. In this case study, two instances are created.



a) Operating State design with Comport mode


ANTI_PINCH
Manual_Open
Auto_Open
Comport_ Open
ComponentColseEnd==True || CloseKey == False
Hard_Tilt_Up
Oper_Ready
AutoCloseEnd == True && CloseKey == True
Comport_ Close
POS_INIT
PosInitStatus == POS_HARD_TILTUP
Manual_ Tilt_Up
KEY_AUTO_TILTUP && Timer(AUTOTILTUP_DELAY_TIME)
Manul_Close
Auto_Close
Auto_Tilt_Up

b) Operating State design without Comport mode

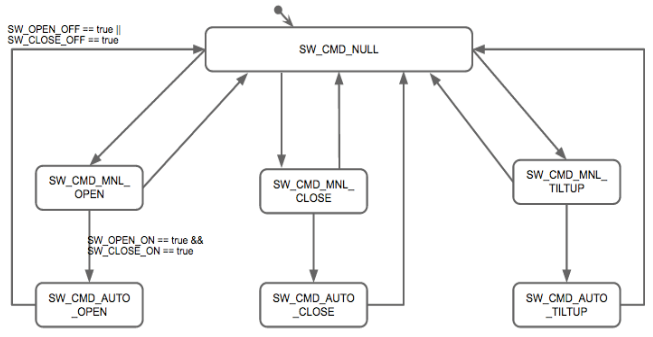

ANTI_PINCH
Manual_Open
Auto_Open
Comport_ Open
ComponentColseEnd==True || CloseKey == False
Hard_Tilt_Up
Oper_Ready
AutoCloseEnd == True && CloseKey == True
Comport_ Close
POS_INIT
PosInitStatus == POS_HARD_TILTUP
Manual_ Tilt_Up
KEY_AUTO_TILTUP && Timer(AUTOTILTUP_DELAY_TIME)
Manul_Close
Auto_Close
Auto_Tilt_Up

c) Operating State design with Variability

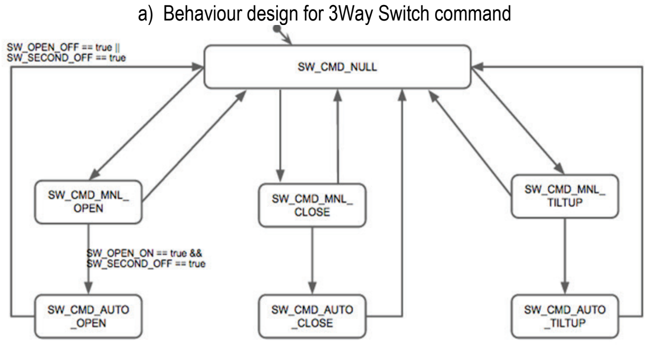
**Figure 3** State model with variability mechanism

## 3.4 Module Design for Processing Variable State Transition Information

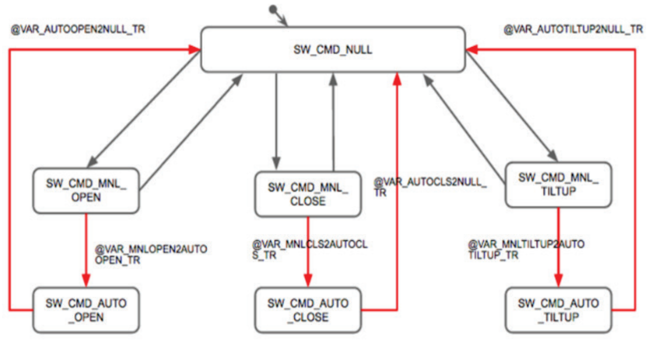
The case where the variable information becomes a state transition condition is similar to that of the variable state condition information discussed in the previous section. The difference is that the state itself is variable information in the former case, whereas the transition condition under the same condition is managed as variable information in this case.



a) Behaviour design for 3Way Switch command



b) Behaviour design for 3Way2nd Switch command



c) Behaviour design with variability

**Figure 4** State Behaviour model with variability mechanism

When the "Switch Command Composer" module has the variable attribute <<VAP>>. In other words, it has alternative variable elements inside. Now it is time to check what the variable elements are. This module plays the role of combining specific switch commands according to the type of input switch. As the domain model suggests, the slide product line supports two types of switches: "3 Way Switch" and "3 Way 2nd Switch." However, regardless of the switch type, the number of output commands is the same at eight (Manual Open, Manual Close, Manual Tilt Up, Auto Open, Auto Close, Auto Tilt Up, Switch Fault, None).

Based on this, the output command can be deemed as a state and the corresponding module can be designed using the state transition diagram. As a result of comparing the above two figures, the state remains the same as previously identified, but the conditions for transitioning the state are different. Unlike the behaviour model of the previous "XYZ" module, there is a variation point in the transition condition.

When a transition becomes a variation point, the method of adding as many transition lines as the number of variable elements to the state diagram is used in general. However, when using this method, the state diagram may look messy. Therefore, a method in which only the transition condition is described separately may also be used. In this case study, the second method is used to design the control behaviour for generating the switch command of the product line module as shown in the Fig. 4.

## 3.5 Module Design for Processing Variable Algorithm Information

This section briefly explains how to perform detailed design when an algorithm variation point is identified in the design target module.

For the module implementing the algorithm, the detailed design is divided into two parts.

- Declaration of interface function
- Algorithm function

The declaration of the interface function is a technology for easily selecting a module implementing a specific algorithm for each product to be produced and reflecting it in the product architecture. This part allows the processing of the variable information of the module that has the algorithm itself as a variable element.

The algorithm function is almost dependent on the inside of the module and operates as a variable element itself. Therefore, this part needs to be designed in the same way as designing general software modules. The flow can be described in detail or the algorithm can be described using Pseudo Code or Flow Chart.

## 3.6 Module Design for Processing Variable Operational Condition Information

It is common to define a layer composed of modules in charge of logic operations under the principle of separating control and operation in the module architecture. Since these modules generally have an algorithm for operation, unlike the previous state management type, it is not possible to design the behaviour of a module based on a certain specification. If the logic that performs the module function is divided into several branching statements by the same condition and the actions performed in each branching statement are similar, it is possible to find a variation point in them.

1.x.x. Flashing Control[CA]
    1.x.x.1. Flashing Request Priority [CA][A]
    1.x.x.2. Flashing Behaviour [CA][A]
  (Some more)

**Figure 5** Feature model with alternative feature

The "ABC" module constituting the software is responsible for performing a specific operation according to the state of the module performing the logic operation. This module has the variable attribute <<VAP>>. This means that there are alternative elements inside the module. The feature with the variable attribute <<VAP>> is as Fig. 5.

The above feature information shows that the priority of flashing requests and the type of command for error handling vary depending on the product. Because of checking how the previously developed system reflects the priority change factor in the system, the method of directly modifying the operation logic that generates the conditional statement and the instruction corresponding to the condition of the module has been used. That is, two variable factors have been identified.

- Command priorities
- Behaviour attributes for operation processing (number of times, waiting time, etc.)

If the source code is directly modified whenever there is a change in the two variable elements identified, the efficiency of development can be reduced through repeated modification of the module and tests for each product development. In this case, the most effective way to remove the variable information from the source code is to separate the priority information and the command information created according to the combination from the source code, allowing modification of such information from outside the source code. This method allows only the logic that reads and processes priority data, as well as the logic that determines the command to be generated in response to the key input based on the priority, to be implemented in the corresponding module. The "ABC" module designed by reflecting this concept is as Fig. 6.

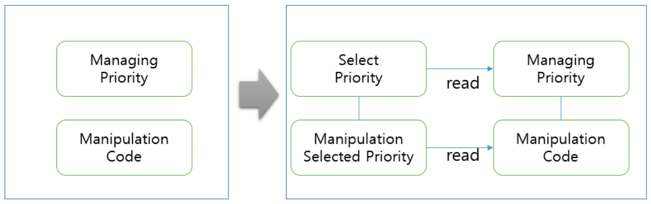


**Figure 6** Refactoring to separate command and behaviour for implementing a command

The "ABC" module has two sub-modules. The first "Flashing Priority Elector" selects the request with the highest priority among the errors to be inspected. Information about priorities is not included in the module. The second "Command Maker" module generates commands to fulfil the selected request. Command information that must also be created is managed separately from the module. Such a module design separates variable elements from the source code to allow the module to accommodate and process various combinations.

## 4 RESULTS AND DISCUSSION

Product line module implementation serves two purposes. First, it aims to create concrete implementation models based on the conceptual modules acquired from architecture design. Second, it aims to determine the technique for implementing the variable information reflected in the design. This section describes the implementation rules based on the C language used for automotive electrical system software development.

### 4.1 Definition of Implementation Rules for Logical Model

An object-oriented language such as Java provides various methods to implement the module defined in the architecture as an independent component at the language level. However, as for the C language, there is a limit to implementing the logical module defined in the architecture as an actual independent module at the language level. This is because calling it an actual module makes it impossible to forcibly block the access restrictions of the developed module. As a result, developers make arbitrary use, and architecture and consistency are gradually lost as development progresses. In order to prevent such a problem in advance, it is necessary to define the rules for developing modules in the C language.

Things to consider when writing the rules are as follows:

- What is the physical implementation unit of the module? Is it a folder? Is it a file? Is it a function?
- How should the layer of design be implemented in the architecture?
- How should the interface defined in the architecture design be implemented?

As there are more team members involved in software development, it becomes increasingly difficult for architectural design to be consistently implemented into codes in the absence of such rules. If some developers implement the module as a file while other developers implement the module as a function included in the file, module management becomes impossible from then on. In this case study, the following criteria are applied based on the module architecture for implementation.

**Table 1** Implementation strategy for architecture

| Architecture Element | Implementation Unit |
|---|---|
| Layer | folder/Package |
| Module | Module file (.c) |
| Module Interface | Header file (.h) for each module |
| Internal Module in | Function |

The implementation based on the above criteria leads to the development of the following code structure. There should be as many C files as the number of modules defined in the module architecture in the package.

### 4.2 Determination of the Mechanism of Variable Element Implementation Inside of Module

How to implement the variable elements inside the module described in the product line module design should be determined? The most easily used variable information processing mechanism in the C language is to utilize the macros and the pre-processor provided by the C compiler. A macro is the easiest way for organizations or developers to deal with variable elements for the first time. However, with an increase in the number of features, the source code becomes messy, reducing readability. Accordingly, a more advanced technique should be applied thereafter.

The mechanism applied to implement the variable information applied in this case study is as follows.

In this case study, three implementation mechanisms were mainly used: macro, declaration of data, and external HMI program.

**Table 2** Implementation mechanism for variability type

| Variability Type | Variable unit | Implementation Mechanism |
|---|---|---|
| Control State | Behaviour Model | Macro Macro definition for feature |
| Transition Condition | Behaviour Model | Macro<br>Macro definition for feature |
| Operational Condition | Priority in Key combination<br>Command based on key combination | Text script<br>Data Definition |
| Parameter | Parameter value | External HMI Program development |
| Algorithm | Implementation logic | Function<br>Macro |

An example of processing variable information using the macro pre-processor provided by the C language can be found in the Switch Command Composer module. The detailed design of the module shows that all variable elements are included in the module. Each variable element has a structure in which one variable element is selected according to feature selection. When implementing a module with this structure, macros are used in general.

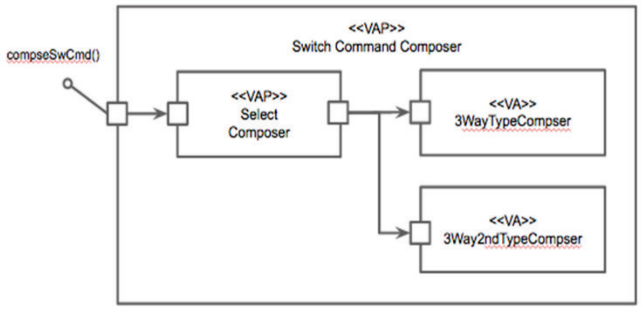


**Figure 7** Variability management in module inside

The source code implementing the above design is as follows. In the _selectComposer() function, the function to be called is determined based on the feature selection.

When information related to operating conditions is completely removed from the source code, variable information can be changed using a simple text-based script. The written text script can be used in various ways depending on the development environment. In this case study, the method of generating the corresponding data based on the script file is selected due to the limitation in the performance of the hardware. The sample code below is generated based on a text script that specifies the priority of commands. There are 11 predetermined commands, and the priority of each command and the corresponding behaviour attributes are specified and initialized as variables using the data structure

In this case study, the variability of the parameter value is handled by developing an HMI program that sets the parameters using the LIN network before operating the product. The Fig. 9 shows the developed HMI program.

```
/**************************
*       File name   : SwitchCommandComposer.c
*       Purpose     : Demonstration of a simple program.
*
*******************/

#ifndef      __SWITCHCOMMANDCOMPOSER_C__
#define      __SWITCHCOMMANDCOMPOSER_C__

#include "SwitchCommandComposer.h"

void compoSwCmd(void) {
        _selectComposer();
}

void _selecComposer(){
#ifdef FEATURE_3WAY_TYPE
        _3WayTypeComposer();
#elif FEATURE_3WAY2ND_TYPE
        _3Way2ndTypeComposer();
#endif
}
```

```
void _3WayTypeComposer(void){

switch (enuSwCmd)      {
       case SW_CMD_NULL :
        if(COND_SW_OPEN_ON)
               enuSwCmd = SW_CMD_MNL_OPEN;
        else if(COND_SW_CLOSE_ON)
               enuSwCmd = SW_CMD_MNL_CLOSE;
        else if(COND_SW_TILTUP_ON)
               enuSwCmd = SW_CMD_MNL_TILTUP;
        else
               enuSwCmd = SW_CMD_NULL;
        break;
        (...)
        default :      break;
        }
}
void _3Way2ndTypeComposer(void) {
     switch (enuSwCmd) {
           case SW_CMD_NULL :
        if(COND_SW_OPEN_ON)            {
          enuSwCmd = SW_CMD_MNL_OPEN;
        else if(COND_SW_CLOSE_ON)
          enuSwCmd = SW_CMD_MNL_CLOSE;
        else if(COND_SW_TILTUP_ON)   enuSwCmd = SW_CMD_MNL_TILTUP;
        else
          enuSwCmd = SW_CMD_NULL;
        break;
        (...)
        default :              break;
        }              // end of switch(enuSwCmd)
}
```

**Figure 8** Example of variability implementation (determination of function call based on feature selection)

## 4.3 Creation of Design Documents using Source Code Comments

A case of application to simplifying detailed design using comments when implementing source code is presented. For electrical system software developers, this may be a strategy that can be used in situations where it is not possible to spend a lot of time documenting the design content.

As of now, tools that automatically generate documentation based on source code comments include JavaDoc and Doxygen. Since JavaDoc is dependent on the Java language, Doxygen is used when development is based on the C language. As Doxygen provides various comment formats and commands, it has the advantage of automatically obtaining various types of documents. Doxygen, however, cannot contain all the details of the design, and it should be used for the purpose of assisting the simplification of the design.

```
/**********************/
/*     Macro */
/************************/
#define PRIORITY_MAX_NO 11

#define COMMAND1      0x00
#define COMMAND2      0x01
#define COMMAND3      0x02
#define COMMAND4      0x04
#define COMMAND5      0x08
#define COMMAND6      0x10
#define COMMAND7      0x20
#define COMMAND8      0x40
#define COMMAND9      0x80
#define COMMAND10    0x100
#define COMMAND11    0x200

#ifdef VARIANT_TABLE
```

```
/*********************************************/
/*     Priority Logic variant Table */
/*****************************************/

typedef struct {
unsigned char Priority;
unsigned char Mode;
unsigned char Work_Count;
unsigned char On_Time;
unsigned char Off_Time;
unsigned char failcheck;
unsigned char Stop_Control;
} fault_Handling_Table

const fault_Handling_Table priority_TBL[PRIORITY_MAX_NO] = {
/* Priority, Mode, Work_Count, On_Time, Off_Time, Fail_check,
Stop_Control */
{ 0, 0, 0, 0, 0, 0, 0},
…
};
#endif
```

**Figure 9** Example of variability implementation (determination of function call based on feature selection)

An example of writing comments on source code using the comment command of Doxygen is shown Fig. 10. Comments can be added to the beginning of files, functions, and variables. Below is an example that was written at the beginning of the file.

The command supported by Doxygen starts with @ to write a brief introduction, date of creation, version, rights, etc. Additional information can be included by using the @note and @par commands. In this case study, additional information such as @par Feature:Fault Priority is added to indicate that the module implements a specific feature.

In the comments applied to the function, the @note command can be used to specify whether the function is an internal module function or an interface function, in addition to the basic description of the function. Although these comments are not enforceable, they can be used to make a promise between the developers that the module will be used through the interface by providing such information to the developer. The following function is the interface function provided by the ABC module. In order for another module to interact with this module, this function must be called. When this function is called, the function that internally determines the priorities and the internal function that executes the selected command are called in turn.

```
/**
*****************
* @brief   ABC Module.
* @details        xxx Software
*         Product Line Project.
* @author        ooo
* @version 1.0.0
* @file     abc.c
* @copyright    Copyright by ooo.
* @warning Target is unknown.
*         Complier is unknown
* @note
* @par
* Project: xxx Software Product Line Project.
* @par
* Target: Unknown.
* @par
* Complier: Unknown.
* @par
* Features: Fault Priority
************************/
```

**Figure 10** Example of Variability management of document based on feature section

## 4.4 Discussions

To evaluate suggested design method, two products are selected among the products defined in the product line to introduce the virtual process of production activity and the implications.

The task of selecting a feature for product development from the feature model is performed. The criterion for selecting a feature is the customer's requirements. The requirements of the customer need to be analysed to determine which features satisfy the requirements. In reality, product requirement analysis and feature selection are performed almost simultaneously. Examples of possible product configurations based on the variable elements examined in the product line module development are as Tab. 3.

**Table 3** Association between feature and product instance

| Feature | Product A | Product B |
|---|---|---|
| Comport Position Control | O | X |
| Switch Type | 3 Way Switch | 3 Way 2nd Switch |
| Fault Priority | Priority Script 1 | Priority Script 2 |
| Parameter | Product A Parameter value | Product B Parameter value |
| Slide Position Calculation Method | XX Device based Algorithm | YY Device based Algorithm |

The product architecture instance development process receives the product feature list, product requirements, and product line architecture created in the previous tasks as inputs, and proceeds in the following order:

- Creating product architecture instances using the product feature list
- Modifying the product architecture considering new product requirements
- Only the results of the first task are presented in this study.

When creating a product architecture based on product features and product line architecture, it is necessary to ensure that selectable modules are present in the product line architecture. In other words, it is necessary to check whether a module with a variable attribute <<VO>> is included in the design. As for such a module with a variable attribute, the module itself can be excluded from the product architecture according to the feature selection. After removing the unselected module, all the variable attribute information in the product line architecture is removed.

The Fig. 11 shows the difference between the product module architecture derived from product line module architecture. There was no module with a variable attribute <<VO>> in the architecture of the slide control software. This means that the product architecture is consistent with the product line architecture.

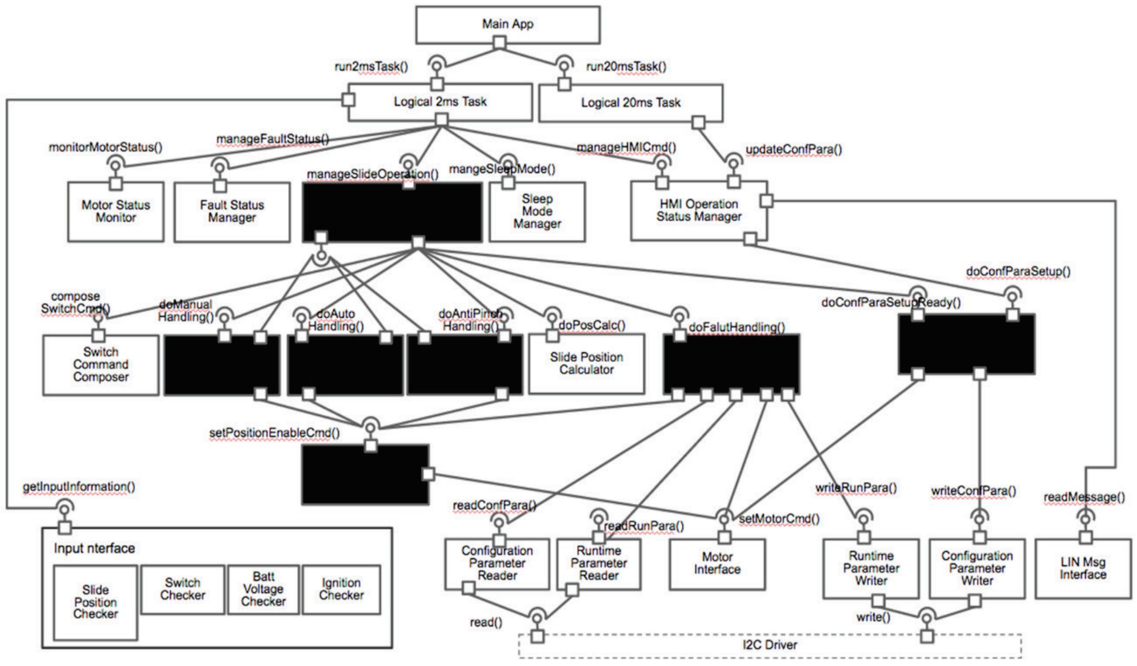


**Figure 11** Product module architecture derived from product line module architecture

## 5 CONCLUSIONS

In this paper, domain implementation methods based variability management techniques are suggested to make the component to be more reusable based on feature selection at product instantiation phase. In SPL, a product based on selected features can be generated or composed using the components developed in domain implementation [12-16].

Software product line engineering is not a treasure trove over the rainbows. This technology is a development paradigm that has emerged to improve the reusability and maintainability of software and can be embodied in various forms. The implementation may be a software platform, or it may be software developed in a component-based method. Software product line engineering has been considered as powerful methodology for platform development and reusable architecture construction. However, construction of software product line requires high qualified engineers and large investment in time and cost. As other software development methodology, guides for adopting the software product line engineering and techniques for variability management are the critical. Many software product line have been constructed in software intensive domain, therefore, there are many guidelines and variability management techniques in software intensive domain. However, very few case studies are available for embedded software domain so that small-sized companies and novice engineers at software product line engineering are not easy to manage the variability.  In this paper, design methods for variability management in embedded software were suggested and a case study of automotive sunroof control software product line is described as domain implementation guides. State, state transition, algorithm are the main variable parts in embedded software domain. Data structure and interface are also variable parts but these are already defined in other domains. Suggested mechanisms contribute to help the embedded engineers to identify the variable parts. Also, these helps to decide the implementation strategies and patterns of software modules to be more extensible including variable parts.

**Acknowledgments**

This work is supported by the Korea Agency for Infrastructure Technology Advancement (KAIA) grant funded by the Ministry of Land, Infrastructure and Transport (Grant 21RSCD-C163348-01).

## 6 REFERENCES

[1] Apel, S., Kästner, D. B. C., & Saake, G. (2013). *Feature-oriented software product lines Sven Apel Don Batory Christian Kästner Gunter Saake concepts and implementation*. Springer. https://doi.org/10.1007/978-3-642-37521-7

[2] Berger, T., Nair, D., Rublack, R., Atlee, J. M., Czarnecki, K., & Węsowski, A. (2014). Three cases of feature-based variability modeling in industry. In: *International conference on model driven engineering languages and systems, MODELS*, 302-319. https://doi.org/10.1007/978-3-319-11653-2_19

[3] Braga, S. T. V., Rodrigues Germano, F. S., Pacios, S. F. & Masiero, P. (2007). AIPLE-IS: an approach to develop product lines for information systems using aspects. *Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2007)*, 17-30.

[4] Chrissis, M. B., Konrad, M. D., & Shrum, S. (2011). *CMMI for development: guidelines for process integration and product improvement*, Third Edition.

[5] Clements, P. & Northrop, L. (2001). *Software product lines: Practices and patterns, reading*. USA: Addison-Wesley

[6] Gai, P. & Violante, M. (2016). Automotive embedded software architecture in the multi-core age. *The 21st IEEE European Test Symposium (ES)*. https://doi.org/10.1109/ETS.2016.7519309

[7] Ignaim, K. & Fernandes, J. M. (2019). An industrial case study for adopting software product lines in automotive industry: An evolution-based approach for software product lines (EVOA-SPL). *SPLC'19: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, 83-190. https://doi.org/10.1145/3307630.3342409

[8] ISO 26262-1:2018 - Road vehicles — Functional safety.

[9] ISO/IEC 26550:2013 - Software and systems engineering — Reference model for product line engineering.

[10] Kang, K., Kim, S., Lee, J., Kim, K., Shin E., & Huh, M. (1998). FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5, 143-168. https://doi.org/10.1023/A:1018980625587

[11] Lee, K., Kang, K. C., & Lee, J. (2002). Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: Gacek, C. (eds) *Software Reuse: Methods, Techniques, and Tools, ICSR 2002*. *Lecture Notes in Computer Science*, vol 2319. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-46020-9_5

[12] Lindohf, R., Kruger, J., Herzog, E., & Berger, T. (2021). Software product-line evaluation in the large. *Empirical Software Engineering*, 26, 30. https://doi.org/10.1007/s10664-020-09913-9

[13] Machado, L., Pereira, J., Garcia, L., & Figueiredo, E. (2014) SPLConfig: Product configuration in software product line. *Brazilian Congress on Software (CBSoft), Tools Session*, Maceio, 85-92.

[14] Metzger, A. & Pohl, K. (2014). Software product line engineering and variability management: Achievements and challenges. *Proceedings of the Future of Software Engineering*, 70-84. https://doi.org/10.1145/2593882.2593888

[15] Nešić, D., Krüger, J., Stănciulescu, Ş., & Berger, T. (2019). Principles of feature modeling. In: *Joint meeting on European software engineering conference and symposium on the foundations of software engineering, ACM, ESEC/FSE*, 62-73. https://doi.org/10.1145/3338906.3338974

[16] Pohl, K., Böckle, G., & van Der Linden, F. J. (2005). *Software product line engineering: Foundations, principles and techniques*. Springer. https://doi.org/10.1007/3-540-28901-1

[17] Ravichandran, T. & Rothenberger, M. A. (2003). Software reuse strategies and component markets. *Communications of the ACM, 46*(8), 109-114. https://doi.org/10.1145/859670.859678

[18] Sainzaya, D., Shin, S.-H., Lee, S. (2018). A MAM net Model of Software Product Line for Project Management. *International Journal of Control and Automation, NADIA, 11*(2), 11-20. https://doi.org/10.14257/ijca.2018.11.2.02

[19] Siegmund, N. (2011). SPL conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal, 20*(3-4). https://doi.org/10.1007/s11219-011-9152-9

[20] van der Linden, F., Schmid, K., & Rommes, E. (2007). *Software product lines in action*. Berlin, Heidelberg, New York: Springer. https://doi.org/10.1007/978-3-540-71437-8

[21] Verband der Automobilindustrie e. V. (VDA) (2017) VDA Automotive SPICE® Guidelines

[22] Zhang, H. & Wang, F. (2016). Timed Model Checking Service-Oriented Product Lines. *International Journal of u- and e-Service, Science and Technology, NADIA, 9*(7), 335-348. https://doi.org/10.14257/ijunesst.2016.9.7.34

**Author's contacts:**

**Jeong Ah Kim**, Professor (Ms)
Computer Education Department, Catholic Kwandong University,
579 BeonGil 24, GangNeung,
Kangwon Province, Gangwon-do, 25601, South Korea
clara@cku.ac.kr