Technical Report of AGSE June 2013

Variability Erosion and Improvement: from Conditional Compilation to Parameterized Inclusion

Bo Zhang
Software Engineering Research Group
University of Kaiserslautern
Kaiserslautern, Germany
bo.zhang@cs.uni-kl.de

***Abstract*—Conditional Compilation (CC) is frequently used as a variation mechanism in software product lines (SPLs). However, as a SPL evolves the variable code realized by CC erodes in the sense that it becomes overly complex and difficult to understand and maintain. As a result, the SPL productivity goes down and puts expected advantages more and more at risk. To investigate the variability erosion and keep the productivity above a sufficiently good level, in this paper we 1) investigate several erosion symptoms in an industrial SPL; 2) present a variability improvement process that includes two major improvement strategies. While one strategy is to optimize variable code within the scope of CC, the other strategy is to transition CC to a new variation mechanism called Parameterized Inclusion. Both of these two improvement strategies can be conducted automatically, and the result of CC optimization is provided. Related issues such as applicability and cost of the improvement are also discussed.**

Keywords—variability erosion; product line improvement; code refactoring; conditional compilation

I. INTRODUCTION

In software product line (SPL) engineering, similar products are created in a cost-efficient way so that their common and varying characteristics are fully utilized. Ideally, a SPL should contain a majority of commonality, which is shared and reused all products, and a minority of variability, that is currently required by existing products. However, trying to maximize the commonality tends to fail in evolution. When being exposed to a number of products, application engineers tend to invent new combinations of features, new improvements to existing features, and completely new features that are difficult to predict during domain engineering. This drives the increase of variability in SPL evolution to allow matching the changing development requests.

On the downside, the newly added variability makes the variability realization in the product line more complex, and the increased complexity tends to reduce the overall SPL productivity. If no corrective actions are done, the SPL becomes less competitive and new products will eventually start to use solutions beyond the product line core or efforts to create a new product line will be started. Based on our practical experience, it is often the better choice to evolve an existing asset base than try to create a new product line, especially in the light of long-lived products. However, there is a lack of methods and tools to support efficient SPL evolution in current industrial practice, e.g. in analyzing existing asset bases to detect erosion problems and planning and executing necessary improvement strategies. Also a lot of research results focus on engineering new product lines rather than supporting evolving existing product lines to target new requirements.

Once SPL evolution is not conducted appropriately, variability realizations will gradually erode over time. The concept of erosion was introduced by Perry and Wolf in [17] to indicate software architecture problems. In our context of SPL evolution, erosion means that realization artifacts become overly complex due to unforeseeable changes (also known as software aging [15]). Considering the frequently used variation mechanism of Conditional Compilation (CC), eroded variability realizations comprise nested, tangled, and scattered #IFDEF blocks with complex interdependencies [11]. They lead to practical problems in evolution as it becomes more difficult to assess change impact on core assets and to adapt the change with other related VEs [14] in a consistent manner. This increases the risk for missing, obsolete or incorrect variability realizations in core assets. Besides change propagation, the effort of Quality Assurance (QA) after change is also non-trivial, especially when the products fall under some regulation. If a code change involves multiple independent variable features and each feature has multiple optional values, then it will cause tremendous QA efforts because all possible configurations have to be checked.

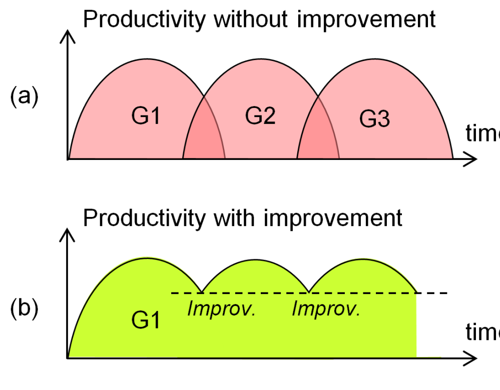


Fig. 1. SPL Productivity with and without Improvement.

The overly complex variability realizations cause a serious impact on productivity of SPL maintenance, and put the expected SPL advantages more and more at risk. In the worst case, a new generation of the SPL is created to develop new products, while the old SPL infrastructure still needs to be maintained to support the old products, as illustrated in Fig. 1(a). In order to sustain SPL productivity above a sufficiently good level as illustrated in Fig. 1 (b), we investigate variability realization erosion and propose corresponding improvements in this paper. Specifically, our research contributions are as follows:

- We identify and automatically measure typical variability erosion symptoms in an industrial SPL.
- We propose an improvement strategy to optimize variable code realized in Conditional Compilation (CC), including standardizing, merging, pruning, and splitting #IFDEF statements.
- We propose another improvement strategy to transition CC to a new variation mechanism called Parameterized Inclusion (PI).
- We discuss advisable improvement strategies in different variability evolution scenarios, so that they can be selectively adopted in variable code realized in CC.

The paper is organized as follows: Section II introduces related research, including various variability realization erosion symptoms identified in an industrial SPL and studies related to variability realization improvement. Section III presents two variability realization improvement strategies against the erosion problems, and the QA issue after conducting the improvements is discussed as well. Section IV discusses the adoption of a specific improvement strategy in different SPL evolution scenarios, while the conclusion and future work is presented in section V.

II. Related Research

In this section, we introduce several variability realization erosion symptoms identified from our previous case study of an industrial SPL, based on which our variability improvement strategies are motivated and derived. Moreover, we also discuss current studies related to variability realization improvement and compare them with our approach.

A. *Variability Realization Erosion*

In variability realizations, CC is a frequently used mechanism to enable or disable variant code [5] [9] [16]. To be specific, macro constants are defined in product configurations and used in #IFDEF expressions (in this paper we use #IFDEF to imply the directives of #if, #elif, #ifdef, and #ifndef) in code core assets. Based on the defined values of macro constants, a C-preprocessor adapts the core assets by enabling or disabling enclosed code fragments in an intuitive way.

In our previous study [23], we have considered various types of Variability Elements (VEs) as illustrated in Fig. 2, i.e., Variability (Var), Variation Point (VP), and Variation Point Group (VPG). A Var represents a variable feature in the problem space and is realized in VPs (solution space) to enable or disable enclosed code fragments (i.e., variants). In CC a Var is represented by a macro constant, while a VP is represented by a #IFDEF block. Moreover, VPs with logically equivalent #IFDEF expressions are clustered as a VPG because they have the same effect on their enclosed code fragments. As shown in Fig. 2, each Var can be used in multiple VPGs, while each VPG may contain multiple VPs. These VPs may be scattered in either one or multiple code files. The concept of VPG helps to understand the alignment of VEs between problem space and solution space.

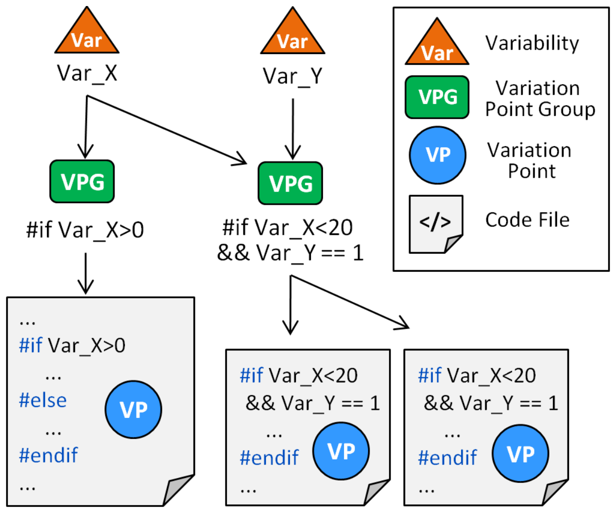


Fig. 2. Variation Mechanism: Conditional Compilation.

Although CC is a straightforward mechanism to implement variation, its usage also brings negative effects: i) the realizations of commonality and variability (#IFDEF code) are mixed in the same core code file; ii) adding a new variant needs to change the entire VP, which is known as closed variation [10]; iii) a VP written as a single #IFDEF block innately only supports one optional variant (i.e., an #IFDEF block without #else or #elif) or two alternative variants (i.e., the positive and negative branch of the #IFDEF block). A VP with multiple co-existing or alternative variants (i.e. OR/XOR) in CC needs to be implemented using multiple #IFDEF blocks.

Moreover, the aforementioned disadvantages of CC cause variability erosion during SPL evolution. In our previous case study [23], several erosion symptoms, such as variability nesting, tangling, and scattering, have been investigated in an industrial SPL of Danfoss in the embedded system domain. All these symptoms indicate the increasing code complexity, which makes the code difficult to understand and maintain [6] [13]. We classify the symptoms of variability erosion in the following subsections.

1) *Variability Nesting*

VPs realized by #IFDEF blocks can be nested to another. If a VP is deeply nested, then its logic would be very complex, which impair program comprehension and make the code maintenance error-prone[11]. In the case study [23], most VPs (94%) are not nested (nesting level equals one) or only nested in one degree (nesting level equals two), and the average nesting degree is 1.50, which is not very high in general. However, a few VPs (6%) are nested with a high nesting degree (up to 5), which are difficult to understand due to their complex logic.

2) Variability Tangling

The #IFDEF expression of a VPG may contain only one or multiple tangled Vars. The tangling degree is defined as the number of Vars that are used in the #IFDEF expression of a given VPG. For instance, in Fig. 2 the tangling degree of the VPG "#if Var_X<20 && Var_Y==1" is two, since it has two Vars. If the #IFDEF expression of a VPG uses too many Vars, then its logic would be very complicated to understand. In the Danfoss case study, most VPGs (98%) only contain one or two Vars in their #IFDEF expressions, and average tangling degree of all core code is 1.21, which also indicates a low tangling level. However, there are a few VPGs (2%) with a high tangling degree (up to 11), which are very difficult to understand.

3) Variability Scattering

In conditionally compiled core code, Vars are scattered in different VPGs as well as in different files. The issue of Var scattering in VPGs depicts the situation when a Var is used in the #IFDEF expressions of different VPGs. It indicates the change impact of a Var on VPGs in the scenario of SPL maintenance. For instance, in Fig. 2 the Var "Var_X" is used in two different #IFDEF expressions, and thus the scattering degree is two. In the Danfoss case study, most Vars (88%) are scattered in one or two VPGs, and the average scattering degree is 1.68. However, some Vars (12%) have significantly higher scattering degree (up to 21), which means each of these Vars crosscuts the variability realizations in many different #IFDEF blocks. Note that since the change propagation of a Var on different VPGs might be different, the Vars with high scattering degree on VPGs would cause a tremendous maintenance effort if they are changed.

Besides analyzing Var scattering in VPGs, we also investigate Var scattering in files. For instance, in Fig. 2 the Var "Var_X" is scattered in three files. This scattering degree indicates the change impact of a Var on files, because the change of a Var needs to be propagated to all relevant files. Since each file shall be quality-assured after change, this metric also indicates the QA effort during SPL maintenance. In the Danfoss case study, most Vars (86%) are scattered in up to ten files, while some Vars (14%) have a significantly higher scattering degree (up to 144). The average scattering degree on file is 6.20, which is also high compared to the scattering degree on VPGs.

4) Complex Variable Code

As a SPL evolves, the variable code files become complex in the sense that they contain more and more VEs, making these files difficult to understand for adapting changes in SPL maintenance. In our previous study, we measured the number of Vars used in #IFDEF expressions and the number of VPs in each variable code file. For instance, the left file in Fig. 2 contains one Var and one VP. In the Danfoss case study, most files (75%) contain one to three Vars, and the average number of Vars per file is 3.91. However, some more complex files (25%) contain up to 118 Vars, which would be very difficult to understand and error-prone during maintenance.

Moreover, the number of VPs in a code file indicates the complexity of a code file. For instance, given N VPs being independent and not nested in one method of a file, the Cyclomatic Complexity of the method is propositional to N. In the Danfoss case study, most files (76%) contain up to ten VPs, and the average number of VPs per file is 10.57, which is already a non-trivial problem. Additionally, some more complex files (24%) contain from 11 up to 407 VPs in the extreme case. Understanding and maintaining such files would be very time-consuming and error-prone.

B. *Variability Realization Improvement*

To the best of our knowledge, there is not many studies in the field of variability realization improvement. Patzke [16] defined the term variability refactoring as a family engineering activity to change a SPL infrastructure in order to evolve or reuse it in a more cost-efficient way. He claimed that while conventional refactoring is to make existing artifacts easier to use, variability refactoring is to make existing core assets easier to reuse. Moreover, he summarized a number of variability refactoring activities. Among these activities, our first strategy of CC optimization is related to the activities of renaming VPs and increasing VP visibility, while our second strategy of transitioning CC to PI is related to the activities of separating variants from commonality, separating variants from each other, replacing closed with open variants, and extracting reuse hierarchy. However, all the refactoring activities were only discussed conceptually without implementation.

Besides, Alves [2] et al. presented a catalogue of Feature Model refactorings in terms of configurability. Based on that Borba [4] extended it into a SPL refactoring catalogue including formalized transformation templates not only for Feature Models, but also for other artifacts such as Configuration Knowledge. The refactoring activities in their study are aiming at deriving a SPL from existing products (extractive SPL development) and extending an existing SPL to encompass another product (reactive SPL development). Thus their refactoring activities are driven by product characteristics that need to be included as SPL features. On the Contrast, our goal is to improve variability realizations of an existing SPL mainly in terms of maintenance and sustain its productivity during evolution.

In our previous study [21], we analyzed variability-related CPP code in both core assets and product realizations, and extracted a realization variability model including variabilities, variation points, and a graphical tree of variation points based on #ifdef nesting. Moreover, we also presented a feature correlation mining approach [22] to identify implicit variability interdependencies from existing product configurations. Both of these two approaches help to understand complex variability realizations via automatic analysis.

Beyond the scope of variability realization improvement, there are several studies focusing on preprocessor code analysis and refactoring. Baxter and Mehlich [3] presented an automated approach to detecting and removing obsolete #IFDEF statements (also known as dead #IFDEFs) based on heuristic transformation rules. Therefore, their approach can only deal with preprocessor code that satisfies certain patterns.

Similarly, Garrido [7][8] presented an automated approach to conduct various refactoring activities on preprocessor-aware C code. The refactoring activities on preprocessor code involves not only CC, but also file inclusion and macro expansion. The refactoring of CC includes removing obsolete #IFDEF statements and fixing incomplete #IFDEF blocks, which is also based on heuristic rules. Liebig et al [12] also discussed the issue of incomplete (also known as undisciplined) #IFDEF code based on an comprehensive code analysis of 40 open source systems. Based on extraction of an AST (Abstract Syntax Tree) from the preprocessor code, it turned out that 84% of #IFDEF code in their analysis is disciplined. Regarding the 16% undisciplined #IFDEF code, they recommended to manually rewrite them into disciplined annotations.

Adams et al. [1] investigated the feasibility of replacing #IFDEF blocks with aspects considering the issues of #IFDEF tangling and scattering. They provided abstract refactoring ideas in term of different patterns of CC usage, but did not apply them in real systems. In general, the idea of transitioning CC to Aspect Orientation (AO) in variability realizations is helpful to separate common and variable code. However, this refactoring also faces several practical challenges: i) AO is not applicable to arbitrary preprocessor code; ii) it could be difficult to automatically apply this refactoring even to limited preprocessor code; iii) This refactoring will change the compiled source code, which therefore needs to be quality assured afterwards.

III. VARIABILITY REALIZATION IMPROVEMENT

Given the variability erosion symptoms discussed in section II, we present two major strategies of variability realization improvement as illustrated in Fig. 3. The first strategy is to optimize the variation mechanism of CC, and the second strategy is to refactor the #IFDEF blocks by transitioning CC to a new variation mechanism called Parameterized Inclusion (PI). These two improvement strategies are introduced in the following subsections.

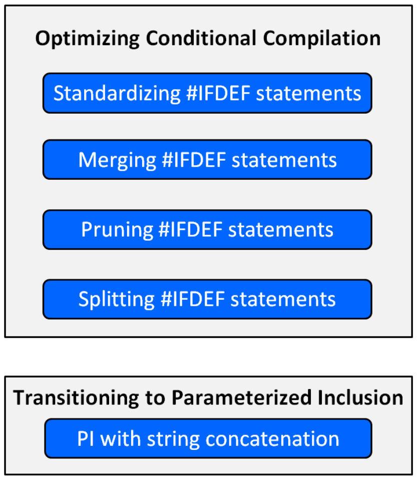

Optimizing Conditional Compilation
Standardizing #IFDEF statements
Merging #IFDEF statements
Pruning #IFDEF statements
Splitting #IFDEF statements
Transitioning to Parameterized Inclusion
PI with string concatenation

Fig. 3. Variability Realization Improvement Strategies.

A. *Optimizing Conditional Compilation*

In the variation mechanism of CC, Vars are defined as macro constants and used in different #IFDEF statements. An #IFDEF statement consists of a C preprocessor directive (i.e., either #if, #elif, #ifdef, or #ifndef) and an arbitrary Boolean predicate (called #IFDEF expression in this paper). From our practical observation, the #IFDEF statements are usually written in an ad-hoc way, which makes them complex (e.g., variability tangling) and difficult to understand. Therefore, we propose to a set optimization activities on #IFDEF statements, which are presented in the following subsections.

1) *Standardizing #IFDEF Statements.*

Since such #IFDEF statements are usually originated and maintained by different developers over a long time span, they are probably written in very different programming styles during SPL evolution. Moreover, #IFDEF statement written in a different style may contain redundant characters, such as “#if ((Var_X)>0)” compared to “#if Var_X>0”. Although such #IFDEF statements are correct and acceptable to the C preprocessor, the different programming styles lack consistency and may cause unnecessary code complexity, which impairs code comprehensibility. Especially, the different programming styles lead to VPs with syntactically different #IFDEF statements but sharing the equivalent inclusion logic. In this case, clustering such VPs into VPGs would be difficult, which will impair the change impact assessment of a Var on VPGs during code maintenance. Before adopting any semantic analysis, we propose to conduct several syntactical standardization activities on #IFDEF statements in order to adapt them into a canonical form.

a) Rewriting #ifdef and #ifndef statements. The C preprocessor accepts both the keyword “defined” and the directives of #ifdef and #ifndef. While some developers are used write “#ifdef Var_X”, other developers prefer to write “#if defined(Var_X)”. However, these different forms cause a problem in clustering VPs into VPGs. There is a dilemma that, on the one hand the clustering of VPs should only depend on #IFDEF expressions and regardless of the directives like #if and #elif; but on the other hand the directives of #ifdef and #ifndef affect the logic of the corresponding expressions. For instance, “#ifdef Var_X” and “#if Var_X” have different semantics, and thus should be clustered into different VPGs. As a result, the Var scattering degree on VPGs will be either overestimated (if VPGs are clustered by #IFDEF statements) or underestimated (if VPGs are clustered by #IFDEF expressions).

In order to solve this issue, we propose to unify these different forms by i) rewriting #ifdef statements like “#ifdef Var_X” as “#if defined(Var_X)”; and ii) rewrite #ifndef statements like “#ifndef Var_X” as “#if !defined(Var_X)”. After this adaptation, the #IFDEF statements will only contain “#if” and “#elif” directives, which do not affect the logic of #IFDEF expressions. Therefore, it is necessary to conduct this adaptation at the beginning of CC optimization, so that we can focus on #IFDEF expressions in the following steps.

We have analyzed #IFDEF statements of all 13970 VPs in the Danfoss SPL, and found 13 #ifdef directives and 5 #ifndef directives. After rewriting these 18 #ifdef and #ifndef statements, we continue to conduct further standardization activities on #IFDEF statements (which actually only contain #if and #elif directives).

b) Removing Extra Spaces. In the variable code of Danfoss SPL, we find that some #IFDEF expressions are only different in spaces. Therefore, we propose to remove the extra spaces based on two rules: i) no space character between a parenthesis/bracket and another token; ii) only one space character between two tokens that are neither parentheses nor brackets. We have conducted this adaptation automatically in the Danfoss SPL, and removed extra spaces in the #IFDEF expressions of 128 VPGs (9.1%). After this step, the number of VPGs is reduced from 1287 to 1221 (66 equivalent VPGs merged).

c) Removing Extra Parentheses. #IFDEF expressions might be different in parentheses, and we propose to remove extra parentheses in three cases: i) parentheses enclosing a single token; ii) parentheses enclosing an entire #IFDEF expression; iii) parentheses directly enclosing another pair of parentheses, e.g., ((Var_X>0)). We have analyzed the 1221 different #IFDEF expressions from the previous optimization step, and have removed redundant parentheses in 47 #IFDEF expressions. After this step, the number of VPGs is reduced from 1221 to 1196 (25 equivalent VPGs merged).

d) Unifying Negation Expressions. While an #IFDEF expression can be written as negation of a proposition, the negation sign can also be moved into each part of the proposition with equivalent semantic. For instance, based on De Morgan's theorem expressions like "!(Var_X && Var_Y)" are equivalent to "!Var_X || !Var_Y". To unify these two forms, we rewrite each negation expression (in the form of "!(proposition)") by moving the external negation sign inside the proposition and then removing the parentheses. In the variable code of Danfoss SPL, there are six negation expressions, all containing only a single equation (i.e., in the form like "!(Var_X==0)"). Thus they are rewritten into the form like "Var_X!=0". After this step, the number of VPGs is reduced from 1196 to 1193 (3 equivalent VPGs merged).

2) Merging #IFDEF Statements

As introduced in section II.A.3, a Var is usually used in different #IFDEF expressions, and its change impact on VPGs should be measured by the number of semantically different #IFDEF expressions that use a given Var, i.e., the scattering degree on VPGs. Otherwise, if the calculation of Var scattering on VPGs is simply based on syntactic clustering of #IFDEF expressions, then the syntactically different but semantically equivalent expressions will be counted as different VPGs, and thus change impact of each Var in the expressions would be overestimated. This overestimation will cause unnecessary effort in change propagation. Therefore, we propose to merge equivalent #IFDEF expressions into a unique form (which is the shortest expression in the equivalence class).

In our previous optimization activities, #IFDEF statements are syntactically standardizing into a canonical form, so that slight differences in similar #IFDEF expressions (e.g., extra parentheses and spaces) will be eliminated. However, there might be still equivalent #IFDEF expressions after the standardization activity, such as "#if Var_X>0 && Var_Y<1" and "#if 1>Var_Y && Var_X>0". In order to analyze the equivalence of these #IFDEF expressions, for each pair of expressions <expr1, expr2> we first parse them in Python, and then check the satisfiability of their equivalence (i.e., "expr1==expr2") in Z3 [20], which is an open source theorem prover.

Currently, we have managed to automatically analyze the equivalence of all #IFDEF expressions in the Danfoss SPL, which are simple conjunctive or/and disjunctive propositions (no nested conjunction or disjunction in parentheses). Finally, 12 equivalent #IFDEF expressions are identified in the Danfoss SPL. Moreover, if we assume each related Var as a binary constant, then another 27 equivalent #IFDEF expressions are identified. These 39 expressions are rewritten and merged into their corresponding VPGs. However, considering the fact that an expression might involve other functions or code structures, comparing the equivalence of two arbitrary expressions is equivalent to comparing the equivalence of two arbitrary programs, which is theoretically an undecidable problem [19].

TABLE I. #VPG REDUCTION BY STANDARDIZATION AND MERGING

Tangling Degree	1	2	3	4	7	11	Total
#VPG Reduction	96	33	3	1	0	0	133
Reduction Percentage	9%	17%	13%	25%	0%	0%	10%

Note that even for the purpose of merging equivalent #IFDEF statements, the former standardization activity is necessary, because some expressions might be too complex to parse semantically but can be standardized. After conducting the optimization of standardizing and merging #IFDEF statements, relevant expressions are rewritten, and the number of VPGs are reduced. As shown in Table I, most merging activities (i.e., #VPG reduction) are conducted among #IFDEF statements at the tangling degree of one or two. Totally 133 (10%) equivalent #IFDEF statements are merged into their corresponding VPGs.

Besides standardizing and merging #IFDEF statements, we have two further optimization activities, which have not been applied to the Danfoss SPL.

3) Pruning #IFDEF Statements

As introduced in section II.A.2, the issue of variability tangling is that an #IFDEF expression contains too many Vars. If the expression is a conjunctive or disjunctive proposition and not all the premises of the proposition affect the inclusion of variant code, then the irrelevant premise(s) should be removed from the #IFDEF expression. On the one hand, for conjunctive #IFDEF expressions like "P1 && P2 && ... && Pn", if a Pi is always true (perhaps due to a mandatory feature) in all product configurations then it can be removed. On the other hand, for disjunctive #IFDEF expressions like "P1 || P2 || ... || Pn", if a Pi is always false (perhaps due to an obsolete feature) in all product configurations then it can be removed.

For instance given the #IFDEF statement "#if defined(Var_X) && (Var_X>0)", if Var_X is always assigned a positive value when defined, then the statement can be pruned as "#if defined(Var_X)". Especially, if premises in a conjunctive #IFDEF expression are proved to be implicatively correlated, then the consequent of the implication should be removed from the expression. For instance given the expression "#if defined(Var_X) && Var_Y>0", if the feature correlation "(Var_Y>0) $\Longrightarrow$ defined(Var_X)" is specified in the variability model, then the #ifdef statement can be pruned and rewritten as "#if Var_Y>0". This is a typical scenario in practice when Var_Y is a child feature of Var_X.

The opposite scenario is that if a premise is constantly false (or true) in a conjunctive (or disjunctive) #IFDEF expression, then the expression will be constantly false (or true). In this case, the entire #IFDEF statement should be removed because the variant code becomes dead (or common code). In conclusion, this optimization can be conducted in either conjunctive or disjunctive #IFDEF statements, and it requires further domain knowledge.

4) Splitting #IFDEF Statements

The optimization activity of splitting #IFDEF statements is to decompose a conjunctive #IFDEF statement into two #IFDEF statements and rewrite the entire #IFDEF block into two nested blocks. For instance, the #IFDEF block "#if defined(Var_X) && Var_Y>0 ... #endif" can be rewritten as "#if defined(Var_X) #if Var_Y>0 ... #endif #endif". While this activity can automatically relieve the issue of variability tangling without domain knowledge, it will create another issue of variability nesting. Therefore, it is only suitable for conjunctive #ifdef statements with high tangling degree and low nesting degree.

B. Transitioning CC to Parameterized Inclusion

Although the aforementioned optimization strategy helps to standardize #IFDEF statements and reduce their complexity, the improvement is limited and still within the scope of CC. Considering the disadvantages of CC, we propose to selectively refactor some VPs using a new variation mechanism, Parameterized Inclusion.

1) Parameterized Inclusion.

The variation mechanism of Parameterized Inclusion (PI) is similar to Module Replacement [16], but it is applied at preprocessing time instead of link time. The idea is to encapsulate each variant code fragment into a separate file, and use the C preprocessor directive "#include" to dynamically include the corresponding variant code fragment into the main core code.

In order to transition from CC to PI, each VP (i.e., an #ifdef block) is indexed with a number i, and replaced by a statement such as "#include VPi_File" in Fig. 4. The VPi_File is defined as a macro constant in each product code, and its value is determined by the #IFDEF expression of the VP. For instance, the VP "#if Var_X>0" in Fig. 2 is replaced by "#include VP1_File", and the constant VP1_File will be defined as either "V_1_1.inc" or "V_1_0.inc" in different products depending on the value of Var_X. The two .inc files are positive and negative variants of this VP, containing respectively the positive and negative branch of the corresponding #IFDEF block. If a VP only has the positive variant (i.e., no #else or #elif statement), then the negative variant is bound to an empty file, such as "Null.inc" in Fig. 4.

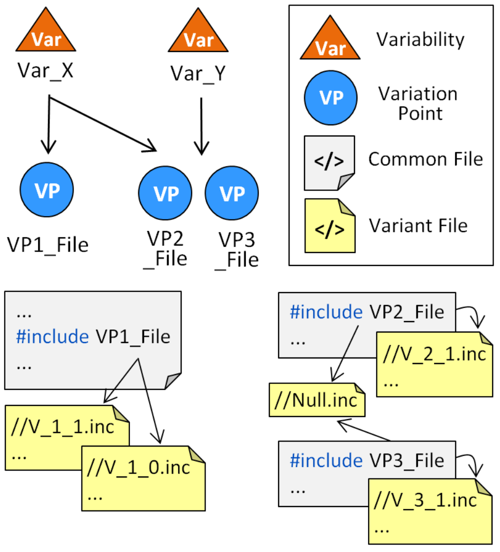


Fig. 4. Variation Mechanism: Parameterized Inclusion.

The advantages of PI are numerous as follows.

a) Explicit variant binding without complex #IFDEF statements. Since each variant is explicitly selected by defining the corresponding VP parameter VPi_File in product configurations, there will be no #IFDEF code in VPs using PI, and thus the issues of #IFDEF nesting and tangling are automatically solved.

b) The common code and variable code are separated. As the common code and the code of each variant are separated in different files, they can be maintained in parallel. Moreover, since each code change action is restricted in a smaller scope, the maintenance process should be less error-prone. Particularly, this shall provide more protection to the common code, because they are usually less frequently modified than variable code.

c) PI is an open variation mechanism. Since the code of each variant is separated in an individual file, adding a new variant or modifying an existing variant does not contaminate other variants in terms of code quality assurance.

d) Stronger support for multiple variants. While one #IFDEF block in CC innately only support two variants in the positive and negative branch, in PI a VP may be linked to any number of varaints by creating the relevant .inc files.

Despite the advantages of PI, its disadvantages should also be considered. One disadvantage of PI is that if the variant code is strongly interrelated with the common code, then it might be difficult to understand both the common code and the encapsulated variant code. Moreover, another disadvantage is that for each VP an additional macro constant (e.g., VPi_File in

Fig. 4) needs to be created. As a result of this overhead, the consistency between the constant VPi_File and its values (i.e., the .inc files) as well as their binding rules must be maintained, which could be a non-trivial effort.

To maintain the binding rules between VPs and variants, a variation binding table can be created as an auxiliary documentation. As shown in Fig. 5, for each VP refactored from CC, the macro constant VPi_File is linked to its positive and negative variants (i.e., the .inc files) in the format of a ternary expression. If there is no negative variant, then the null variant (i.e., Null.inc) can be omitted.

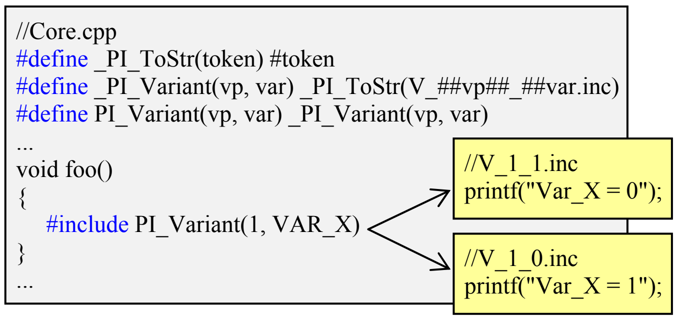
```
VP1_File: Var_X>0 ? V1_1.inc : V1_0.inc;
VP2_File: Var_X>0 && Var_Y==1 ? V2_1.inc;
VP3_File: Var_X>0 && Var_Y==1 ? V3_1.inc;
```

Fig. 5. A variation Binding Table.

2) Parameterized Inclusion with String Concatenation. Although the mechanism of PI avoids the complex #IFDEF blocks and adapts the variable code in a modularized style, it hands over the work of variation binding to developers during product configuration (in CC it is automatically done via #IFDEF statements). This increases the risk of incorrect variation binding. In order to facilitate variation binding, we have improved PI with the idea of automatic string concatenation.

For any VP in which each variant depends on a different value of a Var, then the value of the VP constant VPi_File can be dynamically generated by concatenating the index of the VP and the corresponding Var value. For instance given the VP "#if Var_X>0" in Fig. 2, if Var_X is a Boolean constant, then the positive variant is actually bound under the condition "#if Var_X==1", while the negative variant is bound under the condition "#if Var_X==0". In this case, it is not necessary to define the VP constant VP1_File, because the variant file name "V_1_1.inc" and "V_1_0.inc" can be dynamically concatenated using the VP index (1) and the value of Var_X (1 or 0), as shown in Fig. 6.

Note that since this string concatenation is conducted during preprocessing time, the code is preprocessed as merely tokens, and the normal string concatenation operations supported by C/C++ cannot be used. To concatenate tokens as a string file name during preprocessing time, a feasible solution is to define macros using the "##" and "#" operators [5] [18] as shown in Fig. 6. While the "##" operator helps to concatenate tokens (of VP index and the Var value), the "#" operator helps to convert a token into a string (of the variant file name).



```
//Core.cpp
#define _PI_ToStr(token) #token
#define _PI_Variant(vp, var) _PI_ToStr(V_##vp##_##var.inc)
#define PI_Variant(vp, var) _PI_Variant(vp, var)
...
void foo()
{
    #include PI_Variant(1, VAR_X)
}
...
```

```
//V_1_1.inc
printf("Var_X = 0");
```

```
//V_1_0.inc
printf("Var_X = 1");
```

Fig. 6. Parameterized Inclusion with String Concatenation.

C. Code Validation after Improvement

We have introduced two variability realization improvement strategies, optimization of #IFDEF code and transitioning from CC to PI. Applying either of these strategies would change the core code. Theoretically, the changed code needs to be re-validated (typically via testing) in order to ensure that the refactoring is behavior-preserving and does not affect the code quality (i.e., correctness, reliability, etc.). However, we argue that such Quality Assurance (QA) is unnecessary because these two improvement approaches do not change the product code after preprocessing.

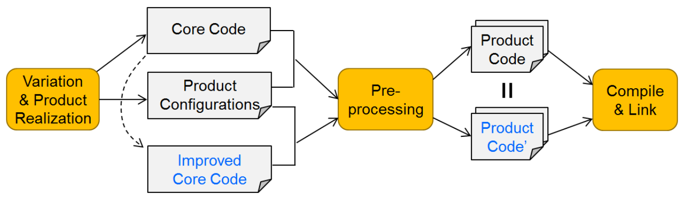


Fig. 7. Validation of Refactored Code.

As illustrated in Fig. 7, the core code is developed during variability realization and reused by various products. If the variation mechanism is CC or PI, then variants are bound at preprocessing time depending on product configurations. Since the optimization of #IFDEF statements and the transition from CC to PI only change the organization of variant code in each VP instead of the variant content, the core code with and without improvement should generate exactly the same product code. In another words, the variable code improvement process should neither affect the product code nor affect the derived software products. In that case, there will be no additional cost of conducting the two variability improvement strategies.

IV. ADOPTION OF IMPROVEMENT STRATEGIES

As discussed in section II, variability realizations often becomes overly complex during SPL evolution, which is known as variability erosion [23] and makes related core code difficult to understand and maintain. In other words, this problem only exists when the core code is going to be read and changed manually. Otherwise, if some variable code is only developed once and does not need to be changed, then it is unnecessary to improve the code no matter how complex it is.

In this paper, we have introduced different strategies for variability realization improvement for different purposes. They should be only applied to evolving core code that is going to be changed manually. Moreover, the adoption of an improvement strategy should depend on how the core code is going to be changed, i.e., specific variability evolution scenarios. Given the context of variability realizations using CC, we discuss following basic evolution scenarios and their suitable improvement strategies. The evolution scenarios are categorized based on concrete VEs defined in Fig. 2, including Var, VP and variant (VPG is an abstract concept and does not count).

A. *Adding a Var or VP*

Since a Var is used in VPs to enable or disable variants, a new Var can be created and used in either a new VP or an existing VP, while a new VP can be added into the core code that contains either new Vars or existing Vars. In either of these two evolution scenarios, if the code operation is independent and does not involve further propagation, then none of our proposed improvement strategies needs to be conducted because it is not sure whether the added code is going to be read and changed manually afterwards.

B. *Changing a Var*

In the context of CC, changing a Var is to rename or delete a macro constant used in #IFDEF expressions of existing VPs. This operation requires understanding of #IFDEF expressions, and is usually conducted not only on a single VP but on VPs with equivalent #IFDEF expressions, i.e., a VPG. In order to correctly determine the change impact of a Var on a certain VPG, it is advisable to *standardize and merge #IFDEF statements of VPs*, so that VPGs can be calculated precisely. Besides, if the related #IFDEF statements are complex containing many Vars, it is advisable to adopt the optimization strategy of *#IFDEF pruning and splitting*.

C. *Changing a VP*

In the context of CC, changing a VP is to change the corresponding #IFDEF block, which involves two possible operations. One operation is to change the predicate of the VP, i.e., the #IFDEF statements, which affects variant binding. Besides, another possible operation is to change the cardinality of the VP, e.g., to change its alternative variants into optional variants. Both of these two operations are manual and require understanding of the corresponding #IFDEF statement. Therefore, if the #IFDEF statement is complex containing many Vars, it is advisable to adopt the optimization strategy of *#IFDEF pruning and splitting*.

D. *Adding a variant*

If a variant is only added into a given VP, then it is usually unnecessary to conduct any improvement strategies because it is not sure whether the added variant will be changed afterwards. However, since the mechanism of CC innately only support VPs with two variants, one exception is that if a third variant is added then it is advisable to *transition the variation mechanism of that VP from CC to PI*, which supports multiple variants.

E. *Changing a variant*

In CC variants of a VP are written in one code file, which may even include other VPs as well as common code. As a consequence, the code change tends to be error-prone due to the file complexity, and after changing one variant the entire code file needs to be quality assured. To mitigate these issues, it is advisable to *transition the variation mechanism of the corresponding VP from CC to PI*, where each variant is encapsulated in a separate file. Considering the advantages and disadvantages of PI (discussed in section III.B), this strategy can be applied to each VP satisfying the following criteria.

1) There is certain evidence that variants of *the VP is expected to be frequently changed*. This trend can be either predicted by domain experts or by analyzing the evolution history of the corresponding SPL.

2) Changing the variant code in CC is effort-consuming and error-prone. E..g, the variant has a huge size or high cyclomatic complexity, or the code file containing the VP is variability-intensive with numerous Vars or VPs.

3) Its variant code should have low coupling and high cohesion, so that the variant encapsulation will not impare code comprehensibility.

V. Conclusion and Future Work

In this paper we have introduced several variability erosion symptoms in practice, and propose a variability improvement process including two improvement strategies to solve these issues. While one strategy is to optimize variable code within the scope of CC, the other strategy is to transition CC to a new variation mechanism call Parameterized Inclusion. Both of these two improvement strategies can be conducted automatically. Since these improvement strategies are behavior-preserving and do not change the product code after preprocessing, no QA effort is required after improvement. Besides, the adoption of a specific strategy should depend on different SPL evolution scenarios.

Our future work includes: 1) conducting the transition from CC to PI in an industrial SPL; 2) obtaining the product code with and without improvement after preprocessing, and validate their identity; 3) investigating further variability improvement strategies.

Acknowledgment

This work is within the MOTION project of “Innovationszentrum Applied System Modeling”, sponsored by the German state of Rhineland-Palatinate and Fraunhofer IESE. See http://www.applied-system-modeling.de/.

References

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan, "Can we refactor conditional compilation into aspects?" in Proceedings of the 8th ACM international conference on Aspect-oriented software development, ser. AOSD '09. New York, NY, USA: ACM, 2009, pp. 243-254.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring product lines," in Proceedings of the 5th international conference on Generative programming and component engineering, ser. GPCE '06. New York, NY, USA: ACM, 2006, pp. 201-210.

[3] I. D. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), ser. WCRE '01. Washington, DC, USA: IEEE Computer Society, 2001.

[4] P. Borba, "An introduction to software product line refactoring," in Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III, ser. GTTSE'09. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1-26.

[5] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," IEEE Transactions on Software Engineering, vol. 28, no. 12, pp. 1146-1170, Dec. 2002.

[6] J. M. Favre, "The CPP paradox," in 9th European Workshop on Software Maintenance, 1995.

[7] A. Garrido and R. Johnson, "Refactoring c with conditional compilation," 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), vol. 0, p. 323, 2003.

[8] A. Garrido, "Program refactoring in the presence of preprocessor directives," Ph.D. dissertation, Champaign, IL, USA, 2005.

[9] C. Gacek and M. Anastasopoulos, "Implementing product line variabilities," SIGSOFT Softw. Eng. Notes, vol. 26, no. 3, pp. 109-117, May 2001.

[10] J. Van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in Proceedings of the Working IEEE/IFIP Conference on Software Architecture, ser. WICSA '01. Washington, DC, USA: IEEE Computer Society, 2001.

[11] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, “An analysis of the variability in forty preprocessor-based software product lines,” in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 105-114.

[12] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of c code," in Proceedings of the tenth international conference on Aspect-oriented software development, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 191-202.

[13] P. E. Livadas and D. T. Small, "Understanding code containing preprocessor constructs," in Program Comprehension, 1994. Proceedings., IEEE Third Workshop on, 1994, pp. 89-97.

[14] S. Livengood, "Issues in software product line evolution: complex changes in variability models," in Proceeding of the 2nd international workshop on Product line approaches in software engineering, ser. PLEASE '11. New York, NY, USA: ACM, 2011, pp. 6-9.

[15] D. L. Parnas, "Software aging," in Proceedings of the 16th international conference on Software engineering, ser. ICSE '94, vol. 7, no. 4. Los Alamitos, CA, USA: IEEE Computer Society Press, Dec. 1994, pp. 279-287.

[16] T. Patzke, "Sustainable evolution of product line infrastructure code," Ph.D. dissertation, 2011.

[17] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," SIGSOFT Softw. Eng. Notes, vol. 17, no. 4, pp. 40-52, Oct. 1992.

[18] S. Prata, C Primer Plus (5th Edition). Indianapolis, IN, USA: Sams, 2004.

[19] Undecidable problems. http://www.cs.rochester.edu/~nelson/courses/csc_173/computability/undecidable.html (retrieved in May 2013)

[20] Z3. http://z3.codeplex.com/. (retrieved in May 2013)

[21] B. Zhang and M. Becker, "Code-based variability model extraction for software product line improvement," in Proceedings of the 16th International Software Product Line Conference - Volume 2, ser. SPLC '12. New York, NY, USA: ACM, 2012, pp. 91-98.

[22] B. Zhang and M. Becker, "Mining complex feature correlations from software product line configurations," in Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ser. VaMoS '13. New York, NY, USA: ACM, 2013.

[23] B. Zhang and M. Becker, "Variability Evolution and Erosion in Industrial Product Lines - A Case Study," submitted to 17th International Software Product Line Conference. SPLC '13. New York, NY, USA: ACM. Unpublished.