RECoVar: A Solution Framework towards Reverse Engineering Variability

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

Martin Becker  
Fraunhofer Institute for Experimental Software Engineering (IESE)  
Kaiserslautern, Germany  
martin.becker@iese.fraunhofer.de

Abstract—As a Software Product Line (SPL) evolves variability specifications in problem space and variability realizations in solution space erode over time and impact productivity during development. On the one hand, the variability model tends to be incomplete and inconsistent with the core assets; on the other hand, the core assets become overly complex, which make them difficult to understand and maintain. In this paper, we present the RECoVar framework towards reverse engineering of SPL variability. The framework includes two approaches: a) code-based variability model extraction; and b) complex feature variability information. These two approaches help to extract various variability information, so that variability specifications and realizations can be maintained in an efficient way.

Index Terms—Variability modeling; reverse engineering; product line analysis

I. PROBLEM STATEMENT

During development of Software Product Lines (SPLs), variability artifacts are created both in problem space and solution space. Based on the definition of Czarnecki and Eisenecker [3], the problem space contains specification artifacts, while the solution space contains realization artifacts. In Family Engineering (FE), a variability model (also known as feature model [9]) is created based on domain knowledge. Then it is developed into core code assets and interlinks with variation points in the code. In Application Engineering (AE), different products are configured by selecting features from the variability model and assigning feature values if needed.

Nowadays SPL development is often conducted in an incremental way, in which the variability artifacts evolve both in space and in time [10]. During evolution the variability supported by the SPL infrastructure is changing over time, which requires iterative development and maintenance of variability specifications and realizations. However, experience in evolving SPLs shows that variability specifications and realizations tend to erode in the sense that they become overly complex and inconsistent with each other. This causes practical challenges in both problem space and solution space.

A. Challenges in Problem Space

A variability model documents all features in a SPL and their correlations. Theoretically the variability model should be consistent with core assets to guide development in FE and to facilitate product configuration in AE. However, as the variability realization gets overly complex, it is difficult to maintain the consistency between the variability model and core assets. A typical scenario in practice is that a core asset is changed during SPL evolution, while the respective variability model is not updated accordingly (so far the update cannot be fully automated). As a result, the variability model is inconsistent with the core assets, which means it either lacks certain feature information or contains incorrect feature information.

This issue is also discussed in other studies. The investigation of Patzke et al. on industrial SPLs [16] reveals that in practice the variability model tends to be incomplete or even missing due to ambiguous variation points and inexplicit variant elements. Moreover, Tartler et al. [18] addressed the problem of zombie features in Linux kernel, where the variability model includes features that can neither be enabled nor disabled.

Besides, the inconsistency problem of a variability model will further affect product configurations because they are derived from the model. If a variability model includes incomplete or inconsistent feature correlations, the derived product configurations might either lack certain dependent features or contain certain undesired features. Considering the increasing number of features and complex feature correlations in real SPLs, it is very difficult to identify such misconfiguration manually. This is a practical problem and also being addressed in [6] and [16].

B. Challenges in Solution Space

In variability realizations, Conditional Compilation (CC) is a frequently used mechanism to enable or disable feature code [8]. To be specific, macro constants are defined in product configurations and used in #ifdef statements. As a result, #ifdef also implies similar directives such as #if and #elif in code core assets. Based on the defined values of macro constants, a preprocessor adapts the core assets by enabling or disabling enclosed code fragments in a flexible way.

However, the usage of preprocessor code also brings negative effects. Liebig et al. [11] analyzed the preprocessor code of forty open source SPLs and addressed the issues of #ifdef nesting, tangling and scattering. Besides, they also found that some preprocessor code is undisciplined [12]. These issues obfuscate the normal code layout, making the core assets overly complex and difficult to understand. For instance, Fig. 1 shows a snippet of core code from industry, where #ifdef
blocks are colored according to their #ifdef statement using Feature Commander [7]. These #ifdef blocks are nested and scattered across the core code, and sometimes multiple macro constants are tangled in #ifdef statements. Such #ifdef code is difficult to understand even with the support of #ifdef coloring.

During SPL evolution, this understanding problem with preprocessor code in core assets further causes maintenance problems. Experience from industry [2][16] shows that variation points (e.g., #ifdef blocks in CC) and their interdependencies are often inexplicit, and such information is difficult to detect from code manually. Therefore, once there is a change in the core assets, it is difficult to identify the change impact [13]. As a SPL evolves, there might be obsolete or even erroneous code in core assets, and fixing such problem is time-consuming. This causes a serious impact on SPL productivity and puts the expected SPL advantages more and more at risk. Refactoring the core assets would be appropriate and feasible, but too expensive if done manually. Sometimes there are several code refactoring activities possible, but it remains unclear what is the best one because the respective methods and tools are still in infancy.

II. SOLUTION FRAMEWORK

Given the aforementioned challenges both in problem space and in solution space, we present a solution framework named RECoVar (Reverse Engineering Configurations and Variability). In general, our objective is to increase SPL productivity and avoid erosion of variability artifacts during SPL evolution. To be specific, our idea is to recover various variability information using reverse engineering approaches and make it traceable with core assets and product configurations, so that variability specifications and realizations can be maintained in an efficient way. As illustrated in Fig. 2, the ReCoVar framework includes two approaches: a) code-based variability model extraction [20]; b) feature correlation mining [21]. The first approach extracts a variability model from preprocessor code, while the second approach identifies complex feature correlations from SPL configurations. We will introduce these two approaches respectively in the following sub-sections.

A. Code-based Variability Model Extraction

In practice SPL variability is often implemented using CC, where variant code fragments in core assets are conditionally compiled in #ifdef blocks (which could be nested). Each #ifdef block uses one or more macro constants in its #ifdef statement. These macro constants are defined in product code with various values in order to flexibly enable or disable corresponding #ifdef blocks. Therefore, each #ifdef block is considered as a variation point, while a macro constant referenced in the #ifdef block is considered as a variability element. In this approach, we analyze variability-related preprocessor code both in core assets and in product realizations, and extract a realization variability model including 1) variability elements; 2) variation points; and 3) a variability tree. Moreover, we also conduct quantitative measurement on extracted variability elements and variation points respectively.

First, variability elements, i.e., macro constants, are extracted from the #defines of existing product code with their values if defined. The number of values indicates the cardinality of this variability element. Second, we parse the #ifdef blocks in core assets to identify variation points and measure their occurrence, complexity, code impact, and so on. We also measure the usage of macro constants in #ifdef statements in order to investigate the relationship between variability elements and variation points. Third, we further extract interdependencies between variation points in a tree structure. As shown in Fig. 3, #ifdef blocks in core code are mapped to a variability tree based on #ifdef nesting, in which nested #ifdef blocks are children nodes. If there is an #else or
#elif statement in an #ifdef block, then this variation point contains positive and negative branches of code fragments. It is similar with the notion of alternative groups in a feature model.

Fig 4. Visualization of a Variability Tree

The variability tree is stored in XML format and visualized using the Treeviz tool [19]. Fig. 4 demonstrates the visualization of a variability tree derived from an industrial SPL in embedded system domain. Each tree node represents homogenous variation points (#ifdef blocks with the same #ifdef statement) at a certain nesting level. The tree nodes are visualized with different colors, which indicate one of the metrics of the variation points. For instance, the color in Fig. 4 indicates refCount, which is defined as the number of associated #ifdef blocks in each tree node. The metric of refCount reflects the impact of homogenous variation points in the core code. Therefore, the variability extraction and visualization helps to understand variability realizations and facilitate SPL maintenance in practice.

B. Complex Feature Correlation Mining

Besides extracting variability information from realization artifacts in the solution space, we also consider such extraction from the problem space. As the variability model is often used to derive product configurations for application engineering, feature correlations in a variability model are satisfied in the derived product configurations. Given the problem of missing or inconsistent feature correlations (#ifdef blocks with the same #ifdef statement) at a certain nesting level, the tree nodes are visualized with different colors, which indicate one of the metrics of the variation points. For instance, the color in Fig. 4 indicates refCount, which is defined as the number of associated #ifdef blocks in each tree node. The metric of refCount reflects the impact of homogenous variation points in the core code. Therefore, the variability extraction and visualization helps to understand variability realizations and facilitate SPL maintenance in practice.

Fig 5. Feature Correlation Mining Approach

TABLE I. EXAMPLE CONFIGURATION MATRIX

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>defined</td>
<td>defined</td>
<td>defined</td>
</tr>
<tr>
<td>F2</td>
<td>30</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>F3</td>
<td>512</td>
<td>512</td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td>EUR</td>
<td>USD</td>
<td></td>
</tr>
</tbody>
</table>

During the correlation mining process, we use data mining techniques to identify significant association rules as potential feature correlations in two steps. In the first step, frequent itemsets of feature assignments are calculated that satisfy a specified threshold of minimum support. A frequent itemset is a set of feature assignments that exists with at least a specified percentage (called support) across all transactions (products). In the second step, these frequent itemsets are used to identify strong association rules that satisfy a specified threshold of minimum confidence. The confidence of an association rule is defined as the number of products that satisfy the entire rule relative to the number of products that only satisfy the antecedent of the rule. The correlation mining process is implemented by the Orange tool [15].

After correlation mining, the identified association rules have to be pruned because there are many sub-rules that can be implied by a parent rule with equal or larger confidence. Therefore, these sub-rules do not provide any predictive advantage and should be removed from the set of feature correlations. From our experience, most derived association rules are redundant and can be pruned automatically. Finally, each feature correlation is checked manually by domain experts to decide whether there exists a semantic relationship or that is only a coincidence. Moreover, since the threshold of minimum support and minimum confidence might be adjusted, the entire correlation mining process can be conducted iteratively. This approach is presented in details in another paper [21].

III. RELATED WORK

There are several related works towards reverse engineering SPL variability. Czarnecki et al. [4][5] introduced the notion of Probabilistic Feature Model (PFM) and extract hard as well as soft feature constraints from product configurations. These constraints do not involve feature values, so that the configuration matrix is transformed into a dataset that is suitable for correlation mining.
used to build an implication graph, which is DAG and similar to a feature model. Lora-Michiels et al. [14] proposed to identify structural and transversal feature dependencies from product configurations. In their approach, pairwise feature correlations (between single features) are calculated using the Apriori Algorithm. Acher et al. [1] presented an approach to reverse engineering feature models. However, additional domain knowledge (e.g., feature hierarchies) is required during model generation. Since the same feature constraints in propositional logic can be mapped to different feature model structures, such domain knowledge is necessary. While most approaches conduct variability extraction from product configurations in the problem space, it is still an open question how to reverse engineer variability automatically from the solution space.

IV. CONCLUSIONS

In this paper, we present a solution framework named RECoVar to extract a variability model from SPL realizations and product configurations. After investigating how variability is implemented in core assets and presented in products, we conduct an automated analysis to identify variability elements and variation points in preprocessor. The variation points are then visualized to help understanding their interdependencies and code impact. Moreover, complex feature correlations are extracted from product configurations to supplement the variability model. Although it is difficult to fully understand the semantic of variability realizations, the extracted variability information helps to maintain variability specifications and realizations in SPL evolution.

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


