Code-based Variability Model Extraction for Software Product Line Improvement

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
Successful Software Product Lines (SPLs) evolve over time. However, one practical problem is that during SPL evolution the core assets, especially the code, tend to become complicated and difficult to understand, use, and maintain. Typically, more and more problems arise over time with implicit or already lost adaptation knowledge about the interdependencies of the different system variants and the supported variability. In this paper, we present a model-based SPL improvement process that analyzes existing large-scale SPL reuse infrastructure to identify improvement potential with respective metrics. Since Conditional Compilation (CC) is one of the most widely used mechanisms to implement variability, we parse variability-related facts from preprocessor code. Then we automatically extract an implementation variability model, including product configuration and variation points that are structured in a hierarchical variability tree. The extraction process is presented with concrete measurement results from an industrial case study.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering.* D.2.8 Metrics – *complexity measures.* D.2.9: Management – *software quality assurance.* D.2.13: Reusable Software – d*omain engineering.*

General Terms
Measurement, Design, Management.

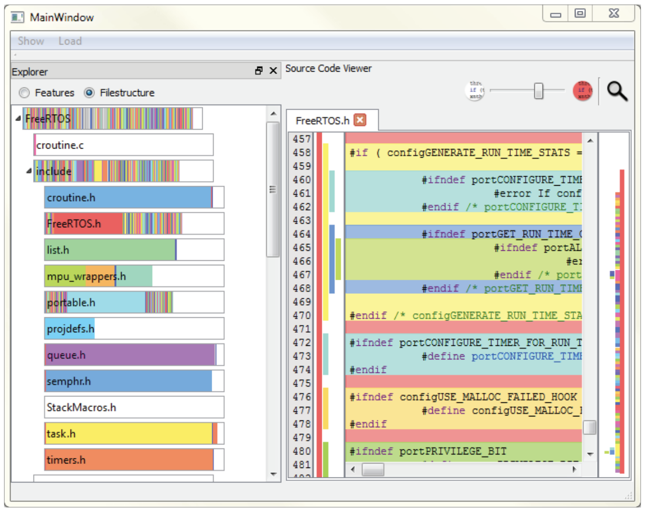
Keywords
Software product line maintenance, variability model, conditional compilation.

1. INTRODUCTION
Nowadays the development of Software Product Lines (SPLs) is often conducted in an incremental way, in which artifacts including specification and implementation evolve both in space and in time [9][13][20]. The SPL specification includes in one way or the other a variability model that defines variability and interlinks it with respective variation points in the SPL core assets to support the adaptation of the latter in SPL evolution. Experience in evolving SPLs shows the problems that 1) the existing variability model becomes inconsistent with evolved SPL core assets and 2) the variability realization in the core assets becomes overly complex. This 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 refactoring activities possible, but it remains unclear what is the best one. The investigation of Patzke et al. on industrial SPLs [15] reveals that in practice the variability model tends to be missing or incomplete due to ambiguous variation points and inexplicit variant elements. Moreover, Tartler et al. [21] also addressed the problem of inconsistency between variability model and variability realization in Linux kernel.

Especially in C/C++ based systems, Conditional Compilation (CC) is one of the most frequently used mechanisms to implement variability in SPLs [10][21]. In this case, macro constants are defined in products configurations and used in #ifdef[1] statements in code core assets. The C preprocessor then adapts the core assets by enabling or disabling enclosed code fragments. Thus each #ifdef block, controlled by one or more macro constants in the #ifdef statement, can be considered as a variation point. Since different #ifdef blocks may have identical #ifdef statement, logically they belong to the same variation point. Therefore, a variation point implemented by CC may involve multiple locations in variability realization, and the mapping between abstract variability information (e.g., variation points) and variability realization needs to be documented in a traceable way.

While the variability mechanisms offered by CC are fairly simple, experience with large-scale and evolving SPLs shows that the respective variability realization tends to become overly complex, due to nested, tangled, and scattered variability realization as shown in Fig. 1 (from our previous work [24]), where #ifdef blocks in FreeRTOS [4] are colored using Feature Commander [3]. In Fig. 1 the colored #ifdef blocks are nested with maximum four levels. Moreover, multiple macro constants are used and tangled in the same conditional statements of a #ifdef blocks, and the same constant is also used in multiple #ifdef blocks scattering in different files. This situation is called #ifdef hell in [19] and discussed in [10]. Even worse, the corresponding variability models are often incomplete, inconsistent, or missing, which makes it even more difficult to understand and maintain variability realization of large-scale SPLs in practice.

[1] In this paper #ifdef also implies similar directives, e.g., #if and #elif.



```
#if ( configGENERATE_RUN_TIME_STATS =

        #ifndef portCONFIGURE_TIM
                #error If conf
        #endif /* portCONFIGURE_T

        #ifndef portGET_RUN_TIME_
                #ifndef portAL
                        #er
                #endif /* port
        #endif /* portGET_RUN_TIM

#endif /* configGENERATE_RUN_TIME_STA

#ifndef portCONFIGURE_TIMER_FOR_RUN_T
        #define portCONFIGURE_TIM
#endif

#ifndef configUSE_MALLOC_FAILED_HOOK
        #define configUSE_MALLOC_
#endif

#ifndef portPRIVILEGE_BIT
```

Figure 1. Colored #ifdef blocks in SPL Code.

Unfortunately, improving this situation is not trivial at all and respective methods and tools are still in infancy. Therefore the objective of our work is to provide a systematic approach that help to identify improvement potential in evolving SPLs and plan respective refactoring in a goal-oriented manner. In this paper, an SPL improvement process is presented based on the idea of the reflexion model [14] that derives certain abstraction from software realizations. In our improvement process, we identify SPL improvement goals and respective questions and metrics following the GQM approach [18], which contributes to defining high-level goals and supporting them with low-level analysis and measurement. Then we extract an implementation variability model by analyzing existing large-scale SPL reuse infrastructure. Due to the pervasion of CC, we focus on preprocessor code in a first step. By extracting variability-related facts from preprocessor statements in the code bases, we build up a hierarchical model of the variability realization automatically and identify improvement potentials with respective metrics therein. This code-based variability model extraction is introduced with concrete results from an industrial case study.

The paper is organized as follows. The overall SPL improvement process with underlying models is presented in Section 2. As an instantiation of the process, the automatic analysis of variability realization in CC (i.e., the preprocessor code) is introduced in Section 3. Then the extraction of the implementation variability model with related measurement is discussed in section 4. Finally related work is discussed in section 5, and conclusion is presented in section 6.

2. Model-based Product Line Improvement

Reuse infrastructure of SPL typically consists of different types of core assets (e.g. code, specification, design, tests, and documentation artifacts), in which different variability mechanisms (e.g. CC, Conditional Execution, and Module Replacement) are used to realize the required variability. Holistic improvement considerations hence need to be based on the facts we get from the various assets (e.g. what is the Fan-Out of a variability A on the core assets), the usage scenarios of the assets (how often must core asset B be understood and modified by a SPL engineer), and the respective costs.

In order to facilitate these considerations, especially the exploration of possible refactoring (e.g. changing the structure of the core assets to localize the variability realization in order to reduce QA effort) and their expected effects in different SPL settings, we propose to base them not directly on the concrete SPL artifacts, but on a corresponding reflexion model [14] of the SPL (PLRM). This model, on the one hand, should abstract from irrelevant details and provide us with just the right amount of information via respective views. On the other hand, it should enable necessary refactorings on the artefacts via the PLRM. Following this approach, we target some (semi-)automated and continuous SPL improvement in a mid-term perspective. Obviously, neither the SPL variability model nor the SPL architecture is sufficient to this end.

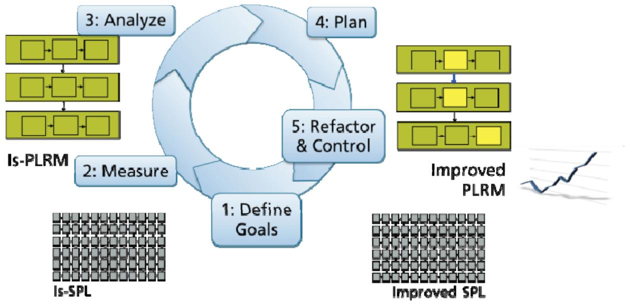


Figure 2. Model-based SPL Improvement Process.

Fig. 2 provides an SPL improvement process that reveals the intended usage of the PLRM. To start with, the PLRM of existing SPL infrastructure and system instances (is-SPL) is built up using the tools of Fraunhofer Variant Analysis [2] and the Variability Analysis introduced in this paper. Then we get an is-PLRM representing the existing SPL commonality as well as variability specification and realization with derived members. Based on the is-PLRM we can run improvement potential analysis and discuss specific concerns with the respective SPL engineers. The space of possible refactoring can be explored in the PLRM and investigated for refactoring activities that meet current and past demands in the best possible way. The effect of the refactoring activities can be simulated and documented in an improved PLRM, and quantitative improvement results can be estimated. After a refactoring activity has been selected and planned, the corresponding execution steps can be conducted semi-automatically on the artifact level. To our best knowledge, there is no comparable approach followed in research and development for SPLs.

A plethora of different metrics can be easily measured from SPL repositories. Some of them might be helpful indicators for variability improvement, others are not. In order to systematically research efficient and effective analysis approaches in the SPL context we have followed the Goal-Question-Metric (GQM) approach [18] to get a clear understanding of the improvement goals and derive the corresponding metrics from them. The GQM model supports goal-oriented software measurement with three levels, i.e., goal, question, and metric. The goal level defines tasks to be achieved, which are refined into questions on the question level to assess the achievement of these goals. Moreover, related metrics are defined on the metric level to answer each question in a quantitative way.

Given the practical problem that variability realization using CC in code assets is often complicated and thus difficult to understand and maintain, while respective variability models are missing or inconsistent, we have developed the GQM model as illustrated in Fig. 3. Our overall goal is to improve variability-related activities in the context of evolving SPLs. The activities include SPL configuration, maintenance, and quality assurance (QA). To achieve these goals, various variability-related questions are addressed, and associated metrics are provided to answer these questions in a quantitative way.

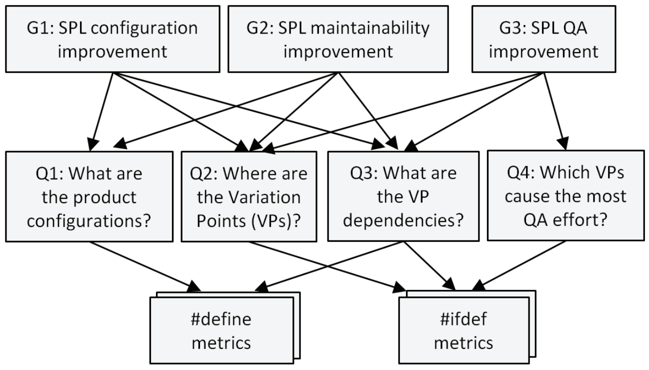

G1: SPL configuration improvement
G2: SPL maintainability improvement
G3: SPL QA improvement
Q1: What are the product configurations?
Q2: Where are the Variation Points (VPs)?
Q3: What are the VP dependencies?
Q4: Which VPs cause the most QA effort?
#define metrics
#ifdef metrics

Figure 3. The GQM Model of SPL Improvement.

The goal of SPL configuration improvement (G1) is to facilitate the product configuration process by considering interdependencies that originate from the SPL code core assets. To achieve this goal, both variation points in the core code (Q2) and their interdependencies (Q3) need to be explicit to guide the configuration process. Moreover, it is also necessary to extract configuration elements (e.g. macro constants in CC) from their definition in existing products (Q1), in which both the name and value of each configuration element can be extracted. The values of existing configuration elements can be used as candidate values for configuring a new product. Furthermore implicit interdependencies between configuration elements can be derived from the existing product configurations and validated with respective experts. Finally, the goal (G1) is achieved if a new product can be configured efficiently based on the extracted variability information.

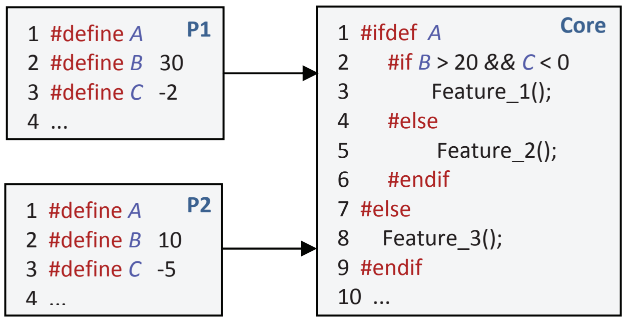
Another goal (G2) is SPL maintenance improvement. To this end, the impact of code changes in the SPL and its members needs to be identified and assessed and the respective changes needs to be conducted correspondingly. The detection of change impact requires dependency analysis on variability realization. Since the dependency exists between the definition of existing product configuration (Q1) and its usage in variation points (Q2), both of them need to be available. In this way, if a variant element in the core asset is changed, the products selecting this variant can be identified and their instantiated assets can be updated correspondingly. Moreover, the code dependencies between variation points (Q3) also need to be extracted. Finally, the goal (G2) is achieved if the change impact can be identified correctly in SPL code.

The third goal (G3) is to improve the QA process by detecting certain variation points. The idea is first extracting variation points (Q2) and their interdependencies (Q3), and then finding out the most effort-consuming variation points in QA process (Q4). The high QA effort of variation points may be caused by either intensive impact on code core assets (e.g., a lot #ifdef blocks with the same conditional statement) or high complexity (e.g., long and over-nested #ifdef blocks). This goal is achieved if such variation points are identified from core assets, and then a certain refactoring can be planned and performed on these variation points to reduce the QA effort. The refactoring may require domain knowledge and will be discussed in our future work.

Given the four questions presented in Fig. 3, metrics of both #defines and #ifdef blocks in preprocessor code are provided in the GQM model. While the definition of macro constants (i.e., #defines) is measured to identify product configuration (Q1), their usage (i.e., #ifdefs) is measured to identify variation points (Q2). To measure interdependencies between variation points (Q3), the number of #ifdef blocks using the same macro constant (i.e., #References) is calculated as a #define metric. Moreover, the hierarchical dependency between nested variation points (#ifdefs) is also calculated as an #ifdef metric. Furthermore, the complexity of variation points is measured by some #ifdef metrics to indicate their QA effort (Q4). The two groups of metrics will be discussed in section 3 in detail.

3. Variability Code Parsing

In our approach, variability-specific code assets are used as input for extracting variability information and identifying SPL improvement potential. Since CC is a frequently used mechanism for implementing variability in SPLs, we focus on parsing variability code assets implemented by CC.



```
#define A
#define B  30
#define C  -2
...
```

```
#define A
#define B  10
#define C  -5
...
```

```
#ifdef A
    #if B > 20 && C < 0
        Feature_1();
    #else
        Feature_2();
    #endif
#else
    Feature_3();
#endif
...
```

Figure 4. An Example of Variability Code in CC.

In CC, a macro constant is defined with the directive #define and is normally assigned a value. Then the constant can be used in an #ifdef statement. To implement variability in a SPL, constant definition usually occurs in product code, while constant usage usually occurs in core code, as shown in Fig. 4. In this way, when the core code is reused and instantiated by products, different variant elements in #ifdef blocks are conditionally compiled into corresponding product code.

3.1 Parsing #defines

In our approach, the definition of all macro constants (i.e., #defines) in product code is automatically parsed using Python scripts. If the constant is assigned a value, the value is stored. Assuming that all constant names are unique in a SPL, we finally collect a list of constant names with different values if assigned. This result is considered as product configuration and will be used for further measurement. It is also possible that a constant is defined without assigning any value, which can be used in #ifdef or #ifndef statements. Note that the C/C++ language also supports the #undef directive that deletes a previously defined macro constant. The #undef statements are also parsed in our approach, although they are rather less used in practice.

3.2 Parsing #ifdefs

Similar to parsing #defines, the usage of macro constants in #ifdef blocks is also parsed. However, the code parsing process is more complicated. An #ifdef block starts with different conditional directives (e.g., #ifdef, #if, and #elif), and ends with #endif. Such a block may include both positive and negative branches separated with #else. During compilation the branches are selected and compiled depending on the Boolean value of the conditional expressions in corresponding #ifdef statements. Based on the defined values of constants used in these #ifdef expressions, the Boolean value can be calculated automatically and thus the selected branches are detected. In our approach, we use Python scripts to parse #ifdef blocks. Note that the compound conditional directive #elif, equivalent to a concatenation of #else and #if in C/C++ language, is also parsed. For each #ifdef block we detect its length, parent #ifdef block (if any) and existed conditional branches. We will further support the automatic evaluation of #ifdef expressions to detect the branch selection of a given product.

```
ParseCode(code)
{
    refList, tempRefList;
    for each line in code {
      ... // parsing #defines
      if(line.match("#ifdef|#ifndef|#if|#elif")) {
          ref = new ConstRef(line);
          ref.type = OPTIONAL;
          if(line.match("#elif"))
              tempRefList[-1].type = ALTERNATIVE;
          ref.parent = tempRefList[-1]; // last element
          if(ref.parent.type == OPTIONAL)
              ref.branch = POSITIVE;
          else
              ref.branch = NEGATIVE;
          tempRefList.append(ref);
      }
      else if(line.match("#else")) {
          tempRefList[-1].type = ALTERNATIVE;
      else if(line.match("#endif")) {
          ref = tempRefList[-1];
          do {
              ref.end = line.end;
              refList.append(ref);
              tempRefList.delete(ref);
              ref = tempRefList[-1];
          } while(ref.prefix == "elif");
       } // elseif
    } // for each line in code
} // ParseCode
```

Figure 5. The Algorithm for parsing #ifdefs.

Fig. 5 shows the pseudo code of our algorithm for parsing #ifdefs. For each #ifdef block, basic information including block location, length, and used macro constants is stored. Since an #ifdef block can be mapped to a variation point, it is either optional (only with the positive branch) or alternative (with both positive and negative branches). Furthermore, if an #ifdef block is nested into another one, their hierarchical relationship is stored. Note that since #elif is equivalent to the concatenation of #else and #if, an #elif block is considered as a nested child of the previous #ifdef block although they share the same #endif at the block end.

Moreover, due to the conditional branching in CC, an #ifdef block may be nested in either the positive branch or the negative branch of its parent #ifdef block. Since in practice most #ifdefs have only one positive branch, an #ifdef block is considered as an optional variation point by default, and the branch position of nested #ifdef blocks is set as positive. When an #else or an #elif statement is matched, the current #ifdef block is considered as an alternative variation point, and the branch position of nested #ifdef blocks is set as negative. Therefore, if the positive branch of a parent #ifdef block is selected during product configuration, its nested #ifdef blocks in the negative branch can be ignored. In this way, the efficiency of product configuration can be improved.

4. Variability Model Extraction

In variability code assets implemented by CC, variant elements are conditionally compiled in an #ifdef block, which is considered as a variation point. Since macro constants are used in #ifdef statements to decide whether the enclosed code is enabled and compiled, defined values of these constants are considered as product configuration. Based on the definition and usage of macro constants parsed from preprocessor code, we extract an implementation variability model consisting of existing product configuration and a variability tree with variation points. Moreover, quantitative measurement on extracted configuration and variation points is performed to help understanding SPL variability and identifying improvement potential.

In order to demonstrate the variability model extraction and relevant measurement on product configuration and variation points, we apply our approach to FreeRTOS [4], an embedded SPL supporting diverse variant products and being used in industry. To implement variability, the mechanism of CC is used in FreeRTOS that includes a large size of preprocessor code (21,129 LOC). In this paper, we will demonstrate the variability extraction and measurement using the FreeRTOS example.

4.1 Extracting Product Configuration

In section 3.1, we introduced the parsing process of #defines in product code which derives a list of constant names and values. These constants are considered as product configuration and used to select variant code elements enclosed in #ifdef blocks. Assuming all constant names are unique, we group all definition of the same constant and measure their definition and usage.

Table 1. Metrics of Macro Constants

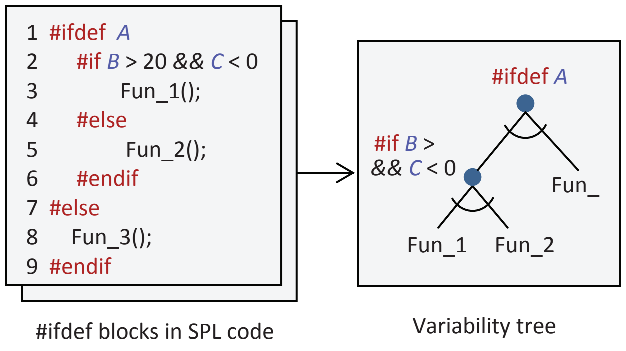
Constant	Values	Cardinality	#References
portSTACK_	1 | -1	2	8
portUSING_	1 | 0	2	8
portBYTE_A	1 | 8 | 2 | 4	4	7
configKERN	1 | 0 | 255	3	7
portLARGE	3	1	6
portCRITIC	1 | 0	2	5
portCOMPAC	2	1	5
portSMALL	0	1	4
portPRESCA	0x00 | 16	2	4

Table 1 shows the measurement results of macro constants which are defined in product code of FreeRTOS. For the sake of brevity only partial constants are listed in this paper, and only first ten letters of their constant names are presented. Each constant may have multiple definitions with different values, separated by "|". The number of the values is considered as cardinality of the constant, which implies various configuration values used in all current products. Moreover, combining with the results of #ifdef analysis we also calculate the number of #ifdef blocks that use a given constant as a reference. The number of referring #ifdef blocks (#References) indicates the code impact of the constant.

As previously mentioned, some constants can be defined without any value, and the constants with only one value (cardinality =1) could be either used in variability-specific #ifdefs, or used in local #ifdefs of a product. Besides, in some industrial SPLs variability-specific constants are defined with a certain naming convention (e.g., portXXX or configXXX), which helps to distinguish between variability-specific constants and local constants. We also notice that in CC product configuration may not only occur in macro constants of source code asseets, but may also exist in configuration files, Makefiles, etc. These configuration elements can be automatically parsed and extracted in the same way. Furthermore, it is possible that some of the extracted macro constants are not used in any #ifdef blocks (#references = 0). If such constants are defined in multiple products, they are probably configuration elements using other variability mechanisms (e.g., Conditional Execution).

4.2 Extracting Variation Points

In our approach an #ifdef block is mapped to a variation point. If multiple #ifdef blocks have the same statement, logically they belong to the same variation point. It means that a variation point may be related to multiple #ifdef blocks in variability code. Based on our #ifdef parsing, all #ifdef blocks are identified and their basic information (e.g., #ifdef statement, length, and nesting) is collected. Then these #ifdef blocks are mapped to variation points with a tree structure, as shown in Fig. 6. In this section we introduce the extraction of variation points, while the extraction of the tree hierarchy will be discussed in section 4.3.



```
#ifdef A
    #if B > 20 && C < 0
        Fun_1();
    #else
        Fun_2();
    #endif
#else
    Fun_3();
#endif
```

Figure 6. Mapping #ifdef Blocks to Variation Points.

Since #ifdef blocks with the same statement belong to the same logical variation point, it is necessary to compare #ifdef statements and group #ifdef blocks with the same statement. However, we notice that some syntactically different #ifdef statements are actually equivalent, e.g., "#if (A>0)" and "#if (int) 0<A". At the moment, the detection of equivalent #ifdef statements in our approach is limited on the syntactic level. The semantic analysis and comparison of #ifdef statements can be achieved in theory, but it would require thorough comprehension of the entire core code, which is highly expensive and time-consuming. To mitigate this problem, the #ifdef statements are literally formatted before comparison. For instance, spaces in the code are unified, and multiple code lines of a splitted statement are concatenated.

Parsing the #ifdef blocks of FreeRTOS, we identified a list of variation points and conducted relevant measurement as shown in Table 2. For the sake of brevity only major variation points (with strong code impact) are listed, and only first ten letters[2] of their #ifdef statements are presented. For each variation point we use the #ifdef statement as its name, and measure its code impact and complexity. The code impact is indicated by the number of involved #ifdef blocks (refCount) and the number of involved files of these blocks (fileCount). The complexity is indicated by the average nesting level of these #ifdef blocks (avgLevel) and the sum of block length of these blocks (size). While the metrics refCount and fileCount help to locate the code impact of a variation point in the SPL, the metrics avgLevel and size help to identify extremely complicated variation points that need to be improved by code refactoring. For instance, the variation point in the second row (#ifndef PO) has a huge size of preprocessor code, which could be difficult to understand during development and maintenance.

Table 2. Metrics of Variation Points

Statement	refCount	fileCount	avgLevel	size (LOC)
#ifdef__cp	112	56	2	112
#ifndef PO	63	63	1	7734
#if conf˜1	63	63	2	315
#if conf˜2	42	37	1.07	434
#if conf˜3	16	3	1.06	172
#if conf˜4	15	11	1.33	829
#if conf˜5	15	3	1.27	205
#ifdef THU	13	13	1.38	245

4.3 Extracting Variability Tree

In section 4.2, we discussed the extraction from #ifdef blocks to variation points. In fact, the extracted variation points also include hierarchical information based on #ifdef nesting. For instance, in Fig. 4 the variation point "#ifdef A" is the parent of the variation point "#if B > 20 && C < 0" because "#if B > 20 && C < 0" is nested within "#ifdef A" in code. This hierarchical information can be used to establish a tree structure of the variation points. The variability tree indicates dependencies between variation points and modularizes variation points with a hierarchical structure. Therefore, it helps to answer the question Q3 in Fig. 3 and finally contributes to the related improvement goals in the GQM model. Since the variability tree is extracted only from #ifdef blocks, it does not express any commonality information (i.e., mandatory features) as supported in typical variability models.

In order to create the tree structure, the #ifdef nesting level is mapped to the tree node level. Thus each node in the tree is a variation point, and its children nodes are the nested variation points if existed. However, a direct mapping from a variation point to a tree node may cause cycles in the tree, because different variation points may include the same child variation point at different code locations. For instance, there might be an #if A block enclosing an #if C block in one file, and an #if B block enclosing another #if C block in another file. Once there is a cycle in the variability tree, the order of corresponding nesting level cannot be recognized, and thus there will be neither an entrance nor an exit for configuration in the cycle.

[2] Statements with the same first ten letters are distinguished with serial numbers in the end.

To avoid cycles in the variability tree, each tree node is mapped to a variation point at a specific nesting level. This means that the same variation point (with the same #ifdef statement) may occur in multiple tree nodes, indicating #ifdef blocks with different nesting path (either different parents or different nesting levels). During product configuration, the decision made on #ifdef blocks at one tree node can be propagated to related #ifdef blocks (with the same #ifdef statement) at the other tree nodes.

Moreover, based on the parsed branch position of #ifdef blocks (discussed in section 3.2), the variability tree also distinguishes variation points in positive branches from those in negative branches. Therefore when the #ifdef branch of a variation point is selected during product configuration, its nested children in the other branch can be skipped from the configuration process, because the corresponding nested #ifdef blocks will not be compiled in any case. In this way, the branch position of variation points in the variability tree contributes to improving the efficiency of product configuration.

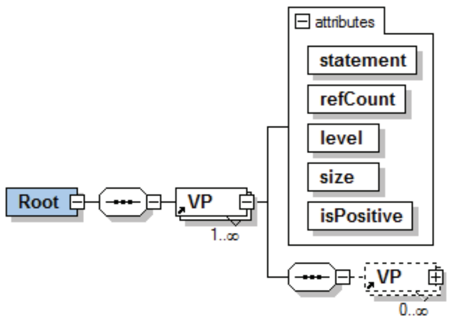


Figure 7. XML Schema of the Variability Tree.

Finally the variability tree is extracted from #ifdef blocks based on their #ifdef statement, nesting, and branch position. The tree is documented in the XML format, and the corresponding schema is shown in Fig. 7. There is an artificial root node consisting of a set of variation points. Each variation point consists of five attributes and a set of nested variation points as children nodes (if any). The attributes include the #ifdef statement (statement), number of involved #ifdef blocks (refCount), nesting level of these blocks (level), sum of block length of these blocks (size), and the branch position (isPositive).

We extract the variability tree from the core code of FreeRTOS, and visualize the XML file by adapting an open source visualization tool Treeviz [22]. Fig. 8 illustrates the visualization of the extracted variability tree that includes 159 nodes, and the highest nesting level found in the code is three. The visualization supports coloring of tree nodes based the value of a given node attribute from the XML schema. For instance, in Fig. 8 the tree nodes are colored based on the value of node size. Most leaf nodes are in blue color, indicating that the size of nested #ifdef blocks is normally small. The root node is in red color with a size of 5,927 LOC, because it consists of all #ifdef blocks in the core code. The yellow nodes are variation points with medium size. The tree visualization demonstrates variation points with their interdependencies in a SPL. Each node of variation point includes metrics to indicate their complexity and code impact. It provides navigation support in product configuration and also facilitates code maintenance and QA in practice.

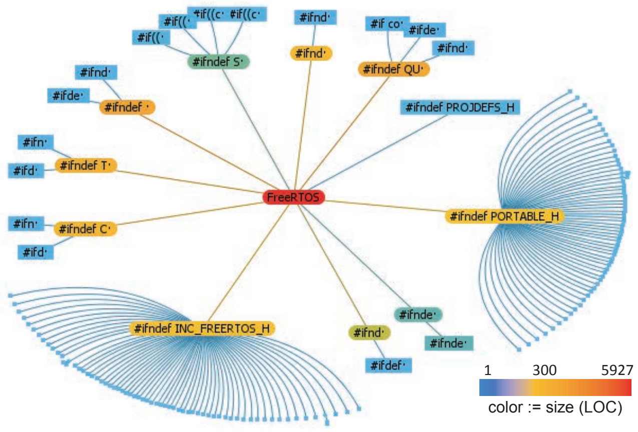


Figure 8. Visualization of the Variability Tree.

5. Related Work

Due to the wide application of CC, various related issues have been addressed in the field. Liebig et al. [10] investigated the use of CC for implementing variability in forty configurable open source systems (they called them SPLs), and measured #ifdef blocks (feature extensions) in term of their complexity, type, etc. Besides, Kästner et al. [6] discussed variability realization at different granularities and their effects, while Liebig et al. [11] addressed another problem of undisciplined preprocessor annotations in CC. These studies show that preprocessor code used in large-scale SPL realizations is often very complicated and especially difficult for human to understand and maintain. However, none of them focused on the extraction of abstract variability information (e.g., variation points) from preprocessor code.

To the best of our knowledge, there are few studies in the area of code-based variability extraction and recovery. Lozano [12] investigated current approaches for detecting variability concepts in code, and it is concluded that mining variation points (and variants) is still an open area with high potential. She et al. [16] proposed a reverse engineering procedure to create a hierarchical feature tree based on the text similarity of feature descriptions in text and requires domain knowledge. Cem et al. [1] mapped preprocessor code to a source code variability model, and proposed a forward engineering approach to automatically generating and updating source code according to operations on the variability model. Galindo et al. [5] presented an automatic approach to map the Debian package dependency language to propositional formulas and consider them as the variability model for Debian based linux distributions. Different from preprocessor code analysis, in that case the variability information and interdependencies of software packages are explicitly specified in configuration files. Sincero et al. [17] extracted Boolean formula from preprocessor code in order to check consistency between the existing variability model and its implementation. Besides, our early research in variability extraction and improvement from preprocessor code is presented in [24].

In addition, Kästner et al. [8] proposed a partial preprocessing approach to resolving macros and file inclusions in order to separate them from CC in preprocessor code. This approach is the first step of their TypeChef project [23], which aims at detecting syntax errors and type errors in preprocessor code. Although the partial preprocessor helps to separate variability implementation from other facilities of the C preprocessor, their code parsing process only focuses on verifying individual preprocessor directives and does not identify abstract variability information of the entire SPL.

6. Conclusion and Future Work

In this paper, we present a SPL improvement process driven by the idea of reflexion model, and extract a variability model from preprocessor code with respective metrics. In the improvement process, SPL improvement potential is investigated and defined in a GQM model, which includes goals, questions, and metrics in a top-down manner. To achieve the improvement goals, we conduct the extraction and measurement on code level. The extraction is based on the automatic parsing of variability code implemented by CC, and the extraction result is a variability model including product configuration and variation points with a hierarchical tree structure, i.e., the variability tree. Moreover, the extracted product configuration and variation points are measured to identify their code impact and complexity. According to the defined GQM model, the extracted variability model with measurement can be used to facilitate product configuration and improve code maintenance and QA in SPL development.

In the future, we will focus on evaluating of the identified improvement potential with empirical studies. Based on the extracted variability model, we will further investigate its application in SPL improvement on various industrial SPL systems and measure the improvement in a quantitative way. Moreover, we are also interested to extend and improve our variability extraction approach. One idea is to support other mechanisms than CC, e.g., Conditional Execution. It is noticed that multiple mechanisms are often combined to implement variability in practice. Therefore only parsing the preprocessor code may be insufficient to obtain complete variability information in a SPL. Another idea is to fill the gap between the implementation variability model and the abstract feature model. Since the implementation model is extracted from code assets, it includes a large amount of low-level information and therefore may be difficult to understand for customers in product configuration. The abstraction of the implementation variability model requires semantic comprehension of SPL code, which probably can be achieved manually with domain knowledge.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] J. M. Cem, C. Fuss, R. Zimmermann, and I. Aktas, "Model-driven support for source code variability in automotive software engineering," 7. Workshop Automotive Software Engineering (ASE 09), vol. Lecture Notes in Informatics (LNI), 2009.

[2] S. Duszynski, J. Knodel, and M. Becker, "Analyzing the source code of multiple software variants for reuse potential," Reverse Engineering, Working Conference on, vol. 0, pp. 303-307, 2011.

[3] FeatureCommander. http://wwwiti.cs.uni-magdeburg.de/~feigensp/xenomai/. Retrieved on May 25th, 2012.

[4] The FreeRTOS project. http://www.freertos.org/.

[5] J. Galindo, D. Benavides, and S. Segura, "Debian packages repositories as software product line models. towards automated analysis," in Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications (ACoTA), Sep. 2010, pp. 29-34.

[6] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in Proceedings of the 30th international conference on Software engineering, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 311-320.

[7] C. Kästner, "Virtual separation of concerns: Toward preprocessors 2.0," Ph.D. dissertation, 2010.

[8] C. Kästner, P. G. Giarrusso, and K. Ostermann, "Partial preprocessing c code for variability analysis," in Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems, ser. VaMoS '11. New York, NY, USA: ACM, 2011, pp. 127-136.

[9] C.W. Krueger, "New Methods behind a New Generation of Software Product Line Successes", In K.C. Kang, V. Sugumaran, and S. Park, "Applied Software Product Line Engineering", Auerbach Publications, 2010, pp. 39-60.

[10] 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.

[11] 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.

[12] A. Lozano, "An overview of techniques for detecting software variability concepts in source code," in ER Workshops, ser. Lecture Notes in Computer Science, O. D. Troyer, C. B. Medeiros, R. Billen, P. Hallot, A. Simitsis, and H. V. Mingroot, Eds., vol. 6999. Springer, 2011, pp. 141-150.

[13] J. D. McGregor, "The evolution of product line assets," Tech. Rep., 2003.

[14] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," IEEE Trans. Softw. Eng., vol. 27, no. 4, pp. 364-380, Apr. 2001.

[15] T. Patzke, M. Becker, M. Steffens, K. Sierszecki, J. E. Savolainen, T. Fogdal, "Identifying Improvement Potential in Evolving Product Line Infrastructures: 3 Case Studies," In Proceedings of the Software Product Line Conference, ser. SPLC'12,. Brazil, September 2012. (To appear)

[16] S. She, R. Lotufo, T. Berger, A. Wkasowski, and K. Czarnecki, "Reverse engineering feature models," in Proceedings of the 33rd International Conference on Software Engineering, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 461-470.

[17] Sincero, R. Tartler, D. Lohmann, and W. S. Preikschat, “Efficient extraction and analysis of preprocessor-based variability,” SIGPLAN Not., vol. 46, no. 2, pp. 33-42, Oct. 2010.

[18] van Solingen, R., Basili, V.R., Caldiera, G., and Rombach, H.D, “Goal Question Metric (GQM) Approach,”. In Marciniak, J.J. (Ed.): Encyclopedia of Software Engineering (2nd Ed.), John Wiley & Sons:578-583, 2002.

[19] H. Spencer and G. Collyer, “#ifdef considered harmful, or portability experience with C News,” in USENIX Summer Technical Conference, Jun. 1992, pp. 185-197.

[20] M. Svahnberg and J. Bosch, “Evolution in software product lines: Two cases,” Journal of Software Maintenance, vol. 11, no. 6, pp. 391-422, Nov. 1999.

[21] R. Tartler, J. Sincero, W. S. Preikschat, and D. Lohmann, “Dead or alive: finding zombie features in the linux kernel,” in Proceedings of the First International Workshop on Feature-Oriented Software Development, ser. FOSD '09. New York, NY, USA: ACM, 2009, pp. 81-86.

[22] Treeviz. http://www.randelshofer.ch/treeviz/.

[23] TypeChef Project. http://ckaestne.github.com/TypeChef/.

[24] Bo Zhang. “Extraction and Improvement of Conditionally Compiled Product Line Code,” In Proceedings of the International Conference on Program Comprehension, ser. ICPC'12, Germany, June 2012. (To appear).