# Feature Mining for Product Line Construction

Yutian Tang
and Hareton Leung

Department of Computing
The Hong Kong Polytechnic University
Hong Kong SAR, China
Email: {csytang,cshleung}@comp.polyu.edu.hk

***Abstract*—Software product line engineering is a promising approach to generate software assets with systematic reuse property resulting in a significant decrease in development cost. Numerous studies and practical work have proved the reliability, reusability, productivity and the reduced R&D cost attributes of product line engineering. Whereas, the adoption rate of the product line is still relatively low considering the complexity and risk of the task given. It is crucial to have effective approaches to migrate legacy software into product line by mining features in the legacy. Nevertheless, common approaches in feature mining are mainly designed for general systems instead of product line. Therefore, in this paper, we firstly highlight characteristics that a well-designed feature-mining algorithm should contain and pinpoint the shortcomings of existing methods. To enhance the performance of existing approaches for product line, we proposed two feasible directions of research in terms of feature mining for product line.**

***Keywords–Software product line; Feature mining; Feature location; Reference checking.***

## I. INTRODUCTION

Software Product Lines (SPL) [1] provides tailored software artifacts with reusable property to stakeholders with minimal R&D effort, considering their general process allows common assets to be shared rather than developing individual systems separately. Among all approaches in generating software product lines, a promising low cost approach should be migrating and refactoring legacy software into a product line, since many fundamentals of legacy could be reused. For the migration, the legacy software has to be reorganized by feature base and then adjusted to a certain pattern to fit the product line. In software product line, variant is a proprietary term for feature and it is designed and specified by domain experts or developers to be either mandatory or otherwise. It assists normalizing the product lines feature model. For instance, in a database product line, a transaction feature should be mandatory for all sub-systems in the product line; however, each sub-system may have its own definition of ranking approach (ranking feature) to process data items.

As reported that successful adoption of the software product line will greatly reduce the cost of generating software products, provide timely service and products to market with reusable characteristics, and decrease the effort for quality assurance. Despite the benefit brought by SPL, the adoption still stays at a low level when compared to other new techniques, including Service Oriented Architecture, Aspect Oriented Programming, and so forth. The underlying reason could be the initial investment, including constructing variants and common assets, in the product line is relative high, and it also costs a lot for developers/software suppliers in terms of risk and complexity [2]. To simplify this process, the variants and common assets could be built by transferring legacy software to product line. The main challenge for migrating is that features in a product line could range from coarse to fine granularity with common and unique purposes like generating various products for different users. For example, in the *MobileMedia* product line, the feature play video is embedded in 24 classes in the system [3]. Therefore, the first and essential step in migrating legacy software into a product line should be locating features and extracting the source code related to the feature concerned given the condition that the code fragments implementing a feature could be scattered in the system instead of concentrating in a single component or file.

In this paper, we will review several existing approaches in feature mining and pinpoint the potential research gaps and the main challenges. Later, we will discuss the lessons learned from our work and other researchers. To focus the research work in feature mining, we will propose two potential directions to consider and investigate with underlying motivations. The contribution of this paper includes following aspects: (1) presenting prospective research gaps in feature mining and shortcomings of current approaches; and (2) proposing some feasible directions for further research.

The rest of this paper is organized as follows: in Section II, we present the essential parts that should be considered in providing feature mining work in product line. Several potential research directions will be presented in Section III. The conclusion will be covered in the last section.

## II. FEATURE MINING

Generally, a feature could be represented/defined by a set of code segments, which implement functions of the feature, and assist interaction with other modules (conjunction part). For example, the feature *transaction* in a bank product line cope with a business transaction, which could be a billing or a transfer. Functions belonging to this feature could be *pay_bill*, and *generate_receipt*. In addition, the conjunction part could be functions in charge of acquiring customers bank account information.

Feature mining in product line engineering is highly related to feature location, concern location, and other related fields, within the specific context and constraints in a product line. Developing a feature mining approach for product line is

 ISBN: 978-1-61208-449-7 

different from designing one for general software products, given the particular circumstance and restriction in product line. Here, we identify several conditions and constraints that will restrict the algorithmic design:

1) Fine granularity design: one significant distinction is that, in product line engineering, a follow-up step is to reconstruct and rewrite legacy code annotated by features into variants for reuse purpose instead of investing on an individual feature. With this constraint, approaches for traditional software products may not be suitable for product line use, as they are designed at the coarse granularity rather than a fine granularity level. Furthermore, the coarse granularity cannot guarantee the consistency of code segments extracted and is not fully adaptable for reconstruction of a product line;
2) Type checking: as reported in [4], type errors are more prone to occur in product lines rather than traditional software. If a product line is ill typed, it may introduce several potential errors during compilation and runtime. These errors include [4]:
   a) *Method invocation*: If a function in class *A* invokes another function in class *B* and the invoked method in *B* is annotated as a variant, deletion of class *B* (removes a variant from a product line, which is allowed in the product line context) will incur undeclared invocation in class *A* as shown in Figure 1;

```
class Painter{
void setPainter(Painter pt,Color col,
    Background bacg){
 canuvs.set(pt,col,bacg.getCurrScope());
}
}
class Canuvs{
#ifdef CANUVS_SET
 boolean set(Painter pt,Color col,Scope scope)
     {....}
#endif
}
```

Figure 1. An example of method invocation type error.

   b) *Referencing types*: Similarly, in referencing-type error, if a class is referenced as a return type or a customized type, when the class is annotated and serves as a variant, the referencing object still remains to be resolved, since the dangling class object will point to null. For the case depicted in Figure 2, if class *Background* is annotated and removed, the object *bacg* will incur a compilation error, since it refers to null;
   c) *Parameters*: Similar to prior case *referencing types*, if an object, which refers to the annotated class, is also annotated, this variant will still fail, since removing this referring object will leave a missing part in the original code slot, which will lead to a compilation error. As shown in Figure 3, in method *set*, the object *bacg* is also annotated as part of variant BACKGROUND, and if variant BACKGROUND is removed from the product line, object *bacg* should be removed as well, which will lead to insufficient parameters for method *set* (three instead of two).

```
class Painter{
 void setPainter(Painter pt,Color col,
     Background bacg){
  canuvs.set(pt,col,bacg.getCurrScope());
 }
}
#ifdef BACKGROUND
class Background{....}
#endif
}
```

Figure 2. An example of referencing type error.

```
class Painter{
 void setPainter(Painter pt,Color col,#ifdef
     BACKGROUND Background bacg #endif){
  canuvs.set(pt,col,bacg.getCurrScope());
 }
}
class Canuvs{
  boolean set(Painter pt,Color col,Scope scope
      ){...}
}
```

Figure 3. Parameter type error.

   d) *Feature interaction*:In general, there are multiple features embedded in a product line. In addition, the connections between features in the feature model show how features can interact and conjunct. Further, in product line, feature dependency will show how features are used and organized. As explained in[5], a practical example of feature dependency is that feature *A* can be selected if and only if feature *B* or *C* is selected without feature *D*, which shows the dependency relation between feature *A*, *B*, *C* and *D* in the feature model.

Next, we will introduce the lessons learned from our study and other related studies. By investigating previous studies in feature mining in a product line, our previous research in feature mining and other approaches in feature location (non product line use) [6], [7], we find following important considerations have been overlooked when preparing a well-defined approach for feature mining:

1) Attention should be given to analyze variability and feature model: Traditional approaches focusing on feature location merely extract features and code segments attached to these features without sufficient effort on feature interaction detection. As known, features do not just interact with each other. Therefore,

 ISBN: 978-1-61208-449-7 

detecting interdependencies and interactions from legacy software is still a pending challenge requiring additional work, as most feature location technique neglects this aspect, which is critical for product line engineering. Here, by detecting feature interaction, we mean exploring the interaction from source legacy automatically rather than just defining it in a feature model.

2) Quality assurance: In migrating legacy software into product line, focuses are often on the procedure of migrating without considering the change of quality in this continuous process. Therefore, we believe providing a set of quality metrics to measure the change in quality, such as reusability, reliability, and readability will assist the practitioners in evaluating the results of feature mining.

## III. Proposed Directions of further study

As presented above, several conditions and constraints must be considered when developing feature-mining approaches in product line engineering. We identify two possible directions, reference detection and data mining approach, to cope with feature mining in the context of product line engineering. The reason we recommend these directions are twofold: firstly, both could explore and mine features at a fine granularity; and secondly, these approaches are currently not well investigated for this context. The essential idea of these methods is to construct a standard graph to represent the system. Furthermore, the problem needed to be solved, namely, finding code segments for feature concerned, could be altered to discovery a set of nodes in the graph. Reference detection applies a searching strategy to the graph, whereas data mining groups programming elements using the concept of similarity.

Prior to providing concrete description of two potential research directions, we will first present the general procedure of feature mining, which is the infrastructure of all feature-mining approaches on legacy code. The general process of feature mining should consist:

- Defining and describing the features concerned and relationship among them in a model;
- A domain export or developer should identify seeds as starting point for each feature. Seed should be a single or a set of representative programming elements (methods, variables) that stand for the feature;
- For a single feature, the seed chosen along with the feature-mining approach will iteratively inspect related code segments and mark the code segments, which belonging to the same feature, with the same label;
- In the last step, the developers or tools will reorganize or rewrite the code fragments to variants.

In this paper, our work only focuses on the third step, which is providing competitive frameworks to cope with feature-mining task.

### A. Reference detection

Reference detection is originally motivated by a simple type of relation in a program named *define-use*, which represents a link between a definition of an instance variable and its later reference to the object [8]. For instance, in a normal object-oriented (OO) language, a single variable could be defined by a statement "*Timer localtimer* = new *Timer*()", which defines a new instance *localtimer* under class *Timer*. A reference statement could be "*this.launch(localtimer)*", which sets the object localtimer as a parameter of launch method. For this scenario, a link should be built between the define-statement and reference-statement in the program. A statement could be set as an attribute with two possible values:*def*, and *ref*. To simplify this procedure, we treat a program as a set of values and operations upon them. Next, we will introduce different types of variables and associated information that should be detected besides the reference relation.

1) *Local variable reference*:A local variable could be declared inside a method or as an argument of a method. For this kind of variable, it could be referenced by other programming elements inside the method and also reference other programming elements inside and outside the method. Specifically, some local variables even have smaller valid boundaries. For example, if a local variable is defined inside a *for* statement block, its valid scope will be within this *for* statement. By checking the location, at which a variable is defined/declared, and the context, it is feasible to obtain the valid scope of the variable.
2) *Instance variable reference*:Instance variables normally hold all attributes of classes and employed as local variables or fields for the class. There are two issues related to the scope concern: the first is the location that this type of instance variable is declared, and the second is whether hierarchical relations exist, which means for a certain class or interface, its super-class, interface, and sub-classes, should also be extracted if any.
3) *Class variable reference*: Class variable is global for all instances of the class and generally its life cycle will finish when the class is destroyed. For a class variable, we should consider the same issues as those of instance variable.
4) *Class/interface field reference*:Fields are the inherent attributes of a class or interface, and are allocated memory when the class/interface is created. The following information should be collected to build the *def-use* link when inspecting a class/interface field: (1) detect the scope of current reference, since a field of a class could be used outside the class/interface in which the field is declared; (2) explore the location where the original class/interface and the field are declared; and (3) build the link between this field reference and instance variable declaration.
5) *Reference variable reference*:When a variable is assigned directly to another variable, these two variables share the same memory location. Thereby, we can use a joint node with two identifiers to represent these two variables, and all reference links to any of them will be redirected to this joint node instead.

After all *def-use* relations are extracted from the source code, we can re-construct and re-build the original program into a standard graph, in which programming elements under reference or defined relations are connected. A simplified example is shown in Figure 4 with some critical information hidden, such as, location information, type information and so forth. After constructing this infrastructure, the original code

 ISBN: 978-1-61208-449-7 

segments are annotated with the reference/define information. Then, numerous algorithms could be proposed based on this framework by checking the fan-in, fan-out, graph topology structure, ranking connected items by granularity and so forth. This framework offers the following advantages: both fine and coarse-granulated programming elements are encapsulated in the graph and only fine granularity information is required. The rational for keeping both fine and coarse granularity information is to ensure completeness of the program and ease to locate a specific programming element. We propose two approaches to resolve this. One is separating the graph into sub graphs by granularity; and another is setting searching conditions, for instance, statement, expression, variable and so forth, when searching related programming elements in the graph. That is, Abstract Syntax Tree (AST) node could be adapted to work as the node in this framework and the type of AST node could be used to search nodes under a specific granularity.

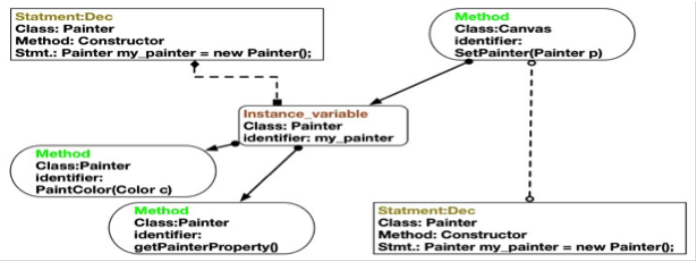


Figure 4. An example of referencing model

**Determining the feature boundary under reference detection**. Technically, features could be connected by a reference link (e.g. call a function), or physically embedded in the same compiling unit from the code base perspective. Specifically, the reference link could be any type listed above. For this case, algorithm *referenceTrack* can be employed to assist in finding the boundary of a feature:

**Algorithm 1** referenceTrack(threshold)

**for** single seeds $s$ **do**
  create $P_s = \{p | p\ references\ s\}$
**end for**
**for** $p \in P$ **do**
  compute uniqueness value $uv_p$ by $uv_p = \left(\frac{\#\ links\ p\ refereces\ s}{all\ p\ reference\ links} - \frac{\#\ links\ s\ references\ p}{all\ s\ reference\ links}\right) * g(p)$
**end for**
Rank elements in $P$, according to $uv_p$, and if the max $uv_t > threshold$, it will be annotated into the feature.
The definition statement $dt$ of $t$ will be annotated.Then set $dt$ as $s$.And recall this algorithm with $dt$.

Specifically, "# links p references s" represents the number of links from $p$ to $s$; "# p reference links" shows the number of links $p$ reference others. $g(p)$ shows the granularity of $p$ and its value is designed for showing the interference from the granularity aspect. The algorithm *referenceTrack* will stop in STEP 3, if $uv_t$ is below the threshold assigned.

### B. Data mining based approach

Apart from the program analysis techniques, machine learning techniques, especially data mining approaches are also recommended for feature mining. Inspiration of using data mining for feature mining comes from the common trait that both attempt to group items into distinct clusters. One basic idea in data mining is to measure the distance between different items to show their "closeness", which is applied to indicate their similarity. In feature mining, this idea should be adjusted to grouping programming elements into different clusters, and each cluster could represent a unique feature. Here, we provide several potential relations that could be used to detect the distance among programming elements:

1) textitReference distance(call graph based):In our model, the reference distance is defined as the number of jumps from one programming element to another and all links connecting intermediate elements are reference relation links.
2) textitControl distance(control graph based):Control graph could be extracted from a program to indicate its control structure. Control distance, in our model, is defined as the jump in a control graph from one programming element to another with the restriction that the distance between two programming elements in the same control branch should be 1.
3) textitText comparsion:Similarity could also be detected from the textual aspect with the assistance of text comparison algorithms[9]. Using different text comparison algorithms, the text distance could be described using the value computed by the token frequency, common strings and so forth.

On the contrary to *def-use* model, the data mining based approach determines closeness from various aspects by building different types of graphs, including call graph and control graph. By utilizing these potential relations, the similarity among programming elements can be determined. Finally, programming elements could be classified into different clusters with each cluster representing a unique feature. In this framework, the seeds selected are used to represent distinct features and other related programming elements are explored iteratively. Here the seed merely serves as the trigger of the algorithm, and is used to detect other potential programming elements.

**Determining feature boundary under data-mining approach.**Different from reference detection, the mining process for a single feature will stop when one of following stopping criteria is met:

1) For a single node in the graph, if all neighbors (data or control) of this node are annotated by other features, the mining process for this node will stop;
2) A threshold could be set to restrict the selection of programming elements and could be used to stop the mining process for the current node. That is, if and only if the distance computed higher than the set threshold, it will be considered to be added to this feature. If scores of all neighbors (data or control) of the current node are lower than the threshold, the mining procedure for this node will stop.

 ISBN: 978-1-61208-449-7 

## IV. CONCLUSION

Despite many research works have investigated feature location and variability detection in product lines, the task of feature mining still poses great challenges, considering the particular circumstance of software product line, complexity of the task, and other special constraints as discussed previously. Mostly, current approaches are insufficient to cope with the fine granularity concern in product line and further detection is required to guarantee feature-mining work flows [8] in legacy software migration. Based on our previous work and other studies in feature mining, we identify key traits that a well-defined product line feature mining approach should contain. The conditions mentioned are highly recommended prior to proposing a well-performing algorithm. With the research gap identified, we proposed two feasible directions to investigate the feature mining. We strongly believe that with general rules and constraints specified in the product line engineering, new approaches can be defined for feature mining in product line and will lead to reduction of risk and effort in transferring legacy software into the product line.

## REFERENCES

[1] K. Pohl, G. Bockle, and F. v. d. Linden, "Software product line engineering foundations, principles, and techniques," 2005.

[2] C. Catal, "Barriers to the adoption of software product line engineering," ACM SIGSOFT Software Engineering Notes, vol. 34, no. 6, 2009, pp. 1–4.

[3] E. Figueiredo, N. Cacho, Sant, C. Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas, "Evolving software product lines with aspects," pp. 261–270, 2008.

[4] amp, C. stner, S. Apel, Th, T. m, and G. Saake, "Type checking annotation-based product lines," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 21, no. 3, 2012, pp. 1–39.

[5] M. Mendonca, A. Wsowski, and K. Czarnecki, "Sat-based analysis of feature models is easy," in Proceedings of the 13th International Software Product Line Conference. Carnegie Mellon University, 2009, Conference Proceedings, pp. 231–240.

[6] M. A. Laguna and Y. Crespo, "A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring," Sci. Comput. Program., vol. 78, no. 8, 2013, pp. 1010–1034.

[7] C. Kastner, A. Dreiling, and K. Ostermann, "Variability mining: Consistent semi-automatic detection of product-line features," Software Engineering, IEEE Transactions on, vol. 40, no. 1, 2014, pp. 67–82.

[8] E. Sderberg, T. Ekman, G. Hedin, and E. Magnusson, "Extensible intraprocedural flow analysis at the abstract syntax tree level," Science of Computer Programming, vol. 78, no. 10, 2013, pp. 1809–1827.

[9] B. Cleary, C. Exton, J. Buckley, and M. English, "An empirical analysis of information retrieval based concept location techniques in software comprehension," An International Journal, vol. 14, no. 1, 2009, pp. 93–130.

 ISBN: 978-1-61208-449-7 