Roberto E. Lopez-Herrejon, Jabier Martinez,
Tewfik Ziadi, Anil Kumar Thurimella, (eds.)

1st International Workshop on Reverse Variability Engineering (REVE 2013)

http://www.sea.jku.at/reve2013/

Pre-proceedings

held on 5th March 2013 in conjunction with the
17th European Conference on Software Maintenance and
Reengineering (CSMR 2013)
Genova, Italy

Variability management of a product family is the core aspect of Software Product Line Engineering (SPLE). The adoption of an SPL approach requires a high upfront investment that will allow to automatically generate product instances based on customer needs

Mining existing assets could dramatically reduce the costs and risks of this adoption. Those existing assets use to be similar product variants that were implemented using ad-hoc reuse techniques such as copy-paste-modify. Bottom-up approaches to automatically extract variability management related artifacts could be proposed, applied, validated and improved in this domain. We propose this workshop to fill the gap between the Reengineering and SPLE communities.

Program Committee

Workshop Organisers

Roberto E. Lopez-Herrejon, Johannes Kepler University Linz, Austria
Jabier Martinez, itemis France SARL, Paris, France
Tewfik Ziadi, UMR CNRS, LIP6-MoVe, Paris, France
Anil Kumar Thurimella, Harman & TU Munich, Germany

This research was partially funded by the Austrian FWF under agreement Lise Meitner Fellowship M1421-N15.

We, the workshop organisers, are grateful to the members of the program committee and the reviewers. We also would like to thank the workshop chairs of CSMR 2013, Walter Cazzola and Michel Wermelinger, for their support and the opportunity to hold REVE 2013 in conjunction with this conference.

Post-proceedings to be published as ECEASST report.

Table Of Contents

A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies

Benjamin Klatt, Martin Küster, Klaus Krogmann
FZI Research Center for Information Technology
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
{*klatt,kuester,krogmann*}*@fzi.de*

***Abstract*—Software product lines are a widely accepted strategy to develop software products with variable features. Deriving a product line from existing customised product copies is still an open issue. Existing approaches try to extract encapsulated reusable components, or analyse features on a product management level. However, support for understanding implementation differences and designing variation points to consolidate the customised copies is still missing. In this paper, we present the analysis concept of our SPLevo approach, to identify related variation points in order to recommend reasonable variation point designs. To illustrate its application, we provide an example analysis.**

***Keywords*-Software Product Line, Reverse Engineering, Software Analysis, Product Copies**

I. Introduction

Productivity, reuse and customisation for individual requirements are major goals in software engineering. The software product line approach has been established to target these goals with explicit and managed variability [1]. A company can serve all their customers in a specific domain with a variable core software product, which can be configured, extended, or parametrised for the customer-specific needs. Building solutions on such a product line aims to deliver a better tested product with less development and maintenance effort in the long term.

Ideally, the required variability of a software product line is identified as part of the requirements engineering process, and suitable variability realisation techniques are chosen during the software design phase [2]. Such a proactive approach allows to benefit from the product line advantages right from the beginning. However, in reality, it requires high upfront investments, and postpones the first product delivery.

As a result, many companies start implementing the first product and copy and customise this code base afterwards for specific needs of other customers. After better understanding the domain and their customers' needs, they have to reactively transform those variants into a common product line [2]. This procedure ensures, that only mature, tested and actually needed features are integrated into the common product line, but is a challenge by itself.

Today, most existing approaches focus on forward engineering or treat variability without respecting the product life cycle at all. Many approaches exist for varibility specification as surveyed by Chen et al. [3] and different variability realisation techniques are available as surveyed by Patzke et al. [4] and Svahnberg et al. [5]. But, support for reactively migrating customised product variants into a common product line is still an open issue. Such a consolidation process is challenging as the number of differences between two product variants is typically high. A product line engineer, responsible for a consolidation, needs to find the product copies' differences, identify the ones relevant for the product line variability, understand their relations, and design proper variation points.

Only a few existing approaches for reactively build product lines try to handle the challenge and automation of variability reverse engineering. The existing approaches focus on a limited scope of the implementations under study or aim to extract encapsulated, reusable assets of the software. For example, Graaf et al. [6] analyse execution traces of product variants, which can only identify differences in the scope of their executed functionality. Koleilat et al. [7] facilitated clone detection to extract reusable assets. None of them considers the product variants as a whole.

In this paper, we present our variability analysis concept developed as part of our overall approach to consolidate customised product copies into a common software product line (SPLevo [8]). We use a graph-based representation of variation points to combine several basic relationship analysis strategies. We further derive design recommendations from the identified variation point relationships to support the product line engineer in his design task.

The contributions of this paper are i) a graph-based composite analysis to recommend variation point aggregations, ii) a set of basic analysis strategies to identify variation point relationships, and iii) a rule-based concept to derive recommendations from the variation point relationship graph.

The rest of this paper is structured as follows: Section II provides a short overview of the SPLevo approach incorporating the outlined concept. Section III introduces our model for describing variation points followed by our concept of how to analyse instances of this model in Section IV. Section V describes the basic variation point relationship analysis strategies, before an illustrating example of their application is given in Section VI. Section VII presents work related to our approach. In Section VIII we clarify our assumptions and limitations and give a conclusion of our work presented in this paper and an outlook on our future work in Section IX.

II. SPLevo Approach Overview

The SPLevo approach aims to support product line engineers in consolidating customised product copies into a common product line [9]. The core idea is a semi-automatic, model-driven process to identify the differences between product variants and to guide the product line engineer in iteratively creating a proper variation point design to later derive refactoring support to perform the product line consolidation [10]. The large amount of fine-grained, source-code-level differences to consider and to assess, makes support for finding relevant and related differences necessary.

Figure 1 presents the main process defined in our SPLevo approach. First, software entity models (i.e. abstract syntax trees (AST) [11] and inventory models [12]) are extracted from the product copies' implementations. Those models are compared in the second step to identify the implementation differences. Next, we initialise a model describing fine-grained variation points based on these differences. This model is then analysed ("variation point analysis") to guide the product line engineer in iteratively refining the variation points until he is satisfied with the variability design. As a last step, refactoring support to change the implemention is derived (e.g. insert "#ifdefs" or change to dependency injection).

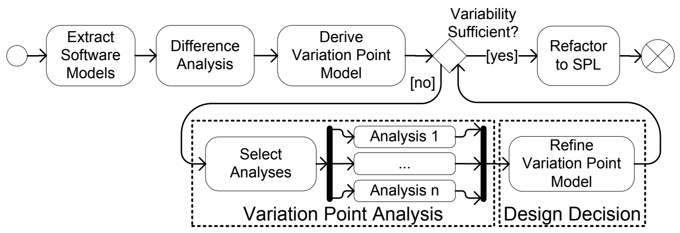


Figure 1. SPLevo Main Process

Such a consolidation will always require manual decisions because of organisational issues or personal preferences on equivalent alternatives in a typically large design space (e.g. because of available implementation techniques, code structuring, or feature granularity). The SPLevo approach supports automated variability reverse engineering, but expects the product line engineer to accept, decline or modify the recommended variation point refinements and to provide individual product line preferences.

III. Variation Point Model

Similar to Svahnberg et al. [5], we explicitly distinguish between features on the product management level and variation points on the software design level [10].

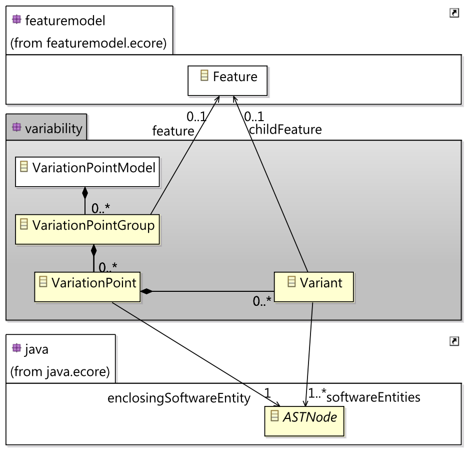


Figure 2. Variation Point Meta-Model (attributes omitted for simplicity)

Figure 2 presents the meta model of the Variation Point Model defined in the SPLevo approach. It is a software design model to describe explicit variation points in a software product line. It allows to correlate and aggregate fine-grained differences into more coarse-grained and better manageable ones.

A *VariantionPoint* represents a location in the software at which a variability mechanism should allow to switch between alternative implementations. The location of the *VariationPoint* is specified by it's reference to an *ASTNode* element of an abstract syntax tree model ("'java.ecore"' in Figure 2). The alternative implementations, available for a *VariationPoint*, are described by it's *Variant* elements. Each of them references one or more software entities (e.g. classes, methods, statements) that implement this specific alternative.

As identified by Svahnberg et al. [5], a single feature can be implemented by one or more variation points in the software. This is reflected in the Variation Point Model by the *VariationPointGroup* element. It groups all *VariationPoints* that might be located at different *ASTNodes* and links to the variable *Feature* they contribute to. In feature models, a variable *Feature* is linked with it's optional or alternative child features. For this, *Variant* elements reference the child feature they contribute to.

The *VariationPointModel* element represents the root element of the model. It contains all *VariationPointGroups* of related *VariationPoints*.

In Figure 2, the referenced *Feature* element origins from the EMF feature model [13], and the *ASTNode* element from the Eclipse MoDisco Java AST model [14]. Both models are used in the currently developed prototype of our approach [8], but the concept itself is not restricted to these concrete feature and software entity models.

As described in the previous section, an initial Variation Point Model is derived from the fine-grained differences between the software entity models. This is done by creating a *VariationPoint* element for each difference with a reference to the parent ASTNode of the differing ASTNodes as the variation point's enclosing software entity. The differing ASTNodes are referenced as software entities by *Variant* elements created for each of the product copies. This initial Variation Point Model is then analysed to identify related variation points and recommend refinements. How this is done is described in the following section.

IV. Variation Point Analysis

The result of the variation point analysis is a Variation Point Model describing a satisfying variability design to serve as input for refactoring the product copies into a software product line.

The initial Variation Point Model, derived from the differences between the product copies, represents all fine-grained differences at the source code level. To achieve a manageable amount of variability in the resulting product line, the Variation Point Model must be refined. Manually analysing all variation points to aggregate them into more corse-grained ones is very time-consuming due to the typically high number of differences.

To support this task, we aim to provide variation point analyses, to identify related variation points and provide recommendations for aggregations. In general, such product consolidations cannot be done in a fully automatic fashion because equivalent alternatives are possible which are selected due to non-technical criteria, such as organisational reasons or personal preferences. To cope with this, the analysis within our SPLevo approach returns only recommendations that the product line engineer can accept, decline, or adapt.

The analysis returns recommendations in terms of variation point aggregations. The following subsections provide details about our graph-based concept to combine and evaluate the different analyses and describe aggregation and filtering techniques as possible variation point refinements.

A. Graph-Based Analysis Composition

Relationships between variation points can exist because of many different aspects, such as their location or type of modification. As our approach is to recommend aggregations of variation points because of their relationships, we consider this scenario as an edge-labelled, undirected multigraph with the variation points as vertices, and their relationships as edges. An edge between two variation point nodes can be created for example because the variation points are located in the same method. The type of a relationship is stored as a label of the according edge (e.g. ”CS” because the relationship is derived from the code structure). Because multiple different relationships can exist between two variation points and each of them is represented as an edge, this leads to the multigraph characteristic. In addition, sub-labels can exist for the edges, to allow to provide additional information about the relationship (e.g. the name of the method the variation points are located in).

As shown in Figure 3, the Variation Point Model is transformed into a graph representation by creating a node for each variation point. Next, this graph is handed over to the intended analyses which are performed in parallel. Each analysis creates edges corresponding to the relationships it has identified. Those edges are labelled with the type of relationship identified, e.g. R1, R2, and R3. In the next step, the edges returned from the individual analyses are merged into a combined graph which now contains all edges.

In the final step of the analysis process, the recommendations to refine the Variation Point Model are derived by detection rules inspecting the combinations of relationship types between variation points. A detection rule is specified for a set of edge labels as it's matching pattern. In addition, it specifies a refinement to recommend in case of a pattern match. If a detection rule matches a subgraph of variation points, a refinement recommendation is created for the involved variaton points according to the rule's specification. This refinement is stored in an overall recommendation set to be later presented to the product line engineer.

Detection rules are always applied in a defined order and if edges are matched by a rule's condition, they are ignored by any rules applied later on to prevent possibly conflicting recommendations. The set of rules to apply as well as their order depends on the individual product line requirements and needs to be adapted to the basic analyses performed in a concrete consolidation scenario. The resulting recommendations can consider two different types of variation point aggregations (grouping and merging) which are described in the following subsection.

B. Variation Point Aggregation

A good variation point design is a trade-off between providing variability for as much product options as possible and minimising the number of variabilities to manage. While the former obviously allows to provide more individual product variants, the latter is a best practice for SPL manageability, also documented by Svahnberg et al. [5].

In our SPLevo approach, we start with a fine-grained Variation Point Model on the level of identified code differences. In order to come up with the requirement to have manageable amount of variation points, this requires to aggregate variation points. We distinguish between ”merge” and ”group” as two types of variation point aggregations which we described in the following subsections.

1) Merge Variation Points: The variation point merge is based on the capability of *Variant* elements to reference several software entities (i.e. *ASTNode* elements) that implement a child feature. *VariationPoints* are merged by consolidating their Variant elements and the referenced software entities in only one of the *VariationPoint* elements. First, one

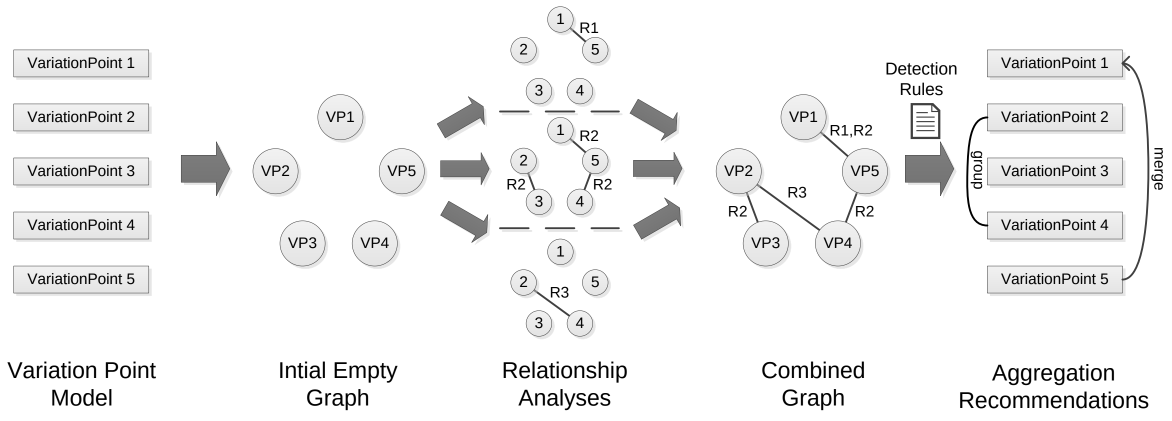


Figure 3. Graph-Based Analyses Composition

VariationPoint element that should survive is selected. Then, for all *Variant* elements in the *VariationPoints*, contributing to the same child feature, the *ASTNode* references are merged into one of the *Variant* elements. It is ensured, that this *Variant* element is contained in the surviving *VariationPoint*. Finally, the remaining empty *Variant* and *VariationPoint* elements are removed from the model.

2) Group Variation Points: The variation point grouping is based on the explicit group structure in the Variation Point Model as described in Section III. Even in the initial Variation Point Model, each *VariationPoint* is assigned to a *VariationPointGroup*. A set of *VariationPoints* is grouped by selecting one of their *VariationPointGroup* elements that should survive and moving all *VariationPoint* elements into this. Afterwards, the remaining empty *VariationPointGroup* elements are removed from the Variation Point Model.

C. Variation Point Filtering

Complementary to the previously described aggregations, variation points can be filtered from the Variation Point Model. This can be done, for example, to remove identified variation points which are not of interest for the product line engineer and his intended product line. In such a case, the variation point is simply removed from the Variation Point Model. A removed variation point, will no longer result in a refactoring advice in the downstream process.
Filtering strategies are not part of the analysis discussed in this paper but mentioned for completeness as an alternative Variation Point Model refinement.

V. Relationship Analyses

As described in the last section, our variation point analysis is a composed analysis of the relationships between variation points. It allows for the use of serveral basic analyses which focus on specific relationship types according to different aspects of software changes.

We have identified a list of different relationship types between variation points and according strategies to identify them:

- Code Structure (CS)
- Program Dependency (PD)
- Data Flow Dependency (DFD)
- Program Execution Trace (PET)
- Change Type (CT)
- Change Pattern (CP)
- Cloned Change (CC)
- Semantic Relationship (SR)
- Modification Time (MT)

Those strategies are discussed in more detail in the following subsections.

A. Code Structure (CS)

The code structure analysis studies the phyisically implemented code structure, such as statements contained in methods and methods contained in classes. Variation points located in the same structure are identified to have a code structure relationship. Examples for such relationships are:

- Statements in the same block statement
- Statements in the same method
- Methods in the same class
- Classes in the same component

Up to the level of classes, this hierarchy is directly provided by the software entity models extracted from the implementations under study. Depending on the product copies under study it might be possible to get higher level structures, e.g. a component architecture, from documentation or reverse engineering techniques. If architecture information is available it could be included in the code structure analysis [15].

B. Program Dependency (PD)

The program dependency analysis identifies dependencies between the variation points' software artifacts based on programming language features. For example, the PD analysis identfies relationships between a variation point containing a varying variable declaration statement and all variation points which describe added, deleted or changed statements consuming this variable. Program dependendencies overlap with data flow dependencies described below but are not the same.

C. Data Flow Dependency (DFD)

A data flow dependency identifies sofware elements which handle the same data object and potentially influence each by this data object. The software entity models extracted from the implementations are sufficient for a static data flow analysis. And, because the data flow is inspected, only data processing model elements need to be considered. Examples for potential dependencies are:

- Statements manipulate the same variable
- Method calls with one using the others' result as input
- A statement depends on a changed class attribute
- A method invocation of a changed method

Adding a data object identifier as an additional sub-label to the data flow dependency relationship label ensures to not mix up relationship edges resulting from different data flow dependencies.

D. Program Execution Trace (PET)

Program execution traces represent a programs execution flow monitored during the execution of one or more specifc features. They can be gathered from instrumenting the program code before it's execution. Alternatively, a profiler can be used, that returns information about the dynamic beahaviour of the software, e.g. method invocations or object instantiations.

Variation points represent locations of variability. If the locations of two variation points are contained in the same execution trace, this is considered as a program execution trace relationship between those variation points. An identifier of the executed feature, respectively of the execution trace is added to the relationship label. This information ensures to not mix up variation point relationships resulting from execution traces of different features.

The execution traces are external information sources which need to be included by the analysis. This also requires to match the elements of the execution traces to elements in the software entity models referenced in the Variation Point Model. This can be done by matching the full qualified names of the software entities.

E. Change Type (CT)

A change type describes a modification of a software entity, e.g. a parameter added to a method signature or a statement removed from a method body. Fluri et al. [16] have developed a taxonomy of such change types to analyse the evolution of a single software system.
However, those change types can also be used for a variation point analysis. In a first step, the differences between the variants of a single variation point can be classified according to a change type taxonomy as provided by Fluri et al.. In a second step, variation points classified with the same change type can be identified and a relationship edge can be created with a change type relationship label. Adding the specific type of change to the relationship label ensures that relatinships resulting from different changed types are not mixed up.

F. Change Pattern (CP)

A change pattern also describes a modification between the variants of a variation point. Change patterns can involve multiple software entities and can be specific to the system under study. For example, a change pattern can describe that a boolean parameter is added to a method signature, checked for a null value in a conditional statement at the beginning of the method, and the control flow is returned if a null value is found. In addition, it is possible to specify low-level change types more specifically. For example, not only all variation points with an added parameter can be detected, but all variation points with an added parameter of a specific data type and a specific name.

If two variation points match to the same change pattern, a relationship edge is created between them and labelled as change pattern relationship combined with an identifier for the specific change pattern as sub-label.

Compared to the change type analysis, the change pattern analysis is not limited to standard software changes. It is able to consider a combination of multiple changes and provides a higher flexibility. The patterns must be specified in advance of the analysis.

G. Cloned Change (CC)

According to Roy et al. [17], *"a code clone are two code fragments which are similar by a given definition of similarity"*. Clone types range from direct copies up to code sections that perform the same computation but have different implementations. Roy et al. further specified four types of clones:

- Type 1: Code Layout & Comments
- Type 2: Literals Changed
- Type 3: Added, Changed, or Removed Statements
- Type 4: Same Computation but other Implementation

The cloned change analysis makes use of this by applying a clone detection to the code changes of the variation points. For example, if two variation points are about a set of added statements, the clone detection is applied to those statements to check if the same code has been added at these variation points. If this is true, a code clone relationship edge is added for those two variation points.

Compared to the previous change type and change pattern analysis, the cloned change analysis makes use of a mining approach without predefined generic or specific change patterns to match. Furthermore, depending on the analysed types of clones, additional similarities which are not possible to be specified as patterns might be detected.

H. Semantic Relationship (SR)

Developers often introduce semantics not only in comments but also in the names of their variables, methods and classes [18]. Semantic code clustering techniques take use of this to find clusters of related code in software products. The semantic relationship analysis makes use of such techniques to identify relationships between variation points. The semantic clustering is applied to the software entities referenced by variants of the variation points under study. The clustering algorithm will return related software entities. The referencing variants will provide the link back to the enclosing variation points. If more than one variation point is connected to such a cluster, a relationship edges labelled as semantic relationship with an identifier for the semantic cluster are created between them.

I. Modification Time (MT)

The modification time analysis investigates in changes performed at the same time. This is applied to software elements referenced by variants contributing to the same child feature. If software entities referenced by variants of two variation points have been modified at the same time, an edge labelled as modification time relationship is created between those variation points.

The software entities' modification time is typically provided by a revision control system such as cvs, git or svn. When a developer has finished a modification, he commits one or more resources, each with one or more modified software entities, into the system. The time of the commit is interpreted as modification time instead of the exact point in time when a developer has modified a single file on his local system. The later is rarely available and the commit provides more useful information because more than one file is involved and a commit typically represents a completed modification.

In addition to the time of the commits, the commit message can be used to identify related commits. This potentially allows to identify even more software entities changed together compared to considering only a single modification.

VI. Illustrating Example

In [15] we have introduced a greatest common devisor (GCD) example program [19] with a native Java and a JScience-based product variant [20]. The two implementations differ in a method named `gcd()`. This method transforms two string parameters into numeric values, calculating their GCD, and returning the result as a string as shown in Listing 1 and 2.

Listing 1. Standard Java Implementation
```
import java.math.BigInteger; //VP1
...
public String gcd(String v1, String v2){
 BigInteger intV1 = new BigInteger(v1); //VP2
 BigInteger intV2 = new BigInteger(v2); //VP3
 BigInteger gcd = intV1.gcd(intV2);  //VP4
 return gcd.toString(); //VP5
}
```

Listing 2. JScience-Based Implementation
```
import org.jscience.mathematics.number.LargeInteger; //VP1
...
public String gcd(String v1, String v2){
 LargeInteger intV1 = LargeInteger.valueOf(v1); //VP2
 LargeInteger intV2 = LargeInteger.valueOf(v2); //VP3
 LargeInteger gcd = intV1.gcd(intV2); //VP4
 return gcd.toString();  //VP5
}
```

Diffing the abstract syntax trees, extracted from the two product copies, detects any changes the implementations. This leads to five variation points which are marked as VP1 to VP5 in the presented code listings. VP1 is a changed import. VP2 to VP4 are about statements declaring variables of different data types. Additionally, in VP5, methods with similar names — `gcd()` — but for variables of differing types are called. While lexical similar, the semantic difference between these calls is detected by comparing the methods' abstract syntax trees and not textual representations only. During the analysis, the graph derived from the initial variation point model contains nodes for each of the variation points as shown by the *Initial Graph* in Figure 4.

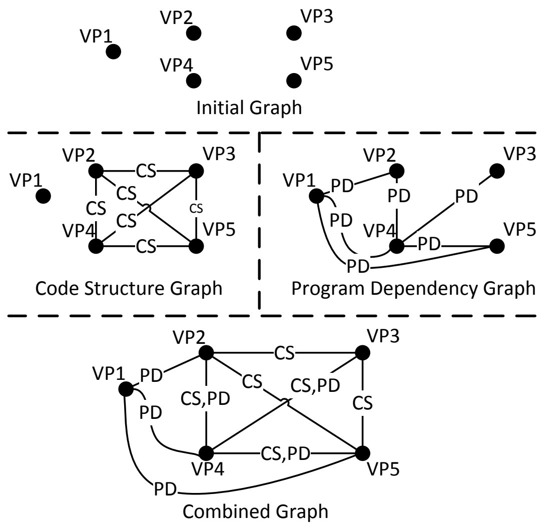


Figure 4. Illustrating Example Graphs (sub-labels omitted for simplicity)

Applying the Code Structure analysis identifies relationship edges, labelled with "CS", between the nodes VP2, VP3, VP4 and VP5, because all of them are located in the method `gcd()`.

Applying the Program Dependency analysis identifies relationships between VP1 and each of VP2, VP3, and VP4 because each of them makes use of the class imported in VP1. Each of the resulting relationship edges is labelled as "PD" and sub-labelled as "import". Furthermore, the Progam Dependency analysis identifies dependencies between VP2 and VP4, VP3 and VP4, as well as VP4 and VP5 because of their variable usages. Relationship edges with the label "PD" and the sub-label "variable-usage" are created.

In the last step, two detection rules are applied in the order as they are described here. The first rule detects edges labelled with "CS" and "PD (variable-usage)". For all matching cliques in the graph, a merge is recommended for the according variation points. VP2, VP3, VP4, VP5 in the example. The second rule applied, detects edges labelled as "PD (import)" and recommends to group the variation points involved in the identified cliques. VP1, VP2, VP3, and VP4 in the example.

Later on, when the recommendations are applied, the recommendation order will take effect. First, VP2 to VP5 are merged into a variation point VP6, and then VP6 is grouped with VP1.

VII. Related Work

Our analysis concept presented in this paper is developed in the context of variability reverse engineering from product copies to be consolidated into a software product line.

A strongly related approach has been developed by Graaf et al. [6] to identify variation points based on program execution traces. Similar to them, we consider feature location techniques based on static or dynamic program graph analysis as done by Rajlich et al. [21], and Wilde et al. [22], as relevant for the detection of program differences relating to a common feature. However, we handle this as one technique beside others to be considered. Furthermore, Graaf et al. do not discuss the influence or the handling of code differences which are not relevant for the product line variability (e.g. code beautifying etc.). In our approach, this can be handled by filtering the according variation points.

Our work focuses on the creation of SPLs from customised product copies with a common initial code-base, comparable to an approach of Koschke et al. [23] facilitating a reflexion method. However, our approach does not depend on a pre-existing module-architecture as the extended reflexion method of Koschke et al.. Furthermore, we aim to take custom SPL requirements ('SPL Profile') into account, such as preferences for variation point design and realisation techniques.

Alves et al. [24] formalised valid, feature-aware refactorings of SPLs. Their approach is complementary to ours and can be considered as part of our refinement step to refactor initially created variation point models to a more homogenious product line. They do not provide support for identifying code differences contributing to a feature. Instead, they assume feature models to be derived from documentation or manual code examination. However, they state automated support for this as desirable.

She et al. [25] have presented an approach for reverse engineering the hierarchal feature model structure and constraints from a given set of features and feature dependencies. We also aim to build a variability model, we do not assume a set of features and dependencies as input as they do, but we aim to reverse engineer this information from the given product copy implementations. While they cluster predefined features based on given dependencies, we cluster variation points based on analysed relationships to identify reasonable variable features.

Yoshimura et al. [26] also worked on identifying related variabilities to improve a product line's manageability. However, their goal is to recommend feature constraints based on customer preferences from the past for an existing product line. In contrast, we aim to identify technical and logical dependencies between varying code entities to design variation points as part of the process of creating a new product line.

VIII. Assumptions and Limitations

The main assumption of our approach is the consolidation of product copies with a common code base. For products developed completly independent from each other with a similar purpose only, the differencing in general and the analysis concept presented in this paper in specific, cannot be expected to return reasonable results.

Furthermore, the approach focuses on systems developed in object-oriented programming languages. For other programming languages, especially the concepts relying on the code structure need to be checked and probably adapted.

Within object-oriented programming languages, the lowest level of granularity we examine are statements. Modifications on the expression level, for example a partly modified composite condition of an if-statement, is interpreted as a modified if-statement rather than a changed sub-expression. This is done to reduce the amount of information to process.

Some changes in customised product copies are not relevant for variable features of the resulting software product line, but might be of interest from a maintenance perspective. For example, an improved code formating or documentation should be adopted in the resulting implementation. However, this requires a simple merge process from the customised variant into the resulting product line, which is not in the focus of this paper and part of our planed future work.

IX. Conclusion

In this paper we have presented our variation point analysis concept developed in the context of the SPLevo approach for consolidating product copies into a common product line. The analysis identifies recommendations to refine an intialy fine-grained Variation Point Model in terms of reasonable aggregations. A graph-based representation of the variation points is used to enable the composition of basic analyses for different relationship types. The identified relationships are represented as labelled edges within an overall graph. The refinement recommendations are derived from this graph by applying detection rules specifying combinations of relationship types as match patterns and the type of refinement to recommend in case of a match.

The analysis concept presented in this paper supports a product line engineer in understanding the differences between the product variants to consolidate and in creating a proper variation point design for the intended software product line.

Currently, we are developing a prototype implementation of the overall SPLevo approach as an Eclipse-based application [8] which already automates the presented example. We furher use and improve it within a case study facilitating the SPL variant of ArgoUML provided by Couto et al. [27]. In this case study we analyse differing implementations generated by their pre-processor facilities and try to reverse engineer the originated product line as a reference. Following this case study, we are going to analyse product copies of our industrial partners with customised product copies created on project contexts and due to organisational reasons. In a further step, we will investigate sets of predefined analysis and detection rule for typical software product line requirements.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF), grant No. 01IS12012B.

References

[1] L. Clements Paul ; Northrop, *Software product lines : practices and patterns*, 6th ed., ser. SEI series in software engineering. Boston, Mass.: Addison-Wesley, 2007.

[2] K. Pohl, G. Böckle, and F. van der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer, 2005.

[3] L. Chen, M. Ali Babar, and N. Ali, "Variability management in software product lines: a systematic review," in *Proceedings of SPLC'09*. Carnegie Mellon University, 2009.

[4] T. Patzke and D. Muthig, "Product Line Implementation Technologies," Fraunhofer IESE, Kaiserslautern, Tech. Rep. 057, 2002.

[5] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Software: Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[6] B. Cornelissen, B. Graaf, and L. Moonen, "Identification of variation points using dynamic analysis," in *Proceedings of R2PL'05*, 2005.

[7] W. Koleilat and N. Shaft, "Extracting executable skeletons," Cheriton School of Computer Science, University of Waterloo, Waterloo, Tech. Rep., 2007.

[8] B. Klatt, "SPLevo Website," 2013. [Online]. Available: http://www.splevo.org

[9] B. Klatt and K. Krogmann, "Towards Tool-Support for Evolutionary Software Product Line Development," in *Proceedings of WSR'2011*, 2011.

[10] ——, "Model-Driven Product Consolidation into Software Product Lines," in *Proceedings of MMSM'2012*, 2012.

[11] OMG, "Architecture-driven Modernization : Abstract Syntax Tree Metamodel (ASTM)," OMG, Tech. Rep. January, 2011.

[12] ——, "Architecture-Driven Modernization : Knowledge Discovery Meta-Model (KDM)," OMG, Tech. Rep., 2011.

[13] Eclipse Foundation, "EMF Feature Model," 2012. [Online]. Available: http://www.eclipse.org/modeling/emft/featuremodel/

[14] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of ASE'10*. ACM, 2010.

[15] B. Klatt and M. Küster, "Respecting component architecture to migrate product copies to a software product line," in *Proceedings of WCOP'12*. ACM Press, 2012.

[16] B. Fluri and H. Gall, "Classifying Change Types for Qualifying Change Couplings," in *Proceedings of ICPC'06*. IEEE, 2006.

[17] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, 2009.

[18] A. Kuhn, S. Ducasse, and T. Girba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[19] B. Klatt, "GCD Calculator Example," 2012. [Online]. Available: http://sdqweb.ipd.kit.edu/wiki/GCD_Calculator_Example

[20] J.-M. Dautelle, "JScience," 2012. [Online]. Available: http://www.jscience.org/

[21] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proceedings of IWPC'2000*. IEEE, 2000.

[22] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal Of Software Maintenance Research And Practice*, vol. 7, no. 1, 1995.

[23] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," *Software Quality Journal*, vol. 17, no. 4, pp. 331–366, Mar. 2009.

[24] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring Product Lines," in *Proceedings of GPCE'06*. ACM, 2006.

[25] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of ICSE'11*. IEEE, 2011.

[26] K. Yoshimura, Y. Atarashi, and T. Fukuda, "A Method to Identify Feature Constraints Based," in *SPLC'10*. Springer Berlin Heidelberg, 2010, pp. 425–429.

[27] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting Software Product Lines : A Case Study Using Conditional Compilation," in *Proceedings of CSMR'11*, 2011.

Extracting variability from C and lifting it to mbeddr

Federico Tomassetti
Politecnico di Torino
C.so Duca degli Abruzzi 24,
Torino, Italy
federico.tomassetti@polito.it

Daniel Ratiu
Fortiss
Guerickestr. 25,
Munich, Germany
ratiu@fortiss.org

***Abstract*—Information about variability is expressed in C through the usage of preprocessor directives which interact in multiple ways with proper C code, leading to systems difficult to understand and analyze. Lifting the variability information into a DSL to explicitly capture the features, relations among them and to the code, would substantially improve today's state of practice. In this paper we present a study which we performed on 5 large projects (including the Linux kernel) and almost 30M lines of code on extracting variability information from C files. Our main result is that by using simple heuristics, it is possible to interpret a large portion of the variability information present in large systems. Furthermore, we show how we extracted variability information from ChibiOS, a real-time OS available on 14 different core architectures, and how we lifted that information in mbeddr, a DSL-based technology stack for embedded programing with explicit support for variability.**

***Keywords*-software product lines; abstraction lifting; embedded software; projectional editors; reverse engineering.**

I. Introduction

For decades the C language has been the language of choice in developing embedded systems[1]. Nevertheless development in C is affected by problems due to the presence of preprocessor directives intermingled inside the C code. The preprocessor complement fundamentally to the expressiveness of the C language permitting conditional compilation, constant definitions, or basic meta-programming support. However the preprocessor favors also the presence of bugs [1] and lead to code that is extremely difficult to understand and analyze. For example, the `ifdef` directive permits to implement variability with the goal of obtaining portability or implementing product lines but it can be easily abused leading to situations in which the code is hard to understand and all possible variants are extremely difficult to analyze [2].

Mbeddr[2] [3] is a technology stack based on language engineering that defines domain specific extensions of C for embedded programing – e.g., components, state-machines, physical units. Mbeddr is built on top of the Meta Programming System (MPS) from JetBrains, which is a projectional language workbench. Since the extensions are based on C, mbeddr can be easily integrated with existing C code. While most of the syntax and semantics of C is preserved in mbeddr some notable features were removed to create a new language easily analyzable. Notably preprocessor directives are not supported and for many common usages of the preprocessor, explicit language constructs are provided.

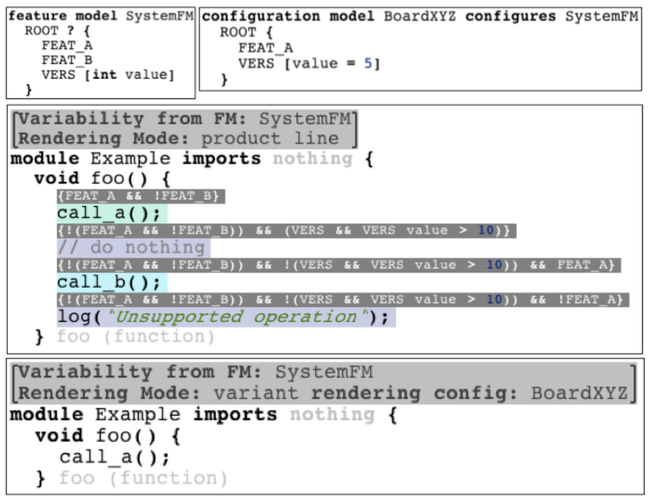
[1]According to the "Transparent Language Popularity Index" it is still the most popular programming language in general in January 2013.

[2]http://mbeddr.com

```
void foo()
{
#ifdef FEAT_A && !FEAT_B
    call_a();
#elif defined(VERS) && VERS > 10
    // do nothing
#else
#ifdef FEAT_B
    call_b();
#else
    log("Unsupported operation");
#endif
#endif
}
```

Figure 1. Variability in C is expressed implicitly through the preprocessor.

In mbeddr is possible to represent explicitly software product line concepts [4] with consequent advantages in terms of analizability (e.g., if feature models are consistent) [5] and comprehension. To operate with the existing code base written in C, an importer is needed to first capture variability expressed using preprocessor directives (like in Figure 1) and then to re-express it using the specific constructs provided by mbeddr (like in Figure 2). As shown in Figure 2, in mbeddr, feature models and configurations are captured through domain specific constructs and the linking of features to code is explicitly represented by annotations on the statements. The annotations contain an expression with references to features which determine if the statement will be included under a particular configuration. These expressions are called presence conditions. Due to the projectional editing capabilities of MPS, programs can be displayed as the entire product line or as individual variants. It means the developer could visualize all possible statements with their presence conditions, or only the statements that will be included when a particular configuration it is used. Importing the C code and variability in mbeddr means to lift the level of abstraction from token level (the level at which the preprocessor operates) to a domain specific level where variability concepts are expressed as first class citizens.



```
feature model SystemFM
  ROOT ? {
    FEAT_A
    FEAT_B
    VERS [int value]
  }
```

```
configuration model BoardXYZ configures SystemFM
  ROOT {
    FEAT_A
    VERS [value = 5]
  }
```

```
[Variability from FM: SystemFM]
[Rendering Mode: product line ]
module Example imports nothing {
  void foo() {
    {FEAT_A && !FEAT_B}
    call_a();
    {!(FEAT_A && !FEAT_B)) && (VERS && VERS value > 10)}
    // do nothing
    {!(FEAT_A && !FEAT_B)) && !(VERS && VERS value > 10)) && FEAT_A}
    call_b();
    {!(FEAT_A && !FEAT_B)) && !(VERS && VERS value > 10)) && !FEAT_A}
    log("Unsupported operation");
  } foo (function)
```

```
[Variability from FM: SystemFM                            ]
[Rendering Mode: variant rendering config: BoardXYZ]
module Example imports nothing {
  void foo() {
    call_a();
  } foo (function)
```

Figure 2. Variability in mbeddr: on top you see the feature model (left) and a configuration of that feature model (right). Below there are two projections of the code: the first one shows all the possible variant, the second one the variant corresponding to the configuration.

The rest of the paper is organized as follows: we start describing how variability is represented in C (Sect. II) and later present an empirical analysis of usages of variability among large C projects (Sect. III). In Sect. IV we report about our experience in extracting variability from ChibiOS source code to mbeddr. Finally we introduce related work (Sect. V) and draw our conclusions (Sect. VI).

II. How variability is expressed in C

The preprocessor when it is invoked can receive a set of parameters, called the *initial configuration*. The *initial configuration* specifies which macros are initially defined and their initial values. During the preprocessing new macros can be defined, and the existing ones can change their value or being undefined. This process is called *configuration processing*. Based on the current configuration (the set of macros being defined and their associated value at a given moment) declarations and statements are included or excluded in the code to be compiled. The expressions determining the inclusion/exclusion of C elements are called *presence conditions*. In addition to that, the preprocessor statements `#error` and `#warning` can be used to issue errors and warnings to the user when a particular configuration is not acceptable or it is deprecated.

Some approaches to extract variability from C (e.g., [6]) require to process the initial configuration and rewrite presence conditions in term of the initial configuration instead of the current configuration. While this is a sound solution for analysis, we conjecture that in C important information is expressed by the combinations of these two different mechanisms: *configuration processing* and *presence conditions*. The first can be used to specify derivation rules that determine the values of symbols later used in *presence conditions*. Consider the example presented in Listing 1. First the *configuration processing* determines that `X_SHOULD_BE_ACTIVATED` will be defined only when the condition `defined(FEATURE_A) && (PARAM_B >0x1010` will have the value `true`. Subsequently, based on the fact that `X_SHOULD_BE_ACTIVATED` is defined or not, two different statements could be included or excluded (both of them located inside the function `foo`). While we could rewrite the presence conditions to `defined(FEATURE_A) && (PARAM_B>0x1010)` and discard all the *configuration processing* we think that this would cause a loss of information which could be useful while maintaining the system.

```
#if defined(FEATURE_A) && (PARAM_B>0x1010)
#define X_SHOULD_BE_ACTIVATED
#endif
void foo()
{
#ifdef X_SHOULD_BE_ACTIVATED
  invoke_x_init();
#endif
  ...
#ifdef X_SHOULD_BE_ACTIVATED
  invoke_x_release();
#endif
}
```

Listing 1. Example of C and preprocessor code containing both configuration processing and presence conditions

III. Empirical analysis of variability in large C projects

To lift the variability information in mbeddr, we need to understand how is it expressed in large C programs, this is the goal of the analysis introduced in this section. In particular we want to investigate if preprocessor directives in the context of large projects are used in a disciplined way to represent variability. If that is the case we could identify usage patterns of the preprocessor that cover the majority of cases in the practice and exploit them in interpreting variability.

A. Research questions

Specifically, we aim to answer the following questions:

RQ1) *Which are the typical building blocks in presence conditions?* This is important in order to understand which kind of expressions we need to support in the higher level configuration language.

RQ2) *Which changes (re-* #define*s and* #undef*s) are operated on a defined symbol?* Depending on changes upon defined symbols, defines can be lifted (or not) as constant configuration values.

RQ3) *Are* #error *and* #warning *used in practice?* If they are, it could be possible to extract feature model constraints from them.

B. Analysis approach

In this section we present the general approach we adopted to answer our research questions. We present the projects we chose to analyze (III-B1), which information we extracted from source files and how (III-B2), how we modelled variation points (III-B3).

1) Projects: To perform our analysis we selected established large projects from different domains. They are:

- **Apache OpenOffice:** it is a suite of six personal productivity applications. It is ported on Windows, Solaris, Linux, Macintosh and FreeBSD. It derives from StarOffice, which was developed since 1984.
- **Linux:** Linux is an OS kernel developed since 1991. It is arguably one of the existing projects which is more portable being available on more than 20 architectures.
- **Mozilla:** Mozilla is a suite of different projects including the Firefox browser for desktop and mobile systems and the Thunderbird e-mail client. It was created by Netscape in 1998.
- **Quake:** it is a videogame released during 1996. It runs on DOS, Macintosh, Sega Saturn, Nintendo 64, Amiga OS.
- **VideoLAN:** It is a multimedia player, supporting a large variety of audio and video formats. It has been ported to Microsoft Windows, Mac OS X, Linux, BeOS, Syllable, BSD, MorphOS, Solaris and Sharp Zaurus. The first release is dated 2001.

Some data about the dimension of projects chosen is reported in Table I. We chose these projects because they are multi-platform projects, from different domains and they are written mainly in C or other languages sharing the same preprocessor. In particular Apache OpenOffice, Mozilla and VideoLAN contain also Objective-C and C++ files. We considered more than 73.000 files with a total of more than 2.1 millions of preprocessor statements.

2) Information extraction: We focus on the `define` statements because they are used to implement *configuration processing* and on the `ifdef`, `ifndef`, `if`, `elif` and `endif` statements because they can be used to express *presence conditions*. We excluded from our analysis statements that, while being of one of the previous types, were not used to express variability. To do that we built for each file a model of the information that are relevant to the preprocessor. We call this model the *preprocessor model*.

Preprocessor model: A *preprocessor model* is an ordered list of the elements contained in a source file. Possible elements are: preprocessor statements, blank lines, comment lines[3] and code lines. Preprocessor statements are recognized from lines not contained in comments which starts with the '#' symbol. They can span across multiple lines when a line is terminated with the '\' character. Comment lines are lines containing whitespaces and comments, blank lines are lines composed only by whitespaces and not included in a multi-line comment. Finally code lines are lines containing some code (they can also include comments). Note that while not parsing the C/C++/Objective-C code the parser have still to be able to handle correctly string and char literals to recognize comments. Data about the number of lines contained in each project are reported in Table I.

Parsing preprocessor expressions: In addition to classify the lines, we parsed the expressions contained in preprocessor statements and inserted them in the *preprocessor model*. In particular we calculated the value of the conditions associated to preprocessor statements `ifdef`, `ifndef`, `if` and `elif` and the expressions specified by `define` statements. While statements `ifdef`, `ifndef` can express only simple presence conditions (based on the presence or absence of one single configuration symbol), `if` and `elif` can specify very complex expressions. To parse those complex expressions we used the grammar presented in Listing 1 (whitespaces and comments were ignored, including the backslash followed by a newline, which is used inside preprocessort statements to continue on the next line). Our parser implementation takes in account the precedence between operators.

```
⟨expression⟩ ::= '(' ⟨expression⟩ ')'
            | ⟨define⟩
            | ⟨flag_value⟩
            | ⟨logical_binary_op⟩
            | ⟨comparison_op⟩
            | ⟨number_literal⟩
            | ⟨char_literal⟩
            | ⟨math_op⟩
            | ⟨bitwise_binary_op⟩
            | ⟨logical_not⟩
            | ⟨bitwise_not⟩
            | ⟨macro_function_call⟩
⟨define⟩ ::= 'defined' '(' ⟨identifier⟩ ')'
            | 'defined' ⟨identifier⟩
⟨flag_value⟩ ::= ⟨identifier⟩
⟨identifier⟩ ::= +[_a-zA-Z][_a-zA-Z0-9]*
⟨logical_binary_op⟩ ::= ⟨expression⟩ ('&&'|'||') ⟨expression⟩
⟨comparison_expression⟩ ::= ⟨expression⟩ ('<='|'>='|'>'|'<'|'=='|'!=') ⟨expression⟩
⟨number_literal⟩ ::= ...
⟨char_literal⟩ ::= ...
⟨bitwise_binary_op⟩ ::= ⟨expression⟩ ('«'|'»'|'&'|'|') ⟨expression⟩
⟨math_op⟩ ::= ⟨expression⟩ ('+'|'-'|'*'|'/'|'%'|'^') ⟨expression⟩
⟨logical_not⟩ ::= '!' ⟨expression⟩
⟨bitwise_not⟩ ::= '~' ⟨expression⟩
⟨macro_function_call⟩ ::= ⟨identifier⟩ '(' (⟨expression⟩ (',' ⟨expression⟩)* )? ')'
```

Grammar 1. Grammar used to parse presence conditions. Definitions of literals are omitted.

Using this grammar we were able to parse correctly a large majority of the conditions expressed: out of more than 185K expressions analyzed we could not parse only three. Expressions that could not be parsed are reported in Listing 2. The same grammar can be used to parse a portion of the `define` statements, when valid expressions are assigned to symbols. In some cases however `define` can assign to symbols arbitrary tokens instead of expressions, for example complete or incomplete statements. This is not a problem relavant for feature model and configura-

[3]Comment lines were included in the model because as future work we aim to associate comments to preprocessor statements (based on adjacency) and import also them.

Description			Files					Lines				
Project	Domain	Version	C	C++	Obj-C	H	Total	Blank	Comm.	Code	PP	Total
AOO	Productivity	V. 3.4.1 Linux DEB	158	11,029	51	12,107	23,345	1.2M	1.4M	5.3M	430K	8.2M
Linux	OS	Vv. 3.6.5	17,448	0	0	14,379	31,827	2.0M	2.2M	9.0M	1.3M	14.5M
Mozilla	Web	Tag FIREFOX_AURORA_19_BASE	3,118	4,502	180	8,050	15,850	807K	991K	3.7M	326K	5.9M
Quake	Gaming	Commit bf4ac424ce...	240	0	0	145	385	34K	43K	123K	8K	198K
VideoLAN	Multimedia	V. 1.3.0	778	255	81	1,311	2,425	97K	110K	426K	45K	678K
Total			21,742	15,786	312	35,992	73,832	4.1M	4.7M	19M	2.1M	29.5M

Table I
SIZE OF THE PROJECTS CONSIDERED. AOO = APACHE OPENOFFICE, H = HEADER, COMM. = COMMENT, PP = PREPROCESSOR.

tion extraction because symbols with syntactic content are not referred inside *presence conditions*, which have to be evaluable expressions. Parsing the values of both object-like and function-like symbols but excluding `define` with empty values, expressions parsable by our grammar ranged from 82% (for VideoLAN) to 95% for Linux.

```
// Mozilla
#if defined(LARGEFILE64_SOURCE) && - \
   _LARGEFILE64_SOURCE - -1 == 1
#if -_LARGEFILE64_SOURCE - -1 == 1
// Apache OpenOffice
#if NFWORK < (NAM$C_MAXRSS + 1)
```

Listing 2. All the expressions not parsed by our grammar

Preprocessor usages excluded: We used the *preprocessor model* to identify the preprocessor statements that were not related to variability. We defined two patterns to be searched in the model, ignoring blank and comment lines. In particular we excluded:

- **Double inclusion guards protecting modules:** we recognized them when a `ifndef` (or an `if` with an expression of type `!defined(SYMBOL)`) was at the very beginning of the file, immediately followed by a `define`, which: i) defined the same symbol used by the previous statement, ii) had no value specified. Finally a `endif` had to be the last element of the file. When recognizing this pattern we ignored the three preprocessor statements involved. This pattern was recognized in most of header files. It is used to prevent an accidental double inclusion of the same header file.
- **Redefinition guards:** we recognized them when a `ifndef` (or an `if` with an expression of type `!defined(SYMBOL)`) was followed by a `define` which defined the same symbol. This line had to be followed by an `endif`. When recognized the first and the third statements were excluded, while the second was marked as a guarded redefinition. Redefinition guards are often used to avoid warnings from the compiler.

3) Calculate conditions of variation points: As we discussed in Section II `if`, `ifdef` and `ifndef` can be used to define presence conditions. These directives open constructs which are terminated by an `endif` and can contain one `else` clause and any number of `elif` clauses. Each of these constructs individuate one or more portions of codes, which can contain other preprocessor statements or C elements. It is possible to insert other conditional constructs inside these portions, i.e., it is possible to have annidated conditional constructs. The portions of code individuate by the constructs are classified in three kinds:

- **Then block:** this is the area between the the `if`, `ifdef` or `ifndef` opening the construct and the first among `elif`, `else` or the `endif` which are parts of the same construct (i.e., we do not consider `elif` or `else` or the `endif` of annidated constructs).
- **Else block:** this is the area between the `else` and the `endif` closing the construct.
- **Elif block:** this is the area between the `elif` and the next `elif`, the `else` or the `endif` block of the same construct.

We map each `if` / `ifdef` / `ifndef` construct to a *Variation point* and each of its block to a *Variation point block.*

For each *Variation point block* a *specificCondition* and a *complexiveCondition* can be calculated. The *specificCondition* of a *ThenBlock* is just obtained from the expression following the `if`, `ifdef` or `ifndef` statement. In the case of the *ElifBlock* it is composed through a logical *and*: i) the condition of the corresponding *ThenBlock* negated, ii) the condition of all the preceding *ElifBlocks* negated, iii) the condition created from the expression following the specific `elif`. In the case of *ElseBlock* it is created composing through a logical *and*: i) the condition of the corresponding *ThenBlock* negated, ii) the condition of all the *ElifBlocks* negated. The *complexiveCondition* corresponds to the *specificCondition* if the block is part of a *VariationPoint* which is not annidated, otherwise it corresponds to the *complexiveCondition* of the block containing the *VariationPoint* in logical *and* with the *specificCondition* of the block.

C. Results and discussion

In this subsection we present how we addressed each RQ and the corresponding results (III-C1, III-C2, III-C3). Later we discuss our findings (III-C4). All the analysis do not consider the statements excluded for the reasons explained in Par. III-B2.

1) Addressing RQ1: Presence conditions: To answer this question we analyzed all the expressions from `if`, `ifdef`, `ifndef` and `elif` statements. For each type of expression (*Identifier*, *NumberLiteral*, *ComparisonOp.*, etc.) we counted in how many of the expressions considered it was used. To do that we looked at the type of the expression itself and the type of all its sub-expressions, recursively.

Results: We report frequencies of the different types of expression in Table II. We can see that, as easily predictable, identifiers are referred in most of the presence conditions (84.6%-98.1%). Presence conditions which instead do not refer to identifiers or macro function calls are constants: they are always true or false, independently of the current configuration. Many of the expressions not referring to identifiers are composed by only one costant, either '0' (false) or '1' (true). An `if` having as expression '0' cause the exclusion of all the elements contained in the *ThenBlock*. An `if` having as expression '1' leaves always untouched the elements in the *ThenBlock*. The remaining expressions without identifiers could still have a documentation role, showing the reasoning process bringing to include or exclude a particular set of statements. The most common operations are logical operations (`!`, &&, ||) which are present in many presence conditions (between 1/5 to 2/3 of the expressions considered). Comparison operations (<,>,<=,>=,==,`!=` are also relevant as well as number literals. Math (+,-,*,/,%,∧) and bitwise operations («,»,&,|,) are very infrequent (they appear in less than 1% of the presence conditions). Quite infrequent are also macro function calls which are not used at all in one of the projects considered and seem to be marginally relevant only in VideoLAN and Linux (being contained in slightly more than 1% of all examined *presence conditions*). Observing the nature of macro functions used in presence conditions we noticed that quite frequently they are just implemented using stringifications[4] to compose different tokens creating a new identifier.

2) Addressing RQ2: Configuration processing: Technically the value of a macro can vary during the execution of the preprocessing. A scenario like the one presented in Listing 3 is therefore possible. In this example two functions (`foo_a` and `foo_b`) have the same *presence condition* (`XYZ` have to be defined) but because of the changes in the definition of `XYZ` (initially defined and then undefined) `foo_a` will be included while `foo_b` will be not.

```
#define XYZ
#ifdef XYZ
void foo_a(){};
#endif
#undef XYZ
#ifdef XYZ
void foo_b(){};
#endif
```

Listing 3. Example of configuration processing varying the value of a macro

The designers of Mbedder considered these consequences of the configuration processing confusing and do not support it in their variant of C; they instead consider configuration values to be constant.

While in general preprocessing symbols can be used in very different ways, we examined how frequently they are used as simple constants. We found three cases in which they behave as simple constants. To individuate instances of these cases we analyzed how a particular symbol was defined, re-defined or undefined in the scope of a complete project (because the preprocessor do not implement local scopes). One condition applies to all these cases: the symbol considered should be never undefined, because it would mean to limit its scope, while we are looking for symbols behaving as constants which are available to the whole system. The cases considered are:

- **Symbols defined once:** symbols that are defined just once in the scope of the project considered.
- **Symbols re-defined always to the same value:** symbols that are defined two or more times but they are assigned always exactly the same expression (possibly the empty value, meaning that they are defined but they have not an associated value).
- **Symbols re-defined under different conditions:** symbols which are defined two or more times, but every time they are defined under a particular condition they are defined to the same value.

To be able to recognize the symbols re-defined under different conditions we had to be able to calculate the *presence condition* under which a particular definition would be used. Consider the example given in Listing 4. In that example the same symbol (`VAL`) could assume different values. The value 1 is assumed only when the condition `defined(A) && defined(B)` is satisfied, the value 2 is assumed when the condition `defined(C) || defined(D)` is satisfied, otherwise the symbol remains undefined. To calculate the *presence condition* of a given definition is not trivial because `if`, `ifdef` and `ifndef` constructs can be annidated and also the role of `elif` and `else` have to be considered. To this operation we used the technique presented in Sub-subsection III-B3.

```
#if defined(A) && defined(B)
#define VAL 1
#endif
#if defined(C) || defined(D)
#define VAL 2
#endif
```

Listing 4. Example of definitions under different presence conditions

Results: Table III reports some data about the number of definitions and undefinitions considered and the number of symbols involved. It is possible to notice that the number of symbols undefined is many times smaller than the number of symbols defined. A symbol can be undefined for different reasons. One reason is to avoid compiler warnings: a symbol already defined can be undefined immediately before a statement defining it again. Another reason is to mimic the concept of scope: by undefining the symbol we are guaranteed previous definitions preceding the undefinition will not affect the code following the undefinition.

[4]See http://gcc.gnu.org/onlinedocs/cpp/Stringification.html for an explanation of stringification.

Expr. Type	AOO	Linux	Mozilla	Quake	VideoLAN	Range
Identifier reference	98.1	95.8	97.1	84.6	93.4	84.6-98.1
Number literal	7.9	7.0	6.5	15.7	10.0	6.5-15.7
Logical op.	66.3	17.9	22.3	28.3	20.9	20.9-66.3
Comparison op.	6.3	3.4	4.0	0.3	3.4	0.3-6.3
Math op.	0.04	0.1	0.1	0	0.3	0-0.3
Bitwise op.	0.01	0.5	0.01	0	0.3	0-0.5
Macro function call	0.01	1.3	0.2	0	1.5	0-1.5
'0'	1.8	3.5	2.7	14.6	5.5	1.8-14.6
'1'	0.2	0.5	0.2	0.8	0.5	0.2-0.8

Table II
PERCENTAGE OF PRESENCE CONDITIONS CONTAINING THE GIVEN TYPE OF EXPRESSION ACROSS THE DIFFERENT PROJECTS CONSIDERED. FOR THE EXPRESSIONS 0 AND 1 IT IS INSTEAD THE NUMBER OF EXPRESSIONS CORRESPONDING EXACTLY TO 0 OR 1. LAST COLUMN REPORT THE RANGE (THE SPACE DETERMINED BY THE MINIMUM AND MAXIMUM VALUES AMONG ALL PROJECTS).

	Definitions						Undefinitions		Errors and warnings		
Project	Sym	Def	D1	D2	D3	D4+	Sym	Undef	Error	Warning	Perc.
AOO	44K	64K	69.9%	25.4%	1.5%	3.1%	0.6K	1.3K	195	0	0.05%
Linux	656K	787K	90.5%	6.6%	1.4%	1.5%	1.9K	3.3K	735	76	0.07%
Mozilla	51K	70K	83.1%	10.9%	2.0%	4.0%	2.0K	3.6K	694	39	0.26%
Quake	3K	5K	72.5%	23.5%	1.9%	2.0%	9	59	0	0	0%
VideoLAN	13K	15K	92.7%	5.5%	0.7%	1.1%	0.5K	0.6K	31	67	0.26%

Table III
DATA ABOUT USAGE OF PRESENCE CONDITIONS AND USAGE OF ERROR AND WARNING DIRECTIVES. DEFINITIONS/SYM = NUMBER OF SYMBOLS DEFINED AT LEAST ONCE, DEF = NUMBER OF `define`, D1 = RATIO OF SYMBOLS DEFINED ONCE, D4+ = RATIO OF SYMBOLS DEFINED FOUR OR MORE TIMES, UNDEFINITIONS/SYM = NUMBER OF SYMBOLS UNDEFINED AT LEAST ONCE, UNDEF = NUMBER OF `undef`. PERC. = PERCENTAGE OF ERROR AND WARNING DIRECTIVES AMONG ALL THE PREPROCESSOR STATEMENTS.

In general preprocessing symbols can be re-defined or undefined multiple times. For example in the Quake project the symbol `INTERFACE` is defined or undefined 46 times, `LOG` is defined or undefined 223 times in Mozilla and `pr_fmt` is defined or undefined 995 times in Linux. This happens for a variety of reasons. In some cases different definitions of the same symbols are contained in header files which are alternatively included in the compiled system. In other cases different definitions of the symbols are used in separated subsystems, so they just happen to have the same name but they are intended as different values to be used in different contexts.

In table IV we report the frequencies of the particular cases described in which we can equiparate symbols to constants. As we can see in all the projects considered the percentage of symbols that are covered by these special cases ranges between 95% and 99%. It means it is feasible to automatic lift a large portion of the symbols, while a minority of them have to be manually converted.

3) Addressing RQ3: Error and warning directives: We simply counted the number of `error` and `warning` directives used.

Results: Data is available in Table III. Only one project (Quake) do not use them at all. In general we can notice that `error` directives are more used than `warning` directives; this is true for 3 out of 4 projects, while in the VideoLAN project `warning` directive are used twice as much as `error` directives. In general these directives constituted between 0.05% and 0.26% of all the preprocessor statements, if they are used at all.

4) Discussion: From our results we can tell that:

- **RQ1** Presence conditions can contain a range set of different expressions. However mathematical and bitwise operations are rarely used, as well as macro function calls, so they have not to be necessarily supported in mbeddr: the expressions not supported (if found in the projects the user want to import) can be manually addressed.
- **RQ2** Most of preprocessing symbols are used in practice as global constants, being never redefined to a different value under the same presence condition and being never undefined, therefore our simplifications appear to be reasonable and would permit to lift in mbeddr most of the symbols.
- **RQ3** `error` and `warning` directives are not necessarily used by all projects. Therefore in some cases constraints between features have to be extrapolated from other information sources or be manually described.

These results suggest it is feasible to extract and lift automatically a large portion of the variability information from C projects, while limited human intervention can be still needed.

IV. EXPERIENCE WITH IMPORTING CHIBIOS INTO MBEDDR

ChibiOS is a real-time operating systems supporting 14 core architectures, different compilers and different platforms. We chose it for our case study because it is a well written, complex embedded system with very high usage of variability to support portability. In our case study we used the code of version 2.5.1. The system is composed from many modules:

Expr. Type	AOO	Linux	Mozilla	Quake	VideoLAN	Range
single definition	69%	90%	80%	72%	90%	69%-90%
re-definition to the same value	23%	6%	7%	24%	2%	2%-24%
definitions under different conditions	2%	2%	9%	2%	4%	2%-9%
total	94.8%	97.9%	96.3%	98.6%	95.1%	94.8%-98.6%

Table IV
SPECIAL CASES IN WHICH MACROS CAN BE LIFTED TO HIGHER LEVEL CONCEPTS.

- *boards*: it contains specific code for 35 boards and a board simulator,
- *demos*: it contains 42 demos for different boards and compilers combinations (some of the demos are then divided in sub-demos),
- *os*: the directory containing the core of the system,
- *test* and *testhal*: contain source code for testing the system under different configurations.
- *tools*: tools which complement ChibiOS.

Given this organization we segmented the global system in sub-systems. In this section we first discuss how we extracted a feature model from the OS Kernel module (IV-A), then how we extracted a configuration for that feature model from the subsystem containing the code of a demo for a particular platform (IV-B). Finally we discuss the experience (IV-C).

A. The OS Kernel module

We start our analysis from the *os/kernel* subsystem because it contains code that have to work with all the architectures, boards and compilers supported by the system. Thereby, this is the module where portability is more important. Examining the code we noticed that most of the presence conditions have the shape `'a-sub-expression' || defined(__DOXYGEN__)`. This has the goal of making visible pieces of code to a tool used to produce documentation. Because this symbol is not related to variability we substituted it with the value `'0'` (which evaluates to false for the preprocessor) and simplified the expressions containing it (for example `defined(A) || defined(__DOXYGEN__)` would become `defined(A)`). This permits to identify as redefinition guards snippets as the one presented in Listing 5.

```
#if defined(A) || defined(__DOXYGEN__)
#define A 123
#endif
```

Listing 5. A redefinition guard polluted by the __DOXYGEN__ symbol

We parsed correctly all the 41 files (18 C files and 23 header files). Excluding the statements described in Par. III-B2 we obtained 246 presence conditions expressions (all parsed correctly) and 233 definitions (both of symbols with object-like or function-like symbols). We examined which symbols were used in presence conditions: they were 54, none of them being a function-like symbol. We then examined all the definitions of these symbols in the subsystem to iteratively look for symbols that were indirectly referred by presence conditions. We found only the symbols `TRUE` and `FALSE` to be indirectly referred by presence conditions. Out of the total 56 symbols used (directly or indirectly) only 3 symbols were defined internally at the module: `CH_DBG_ENABLED`, `TRUE` and `FALSE`. Being the other 53 symbols never defined in the subsystem we know they have to be defined in the modules "using" the kernel module. For this reason we lifted them as features. We therefore created a feature model containing these 53 features.

The three symbols used in presence conditions and defined in the subsystem were instead imported as derived features. For `TRUE` and `FALSE` there was just one definition which was always valid (i.e., the definitions were not inserted in a variation point). `CH_DBG_ENABLED` instead had two definitions. The first one under the condition `CH_DBG_ENABLE_ASSERTS || CH_DBG_ENABLE_CHECKS || CH_DBG_ENABLE_STACK_CHECK || CH_DBG_SYSTEM_STATE_CHECK`, the second one under the opposite condition. In the first case it was defined to `TRUE`, in the second one to `FALSE`. We could import it automatically because the different conditions under which it was defined were disjoint, otherwise we would have needed human judgement to import it.

To complete the feature model we imported the extra constraints from warning and error statements. In the subsystem analyzed there were 6 errors statements and 0 warning statements. For each error we calculated the *presence conditions* and extracted the message.

B. Module demos/ARMCM3-STM32F103ZG-FATFS

The code of this module contains a definition for 31 out of the 53 features present in the feature model extracted by the OS Kernel. Other features could remain undefined or be defined in other modules which are compiled together with this one to produce the final demo: for example the module containing the board-specific code or the code specific for the ARM architecture. All the definitions of these features have a *presence condition* equals to true. It means they are always valid. The features have assigned in 28 cases either the value `TRUE` or `FALSE`. Note that those macros assume the value 1 or 0 (the preprocessor or the C language have not a boolean type). We decided to import them as booleans (which is supported by mbeddr-C) instead of simple integers. One of the remaining three features is defined without providing a value, while the other two assume the numerical values 0 and 20.

C. Discussion

ChibiOS is a system written extremely well and the usage of the preprocessor is very disciplinated. Because of that we were able to extract without human intervention a feature model and a configuration model from two of the modules of the whole system. We could extract some of the constraints of the feature model from `error` directives but most of them had to be manually specified. The type of values that can be associated to features was obtained indirectly, looking at how the symbol was used in the configuration and updating the feature model.

V. Related work

We classify the related work along four directions:

Substitute preprocessor directives Kumar et al. [7] discuss how to substitute some usages of preprocessor directives with features of the new standard of C++ (C++11). ASTEC [8] is a variant of C with support to syntactic macros. McCloskey et al. explain how they analyzed C code and refactor the preprocessor directives using these extensions. Both approaches target languages with no explicit support for variability. ASTEC in particular lift the level of abstraction from token to syntactic level.

Representing variability There are development tools which provide a more intuitive representation of variability concerns through the usage of background colors (e.g., [9]). Our work could be integrated with such tools.

Variability information extraction Some approaches aims to provide tools able to analyze the preprocessor directives and individuate possible bugs. Among these approaches one of the most relevant is TypeChef [6]. While the goal of these approaches is to achieve maximum accuracy and generality, our approach is to capture the intentions as expressed in the code.

Empirical study about preprocessor usage On this topic it is very relevant the analysis presented by Ernst et al. [10]. While their analysis is very deep and interesting, they considered less lines of code that we did and they did not address large projects as we did. The goal of their analysis is more general, while we focus specifically on variability. However we can confirm one of their findings: they reported that 86% of preprocessor symbols were defined just once among the 26 systems they considered. We found this value to range between 69% and 90%.

VI. Conclusions and future work

In this paper we examined how in practice large, long established projects use preprocessor directives to represent variability. Results suggest that this category of projects are quite disciplined in using them, as consequence idioms and patterns can be identified and exploited to extract a large part of the variability information present in the code using simple heuristics. Our approach aims to preserve the readability and the original intent expressed in the code: while it has some theoretical limitations, it seems to be applicable in practice, as suggested also by the results of our experience with ChibiOS.

Our approach have still to be improved and completed: we have to use the variability information extracted to decorate the statements with presence conditions. Possible improvements could include attributing automatically a type to the extracted features. As future work we plan to perform a case study on a project involving industrial partners.

Acknowledgments

We would like to thank all the contributors to the mbeddr project at Itemis and Fortiss for their support, Markus Völter for his suggestions and Christian Kästner for the useful discussions we had with him. We would like also to thank DAAD that partially financed this work. Mbeddr has been supported by the German BMBF, FKZ 01/S11014.

References

[1] K. Nie and L. Zhang, “On the relationship between preprocessor-based software variability and software defects,” in *High-Assurance Systems Engineering (HASE), 13th Int. Symp. on*, 2011, pp. 178 –179.

[2] H. Spencer and G. Collyer, “#ifdef considered harmful, or portability experience with c news,” in *Proc. of the Summer 1992 USENIX Conf.*, San Antionio, Texas, 1992, pp. 185–198.

[3] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, “mbeddr: an extensible c-based programming language and ide for embedded systems,” in *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity*, ser. SPLASH ’12. New York, NY, USA: ACM, 2012, pp. 121–140.

[4] M. Voelter and E. Visser, “Product line engineering using domain-specific languages,” in *Software Product Line Conference (SPLC), 15th Int.*, 2011, pp. 70 –79.

[5] D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb, “Language Engineering as Enabler for Incrementally Defined Formal Analyses,” in *FORMSERA’12*, 2012.

[6] A. Kenner, C. Kästner, S. Haase, and T. Leich, “Typechef: toward type checking #ifdef variability in c,” in *Proc. of the 2nd Int. Workshop on Feature-Oriented Software Development*, ser. FOSD ’10. New York, NY, USA: ACM, 2010, pp. 25–32.

[7] A. Kumar, A. Sutton, and B. Stroustrup, “Rejuvenating c++ programs through demacrofication,” in *Proc. of the 28th IEEE International Conference on Software Maintenance*, 2012.

[8] B. McCloskey and E. Brewer, “Astec: a new approach to refactoring c,” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 21–30, 2005.

[9] J. Siegmund, N. Siegmund, J. Fruth, S. Kuhlmann, J. Dittmann, and G. Saake, “Program comprehension in preprocessor-based software,” in *Computer Safety, Reliability, and Security*, ser. Lect. Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7613, pp. 517–528.

[10] M. Ernst, G. Badros, and D. Notkin, “An empirical analysis of c preprocessor use,” *Software Engineering, IEEE Transactions on*, vol. 28, no. 12, pp. 1146 – 1170, 2002.

Identifying Traceability Links between Product Variants and Their Features

Hamzeh Eyal-Salman, Abdelhak-Djamel Seriai, Christophe Dony and Ra'fat Al-msie'deen
UMR CNRS 5506, LIRMM
Université Montpellier 2 Sciences et Techniques
Place Eugène Bataillon, Montpellier, France
{Eyalsalman, Seriai, Dony, Al-msiedeen}@lirmm.fr

***Abstract*— usually a software product line (SPL) is developed by exploiting available resources of a set of software variants that deem similar. In order to reengineer such variants that are developed by ad-hoc reuse into software product line that are developed by systematic reuse, it is necessary to identify traceability links between features and source code in a collection of product variants. Information retrieval (IR) methods are used widely to achieve this goal. These methods handle product variants as singular entities. However when product variants are considered together, we can get additional information that improves IR results. This paper proposes an approach to improve IR results when they are applied to identify traceability links in a collection of product variants. The novelty of our approach is that we exploit commonality and variability across product variants at feature and implementation levels to apply IR methods in efficient way. The obtained results proved that our approach significantly outperforms direct applying IR technique in conventional way in term of precision and recall metrics.**

Keywords- Traceability links, features, source code, object oriented, variability, software product line, latent semantic indexing, product variants.

I. INTRODUCTION

SPL aims to reduce development cost and time by producing a family of software products at a time. According to software engineering institute (SEI) definition, a SPL is "a set of software-intensive systems sharing a common, managed set of features that satisfies the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [3]. Usually SPL is developed by exploiting available resources of a set of software variants that deem similar to build SPL core assets. Such resource includes: source code, design documents, features and so on [8].

Software product variants represent a set of similar products that are developed by ad-hoc reuse techniques such as "clone-and-own". These variants share some features and also differ in others to meet specific needs of customers in a particular domain. For example, Wingsoft Financial Management System (WFMS) was developed for Fudan University and then evolved many times so that all evolved WFMS systems have been used in over 100 universities in China [5].

At first glance, clone-and-own technique represents an easy and fast reuse mechanism so that it provides the ability to start from existing already tested code and then making required modifications to produce a new variant. However when number of product variants and features grows, such ad-hoc reuse technique causes critical problems such as: maintaining efforts will be increased because we should maintain each variant separately from others, and sharing features in new products will be more complicated. When these problems accumulate, it is necessary reengineering product variants into a SPL for systematic reuse.

The first step in this reengineering process is to identify source code elements that implement a particular feature across product variants. This mapping between features and corresponding source code elements is known as traceability links.

The identified traceability links can be used to facilitate products derivation process from SPL core assets, find dependency between features, facilitate program comprehension process and also no maintenance task can be completed without identifying source code elements that are relevant to the task at hand[11].

Numerous approaches that are based on IR techniques have been proposed to identify links between source code and features [6]. These approaches handle product variants as singular entities (one product at a time). However when product variants are considered together, we can get additional information that can help to improve IR techniques results. This information is about commonality and variability across product variants at feature and implementation levels.

In this paper, we propose new approach to identify traceability links between object oriented source code of a set of product variants and given features of these variants using latent semantic indexing (LSI). Our approach aims to divide LSI search space at feature and implementation levels for each variant into two partitions (or subspaces): common and variable partitions. At features level, common partition represents a set of features that are shared by all variants (common features) while other features in the same variant represent variable partition (optional features). At implementation level, common partition refers to source code elements that realize common features while other implementation in the same variant represents variable partition that realize optional features. Source code elements implementing common features are called common source code elements while source code elements implementing optional features are called variable source code elements. The intuition behind this dividing process is to isolate common features and their corresponding code in each product variant. Consequently, we can also isolate optional features and their corresponding code in each product variant. The experimental results show that our approach gives promised results comparing with applying LSI in conventional way (a variant as atomic chunk).

The remainder of this paper is organized as follows. Section 2 presents background and related work. Section 3 presents our approach. Section 4 shows experimental results and evaluation. Finally, Section 5 presents conclusion and future work.

II. BACKGROUND AND RELATED WORKS

This section describes LSI and traceability links in software engineering, and discusses related works.

A. *Traceability Links*

Traceability is the ability to describe and follow the life cycle of an artifact (requirements, design models, source code, etc.) created during the software life cycle in both forward and backward directions [13]. Traceability relations can refer to overlap, satisfiability, dependency, evolution, generalization/refinement, conflict or rationalization associations between various software artifacts. In general, traceability relations can be classified as horizontal or vertical relations. The former type refers to relation among artifacts at different levels of abstraction (e.g. between requirements and design) and the latter type refers to relation among artifacts at the same level of abstraction (e.g. among related requirements) [14].

Identifying traceability links among software artifacts at different levels of abstraction of product variants provide important information about development and maintenance a SPL. Such traceability is useful to derive concrete products from SPL core assets.

B. *LSI in Software Engineering*

Several IR methods exist such as: probabilistic method (PM), vector space method (VSM) and LSI [15]. All of these methods assume that all software artifacts are in textual format. In each method, one type of software artifact is treated as query and another type of artifact is treated as document. IR methods rank these documents against queries by extracting information about the occurrences of terms within them. The extracted information is used to find similarity between queries and documents. In the case of recovering traceability links, this similarity is exploited to recover traceability links that might exist between two artifacts, one of them is used as query.

LSI is an advanced IR method. The heart of LSI is singular value decomposition technique (SVD). This technique is used to mitigate noise introduced by stop words like (the, an, above, etc.) and to overcome two classic problems arising in natural languages processing: synonymy and polysemy. The intuition behind SVD is rather complex to be presented here and see [16] for further details.

We chose LSI because it already has positive results to address maintenance tasks such as concept location [15], detection in software [17], and recovery of traceability links between source code and documentation [1].

C. *Related Works*

A comprehensive survey about techniques that have been proposed to identify source code elements relevant to a feature can be found in [9]. These techniques depend on static, dynamic or textual analysis, or a combination of these. Static analysis examines structural information such as control or data flow. Dynamic analysis relies on execution trace according to scenarios related to specific feature(s). Finally, textual techniques examine words in source code using IR methods. All these techniques identify traceability links between source code and corresponding features in a single product while our approach considers a set of product variants. The following works represent the most relevant works to us.

Ghanam et al. [4] have proposed a method to keep traceability links between feature model (FM) of a SPL and source code up-to-date. When SPL evolves, the traceability links become broken or outdated due to evolution at features and implementation levels. Their method is based on executable acceptance tests (EAT). EAT refers to English-like specifications (such as: scenario and story tests).These EATs represent the specifications of a given feature and can be executed against the system to test the correctness of its behavior. Their approach starts from already existing links to make them up to date while our approach is differ from this work where we start from scratch and assume that no already existing links.

Rubin et al. [12] focused on locating distinguishing features of two product variants realized via code cloning. Distinguishing features mean those features that are present in one variant and absent in another. Their approach relies on capturing the information about unshared part of the code between two products. This unshared part can be obtained by comparing a variant’s source code that has the features of interest to another one that does not has. The distinguishing features between to variants reside on this unshared part of code. Their work aims at isolating source code that corresponds to distinguishing features and then apply feature location techniques in efficient way. However, if the number of distinguishing features is large their approach becomes infeasible because in this case we map a large number of features with a large part of code.

III. THE PROPOSED APPROACH

In this section, we describe input data of our approach; discuss how to divide each variant at feature and implementation levels into two partitions and how to apply LSI for recovering traceability links.

A. *An Illustrative Example*

Consider a collection of four variants of text editor system as shown in table 1 below. The initial product in this collection is T_Editor_V1.0. It supports just core features for any text editor such as: open, save and create a file. The initial product is enhanced to be *T_Editor_V1.1* by adding s*earch* and text *edit* features. *T_Editor_V1.2* is another enhancement of initial product. *T_Editor_V2.0* is an advanced variant of text editor. It supports all previous features in addition to *replace* feature.

Table1. Features Set of Four Text Editor Variants.

Product Name	Features
T_Editor_V1.0	Core (Open, Save, Create).
T_Editor_V1.1	Core, Search, Edit.
T_Editor_V1.2	Core, Print, Edit.
T_Editor_V2.0	Core, Search, Edit, Replace, Print.

B. *Input Data*

Our approach takes object oriented source code of a set of product variants and a given set of features of these variants as input (like table 1). Each feature is identified by its name and description. Feature description is a natural language description. This information about feature represents a domain knowledge that is usually available from product variants documentation. In our work, feature description consists of small paragraph or some sentences.

C. *Feature versus Object-Oriented Elements*

In the literature, there are many definitions of feature. In this work, we rely on the following definition: a feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [19]. We adhere to the classification given by [18] which distinguishes three categories of features: firstly, functional features express the behavior or the way users may interact with a product. Secondly, interface features express the product’s conformance to a standard or a subsystem. Finally, parameter features express enumerable, listable environmental or non-functional properties. In our work, we deal with functional aspects of features where functionalities are grouped together into at a high level of abstraction to form features.

As there are several ways to implement features [7], we assume that functional features either common or optional are implemented at the programming language level. Thus in an object oriented source code, functional feature can be implemented by different object oriented building elements (OOBEs). OOBEs include packages, classes, methods, attributes, etc. A feature has coarse granularity elements when its implementation consists of high level building units such as: packages, classes and interfaces. On the other hand, a feature is fine-grained when its code is composed by lower level units, such as methods, attributes, statements. In our work, we consider that a feature is realized at implementation level by a set of classes because the class represents a main building unit in any object oriented language.

D. *Identifying Common and Variable Partitions at Feature and Implementation Levels For Each Variant*

The goal of our approach is to reduce LSI search space as much as possible at feature and implementation levels. The

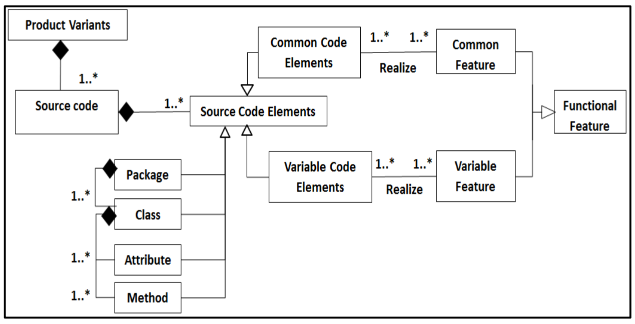


Figure1. Feature-Source code mapping model in product variants.

underlying intuition behind this goal is to map less features to less implementation in order to apply LSI in efficient way. In our previous work published in [22], we deal with product variants as a collection and divided this collection at feature and implementation levels into two partitions (common and variable partitions). At feature level, common and variable partitions represent common and optional features across product variants respectively. At implementation, common and variable partitions implement common and optional features respectively. During this current work, we will divide each product variant at feature and implementation levels into two partitions considering variability and commonality distribution across product variants.

Presence or absence a feature in product variant should be reflected in the implementation by presence or absence corresponding source code elements. Thus, we proposed an approach to divide each product variant at feature and implantation levels into two partitions as follow:

1) *At Feature Level*

We rely on lexical similarity of feature names and descriptions to determine common features across product variants such as core features in our illustrative example (*open, save and create*).

Algorithm1: ICFeatures.

Input:features sets of product variants PVF = { PF_1,…,PF_n}.
Output:common features (F_{sn}).

```
1:  $F_{sn} := PF_1$
2:  for  i := 2  to  length [PVF] do
3:        $F_{sn} := F_{sn} \cap PF_i$
4:  return  $F_{sn}$
```

For a set of features of set of product variants, our approach firstly defines a subset of same name features (*F_{sn}*) according to given above *ICFeatures* algorithm. *ICFeatures* takes as input sets of features of product variants (*PVF*) and return common features (F_{sn}). *PVF* represents a multiset data structure where each set corresponds to specific product variant. For instance, *PF_1* corresponds to product variant1 features. Step three compute shared features by conducting an intersection among all product variants features.

A feature may be renamed to response changes in software environment or the adoption of different technology [2]. Our approach considers this issue into account by computing lexical similarity pair-wisely for those features that don't have same name based on longest common subsequence (LCS) of their feature descriptions [21]. For example, for two features f_1 and f_2 where $f_1 \in PF_1$, $f_2 \in PF_2$ and f_1.name $\neq f_2$.name, if LCS for description of f_1 and f_2 has the same subsequence terms we can consider f_1 and f_2 represent the same feature.

By identifying common features across product variants, the rest features in any variant represent optional features. In our illustrative example, *core* features are common features across product variant while *search* and *edit* features represent optional features in *T_Editor_V1.1*.

2) At Implementation Level

Our approach analyzes source code of a set of product variants itself. Source code for each product variant is decomposed into a set of elementary construction units (ECU). ECU considers packages and classes. ECU takes the following format:

ECU = { Package.Name_Class.Name}.

This representation is inspired by the model construction operations proposed by [20]. Each product variant P_i is encoded as a set of ECUs, i.e. $P_i=\{ECU_1,ECU_2,\ldots,ECU_n\}$. We can note that our ECU can appear any structural changes at package and class levels. We call these changes as variations levels in the source code. These variations can reflect any changes at feature level (e.g. add or remove features) directly in the implementation level by adding or removing corresponding OOBEs.

Algorithm2: *IC_ECUs*

Input: Set of product variants (AllPV) abstracted as ECU. AllPV = $\{P_1,P_2,\ldots,P_n\}$.
Output: Common ECUs (C_ECUs)

```
1:  C_ECUs := P_1
2:  for i := 2  to  length [AllPV] do
3:        C_ECUs:=C_ECUs ∩ P_i
4:  return  C_ECUs
```

In order to identify common *ECUs* shared by all product variants, we proposed above *IC_ECUs* algorithm. *IC_ECUs* takes as input a set of product variants abstracted as a set of *ECUs* and returns common *ECUs* (common source code elements) across product variants. Step three compute shared *ECUs* by conducting an intersection among all *ECUs* of a set of product variants. These shared *ECUs* implement common features.

By identifying common *ECUs* (C_ECUs) across product variants, the rest ECUs in any variant represent variable *ECUs* (V_ ECUs). V_ ECUs represents variable source code elements. In our illustrative example, if we consider that *T_Editor_V1.0* is implemented by $\{ECU_1, ECU_2\}$ and *T_Editor_V1.1* is realized by $\{ECU_1, ECU_2, ECU_3, ECU_4, ECU_5\}$. Our approach reports that C_ *ECUs* = $\{ECU_1, ECU_2\}$ while *V_ECU* = $\{ECU_3, ECU_4, ECU5\}$.

E. Recovering Traceability Links By LSI

Domain knowledge and concepts are recorded in the source code through identifiers. Thus, our approach uses LSI for analyzing these elements to identify traceability links between common features and common source code elements (classes), and between optional features and variable source code elements in each product variant. Our applying of LSI is similar to [1]. It involves building LSI corpus and queries.

1) Building LSI Corpus

Our approach depends on four steps to process source code: (1) Identifiers extraction. (2) Tokenization. (3) Tokens manipulation. (4) Determining document granularity.

Firstly, identifiers extraction needs a parser to extract all source code information. During our work we used a Java parser to build abstract syntax tree of the source code that can be queried to extract required identifiers.

Secondly, identifiers must be tokenized. We considered two commonly styles for identifiers: one is the combination of words using underscore as delimiters (e.g., traceability_links); and the other is the combination of tokens using letter capitalization for separation (e.g., TraceabilityLinks). All identifiers that follow these rules are tokenized into singular tokens (e.g., traceability links for the above examples).

Thirdly, tokens are manipulated by reducing every token to its root. For example, *take, took* and *taken* are reduced to the same root *take*. Finally, we choose each class to be a separate document. A document contains lines of all identifier inside a class.

After source code processing, each product variant (*P*) is decomposed into a set of documents. These documents represent LSI corpus.

2) Building Queries

In our approach, LSI uses feature name and description as query to retrieve classes relevant to a specific feature. Our approach creates a document for each feature. This document contains feature name description, and it also is manipulated likes source code. Our approach extracts tokens from feature name and description. It uses white space and punctuation marks as delimiters. Then it reduces every token to its root.

3) Establishing traceability links

We feed LSI with documents and queries to build topic model. LSI builds a vector of weights for each document and query. Each weight represents a probability of affiliation for a given document and a query to the same topic. Then, LSI measures the similarity between queries and documents using cosine similarity. It returns a list of documents ordered based on their similarity against each query. In our work, we consider the most widely used threshold for cosine similarity that equals to 0.70 [1], i.e., documents that will be retrieved have a similarity with a query greater than or equal to the threshold value.

IV. EXPERIMENTAL RESULTS AND EVALUATION

In this section, we show the case study used for the evaluation of our approach, present the evaluation metrics and discuss the experimental results.

A. Case Study

We have applied our approach on a set of product variants from ArgoUML modelling tool. These variants represent members of ArgoUML-SPL[1] published in [10]. We generated four variants from ArgoUML-SPL as shown in table 2 below.

The four variants provide two common features (class diagram and cognitive support features) and seven optional features (state, collaboration, usecase, activity, deployment and sequence diagram features). The advantage of ArgoUML-SPL is that it implements features at different levels where features are implemented at package, class, method and attribute levels. Preprocessor directives have been used to annotate the source code elements associated to each feature. This pre-compilation process allows us to establish the truth links (real implementation for each feature) to evaluate the effectiveness of our approach.

Table 2: Set of ArgoUML-SPL members.

Products	Features
Product1	Class, cognitive, sequence, usecase, state, activity.
Product2	Class, cognitive, sequence, usecase, collaboration, activity.
Product3	Class, cognitive, collaboration, deployment, state.
Product4	Class, cognitive, state, activity, collaboration, deployment.

B. Evaluation Measures

We have used two measures to evaluate our approach: *Precision and Recall.* These measures are commonly used to evaluate IR methods [1].

1) Precision

Precision describes the precision of retrieved traceability links for a given feature. Precision is the percentage of correctly retrieved links (classes) to the total number of retrieved links. Equation 1 below represents *precision* metric equation where *i* ranges over the entire features set.

$$\text{Precision} = \frac{\sum_i \text{Correctly Retrieved Links}}{\sum_i \text{Total Retrieved Links}} \% \qquad EQ(1)$$

Precision values can have any value in the interval [0, 1]. Higher precision values mean better results for the approach that establishes traceability links.

[1] http://argouml-spl.tigris.org/

2) Recall

Recall quantifies number of relevant links that are retrieved for a given feature. Recall is the percentage of correctly retrieved links to the total number of relevant links. Below given equation 2 represents *recall* metric equation where *i* ranges over the entire features set.

$$\text{Recall} = \frac{\sum_i \text{Correctly Retrieved Links}}{\sum_i \text{Total Relevant Links}} \% \qquad EQ(2)$$

Recall values can have any value in the interval [0, 1]. Higher recall values mean better results for the approach that establishes traceability links.

C. Performance of Our Approach

LSI associate related tokens into topics based on their occurrences in the documents in a corpus. The most important parameter to LSI is the number of topics that should be used for topic-model building. We need enough topics to catch real term relations. Too many topics lead to associate irrelevant terms. Small number of topics lead to lost relevant terms. According to Dumais et al. [23], the number of topics is between 235 and 250 for natural language. For a corpus of source code files, Poshyvanyk et al. [24] recommended that the number of topics is 750.

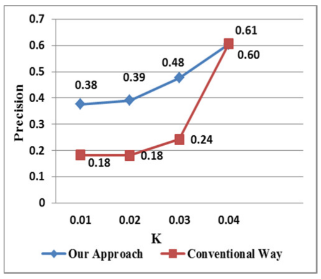
In this work we cannot use a fixed number of topics for LSI because we have different size of partitions. Thus, we use a factor *k* between 0.01 and 0.04 to determine number of topics. The number of topics (#topics) = k × doc_d, where doc_d is document dimensionality of term-document matrix that is generated by LSI. We evaluate the performance of our approach for #topic at *k= 0.01, 0.02, 0.03 and 0.04.*

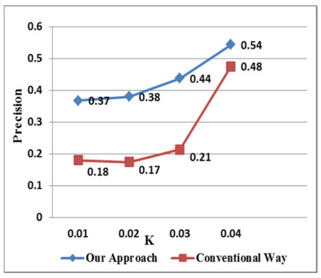
Figures 2 and 3 compare the precision and recall results for our approach against applying LSI in conventional way. The graphs *A,B,C,D* given in figures 2 and 3 corresponds to *Product1,Product2,Product3 and Product4* respectively. The X-axis in the graphs represents the number of LSI topics while Y-axis in the figures 2 and 3 correspond to precision and recall respectively. It can be noticed that our approach always gives a better precision and recall results than applying LSI in conventional way.

The threat to the validity of our approach is that if developers don't use the same vocabularies to name source code identifiers across product variants. This would means that lexical matching at implementation level will be effected. However, when a company has to develop a new product that is similar, but not identical, to existing ones, an existing product is cloned and later modified according to new demands.

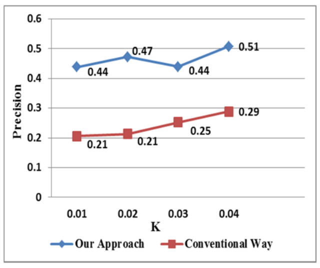
V. CONCLUSION AND FUTURE WORK

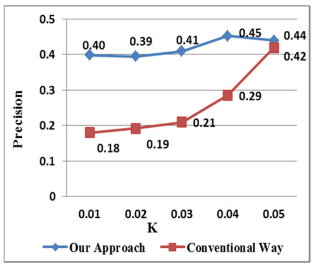
In this paper, we presented an approach that uses LSI in effective way to establish traceability links between the object oriented source code of a collection of product variants and a given features of these variants. Our approach exploits variability and commonality distribution of product





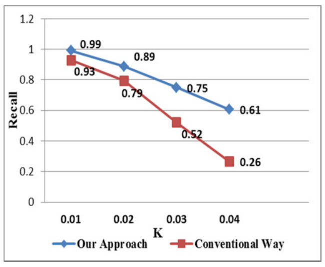
A B

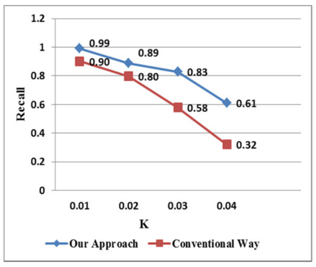




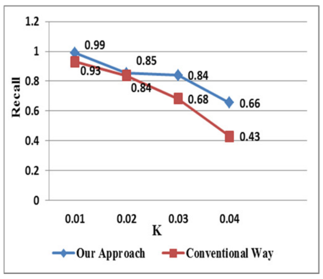
C D

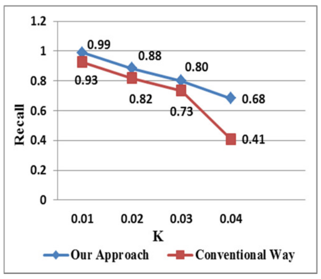
Figure 2. Precision results for our approach against applying LSI in conventional way.





A B





C D

Figure 3.Recall results for our approach against applying LSI in conventional way.

variants to reduce features and implementation spaces such that LSI can be applied in efficient way. The evaluation of our approach with a collection of four ArgoUML-SPL products showed that our approach significantly outperforms applying LSI in conventional way according to the precision and recall metrics.

In our future work, we plan to use existing relationships between source code elements (e.g., method call, class inheritance and son on) to improve the relevance of identified traceability links. This will require a definition for semantic similarity measure between source code elements.

REFERENCES

[1]. A. Marcus and J.I. Maletic. Recovering documentation-to-source code traceability links using Latent Semantic Indexing. ICSE 2003, pp.125-137.

[2]. Y. Xue, Z. Xing, and S. Jarzabek. Understanding feature evolution in a family of product variants. WCRE 2010, pp.109-118.

[3]. P.Clements and L.Northrop. Software product lines: practices patterns. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2001

[4]. Y. Ghanam and F.Maurer. Linking feature models to code artifacts using executable acceptance tests. In Proceedings of the 14th international conference on Software product lines: going beyond (SPLC'10), Jan Bosch and Jaejoon Lee (Eds.). Springer-Verlag, Berlin, Heidelberg,2010, 211-225.

[5]. P.Ye, X.Peng, Y.Xue and S. Jarzabek.: A Case Study of Variation Mechanism in an Industrial Product Line. ICSR. 2009,126-136.

[6]. D.Andrea, F.Fausto, O.Rocco and T.Genoveffa. Recovering traceability links in software artifact management systems using information retrieval methods. ACM Trans. Softw. Eng. Methodol. 16, 4,20007, Article 13 .

[7]. D.Beuche, H.Papajewski, S.Wolfgang.. Variability management with feature models. Sci. Comput. Program. 53, 3 (December 2004), 333-352. 352.

[8]. I. John and M. Eisenbarth, “A decade of scoping: a survey,” in SPLC, ser. ACM International Conference Proceeding Series, D. Muthig and J. D. McGregor, Eds., vol. 446. ACM, 2009, pp. 31–40.

[9]. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey, JSME, 28 Nov, 2011.

[10]. M. V. Couto, M. T. Valente, and E. Figueiredo, “Extracting software product lines: A case study using conditional compilation,” in CSMR, T. Mens, Y. Kanellopoulos, and A. Winter, Eds. IEEE Computer Society, 2011, pp. 191–200.

[11]. S. Wang, D. Lo, Z. Xing, and L. Jiang. Concern localization using information retrieval: An empirical study on Linux kernel. WCRE 2011, pp. 92-96.

[12]. J. Rubin and M. Chechik. Locating distinguishing features using diff sets. InProceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering(ASE 2012). ACM, New York, NY, USA, 242-245.

[13]. G. Orlena and F. Anthony. An analysis of the requirements traceability problem. In Proceedings of 1st International Conference on Requirements Engineering (Colorado Springs, CO). IEEE Computer Society Press,1994, Los Alamitos, CA, 94–101.

[14]. S. George and Z. Andrea. Software Traceability: A Roadmap, in Handbook of Software Engineering and Knowledge Engineering, Chang, S. K., Ed. World Scientific Publishing Co,2004 , pp. 395-428.

[15]. W.Shaowei, L.David, X.Zhenchang and J.Lingxiao .Concern Localization using Information Retrieval: An Empirical Study on Linux Kernel. In Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE '11). IEEE Computer Society, Washington, DC, USA, 92-96.

[16]. S.Gerard and M.Michael. Introduction to Modern Information Retrieval. McGraw-Hill, Inc.,1996, New York, NY, USA.

[17]. M.Andrian and M.Jonathan. "Identification of High Level Concept Clones in Source Code", in Proceedings of Automated Software Engineering (ASE'01), San Diego, CA, November 26-29 2001, pp. 107-114.

[18]. M. Riebisch, “Towards a more precise definition of feature models,” in Modelling Variability for Object-Oriented Prod- uct Lines, M. Riebisch, J. O. Coplien, and D. Streitferdt, Eds. Norderstedt: BookOnDemand Publ. Co, 2003, pp. 64–76.

[19]. K. Kyo, C.Cohen,H.James,N.William and P.Spencer.Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990,Carnegie Mellon University.

[20]. X. Blanc, I. Mounier, A. Mougenot, and T. Mens, “Detecting model inconsistency through operation-based model construction,” in ICSE, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds.ACM, 2008, pp. 511–520.

[21]. L. Bergroth and H. Hakonen and T. Raita. A Survey of Longest Common Subsequence Algorithms. SPIRE 2000, pp. 39–48.

[22]. E.Hamzeh, S.Abdelhak-Djamal, D.Christophe and A.Ra'Fat. Recovering Traceability links between Feature Models and Source Code of Product Variants. ACM VARY Workshop (VARY: VARiability for You @ MODELS 2012, Sept. 30th - Oct. 5th, 2012 - Innsbruck, Austria.

[23]. S.T. Dumais. LSI meets TREC: A status report, in Proceeding of Text Retrieval Conference, pp. 137-152. 1992.

[24]. D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Guéhéneuc, and G. Antoniol. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. ICPC, pp. 137-148, 2006.