Study on Analysis of Program Collection for Classifying and Understanding Relations

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2016

Tetsuya KANDA

Abstract

Throughout a software development historic-lifecycle, a large amount of software and libraries can be generated. Those outputs are collected or categorized to form a "program collection". This dissertation dealt with two types of program collections. "Program collection with time series" is a set of the programs that have the same origin, but now containing multiple versions through branching and updating. "Snapshot of program collection" is a set of programs that is provided as ready to (re)use in deveoping another program. This category contains applications for specific devices and library set.

Well-managed program collections are useful for further development, as a target of software reuse. Maintaining program collections and keeping their value as an asset is an important thing, to prevent them from obsolescence. Thus, analysis technique for understanding their characteristics and revealing hidden relations may be helpful.

This dissertation describes four researches on analysis of program collections for classifying and understanding their relations. Each research analysis uses very limited inputs, mostly the program code, and reveals important characteristics of program collections. We believe that these results support developers to understanding existing program collections.

In the first research, we present a method to approximate evolution history of product family using only the source code of them. Since the history of product family would be lost in typical cases, it is hard for developers to understand these product family. A proposed method only requires the source code of target product family and clarify the branching and latest versions of the software products. The study showed that about 80% of the edge in the approximated evolution history is consistent with the actual evolution history of the products.

In the second research, we present a semi-automatic method to extract features from Android applications. Many Android applications with similar purpose are available, however, those applications are developed by independent developer so it is difficult to compare. The method extracts sequences of API calls from source code of Android applications and consider those sequences as features of applications. A case study showed

important differences among applications.

Thirdly, this research examined the quality of Java library set Maven2, the popular Java library repository. Maintaining library set is important, but Java library files can contain another library file and it is invisible for the library users. We meajured the nubmer of nested library files and count duplication of them. Analysis revealed that there are many copies of Java library files among the nested library files.

Finally, this research compared the characteristics of C and Java library set. Especially this research spotting identifier names, because identifier names are important source for program analysis and comprehension. The analysis of identifier definitions in C and Java library APIs reveals that they have different tendency of definition.

List Of Publications

Major Publications

1. Tetsuya Kanda, Takashi Ishio, Katsuro Inoue. "Approximating the Evolution History of Software from Source Code", IEICE TRANSACTIONS on Information and Systems, Vol.E98-D, No.6, pp.1185-1193, June 2015.

2. Tetsuya Kanda, Yuki Manabe, Takashi Ishio, Makoto Matsushita, Katsuro Inoue. "Semi-Automatically Extracting Features from Source Code of Android Applications", IEICE TRANSACTIONS on Information and Systems, Vol.E96-D, No.12, pp.2857-2859, December 2013.

3. Tetsuya Kanda, Daniel M. German, Takashi Ishio, Katsuro Inoue. "Measuring Copying of Java Archives", in Proceedings of the 8th International Worskhop on Software Clones (IWSC 2014), Antwerp, Belgium, February 2014.

4. Tetsuya Kanda, Daniel M. German, Takashi Ishio, Katsuro Inoue. "Comparing Frequency of Identifier Definition in C and Java APIs", IEICE TRANSACTIONS on Information and Systems, 2016
(in Japanese) (to appear).

Related Publications

5. Yusuke Sakaguchi, Takashi Ishio, Tetsuya Kanda, Katsuro Inoue. "Extracting a Unified Directory Tree to Compare Similar Software Products", in Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015), pp.165-169, Bremen, Germany, September 2015.

6. Yasuhiro Hayase, Tetsuya Kanda, Takashi Ishio. "Estimating Product Evolution Graph using Kolmogorov Complexity", in Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015), pp.66-72, Bergamo, Italy, August 2015.

7. Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M. German, Katsuro Inoue. "A Method to Detect License Inconsistencies in Large-Scale Open Source Projects", in Proceedings of the 12th Working Conference on Mining Software Repositories (MSR 2015), pp.324-333, Flotrnce, Itary, May 2015.

8. Yasuhiro Hayase, Tetsuya Kanda, Takashi Ishio. "Product Evolution Estimation Based on Kolmogorov Complexity", in FOSE2014, Kagoshima, December 2014 (in Japanese).

9. Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula Gaikovina Kula, Coen De Roover, Katsuro Inoue. "Identifying Source Code Reuse across Repositories using LCS-based Source Code Similarity", in Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), pp.305-314, Victoria, Canada, September 2014.

10. Tetsuya Kanda, Takashi Ishio, Katsuro Inoue. "Extraction of Product Evolution Tree from Source Code of Product Variants", in Proceedings of the 17th International Software Product Line Conference (SPLC 2013), pp.141-150, Tokyo, Japan, August 2013.

Acknowledgement

First of all, I am most indebted to my supervisor Professor Katsuro Inoue for his continuous support and supervision over the years. Without his help, experience and advice, this thesis would never have reached completion.

I would like to express my gratitude to Professor Toshimitsu Masuzawa and Professor Shinji Kusumoto for valuable comments on this thesis. I would also like to acknowledge the guidance of Professor Kenichi Hagihara and Professor Yasushi Yagi while I am in the Department of Computer Science.

I am very grateful to Assistant Professor Takashi Ishio for a lot of valuable support, supervisions, and helpful criticism of this thesis. I am also grateful to Associate Professor Makoto Matshishita for a lot of assistance and invaluable advice.

I would like express my gratitude to Professor Daniel German at the University of Victoria, Canada for his valuable guide in all aspects of my research. I also would like to express my gratitude to Assistant Professor Yasuhiro Hayase at University of Tsukuba and Assistant Professor Yuki Manabe at Kumamoto University for their valuable guide and comments.

I wish to thank Specially Appointed Assistant Professor Kula Raula Gaikovina, Specially Appointed Assistant Professor Ali Ouni, Professor Coen De Roover at Vrije Universiteit Brussel, Associate Professor Norihiro Yoshida at Nagoya University for their supports and advice. I would like acknowledge my appreciation to Dr. Yoshimura Kentaro at Hitachi, Ltd. for his support on my internship at Hitachi.

I would also like to thank Assistant Professor Eunjong Choi at Osaka University, Dr. Yu Kashima at Geniee, Inc., and Dr. Hironori Date for their kind help in Inoue Laboratory. Thanks are also due to many friends, especially students in Inoue Laboratory.

Finally I wish to thank to my family for supporting my school life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer software is now an important element in every part of the world, small devices such as smartphones and other consumer electronics to large-scale systems such as government systems and financial trading.

Throughout software development history, a large amount of software and libraries were developed. It is important to understand existing software to further development.

1.1 Program Collection

Along with the increased amount of the programs, they are collected or categorized to a set. "Program collection" is defined as a set of programs that are selected for some specific use.

Well-managed program collections are useful for further development and as a target of software reuse. Software reuse is an activity based on creating software systems from existing software, rather than building it from scratch [34]. Reusing proven software makes a product reliable and prevents developers from reinventing the wheel. Software reuse is also expected to reduce the cost and speeds up the development.

Understanding a program collection might lead developers to finding out a value of the program collection as an assets. However, maintenance of the program collection and keeping its quality is not easy problem. This dissertation spots and examines two types of program collections and proposes program code analysis methods for them.

1.1.1 A Program Collection with Time Series

During software evolution, a single program has been updated to the new version or sometimes branched into multiple variants which have different features. The set of the programs that have the same origin can be considered as a program collection with time series.

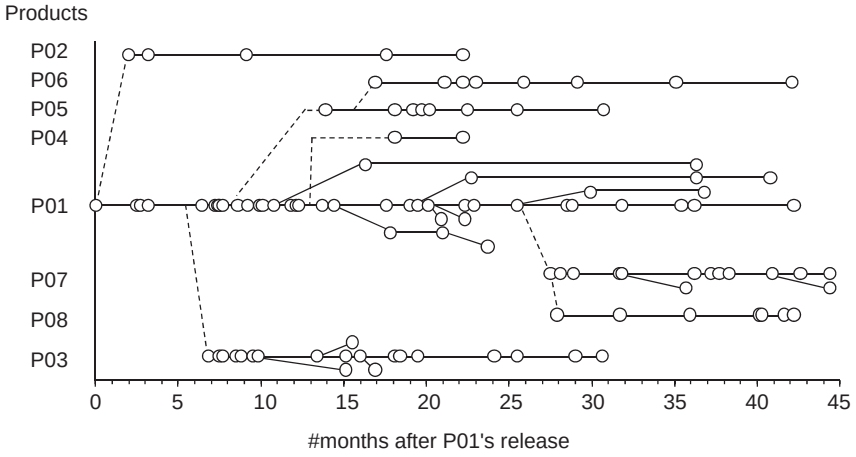


Figure 1.1: A product family derived from a single product [46].

Developers often create a new program by copying and modifying an existing one or importing libraries. Furthermore, they often reuse the developed program to create yet another new program. This method of iteration is called a "clone-and-own" approach. Once a software product has been released, a large number of software products may be derived from the original program.

Figure 1.1 shows a part of the industrial product family analyzed by Nonaka *et al.* [46]. The horizontal axis represents the number of months from the first release of the original product series (P01). The vertical axis represents product series ID in a company, respectively. In Figure 1.1, a circle corresponds to a product. Each dashed edge indicates that the new product series is derived from the original product. A solid edge connecting products indicates that the products are released as different versions of the same product series. This figure shows only 8 major product series and their variations, while the company had 25 series of products. Each series of products has from 2 to 42 versions.

1.1.2 Snapshot of a Program Collection

A snapshot is a set of program that is provided as ready to (re)use. For example, Google Play, previously named Android Market, provides a large number of applications for Android. iTunes App Store also contains huge number of applications for iPhones. As of June 2015, both Google Play and App Store provide more than 1.5 million applications.

Linux distribution such as Debian also contains a set of programs as a package. Those official packages are maintained and their dependencies

are managed so users can install the package without considering complex dependencies.

Libraries for sharing common functionality among programs also form a program collection. The collection of libraries for specific environment provides comfortable development environment. In Debian packages for example, we can see many libraries for software developing in the "Devel" section. In the case of Java development, there are tools for managing libraries and resolving dependencies for them. Apache Maven [41] is a project management tool for Java. Maven has a central repository that contains massive Java libraries. Ruby also has a package management system, RubyGems [53]. Some languages have an archive network and users can easily find out libraries they need, for instance, CPAN [7] for Perl and CRAN [8] for R.

1.2 Maintenance of a Program Collection

Maintaining program collections and keeping their value as an asset is an important activity. Since the evolution of programs is rapid, it is necessary to keep a program collection up-to-date and import new programs into the collection.

1.2.1 Software Product Line

Software Product Line Engineering (SPLE) is a well-known approach for efficient maintenance of a software product family [49]. Following Software Engineering Institute in CMU [55], SPL is defined as:

> a set of software-intensive systems that share a common, managed set of features satisfying 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.

Since the "clone-and-own" approach is very popular, the industry already maintains a large number of derived software products. Management and maintenance of product variants are important, but those tend to be disregarded because developers do not put enough effort for the further maintainability in the initial phase of software development [12]. Construction of a software product line from existing products is a very important problem and many re-engineering methods have been proposed [14, 33, 64].

The construction of a software product line from existing products requires developers to understand the commonalities and variabilities of them [4]. Krueger suggested that developers should start their analysis from a small number of software products, instead of all products at once [35]. Koschke et al. proposed an extension of reflexion method to construct a

product line by incrementally analyzing products [33]. To follow these reasonable approaches, developers must choose representative software products as a starting point. However, history of products would be lost in typical cases and developers have only an access to source code of products in the worst case [36]. Thus, clarifying the difference of products and show the representative ones without relying the history is needed.

1.2.2 Library Updating Problem

If bugs or vulnerabilities are found in the library, possible new version of the library might be released, then it is advisable that library users update. However, through the evolution of a library, the API also changes and sometimes the compatibility is lost [15]. Darcy noted that there are three main kinds of compatibility; source, binary, and behavioral [32]. In a recent study, Dietrich *et al.* [11] pointed out that the recent trend is partially upgrading systems by replacing new library versions, not rebuilding an entire system. In this build style, library incompatibility is still problematic and this causes new category of binary incompatibility.

One solution for users is that using the same set of library versions as existing software. Yano *et al.* visualized a popular combination sets of libraries [62]. Users can avoid problematic combination by using their tool, but maintaining library set itself is still a difficult problem.

1.3 Contribution of the Dissertation

To understand program collections, spotting and revealing hidden relations would be helpful. For each type of program collections, this dissertation describes the analysis methods and those results. Each research analysis very limited inputs, mostly the program code, and reveals important characteristics of program collections. We believe that those results support developers to understanding existing program collections.

1.3.1 A Program Collection with Time Series

Software Product Family

One important step to understand product family is selection of initial analysis target from large number of products. The evolution history of the product family helps this step, but it is not always available.

To deal with this problem, we proposed a method to approximate evolution history of product family using only their source code. The study of 9 datasets, including C and Java projects, showed that about 80% of the edge in the approximated evolution history is consistent with the actual evolution history of the products.

1.3.2 Snapshot of a Program Collection

Application Collection for Specific Target

We proposed a method to extract features from an Android application collection. This method enables users to compare features among similar but different applications.

The basic idea is that applications targeting a specific device use similar set of APIs provided as SDK. The study of 11 Android applications showed that a sequence of API calls can be useful to compare applications released by independent developer.

Libraries from a Large Software Set

Libraries are the component that ready to reuse. Understanding their characteristics would be helpful for analyzing and understanding client applications that are using those libraries.

A quality of library collection is particularly important because libraries work in cooperation. In other words, they have a complex dependency. We examined Java libraries from Maven repository, and found copies of Java library file inside.

From a viewpoint of client application source code, API names are the main linkage to the libraries they use. We analyzed two large library set, C libraries from Debian packages and Java libraries from Maven repository. The analysis clarifies that they have different tendency of definition The analysis also showed that most of C identifiers are unique to a single library. This fact would be useful for light-weight analysis.

1.4 Outline

The rest of the dissertation is structured as follows:

Chapter 2 reports the method to extract "Product Evolution Tree" that approximates evolution history of software products. The extraction depends on only the source code, so that it can be applied to the products that have lost their evolution history.

Chapter 3 describes a semi-automatic approach to extract feature names from Android applications. This approach extracts API calls from the source code of Android applications.

Chapter 4 analyzes the quality of Maven2, the famous Java library repository. Analysis reveals that there are many copies of Java library files inside the library file, and it is invisible for the library users.

Chapter 5 studies the uniqueness and tendency of defining library identifier names. This analysis of identifier definitions in C and Java library APIs reveals that they have different tendency of definition.

Finally, Chapter 6 concludes the dissertation and shows the directions for future work.

Chapter 2

Approximating the Evolution History of Software from Source Code

2.1 Introduction

When developing a software product, clone-and-own approach is one of the major and easy ways to realize software reuse [52]. Developers copy existing code or the whole of the product and then add features, fix bugs, and so on. A software product contains source files, images, documents, and the other resources. We define "a source file" as a source code in the single file and "a software product" as a set of source files.

The new version of the first product is released with slightly different features, so it will have very similar files with the first one. Management of such similar software products is a very important task. They might have the same problems or bugs, or developers can apply same improvement in them. However, developers often copy and modify the software product without using version control systems (VCS) or other management techniques [12] since no one knows whether the product would be successful enough to apply many extensions and derive many variants. Using #IFDEF macro in C language to describe product specific features is one of the solutions, but it is believed to decrease code readability. Clone-and-own approach also gives developers freedom of making changes, without considering making an impact to existing projects.

Many re-engineering methods for existing software products have been proposed [14, 33, 64]. Since analyzing a large number of software products is a difficult task, Krueger *et al.* suggested that developers should start their analysis from a small number of software products [35]. Koschke *et al.* proposed an extension of reflexion method to construct a product line by incrementally analyzing products [33]. To follow these reasonable

approaches, developers must choose representative software products as a starting point. If the history of software evolution is available, developers could recognize the relationships among the products and choose representatives for their analysis. For example, compare products between branches to extract common features and product specific features. In the point of view of re-engineering, understanding the evolution history of software is also an important thing.

However, the history of software products is often not available [36]. Software products are not always managed under the VCS. If the software has branched and managed independently, relationships between branches are not recorded. Some of experts know the whole of the software products, but their knowledge is often incomplete [48]. In the worst case, developers only have access to source code of each product, they cannot get version numbers nor release date for some of the products.

To deal with the situation that evolution history of software products is lost, we propose a method to approximate the evolution history of software products using source code of them. We assume that two successive products are the most similar pair in the products. Similar software products must have similar source files so we analyze the source files and count the number of similar source files between products. We connect the most similar products and construct a tree. This tree is an approximation of the evolution history of software products and two successive products will be connected. Our approach depends only on source files, so we can analyze products whose evolution history is lost; no version numbers, names or release dates.

The contributions of this chapter are follows:

- We have proposed a visualization technique of relationships among software products from their source code.
- We have introduced a weighted function between two software products to reflect the effect of small changes.
- We did an experiment with programs written in C and Java.
- We did a case study with two variants of Linux kernel and found out their origin.

2.2 Related Work

2.2.1 File Similarity

When comparing software products, similarity between source files is a very important metric. To find out the same or similar source code fragments,

many code clone detection tools have been proposed [29, 38]. Using large-scale code clone detection techniques, Hemel and Koschke compared Linux kernel and its vendor variants [21]. They found vendor variants included various patches, but the patches are rarely submitted to the upstream. Another application of code clone detection is detecting file moves occurred between released versions of a software system [36].

Yoshimura *et al.* visualized cloned files in industrial products [63]. They have used an edit distance function as a source file similarity to find out cloned files whose contents are almost the same. Inoue *et al.* [24] proposed a tool named Ichi Tracker to investigate a history of a code fragment with source code search engines. It visualizes how related files are similar to the original code fragment and when they are released. With the visualization, developers can identify the origin of the source code fragment or a more improved version. Our approach enables similar analysis on software products instead of source files.

We have assumed that two successive products are very similar to each other. This observation is shown by Godfrey *et al.* [16]. They detected merging and splitting of functions between two versions of a software system. Their analysis shows that a small number of software entities such as functions, classes or files are changed between two successive versions. Lucia *et al.* reported that most of bug fixes are implemented in a small number of lines of code [40]. Since these studies reported that two successive versions are very similar, we infer that the most similar pairs of products are likely two successive versions.

2.2.2 Software Evolution

Yamamoto *et al.* proposed SMAT tool that calculates similarity of software systems by counting similar lines of source code [61]. They identify corresponding source files between two software systems using CCFinder [29], and then compute differences between file pairs. They applied their tool to a case study of software clustering, and extracted a dendrogram of BSD family. The dendrogram reported which OSs are similar to each other. Tenev *et al.* introduced bioinformatics concepts into software variants analysis [57]. One of them is phylogenetic trees, which visualizes the similarity relations. They constructed dendrogram and cladogram from six of BSD family for example of phylogenetic trees.

They can show the relationship that which product is most similar to another and which products were forked from the release. Although their approaches and goals are similar to our idea, our approach visualizes more concrete relationships among products which are not shown in those related works; which product was first released, their evolution direction, and so on.

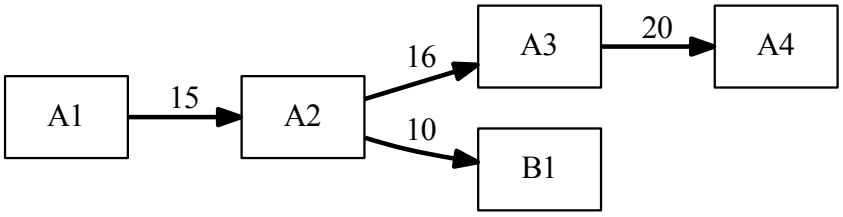


Figure 2.1: An example of a product evolution tree.

2.2.3 Software Categorization

Several tools have been proposed to automatically categorize a large number of software based on their domains such as compiler, database, and so on. MUDABlue [30] classifies software based on similarity of identifiers in source code. MUDABlue employed latent semantic analysis which extracts the contextual-usage meaning of words by statistical computations. LACT [58] uses latent dirichlet allocation in which software can be viewed as a mixture of topics. LACT used identifiers and code comments, but excluded literals and programming language keywords, to improve categorization. CLAN [42] focused on API calls. Its basic idea is that similar software uses the same API set.

While all of these tools are able to detect similar or related applications from a large set of software products, our approach focuses on very similar products derived from the same product, that are likely categorized into the same category by these tools.

2.3 Approach

We define the "Product Evolution Tree" as a spanning tree of complete graph which includes all input products and connects most similar product pairs first. If many files are similar between two products, it means that those products are similar. A simple example of the tree is shown in Figure 2.1. Each node represents a software product. Each edge indicates that a product is likely derived from another product and the direction of derivation: which product is an ancestor and which product is a successor. A label of an edge explains the number of similar files between products. In Figure 2.1, the product branched and there are more similar files between A2 and A3 than A2 and B1.

We construct a Product Evolution Tree from source code of products through four steps as follows.

1. We calculate file-to-file similarity for all pairs of source files of all products.

2. We count the number of similar files between two products.

3. We construct a tree of products by connecting most similar product pairs.

4. We calculate evolution direction based on the number of modified lines between two products.

2.3.1 File Similarity

We calculate similarity for all pairs of files across different products. We do not consider file names because a file may be renamed. To calculate the similarity of two source files, we first normalize each of source files into a sequence of tokens. In a normalized file f_n, which is a sequence of tokens of file f, each line has only a single token. We remove blanks and comments since they do not affect the behavior of products. All other tokens including keywords, macros and identifiers are kept as is. Given a pair of files (a, b), their file similarity $sim(a, b)$ is calculated as follows:

$$sim(a, b) = \frac{|LCS(a_n, b_n)|}{|a_n| + |b_n| - |LCS(a_n, b_n)|}$$

where $|LCS(a_n, b_n)|$ is the number of tokens in the Longest Common Subsequence between a_n and b_n.

We have used a file similarity based on LCS, since we could optimize the calculation as described in Section 2.3.6. Another reason is that LCS-based technique like UNIX diff is one of the most popular choices in comparing source code. There are famous metrics for measuring similarity of documents such as TF-IDF, jaccard similarity, and so on. Of course, those metrics can be applied to the source files (we are using jaccard similarity in optimization), but they are based on the term frequency and do not consider the order of elements. The following computation steps did not depend on the definition of file similarity function; hence, other methods such as code clone detection are also applicable to compute file similarity.

2.3.2 Count the Number of Similar File Pairs

When the file pair has a higher similarity than a threshold, it is a similar file pair. The set of all possible similar file pairs S is defined as:

$$S(P_A, P_B, th) = \{(a, b) \mid a \in P_A, b \in P_B, sim(a, b) \geq th\}.$$

and the number of similar file pairs N between software products P_A and P_B are defined as:

$$N(P_A, P_B, th) = |S(P_A, P_B, th)|.$$

2.3.3 Construction of the Tree

In this step, we construct a spanning tree of products. We first construct a complete undirected graph $G = (P, E)$, P denotes that software product and E denotes set of edges that connects all those products. From this graph, we pick edges with maximum number of similar files and add to the tree, without making a loop, until all nodes are connected. This is the same operation of the well-known algorithm of the minimum spanning tree. As a result, we get a spanning tree $S = (P, E')$ of the graph G. $E' \subseteq E$ is a set of edges which have the largest number of similar file pairs as follows:

$$\sum_{(P_i,P_j)\in E'} N(P_i, P_j, th).$$

If two or more edges have the same weight values, one of them can be arbitrary selected. In our implementation, it depends on the input order.

2.3.4 Evolution Direction

After a spanning tree is constructed, we set the direction on each edge which explains the direction of evolution. Our hypothesis is that source code is likely added, so we count the amount of added code in two software products as follows:

$$ADD(P_A, P_B) = \sum_{(a,b)\in S(P_A,P_B,th)} |b_n| - |LCS(a_n, b_n)|$$

where a_n and b_n are the normalized source files. Evolution direction is defined as follows:

$$\begin{cases} ADD(P_A, P_B) > ADD(P_B, P_A) \Rightarrow P_A \rightarrow P_B \\ ADD(P_A, P_B) = ADD(P_B, P_A) \Rightarrow P_A - P_B \\ ADD(P_A, P_B) < ADD(P_B, P_A) \Rightarrow P_A \leftarrow P_B. \end{cases}$$

Direction "–" means no direction detected.

We put directions and labels which denote the number of similar files on each edge of the tree. The Product Evolution Tree is completed through these four steps.

2.3.5 Weighted Function

The function N explains the number of similar source files. When the software product series goes to maintenance phase, there would be no drastic changes so that changes will not decrease file similarity below the threshold. This means that N cannot explain how much the source code is changed.

To reflect the amount of changes to the function, we define another function N_w that weighting the function N with *sim*:

$$N_w(P_A, P_B, th) = \sum_{(a,b)\in S(P_A,P_B,th)} sim(a, b).$$

sim is already computed in the Step 1 so that we can get N_w without vast amounts of calculating cost. We compare these two functions in the experiment.

2.3.6 Optimization

To reduce the computation time, we introduced an implementation technique that calculates *sim* value only if it seems greater than the similarity threshold. The technique is based on the jaccard similarity of two documents. We introduce the term frequency $tf(f, t)$ which represents how many times term t appears in file f. For example, suppose two tokenized files a_n=AAABB and b_n=ABBBB, where A and B are terms in the files. The term frequencies are $tf(a_n, A) = 3$, $tf(a_n, B) = 2$, $tf(b_n, A) = 1$, and $tf(b_n, B) = 4$. Since $LCS(a_n, b_n)$ can include at most one A and two Bs shared by the sequences, the maximum length of $LCS(a_n, b_n)$ is 3.

The maximum length of $LCS(a_n, b_n)$ is calculated as:

$$\sum_{t\in T} min(tf(a_n, t), tf(b_n, t))$$

and we can get maximum similarity

$$msim(a, b) = \frac{\sum_{t\in T} min(tf(a_n, t), tf(b_n, t))}{\sum_{t\in T} max(tf(a_n, t), tf(b_n, t))}$$

of each file pair (a, b) using term frequency. T represents the set of terms appeared in all source files. The value of $sim(a, b)$ equals to $msim(a, b)$ if all the common tokens appear in the same order in two sequences. If the order of tokens is different from another sequence, then $sim(a, b)$ is smaller than $msim(a, b)$. A fomula $msim(a, b) \geq sim(a, b)$ is always true, hence we compute $sim(a, b)$ only if $msim(a, b)$ is greater than the similarity threshold.

2.3.7 Simple Example

Here is a simple example of the algorithm. In this section, we use two products shown in Figure 2.2. We shorten "Product 1" to P_1 and "File A of Product 1" to P_1-A.

File Similarity

We calculate all file pairs among P_1 and P_2. Table 2.1 shows the similarity value among those producs.

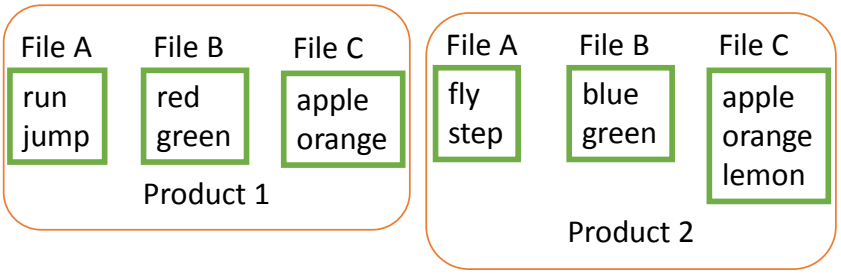


Figure 2.2: An example input.

Table 2.1: Similarity value among example input

	P_1-A	P_1-B	P_1-C
P_2-A	0	0	0
P_2-B	0	0.33	0
P_2-C	0	0	0.66

Count the Number of Similar File Pairs

When we set the similarity threshold $th = 0.5$, only (P_1-C, P_2-C) is the similar file pair. The cost is $N(P_1, P_2, 0.5) = 1$ and $N_w(P_1, P_2, 0.5) = 0.66$.

Construction of the Tree

In this example, we have only two products so we just connect them.

Evolution Direction

In the similar file pair (P_1-C, P_2-C), P_2-C has one more token "lemon" than P_1-C and no unique token in P_1-C. Please note that P_1-B and P_2-B shares some code but those files are "not similar" so the algorithm does not consider the changes between them.

As a result, $ADD(P_1, P_2) = 1$, $ADD(P_2, P_1) = 0$ so the evolution direction is "$P_1 \rightarrow P_2$".

2.4 Experiment

We have implemented our approach as a tool and conducted an experiment. The goal of the experiment is to evaluate how accurately the Product Evolution Tree recovers the actual evolution history. We have used similarity threshold $th = 0.9$ in this experiment, which is experimentally determined.

2.4.1 Datasets

We have prepared nine datasets using open source projects, six of them are implemented in C and the other three of them are implemented in Java.

PostgreSQL [50]. It is a database management system. In the evolution history of PostgreSQL, each major version was released from the master branch after developing beta and RC releases. After a major version had been released, a STABLE branch was created for minor releases and the master branch was used for developing the next beta version. While each release archive contains a large amount of files, we used only source files under "src" directory in this experiment.

The evolution history of PostgreSQL is simple and well-formed so we select four datasets from PostgreSQL to evaluate some kind of situation.

Dataset 1: Pgsql-major is a dataset whose evolution history is straight, *i.e.*, it has no project forks. *Dataset 2: Pgsql8-all* is a dataset whose evolution history is a tree of a single project with a large number of variants. *Dataset 3: Pgsql8-latest* is a dataset that includes only recent products. If a product family has a long history, older products may be no longer available for developers. *Dataset 4: Pgsql8-annually* is another dataset that a full collection of products is not available. Dataset 4 contains releases which have been released around September from 2005 to 2012.

FFmpeg and Libav. They are libraries and related programs for processing multimedia data. Libav is forked from FFmpeg and is developed by a group of FFmpeg developers. They are independently developed, but similar changes have been applied to both products.

Dataset 5: FFmpeg is a dataset whose project has been forked to two projects. This dataset is created to evaluate whether our approach can recover the evolution history of forked projects or not.

4.4BSD, FreeBSD, NetBSD and OpenBSD. These operating systems are derived from BSD, but they are now independent projects. Figure 2.4a shows a part of the family-tree for the versions selected for our dataset. According to the tree, NetBSD-1.0 is not only derived from NetBSD-0.9 but also from 4.4BSD Lite. FreeBSD-2.0 is also based on 4.4BSD Lite. OpenBSD is the forked project of NetBSD. 4.4BSD Lite2 affects other BSD operating systems. For each version, we used source files under "src/sys" directory.

Dataset 6: BSD is a dataset whose project has been forked to more than three projects. The evolution history is the most complex in our datasets and there are releases created by merging source code from more than one product. Since our approach extracts only a tree, our approach must miss such merged edges.

Groovy [19]. This is an agile and dynamic language for Java Virtual Machine. In the evolution history of Groovy, each release has own branch. Since they all branched just before the release and there are no changes in

source files comparing with original branch, we can say that the evolution history of Groovy is very similar to that of PostgreSQL. We used only source files under "src" directory.

Dataset 7: Groovy is a small dataset of Java application. In the VCS, each release has branched from the main branch, but it has completely same source code so we did not consider such small branches.

Hibernate [22]. This is an object relationship mapping library for Java. This evolution history is also similar to PostgreSQL and Groovy. Each major version is developed on their own branches. We used only source files under "hibernate-core" directory.

Dataset 8: hibernate is a large dataset of Java application. This dataset contains 3 branches and 61 versions. Some of them has special version names like "4.2.7SP" and they makes the evolution history bit complex.

OpenJDK [47]. This is an open-source implementation of Java. The OpenJDK project firstly released OpenJDK7, and implement OpenJDK6 from it. We analyze files under "src/share/classes" directory.

Dataset 9: OpenJDK6 is a dataset which represents unusual evolution history. This dataset contains initial OpenJDK6 (the copy of OpenJDK7) and its children. The product starts with OpenJDK7 and modified to implement "old" Java6 standard. So this dataset considered not to follow the standard evolution; implementing new and rich features into later version.

2.4.2 Results Overview

The correctness of the edges and labels is shown in Table 2.2 and Table 2.3. Column "#" denotes the dataset number. Column "H. (History)" denotes the number of edges in the evolution history and "O. (Output)" denotes number of edges in the Product Evolution Tree. Column "Matched Edges" shows how many edges are matched with the actual history without considering direction. In other words, we only checked the shape of the tree. Column "Matched Labels" shows how many correct edges have correct direction. Column "Recall" indicates the proportion of correctly identified edges to edges in an actual evolution history.

We did not calculate precision in this experiment, since the precision is higher than or the same as the recall. This is because the number of edges in the Product Evolution Tree is the same as or less than the number of edges in the actual evolution history. If the dataset which consist of N products does not contain the loop, the number of edges in the dataset is $N-1$ and the number of edges in our tree is also $N-1$. So the number of false positive edges is always the same number of false negative edges and the precision is the same value as the recall. Only the Dataset 6 contains the loop so the number of false positive edges is smaller than the number of false negative edges and the precision is smaller than the recall.

Table 2.2: Result with N

#	H.	O.	Matched Edges		/ Labels		Recall
1	13	13	13	(100%)	13	(100%)	100%
2	143	143	106	(74.1%)	104	(98.1%)	72.7%
3	37	37	24	(64.9%)	24	(100%)	64.9%
4	24	24	20	(83.3%)	20	(100%)	83.3%
5	15	15	1	(6.7%)	1	(100%)	6.7%
6	17	15	11	(64.7%)	11	(100%)	64.7%
7	36	36	28	(77.8%)	22	(78.6%)	61.1%
8	61	61	51	(83.6%)	44	(86.2%)	72.1%
9	15	15	8	(53.3%)	5	(62.5%)	33.3%

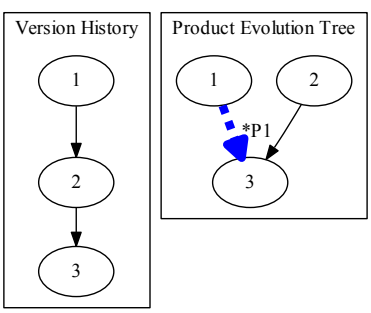
Table 2.3: Result with N_w

#	H.	O.	Matched Edges		/ Labels		Recall
1	13	13	13	(100%)	13	(100%)	100%
2	143	143	137	(95.8%)	132	(96.4%)	92.3%
3	37	37	30	(81.1%)	30	(100%)	81.1%
4	24	24	20	(83.3%)	20	(100%)	83.3%
5	15	15	14	(93.3%)	14	(100%)	93.3%
6	17	15	11	(64.7%)	11	(100%)	64.7%
7	36	36	30	(83.3%)	24	(80.0%)	66.7%
8	61	61	53	(86.9%)	46	(86.8%)	75.4%
9	15	15	13	(86.7%)	7	(53.8%)	46.7%

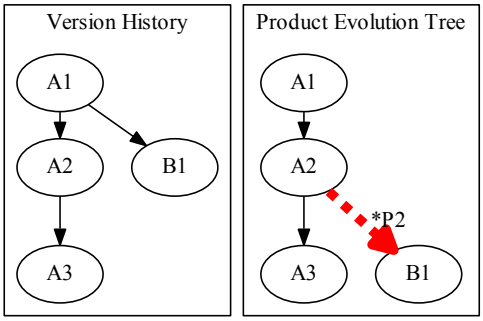
Comparing the result with N and N_w , N_w performed better and Dataset 5 is a case that weighted function has worked most effectively. When the project forks, it has already been in the maintenance phase and few changes are adopted to the forked releases. As a result, all file pairs exceeds the similarity threshold 0.9 and the number of similar files between any two of the dataset are the same value ($N = 618$) so almost all edges showed wrong evolution. Using weighted function N_w , we can reflect the effect of small changes and the tree well approximates the evolution history so we discuss the result with N_w below.

2.4.3 Patterns of Incorrect Edges

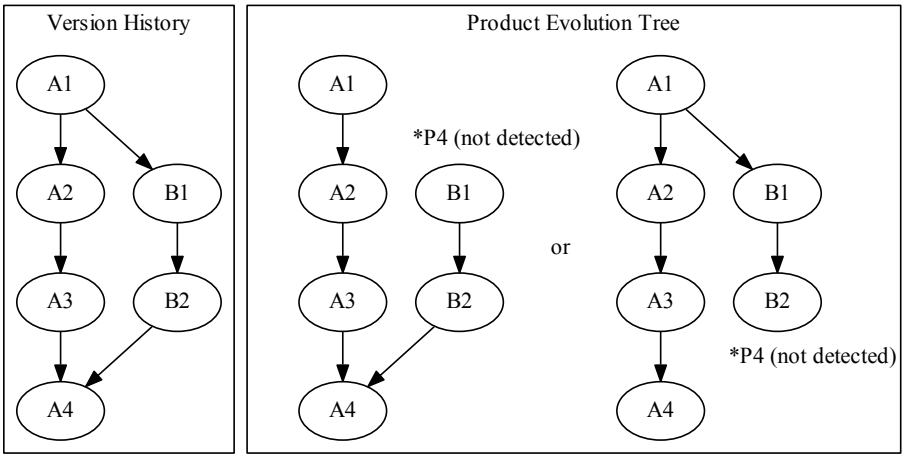
Even though our approach connects most likely similar products, some edges are mismatched with the actual evolution history. To analyze mismatches, we have categorized incorrect edges in Product Evolution Trees into 5 patterns as follows. In Figure 2.3, each left graph shows an actual evolution history and each right graph shows an extracted Product Evolu-



(a) P1: Version Skip



(b) P2: Misalignment of branch



(c) P4: Missing Branch/Merge

Figure 2.3: Patterns of incorrect edges

tion Tree. Thin edges are the connections that exist in the actual history. Thick, dashed edges are extracted by our approach, but they do not exist in the actual history.

P1: Version Skip. This pattern is found in three successive versions; two edges v_1 to v_3 and v_2 to v_3 are detected instead of a path from v_1 to v_3 via v_2. Figure 2.3a shows an example. This pattern happens when v_2 and v_3 have the same N_w value from v_1 or the N_w between v_1 and v_3 is large. In addition, we classify edges into this category only when the edge skips one version. If the edge skips two or more versions, it classified into P5: Out of Place.

In Dataset 9 for example, tags "b13" and "b15" are connected in the tree and "b14" is skipped. One developer said in his blog that "b15" is tagged just for mark as switching VCS to mercurial. There are no difference in any files between "b14" and "b15" so that $N_w(b13, b14, 0.9)$ and

$N_w(b13, b15, 0.9)$ are the same value.

P2: Misalignment of Branch. An edge connects two branches but does not connect actually branched products. In Figure 2.3b, there are two branches A and B. While B1 was actually forked from A1, the origin of branch B was recognized as A2. In this pattern, A2 actually has more similar files, comparing with B1 than A1.

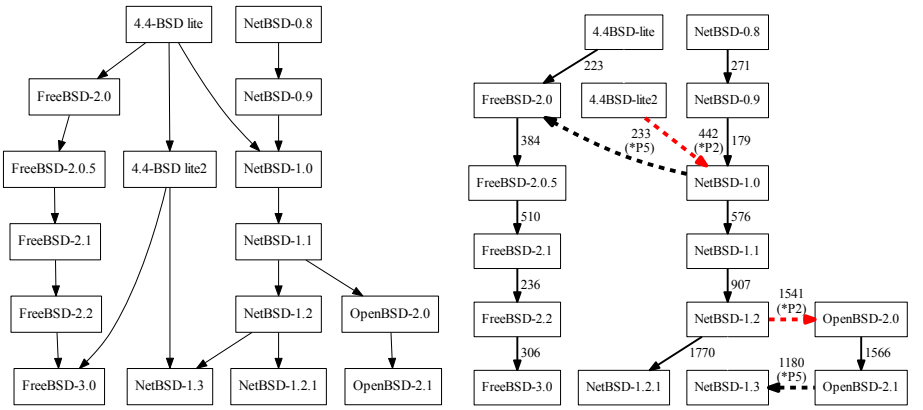
In Dataset 2, almost all edges connecting branches are not matched. We found that this is because branched products share the same changes. For example, 8.2BETA1 is developed on the master branch as the next version of 8.1.0, but extracted tree says this is the next version of 8.1.5. We examined git repository and found that version 8.1.5 is released right after 8.2BETA1. The master branch developing 8.2BETA1 and STABLE branch for 8.1 received 225 commits that are submitted on the same date with the same log message, but there are only 28 commits unique to the master branch. This fact also means that the actual evolution history does not always show functional differences of products.

P3: Misdirection. An edge connects accurate products, but its label shows the reverse direction. It happens when the size of source code or the number of source files decreased by several activities such as refactoring and deletion of dead code. In the other case, if two versions have the same source files, our approach cannot define the evolution direction.

Many of this pattern show reversed direction, but other edges around thems show accurate direction, so it is easy to recognize that those edges connects exact products but the direction is reversed. In the case of Dataset 8, two of misdirection patterns, 4.1.2–4.1.2.Final and 4.3.3Final–4.3.4Final, have no direction. A comment in VCS says that there are no changes but the developer tagged them again.

P4: Missing Branch/Merge. Our Product Evolution Tree cannot detect a branch or a merge of two products derived from a single product. In Figure 2.3c, we can see that the Product Evolution Tree misses branching from version A1 to version A2 and B1 or merging from version B2 to A4. In this pattern, one edge is missing but no wrong edges are output. If an actual evolution history includes a merge (e.g. Dataset 6), 100% recall is not achievable.

This pattern appears in Dataset 6. Figure 2.4a shows the family-tree and Figure 2.4b output of our approach. The Product Evolution Tree included a merge relationship for NetBSD-1.0. It is the next release of NetBSD-0.9 and includs many source files from 4.4-BSD Lite. On the other hand, an edge from 4.4BSD Lite2 to FreeBSD-3.0 is not detected because the Product Evolution Tree does not allow closed paths. In addition, N_w(4.4BSD Lite 2, FreeBSD-3.0, 0.9) = 40 indicated that all except for 40 files are different between two versions. The relationship from 4.4BSD Lite2 to FreeBSD-3.0 in the family tree may not be captured by the source



(a) A family-tree of Dataset 6. (b) A Product Evolution Tree of Dataset 6.

Figure 2.4: BSD Family Tree

Table 2.4: Release date of BSD family.

BSD	date
NetBSD 1.2	1996-10-04
OpenBSD 2.0	1996-10-18
OpenBSD 2.1	1997-06-01
NetBSD 1.3	1998-01-04

code difference.

P5: Out of Place. This pattern is a falsely detected edge which is not classified into previous patterns. There are no relationship between the wrong edge and the actual history.

2.4.4 Discussion

The result shows that 65% to 100% of edges without labels and 47% to 100% of edges with labels are consistent with the actual evolution history.

From the shape of the Product Evolution Tree, developers can learn where the starting point of the evolution is and where they branched. Almost all of the latest products of each branch are represented as leaf nodes, except Dataset 6. Value of the function N_w also provides hints to understand an evolution history. If a vertex has three edges and one of them has a small number of similar files, it may indicate branching and others may indicate the mainline.

Take a look at Figure 2.4b, FreeBSD-2.0, NetBSD-1.0, and NetBSD-1.2 will get attention because they have more than two edges. Leaf nodes 4.4BSD Lite, 4.4BSDLite2, FreeBSD-3.0, NetBSD-0.8, NetBSD-1.3, and

Table 2.5: Incorrect edge patterns with N_w

Dataset	P1	P2	P3	P4	P5	Total
1						0
2		4	5		2	11
3		5			2	7
4		4				4
5		1				1
6		2		4	2	8
7	1	5	6			12
8	4	3	7		1	15
9	2	6				8

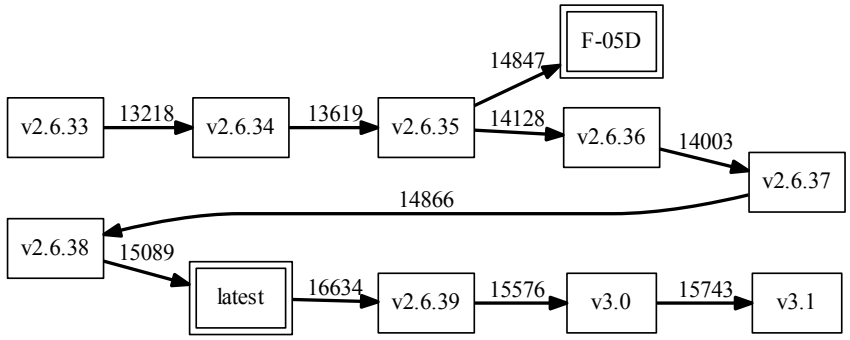
NetBSD-1.2.1 also seem important. The tree suggests that OpenBSD-2.1 is not a characteristic release. It is hard to find out that they are important releases in this dataset.

If the time had passed from previous releases, they would apply the same changes. In Dateset 6 for example, OpenBSD Project is forked from NetBSD 1.1 but its first official release is in October 1996. NetBSD 1.2 is released just before OpenBSD 2.0 was released so we can imagine that there are same changes in NetBSD and OpenBSD. The same things can be said in OpenBSD 2.1 and NetBSD 1.3, showed in Table 2.4.

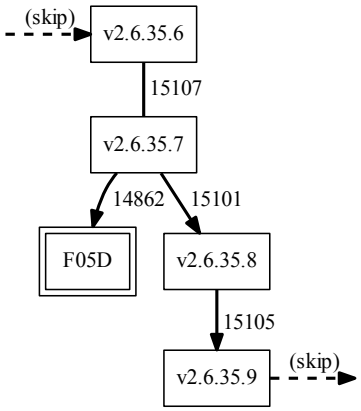
Major error P3 is a counterexample for our hypothesis that "source code is likely added". One reason is that refactoring such as class splitting and merging have been applied. Techniques for detecting refactoring [59] may be helpful to remove incorrect labels caused by this reason. Another reason is non-essential changes [31] such as deleting dead code affect a large number of lines of code, while they are less important than other modification tasks such as feature enhancement. We can conjecture some cases that source code is decreased, but P3 was at most 17% (6 of 36 in Dataset 7) of extracted edges in our experiment. Hence, our method for determining the direction still worked effectively. We did not use release dates since they are not always available, but if release dates are available, all evolution direction would be correctly extracted if edges connect successive products.

Releases with no changes invoke error pattern P1 and P3. Developers easily notice this is an error, since it is hard to think that some files are fixed but total amount of deleted and added code are the same amount.

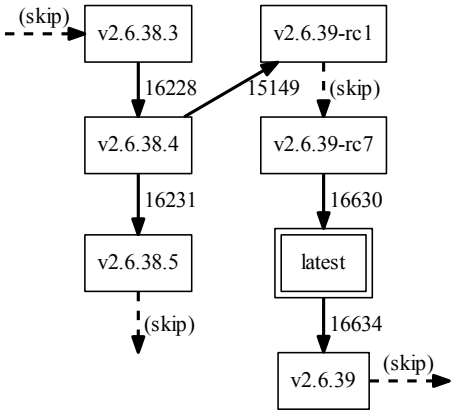
The optimization reduces the execution time greatly. Dataset 1 for example, we need 10 minutes for analysis using optimization. On the other hand, without optimization, our tool runs over an hour for analyzing first four products.



(a) Overview of the tree



(b) Detail of the tree around F-05D kernel.



(c) Detail of the tree around "latest."

Figure 2.5: A case study with Linux kernel and two variants.

2.5 Case Study

The result of experiment shows that our method well approximates an evolution history of software product from their source code with high precision. In the case study, we simulate the situation that finding out the origin of the variants. We continued using similarity threshold $th = 0.9$.

The target is the Linux kernel [39] and two of their variants. One variant is in the kernel repository, labeled "latest", and another variant is kernel files from F-05D Android smartphone [13]. We analyze those two variants with releases of the Linux kernel and check the result with the version number denoted in the Makefile.

Figure 2.5a shows the overview of the Product Evolution Tree and Figure 2.5b and Figure 2.5c shows the detail of the tree around target variants. Those figures show that the F-05D kernel was branched from 2.6.35.7 and latest tag is attached just before 2.6.39 is released. We can see that those

two variants have different history. F-05D kernel was branched and they have had some changes. “latest” tag is assigned for development of 2.6.39 but there are still some changes before 2.6.39 is released.

This result matches the version number denoted in the Makefile and its product history. Makefile of F-05D says that this is 2.6.25.7, and “latest” is tagged between 2.6.39-RC7 and 2.6.39 in the repository. The result of the case study shows that our approach is useful for detecting origin of the variants. With the Product Evolution Tree, we can see that which product is the origin and whether the product is branched or not.

2.6 Threats to Validity

Targets of our experiment are restricted in the OSS with version control system and they have reliable their evolution history. In other words, those projects are considered well maintained. However, our Product Evolution Tree well reflects the development history compared with actual history in some cases. For example, branched timing in the tree follows functional changes in Dataset 2, and we could find completely same versions with different tags in Dataset 7, 8, and 9.

We have used a single threshold 0.9 in the case study, which is determined by a small preliminary experiment. While it works for 9 datasets, a different threshold may be better for a different dataset.

2.7 Conclusions

To help developers understand the evolution history of products, we proposed a method to extract an approximation of the evolution history from source code. It is defined as a tree that connects most similar file pairs. Specifically, we count the number of similar files with Longest Common Subsequence based source similarity and we construct a spanning tree of complete graph which connects all input products.

As a result, 47% to 100% of edges are correctly recovered. We can identify branches and the latest versions of products using our approach, even if the result included incorrect edges. Our methodology and techniques used are simple, but shows promising result in experiments.

Chapter 3

Semi-automatically Extracting Features from Source Code of Android Applications

3.1 Introduction

Android is one of the most popular platforms for mobile phones and tablets. A user can search and choose from more than 600,000 Android applications in Google Play [2]. Because there are so many choices, however, selecting an appropriate application is not a trivial task. For example, in November 2012, at least 1,000 applications could be found when searching with the keyword "calculator" on Google Play.

A simple but important criterion for selection of an application is the set of features it provides. Investigating the features by trying each application, however, is time consuming. Although documentation is an important source of information, many applications are less than adequate in this area.

MUDABlue [30] and LACT [58] are the solutions that enable users to focus on a set of similar applications. These approaches automatically categorize applications with similar features based on characteristics of the source code. While they can extract a set of similar applications, they cannot show a list of the features provided by the applications in a specific category.

Software developers construct an application by combining several features. Developers often make software for a specific platform. When the target platform provides a high level API, developers implement features by combining several API calls. In such a platform, API calls explain the

feature of the application. While a single API call sometimes directly correspond to a single feature, a sequence of API calls corresponds to a single feature in most cases.

In this chapter, we propose a semi-automatic approach to extracting features from Android applications. The premise of our proposed solution is that a feature can be associated with a particular sequence of API calls. API calls are used to control GUI components, network connections, and hardware devices such as a camera, GPS, or touch screen. Although software developers can use arbitrary sequences of API calls, we hypothesize that a popular feature of an application is likely to be implemented by the same sequence of API calls, since similar applications use the same set of APIs [43, 42]. In our proposed solution, therefore, we automatically extract common sequences of API calls in two or more applications, and manually associate each of these with a feature name. We use the associations as a knowledge-base. We then automatically extract API calls from other target applications and, using our knowledge-base, output feature names that are associated with the API calls. As a case study, we have built a knowledge-base from 6 applications and extracted features from other 5 applications. The result shows that our approach is promising to extract features of applications and show important differences among applications.

3.2 Associating API Calls with Feature Names

The objective of our study is to extract a list of features from multiple applications and build a knowledge-base. A user can then more easily compare the features of two or more applications. Our approach has two phases: build a knowledge-base from a set of applications, and, using the knowledge-base, extract and list the features from another set of applications.

Our knowledge-base is defined as a set of associations $\langle S, f \rangle$, where S is a sequence of API calls and f is a feature name. We build a knowledge-base using the following three steps. Figure 3.1 shows an overview.

Step 1: Extraction of sequences of API calls

We translate each application into a set of sequences of API calls. As Android applications are written in Java, we extract a sequence of Android API calls from each method of the application. A method call is identified by its name and the receiver type declared in the source code. We recognize an Android API as any method call whose fully qualified class names start with "`android.`" or "`com.google.android.`". Figure 3.2 shows an example of a sequence of API calls extracted from a method in an application. API calls in a sequence are sorted by line number. If two or more API calls are involved in

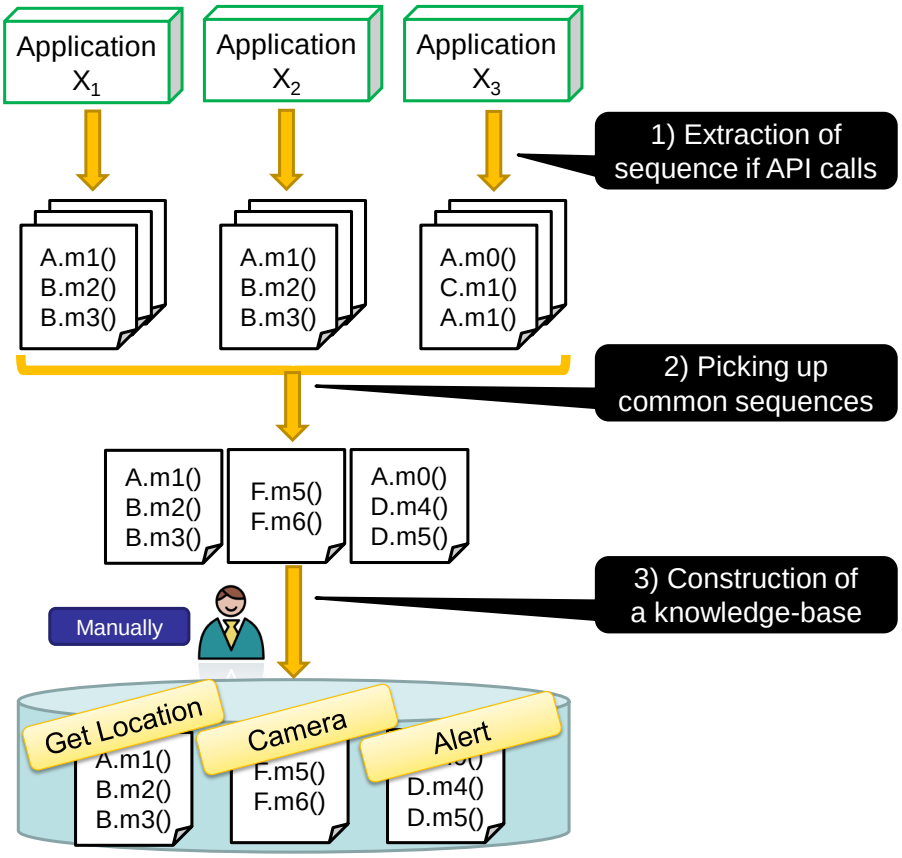


Figure 3.1: Building a knowledge-base

a single line, they are sorted in alphabetical order. As a result, N applications are translated into sets $Apps = \{A_1, \cdots, A_N\}$, where A_i is a set of sequences of API calls extracted from the i-th application.

Step 2: Picking up common sequences

We extract common sequences of API calls involved in at least two applications as candidates for features. We compute a set of common sequences as follows:

$$C = \bigcup_{A_i, A_j \in Apps, i \neq j} \{LCS(s,t) | s \in A_i, t \in A_j\}$$

where $LCS(s,t)$ is the longest common subsequence of two sequences s and t. We exclude sequences that consist of only a single API call from C. We denote the resultant set by $CommonAPI$.

Step 3: Construction of a knowledge-base

We manually associate each sequence S in $CommonAPI$ with a feature name, f, and store the association $\langle S, f \rangle$ in a knowledge-base. A

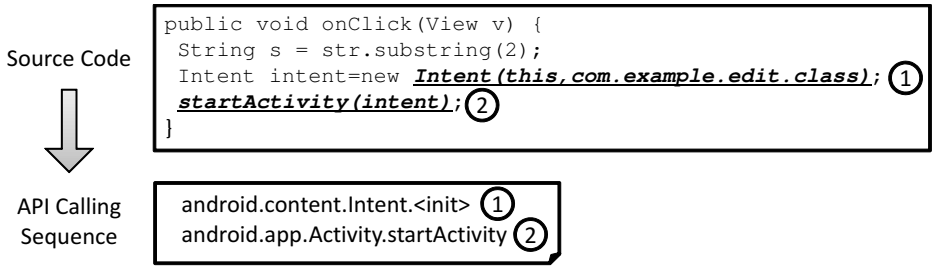


Figure 3.2: Extracting sequences of API calls

feature name can be associated with a sequence if the sequence controls a particular device or system component, because application features are often characterized by devices and components used by the application.

We use the knowledge-base to translate API calls in a target application into feature names. If an application involves a sequence S', including a subsequence S, matching an association $\langle S, f \rangle$ in the knowledge base, we output f as a feature of the application.

3.3 Case Study

We conducted a case study to evaluate whether our approach could extract the features of applications. We collected 11 applications labeled "Map" in Google Code as shown in Table 3.1. We built a knowledge-base from six applications (KB1-KB6) and then used it to extract features from the remaining applications (T1-T5).

We extracted 156 common API calling sequences from the six applications. I manually checked them and could associate names with 23 out of the 156 sequences. Table 3.2 shows an example of the sequences and their feature names. The feature names simply describe what components are controlled by the API sequences. In this example, "Alert dialog," "Submenu," and "Show Toast (pop-up message)" are related to the user interface, while "Get Location" and "Set Location" are related to map features. Using the knowledge-base, we then extracted a list of features for each application (T1-T5). Table 3.3 shows the features found in the target applications. From these results, without using the applications, we could observe that T1 and T5 can specify a location on a map and that T2 is probably not a map viewer.

It should be noted, as we hypothesized, that 18 of the 23 identified API calling sequences are involved in at least one target application. This result is promising because it indicates that a small knowledge-base could cover the popular features of many applications in the same category.

Features found by the tool are considered that the application actually has, but features which are not reported are not always considered that the application doesn't have. API calls which are not registered in a knowledge-base are not found by the tool, so existing features may not be detected if the knowledge-base contains not enough data.

3.4 Related Works

While the proposed method focused on extraction of the features of software, some researches and tools trying to compare the target software.

If source code of the applications is available, comparing them is a one of the solution to compare the features of the applications. UNIX diff[45] is a simple way to compare source code. Unix diff shows only lines which have been changed, and Semantic Diff[26] shows changed lines and its effects on dependence relation between variables. Users can know differences between two versions of software, but these text-based comparisons might extract the whole source code as diff if their design is different even though they use the same programming language.

Grechanik *et al.* proposed a tool Exampler [18], for finding highly relevant software projects from large archives of applications. Exampler create API dictionary from help pages and provide a search engine for finding relevant applications. While my approach did not used the documents even building a knowledge-base, using a well-written documents would be useful instead of naming API calls.

Another use of API calls is application porting from source platform to a target platform. Gokhale *et al.* proposed a method to map APIs between different platforms [17]. They also use an idea that features are related to API calls. Their tool Rosetta get traces of similar applications for different platforms, then the tool identifies API calls with a similar feature. Since their comparison target is applications on different platforms, they consider the call position, call context, edit distance of the method names and so on, while our approach use a simple call sequence only since we compared applications using similar API set.

3.5 Conclusion

We proposed an approach to extracting features from an Android application using a knowledge-base built from source code of applications. The results of a case study showed that our approach could extract features from an application and list them in terms of devices and components used by the applications. Although our approach is promising, we were unable to represent the usage or purpose of the components. We also could not recognize features implemented by general-purpose GUI components. To

resolve this problem, we intend to enhance our approach using information about data names and types used in applications. In addition, we would like to use our approach to understand the variability of software product lines in our future work.

Table 3.1: Applications used in the case study

ID	Application name	LOC	#API calls
KB1	OpenGPSTracker	8122	1099
KB2	mapsforge	37326	1407
KB3	OSMandroid	3150	175
KB4	TripComputer	14487	825
KB5	shareyourdrive	2761	346
KB6	savage-router	1041	66
T1	MapDroid	6387	1160
T2	cycroid	1278	761
T3	yozi	5348	159
T4	maps-minus	1785	218
T5	BigPlanetTw	4139	432

Table 3.2: Example of sequence of API calls

Feature name	Sequence of API calls
Alert dialog	android.app.AlertDialog.Builder.<init> android.app.AlertDialog.Builder.setTitle
Get Location	android.location.Location.getLatitude android.location.Location.getLongitude
Show toast (pop-up message)	android.widget.Toast.makeText android.widget.Toast.show
Set Location	android.location.Location.setLatitude android.location.Location.setLongitude
Submenu	android.view.Menu.addSubMenu android.view.SubMenu.setIcon

Table 3.3: Features identified in five applications

ID	T1	T2	T3	T4	T5
Alert Dialog	✓	✓	✓	✓	✓
Get Location	✓		✓	✓	✓
Show Toast (pop-up message)	✓	✓		✓	✓
Set Location	✓				✓
Submenu					✓

Chapter 4

Measuring Copying of Java Archives

4.1 Introduction

Reusing software components reduces time and cost when constructing new software, and copying the whole of a library into the software development project is one of the major types of reuse. Heinmann *et al.* showed that software reuse is common among open source Java projects and the black-box is the predominant form of reuse [20].

In the case of Java, library archive files often contain their dependent libraries. One reason is that developers want to use specific versions of libraries that might be considered reliable.

Once black-box reuse method has been done, it might not be known which version of which library is included in the library archive file. Davis *et al.* pointed out that the provenance of included components is not clearly stated and they proposed a method to determine the provenance of source code contained within Java archives [10].

However, there is a possibility that developers are also copying duplicated libraries in the reused libraries without knowing that. When developers copy some libraries into their project, they may also unconsciously copy the same version of the library they already have or copy different versions of the library.

Developers might not be aware of inside of the library. If some libraries have a vulnerability then developers will update it to the latest versions, but developers hardly take care of the nested libraries and old versions of libraries might be left inside. If duplicated libraries are different versions, they will contain the classes with the same package name and one of them will be loaded on the runtime but it is not clear that which versions of libraries or classes are loaded.

Although library duplication is potentially problematic, there are less

researches for inside of Java libraries. The mainstream of the software clone research is for the source code [51] and few researches focuses clones of other software artifacts. There are some researches for Java archives [5, 54]. They dealt with a problem that how to compress class files to reduce file size, but they paid no attention to the duplication of class files or whole of archives.

In this chapter, we performed an experiment to measure copying of jar archives in the Maven Central Repository, a collection of open source Java libraries. We set these research questions as a first step of the study of this type of duplication.

RQ1: How many jar files in a large software repository contain jar files inside and how many jar files are reused?

RQ2: Does duplication of reused jar files in other jar files really exist? If so, are those duplicated jar files the same version or different versions?

4.2 Background

Apache Maven [41] is a software project management and comprehension tool. It automatically downloads dependent Java libraries from Maven repositories at build time. Maven Central Repository (Maven2) is the default repository of Apache Maven. Maven2 repository contains many popular libraries and projects.

Java archive file is the typical format used to distribute Java applications and libraries. A Jar file contains Java class files and metadata and resources, and even another jar archive inside.

We define the term "top-level jar file" and "inner jar file" in this paper. A "top-level jar file" is a jar file found in the Maven2, and therefore, it corresponds to a component ready to be reused. An "inner jar file" is a jar file that is included in another jar file, either a "top-level jar file" or an "inner jar file".

Figure 4.1 shows an example of a library with nested jar files. A node corresponds to a jar file. The jar file at the start of the arrow contains the jar file at the end of the arrow. In this case, the top-level jar file *A.jar* is found in the target repository and contains four inner jar files in it; *B.jar*, *C.jar*, *E.jar*. *C.jar* contains *B.jar* which is exactly same file as *B.jar* under *A.jar*. *D.jar* contains *C.jar* so *B.jar* appeared again inside of *C.jar*. *E.jar* contains *B-2.jar* which is the newer version of *B.jar*. In Figure 4.1, all jar files in the right side of *A.jar* are inner jar file of *A.jar*. *B.jar* and *C.jar* are duplicated, and there are two versions of *B.jar* (*B.jar* and *B-2.jar*).

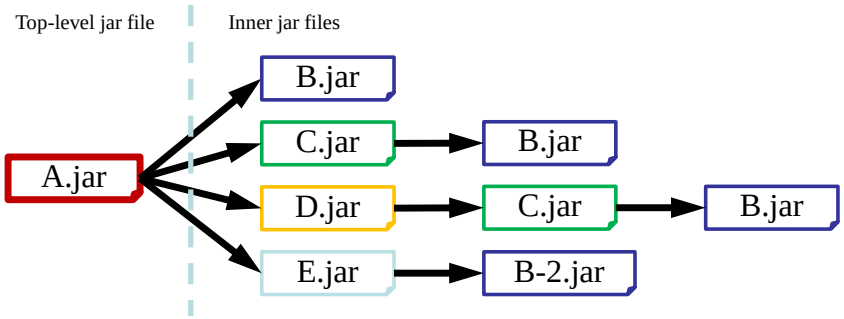


Figure 4.1: Example of nested jar files

4.3 The Experiment

We conducted an experiment to find how many archive files contained duplicate archive files inside. We detected two types of duplication of jar files: the same version of the same library and the different versions of the same library.

Setup We used a framework for Software Bertillonage proposed by Davis *et al.* [10]. The framework extracts metrics of source and archive files. We use two metrics, the filename and SHA1 hash of the file contents, to find jar files with exactly the same contents. If the file is contained in the archive file, SHA1 hash of the parent file is also extracted so that we can find out the contents of jar file.

Inner Jar Files There are 607,319 top-level jar files in the Maven2 repository. Removing exactly the same files, with the same file name and the same hash, we get 599,498 top-level jar files. Checking the contents inside each top-level jar files, we found that 4,747 top-level jar files contain at least one jar file inside. 1,833 of them contains only one jar file and the largest one has 282 jar files in it, 13.1 on average and median was 2. We also found that 118,361 different inner jar files are contained in other jar files and 89,054 of them are found in the Maven2 repository as a top-level jar file. This means that most inner jar files are reused directly from the Maven2 repository.

Detecting Duplication To find the two types of duplication inside jar files, we checked inner jar files using the following method:

First, we identify duplication of the same version of the libraries. If two jar files have the same file name and the same file hash, this means that they have exactly the same contents so they are considered as duplicated and they are the same version. We did not care about the

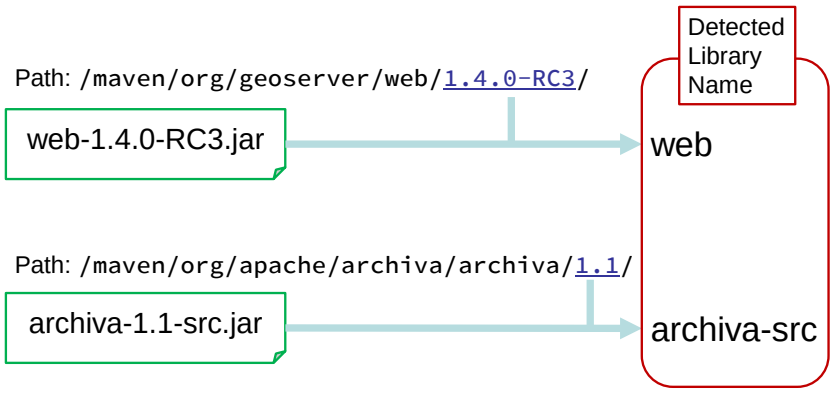


Figure 4.2: Example of how the jar filename was use to identify the name of the library

Table 4.1: Analysis result for A.jar in Figure 4.1

Step	File list
Unique inner jar file	B, C, D, E, B-2
Unique inner jar file without version names	B, C, D, E

nest level of jar file. In Figure 4.1, three *B.jar* are all different nest level counting from *A.jar*, but it does not affect the analysis.

Second, we identify duplication of different versions of the libraries. To detect different versions of the same library, we remove the version information from the jar file name. Version names are not only restricted in the number but also some strings such as "RC" and "SNAPSHOT". We found that many libraries are also found in the Maven2 repository so we use the jar path name in Maven2 to identify its version. In the Maven2, most projects have their own directory, and a subdirectory for each version. We regard the directory name as the version name of the library and remove it from file name of the library. We also remove a leading hyphen or underscore with the version name. Figure 4.2 shows two examples. This step is skipped if the library is not found in the Maven2 repository since we cannot get the version name from the directory name.

Table 4.1 shows the example result of analysis for Figure 4.1. In the example Figure 4.1, *B.jar* appears three times and *C.jar* appears twice. In this case *B.jar* and *B-2.jar* have the same library name so they are determined as different versions of the library B.

Table 4.2 shows the results of the experiment. We count the number of libraries in two ways; counting number of jar files and counting number of

Table 4.2: Duplication of inner jar files

	Contains inner jar	Duplication Type Same	Different	Both	Total duplication
#files	4,747	105	394	30	469
#projects	886	39	49	14	73

Table 4.3: List of inner jar files of nexus-app-1.7.1-tests.jar

antlr-2.7.6 (7)	nexus-3148-1.0.20100111.064938-1
antlr-2.7.7 (5)	nexus-3148-1.0.20100111.065026-2
log4j-1.2.12 (5)	nexus-indexer-1.0-beta-3-20010711.162119-2
log4j-1.2.13 (5)	nexus-indexer-1.0-beta-3-SNAPSHOT
log4j-1.2.13-sources (5)	nexus-indexer-1.0-beta-4
log4j-1.2.14 (5)	nexus-indexer-1.0-beta-4-SNAPSHOT
log4j-1.2.14-sources (5)	nexus-indexer-1.0-beta-4-SNAPSHOT-cli
log4j-1.2.15 (3)	nexus-indexer-1.0-beta-4-SNAPSHOT-jdk14
log4j-1.2.8 (7)	nexus-indexer-1.0-beta-4-SNAPSHOT-sources
log4j-1.2.9 (7)	nexus-indexer-1.0-beta-5-20080711.162119-2
	nexus-indexer-1.0-beta-5-20080718.231118-50
	nexus-indexer-1.0-beta-5-20080730.002543-149
	nexus-indexer-1.0-beta-5-20080731.150252-163
	nonuniquesnap-1.1-SNAPSHOT
	plexus-plugin-manager-1.0-20081125.071530-1
	sonatype-test-evict_1.4_mail-1.0-SNAPSHOT
	very.very.long.project.id-1.0.0-20070807.081844-1
	very.very.long.project.id-1.1-20070807.081844-1

(n) represents the number of files

projects used disregarding their version as described as Step 3.

In total, 469 jar files contain duplicate libraries inside, about 10% of the top-level jar files that contains inner jar files. Counting the number of projects, the result also shows that about 8% of maven projects contain inner jar files that have duplicated libraries in them.

We found both types of duplication in the Maven2 repository: 394 jar files contain the same version of the same library and 105 jar files contain the different versions of the same library. We also found that 30 files have both types of duplication.

Some jar files which have duplication of different versions of the archive files have "test" in their file name. The inner jar files of *nexus-app-1.7.1-tests.jar*, listed in Table 4.3, it contains 28 different inner jar files, including six different versions of log4j library. In total there are 32 inner jar files named log4j inside *nexus-app-1.7.1-tests.jar* and each versions of log4j appeared 3 to 7 times.

4.3.1 Revisiting Research Questions

RQ1 *How many jar files in a large software repository contain jar files inside and how many jar files are reused?*

In the Maven2 repository, there are 4,747 of 599,498 jar files that contain inner jar files. The number of inner jar files is at least one and at most 282 files, 13.1 on average and median was 2. From the point of view of reuse, 89,054 of top-level jar files in the Maven2 repository also appeared as inner jar files.

RQ2 *Does duplication of reused jar files in other jar files really exist? If so, are those duplicated jar files the same version or different versions?*

Yes, 10% of jar files which have inner jar files contains duplicated jar files. We can say that the duplication in libraries are not an unusual problem. Both type of duplication are found in the Maven2 repository.

4.4 Conclusion and Future Work

Developers reuse existing libraries by copying them into the software development project and this style reuse reduces time and cost on constructing new software. On the other hand, there is a possibility that developers are also copying duplicated libraries in the reused libraries without knowing that.

The result of our experiment indicates that the duplication of archive files in a single archive file is not frequent, but it exists. And furthermore, we must remember that many archive files are copied into others so that further duplication can occur. Concretely, we found that about 5,000 jar files in the Maven2 repository contain other jar files in them and about 470 of them contains duplicate libraries, some of them are the same version and some of them are different versions. We also found that about 14% of top-level jar files in the Maven2 repository are copied into other top-level jar files.

Based on this result, we are planning to perform further studies. We found duplication of jar files but did not check all contents of them, and finding out which duplicated archive is most frequently reused is our future work. In addition, we should also analyze other types of archive files. We only used jar archives but the Maven2 repository has .zip, .tar.gz, .war, .ear formats of archives and these are not limited in binary archives but also source archives.

Another interesting fact is that there are some inner jar files and some duplications even though Apache Maven has a system to download needed

jar files at built time. We want to investigate whether it is possible to remove the duplication.

Chapter 5

Comparing Frequency of Identifier Definition in C and Java APIs

5.1 Introduction

Identifiers in the source code are one of the important elements for source code analysis. Identifiers have been used for different purposes. For instance, work by Subramanian *et al.* [56] showed that developers can recognize which library a code snippet is using, and can locate where is the official API document. Their iterative approach determines the fully qualified names of code elements from a code snippet, using identifier names, return types, arguments and so on from the partial code.

Java implements the object-oriented programming style where methods and fields have different namespace containments. Identical identifiers are distinguished by their fully qualified names, usually as a concatenation of the higher package and class name (i.e., `java.io.BufferedReader.close` and `java.io.BufferedWriter.close`
with the same `close` method name). For this reason, existing work describes that they need syntax-tree based analysis to determine identifiers in the code snippet rather than token-based analysis. On the other hand, the C programming language has only one public namespace that includes both global function names and global variable names.

In this chapter, we conducted a large analysis of C and Java libraries and investigated the frequency of public identifier definitions. We found that they have different tendency on definition, C identifier names are rarely duplicate comparing to Java ones.

5.2 Background

This section describes the well-known behavior of public identifiers and show related works on usage of identifiers in software engineering.

5.2.1 Public Identifiers

Identifiers are code element references (i.e., variables, methods, classes and packages) which are defined at different containment levels. A public identifier is visible from outside the scope of its containment (i.e., public methods, classes or global variables). Visibility enables the outside to access specific classes, variables or methods. We define a public identifier to be unique if it is defined only in a single software package. A software package is a component, library or application that can be independently downloaded and (re)used.

Public Identifiers in C

The C programming language has only one public namespace that includes both global function names and global variable names. This is one of the reasons that identifiers that start with '_' (underscore) are considered reserved and not to be used by programmers (see Section 7.1.3 of the 1999 C ISO Standard [6]). A good programming practice in C is to designate non-public identifiers as static (which makes it file scope only—not to be confused with Java's static keyword). Any other global function or variable is considered to be public. In some operating systems, most notably Windows, the public identifiers of a library (DLL) should be documented explicitly using a .def file [44].

This flat space has prompted some projects to issue guidelines to increase readability of their code. For example, the WXwidgets project states that "The prefix wx must be used for all public classes, functions, constants and macros, no exceptions" [60].

Public Identifiers in Java

According to the ISO Standard 3166 and Code Conventions for the Java TM Programming Language (April 20, 1999) [27], Java stresses the use of easy-to-understand names to provide hints on the functionality. According to the conventions, strict rules regarding the prefix and case-sensitivity exist. An example is that the prefix of a unique package name is always written in lower case, while interface names should be capitalized. Also, the use of white space or reserved words is not recommended. Using verbs in camel case are common conventions for a method name (i.e., `closeFile`). Constants are encouraged to be named in uppercase with underscores. Most organizations reflect their internal conventions or structure in the package naming (i.e., `com.apple.quicktime.v2` or `edu.cmu.cs.bovik.cheese`).

Java implements the object-oriented programming style. Methods and fields have different namespace containments. Identical identifiers are distinguished by their fully qualified names, usually as a concatenation of the higher package and class name (i.e., `java.io .BufferedReader.close` and `java.io.BufferedWriter.close` with the same `close` method name).

5.2.2 Identifiers Used in Software Engineering

Identifier names are widely used in software engineering. Lawrie *et al.* studied the role of identifiers in program comprehension [37], which shows that better comprehension is achieved with full word identifiers rather than single letters or abbreviations. Abebe *et al.* defined lexicon bad smells, such as inconsistent identifier use or odd grammatical structure [1]. Arnaoudova *et al.* defined anti-patterns for identifier naming, inconsistencies between method or attribute naming conventions, documentation, and signatures [3]. Both Abebe and Arnaoudova implemented a detector for bad naming.

Subramanian *et al.* [56] proposed a method of linking source code snippets to the API documentation. They analyzed code snippets on Stack Overflow written in Java and JavaScript and showed that their method can link API elements in the snippet to the documentation with high precision. They used an oracle, the Maven repository for Java and seven libraries including the core JavaScript API for JavaScript. A platform proposed by Inozemtseva *et al.* uses links between code elements and resources such as documentation and show the link via a web browser or an IDE [25]. They dealt with small code snippets or identifiers in natural text. While such small code has limited description, their method can detect APIs code elements using.

On the other hand, a study by Dagenais and Robillard describe contrasting results when linking code elements with corresponding learning resources such as API documentation. They reported that simple mechanical matching between the relevant code methods without context of the learning resource would fail. The study found that 89% of all unqualified Java methods were declared on an average in 13.5 different types [9], making it difficult to understand code. The study looked at four Java open source systems. A study on the naturalness of software by Hindle *et al.* [23] suggests that the token frequencies in the source code are very skewed.

5.3 Experiment Design

The purpose of this experiment is to reveal difference of identifier definition in C and Java libraries. We extract identifier definitions from large software sets and compare them.

For the target of the experiment, we used two large software sets. We selected C libraries from Debian 7.5.0 packages. For Java we analyzed libraries from the Maven central repository, the default repository for software project management and comprehension tool Maven.

For Java we analyzed libraries from the Maven repository. Apache Maven [41] is a software project management and comprehension tool that automatically downloads dependent Java libraries from repositories at build time. Maven repository is the default repository of Apache Maven and it contains many popular libraries and projects. The set of library files are provided by the existing research [10]. We consider each binary jar file as a different library.

Our method consists of the following four generic steps.

Step 1) Collect library files from repository

For Debian, library files were distinguished as 1.) files that resided in the following folders `/lib`, `/usr/lib`, and `/usr/lib64`. 2.) files with extensions of`.o`, `.so` or `.a`. In the case of Maven, we consider that any package is a library; therefore we use all the class files in every jar file.

Step 2) Extract identifier definitions

We extract identifier definitions from the library files. During extraction, we also record the types of identifiers such as function declaration or method definition.

For library files from Debian, we ran the `readelf` Linux command for each library files to extract identifier definitions. We extracted the identifier type and identifier name from the output of `readelf` command. We analyzed Java libraries using the `javap` and extract identifier definitions for each library. In this step, we removed package name from definitions. From the analysis result of previous chapter, we understood that there exists much duplication of Java libraries (i.e, different versions of a library being used in one system) in the repository. We identified and ignored these instances.

Step 3) Filter out irrelevant identifiers

We filter out unrelated identifiers from the extracted identifier definitions. In the case of Debian libraries, some library files are not written in C (e.g., C++). We search for source package of library and extract identifiers defined in source files whose extension is `.c`, then we took the intersection of identifiers from the library files and from source files. In the Maven repository, there exist libraries written in languages other than Java such as Scala, Clojure, and so on. So we sited source file name recoded in the library file and removed

Table 5.1: C function names

#Defined Library	#Identifier Names		#Identifier Definitions	
1	886,134	91.80%	886,134	82.29%
2	60,574	6.28%	121,148	11.25%
3	13,483	1.40%	40,449	3.76%
4	2,571	0.27%	10,284	0.96%
5≤	2,477	0.26%	18,824	1.75%
Total	965,239		1,076,839	

Table 5.2: C variable names

#Defined Library	#Identifier Names		#Identifier Definitions	
1	206,111	92.62%	206,111	82.14%
2	10,646	4.78%	21,292	8.49%
3	3,974	1.79%	11,922	4.75%
4	926	0.42%	3,704	1.48%
5≤	875	0.39%	7,904	3.14%
Total	222,532		250,933	

identifiers that did not belong to a Java file (i.e., has extension of `.java`).

Step 4) Calculation

We counted the number of definitions for each identifier name and check how many libraries defined that identifier. For Debian libraries, we counted that how many source packages defines the specific identifier name. For Maven repositories, we counted that how many Jar files defines the specific identifier name. Maven repository contains some libraries that have same name but different versions, so we summed up Jar files with different versions as a single library.

5.4 Analysis Result

Tables 5.1 and 5.2 show the investigation result for C function names and variable names, and Tables 5.3, 5.4, and 5.5 show the result for Java class names, method names, and function names respectively. Column "#Identifier names" shows the aggregate result for counting identifier names and "#Identifier Definitions" shows the result for counting identifier definitions. Table 5.1 for example, the row whose "#Defined Library" is 2 shows that 60,574 of identifier names are defined in 2 libraries, and totally they are

defined 121,148 times.

Most of identifier names are defined in single library in both C and Java, about 90% in C and about 70% in Java in concrete terms.

There are less identifier names that are defined in multiple libraries.

On the other hand, counting identifier definitions, While about 80% of C identifiers are defined once, it was less than 20% for Java method names. We can see that some specific identifier names are defined in multiple libraries in case of Java method names.

The major reason of duplication of identifier names are follows; auto-generated code, including an external library file, using a same framework. In C function names and variable names, names start with "`yy`" are automatically generated by Yacc parser generator. Those identifiers are defined in the packages related to programming languages; php, ruby, golang, postgresql, and so on. C functions `strlcpy` and `strlcat` are defined in BSD libc to provide less error prone replacements for `strncpy` and `strncat` provided by C standard library. However, they are not included in glibc, commonly used C standard library on Linux, so some libraries provides or includes those functions instead.

5.5 Conclusion

Learning the trend of identifier names in the library helps program comprehension, and unique identifier names allow fast and clear understanding of software. We analyzed two library sets, from Debian for C and from Maven repository for Java, to evaluate how unique public identifiers are within there. The result of our experiment indicates that more than 80% of C identifier names and approximately 70% of Java identifier names are unique to single library. However, the specific method names are defined in multiple Java libraries and it is one big difference between C and Java identifiers. We believe that this result provides useful hints on context through providence. We also argued for overlapping identifier names and find some reasons of duplication.

Table 5.3: Java class names

#Defined Library	#Identifier Names		#Identifier Definitions	
1	445,353	75.85%	564,463	43.44%
2	87,538	14.75%	228,600	17.59%
3	27,217	4.59%	107,338	8.26%
4	11,240	1.89%	59,901	4.61%
5≤	22,067	3.72%	338,991	26.09%
Total	593,415		1,299,293	

Table 5.4: Java method names

#Defined Library	#Identifier Names		#Identifier Definitions	
1	1,010,135	66.03%	2,240,606	15.75%
2	260,930	17.06%	1,269,538	8.93%
3	92,622	6.05%	978,365	6.88%
4	48,160	3.15%	536,944	3.78%
5≤	117,860	7.70%	9,196,312	64.66%
Total	1,529,707		14,221,765	

Table 5.5: Java field names

#Defined Library	#Identifier Names		#Identifier Definitions	
1	387,149	70.28%	565,455	29.21%
2	83,367	15.13%	272,301	14.07%
3	33,335	6.05%	179,962	9.30%
4	16,059	2.92%	118,384	6.11%
5≤	30,951	5.62%	799,915	41.32%
Total	550,861		1,936,017	

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation proposed approaches to analyze program collections and reveal hidden relations in them. Each approach challenged to analyze not only a single program but also a set of program code and offer new insight of the program collection. Program code analysis is the base of further analysis and the improvement of analysis technique will be applicable for another area such as Mining Software Repository. We believe that these results support developers to understanding existing program collections, and useful for maintaining and keep its quality.

In this dissertation, four approaches were proposed.

For software product family, this dissertation proposed the method to extract the approximation of evolution history of them. The result of the experiment showed that the proposed method achieved high recall. In addition, the case study with linux kernel variants showed that the proposed method can detect the origin of unknown versions of the product and also can detect whether those products are branched or not from mainstream. The methodology and the technique we used are simple, but shows promising result in experiments.

For Android applications, this dissertation proposed a semi-automatic method to extract features from source code of them. The result of a case study showed that our approach could extract features from Android application and list them in terms of devices and components used by the applications.

For Java libraries in Maven repository, this dissertation reveals that there are many copies of Java library files among nested library files. The result of our experiment indicates that the duplication of archive files in a single archive file is not frequent, but it exists. And furthermore, we must remember that many archive files are copied into others so that further duplication can occur.

For C and Java libraries, this dissertation found that they have different tendency of definition of identifier names and most of C identifiers are unique to single library. The result of our experiment indicates that specific method names are defined in multiple Java libraries and this fact is one big difference between C and Java identifiers.

6.2 Future Work

Based on the studies and results, some future work is needed for further use of program collections. For future work of the study on software product family is dealt with software merge. Moreover, future work includes adaption of product evolution tree to real problem, such as applying patches for vulnerability to each branches or not.

Future work for the study on Android applications includes automation and accuracy improvement. To expand the proposed approach using code analysis, future work includes using documents such as API documents or introduction text of application.

Future work for the study on library analysis includes a library duplication problem. Very recent study shows that copy of Java library files inside the library file have a risk of incompatibility [28], so future work includes the solution for preventing incompatibility in Maven repository. Another direction includes the lightweight origin analysis for C programs using uniqueness of identifier definitions.

Bibliography

[1] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. Lexicon Bad Smells in Software. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, pages 95–99. IEEE, 2009.

[2] Android Apps in Google Play - The year of opportunity. `http://commondatastorage.googleapis.com/io2012/presentations/live%20to%20website/123.pdf`.

[3] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gael Gueheneuc. A New Family of Software Anti-patterns: Linguistic Anti-patterns. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 187–196. IEEE, mar 2013.

[4] Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the 2nd International Conference on Software Product Lines (SPLC)*, pages 257–271, 2002.

[5] Quetzalcoatl Bradley, R Nigel Horspool, and Jan Vitek. JAZZ: an efficient compressed format for Java archive files. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 7–15, 1998.

[6] ISO/IEC 9899:TC3, C programming language standard. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

[7] The comprehensive perl archive network. `http://www.cpan.org/`.

[8] The comprehensive r archive network. `https://cran.r-project.org/`.

[9] Barthélémy Dagenais and Martin P. Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 47–57, 2012.

[10] Julius Davies, Daniel M. Germán, Michael W. Godfrey, and Abram Hindle. Software bertillonage - determining the provenance of software development artifacts. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[11] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 64–73. IEEE, 2014.

[12] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 25–34, 2013.

[13] F-05d open-source software. `http://spf.fmworld.net/oss/oss/f-05d/`.

[14] D Faust and C Verhoef. Software product line migration and deployment. *Software: Practice and Experience*, 33(10):933–955, 2003.

[15] Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 30, pages 426–438, 1995.

[16] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[17] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. Inferring likely mappings between APIs. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 82–91, 2013.

[18] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, volume 1, page 475, 2010.

[19] The Groovy programming language. `http://www.groovy-lang.org/`.

[20] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. On the Extent and Nature of Software Reuse in Open Source Java Projects. In *Proceedings of the 12th*

International Conference on Software Reuse (ICSR), pages 207–222, 2011.

[21] Armijn Hemel and Rainer Koschke. Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 357–366, 2012.

[22] Hibernate. Everything data. - Hibernate. `http://hibernate.org/`.

[23] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[24] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go? — Integrated code history tracker for open source systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 331–341, 2012.

[25] Laura Inozemtseva, Siddharth Subramanian, and Reid Holmes. Integrating software project resources using source code identifiers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 400–403, New York, New York, USA, 2014. ACM Press.

[26] D. Jackson and D.a. Ladd. Semantic Diff: a tool for summarizing the effects of modifications. In *Proceedings of the 1994 International Conference on Software Maintenance (ISCM)*, pages 243–252, 1994.

[27] ISO standard 3166, 1981, Java programming language guidelines. `http://www.oracle.com/technetwork/java/codeconventions-135099.html`.

[28] Kamil Jezek and Jan Ambroz. Detecting Incompatibilities Concealed in Duplicated Software Libraries. In *Proceedings of the 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 233–240. IEEE, aug 2015.

[29] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[30] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. MUDABlue: An automatic categorization system for Open

Source repositories. *Journal of Systems and Software*, 79(7):939–953, 2006.

[31] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceeding of the 33rd international conference on Software engineering (ICSE)*, page 351, New York, New York, USA, 2011.

[32] Kinds of compatibility: Source, binary, and behavioral (joseph d. darcy's oracle weblog). `https://blogs.oracle.com/darcy/entry/kinds_of_compatibility`.

[33] Rainer Koschke, Pierre Frenzel, Andreas P. J. Breu, and Karsten Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17(4):331–366, 2009.

[34] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[35] Charles W. Krueger. Easing the Transition to Software Mass Customization. In *Revised Papers from the 4th International Workshop on Product-Family Engineering (PFE)*, pages 282–293. Springer Berlin Heidelberg, 2002.

[36] Thierry Lavoie, Foutse Khomh, Ettore Merlo, and Ying Zou. Inferring Repository File Structure Modifications Using Nearest-Neighbor Clone Detection. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 325–334, 2012.

[37] Dawn Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a Name? A Study of Identifiers. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC)*, pages 3–12. IEEE.

[38] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[39] The linux kernel archives. `https://www.kernel.org/`.

[40] Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang. Are faults localizable? In *Proceedings of the 9th Working Conference on Mining Software Repositories (MSR)*, pages 74–77, 2012.

[41] Apache maven project. `http://maven.apache.org/`.

[42] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 364–374, 2012.

[43] Collin McMillan, Mario Linares-Vásquez, Denys Poshyvanyk, and Mark Grechanik. Categorizing software applications for maintenance. In *Priceedings of the 27th International Conference on Software Maintenance*, pages 343–352, 2011.

[44] Exporting from a dll using def files. `http://msdn.microsoft.com/library/d91k01sh.aspx`.

[45] Eugene W Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, nov 1986.

[46] Makoto Nonaka, K Sakuraba, and K Funakoshi. A preliminary analysis on corrective maintenance for an embedded software product family. *IPSJ SIG Technical Report*, 2009-SE-166(13):1–8, 2009.

[47] OpenJDK. `http://openjdk.java.net/`.

[48] David Lorge Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 279–287, 1994.

[49] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[50] Postgresql: The world's most advanced open source database. `http://www.postgresql.org/`.

[51] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[52] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference (SPLC)*, pages 156–160, 2012.

[53] RubyGems.org. `https://rubygems.org/`.

[54] Dimitris Saougkos, George Manis, Konstantinos Blekas, and Apostolos V. Zarras. Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments. *IEEE Transactions on Software Engineering*, 33(7):478–495, 2007.

[55] Software product lines | overview. http://www.sei.cmu.edu/productlines/.

[56] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 643–652, 2014.

[57] Vasil L Tenev and Slawomir Duszynski. Applying bioinformatics in the analysis of software variants. In *IEEE International Conference on Program Comprehension*, pages 259–260, 2012.

[58] Kai Tian, M Revelle, and Denys Poshyvanyk. Using Latent Dirichlet Allocation for automatic categorization of software. In *Proceedings of the 6th Working Conference on Mining Software Repositories (MSR)*, pages 163–166, 2009.

[59] Peter Weissgerber and Stephan Diehl. Identifying Refactorings from Source-Code Changes. In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE)*, pages 231–240, 2006.

[60] Coding guidelines - wxwidgets. https://www.wxwidgets.org/develop/coding-guidelines/#wx_prefix.

[61] Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya, and Katsuro Inoue. Measuring Similarity of Large Software Systems Based on Source Code Correspondence. In *Proceedings of the 6th international conference on Product Focused Software Process Improvement (PROFES)*, pages 530–544, 2005.

[62] Yuki Yano, Raula Gaikovina Kula, Takashi Ishio, and Katsuro Inoue. VerXCombo: An interactive data visualization of popular library version combinations. In *Proceedings of the 23rd International Conference on Program Comprehension (ICPC)*, pages 291–294, 2015.

[63] Kentaro Yoshimura and Ryota Mibe. Visualizing code clone outbreak: An industrial case study. In *Proceedings of the 6th International Workshop on Software Clones (IWSC)*, pages 96–97, 2012.

[64] Kentaro Yoshimura, Fumio Narisawa, Koji Hashimoto, and Tohru Kikuno. FAVE: factor analysis based approach for detecting product line variability from change history. In *Proceedings of the 5th International Workshop on Mining Software Repositories (MSR)*, pages 11–18, 2008.