Early Detect Problematic File Groups During Software Evolution

Di Cui*, Ting Liu*, Yuanfang Cai†, Qinghua Zheng*, Wuxia Jin*, Qiong Feng†, Yu Qu*,
*Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China
†Department of Computer Science, Drexel University, Philadelphia, USA
{cuidi,wx_jin}@stu.xjtu.edu.cn; {tingliu,qhzheng,yuqu}@mail.xjtu.edu.cn; yc349@drexel.edu

***Abstract*—During software evolution, some structurally connected file groups frequently incur bugs and changes, consuming significant maintenance costs. We call such group as problematic file group (PFG). In this paper, we proposed a method to identify potential PFGs at early stage during software evolution, by combing structural relations (SR) derived from source code syntax and IR-based relations (IRR) derived from source code lexicon, so that developers can save maintenance costs by fixing these issues in time. First, we collect a set of file pairs as suspect file pairs (SFP) by calculating these two types of software relations. Next, for each SFP, we search its common design rule to generate suspect file unit (SFU). Finally, we recursively merge SFUs to discover suspect file groups (SFG) as PFG candidates. We evaluate our method using 838 releases of 15 open source projects, including 33353 bug reports and 86690 revision commits. The result shows that our identified file groups use 14% of all the file to capture almost 70% of top 10% error-proneness with an enough high precision: 92%. Moreover, our groups also incur 957% bug frequencies and 1050% change frequencies than average in future versions. In summary, our method can effectively and efficiently detect PFGs in time during software evolution.**

***Index Terms*—Software Quality, Software Maintenance, Software Evolution.**

I. Introduction

As software evolves, an increasing amount of maintenance efforts were spent on software quality assurance [1]. However, the rapid growth of complex software structure poses great challenge to maintenance activities. Recent research has shown that bug-prone files are often connected by flawed structure [2]. For a software system, there always exist several structurally-connected file groups frequently incurring bugs and changes, which consumes significant maintenance costs during software evolution [3]. We call such group as problematic file group (PFG). For example, Figure 1 shows four PFGs marked with red color in *AVRO*[1]-1.3.0, causing repeated bugs and changes with a much higher probability. PFG is a typical design flaw caused by quick and dirty implementation but frequent rolling back later. Whereas one PFG may not be burdensome, the accumulation of them will gradually make software harder to maintain, debug, evolve, and eventually cause the ripple effect. If PFGs can be discovered early and fixed via refactoring their flawed structure in time, it is possible to avoid the increasing maintenance cost.

[1]http://avro.apache.org/

Table I shows the evolution of one of these four PFGs from *AVRO*-1.0.0 to *AVRO*-1.3.0. This group consists of six files organized with a pyramid structure. *GenericData* is the top-most file dominating other five files. By tracing the revision history, bug reports and source code of this group, we discovered it was first introduced when fixing *AVRO-110*[2] in 1.1.0. In subsequent versions, this group consumes significant maintenance costs. From 1.2.0 to 1.3.0, developers spent nearly 700 lines of code related to 18 bugs in this group. If this PFG can be detected early and fixed in time, these extra costs may be saved.

Within the best of our knowledge, the state-of-the-art related techniques on diagnosing PFGs can be roughly divided into two families. The first approach is metric-based. One representative work is Decoupling Level Metrics (DL) [4], which evaluates the maintainability of software structure based on the design rule theory [5]. We employ DL in detecting the preceding PFG. As shown in Table I, DL does monitor *AVRO*'s architectural health from 1.0.0 to 1.2.0. However, this high-level approach cannot locate the problematic file group directly.

The second approach is history-based. One of the most recent work is hotspot detection [6]. It can examine several connected file groups strongly correlated with bugs and changes. We also employ hotspot in detecting the preceding PFG. As shown in Table I, this group is identified in 1.3.0. However, hotspot detection relies on adequate revision history, incurring the following two issues: First, PFGs cannot be identified as soon as they were introduced. In the preceding example, this group was detected after significant maintenance costs were paid. By 1.3.0, developers have committed nearly 700 lines of code. Second and most importantly, hotspot misdiagnoses file groups already fixed. In the preceding example again, this group actually has been patched in 1.3.0 and will not contribute to bug-proneness in a long time. However, it is still detected by hotspot [3, 6].

In order to locate PFGs quickly and accurately, we collect 33353 bug reports and 86690 revision commits of 838 versions of 15 open source projects, and systematically investigate the relationship between bug-prone files. We first found a problematic pattern to reveal PFG. Using *AVRO* from 1.0.0 to 1.3.0 as an example shown in Table II, we observed that:

[2]Implement *Comparable* interface and employ singleton design pattern in *GenericData*

1) Bug-prone files are relevant to source code syntax. We collect six types of syntactic relations, referred as structural relations (SR), including inheritance, implementation, method call, field access, type reference, and instance creation. This observation is consistent with Lu et.al's work [2]. As shown in Table II, from 1.1.0 to 1.3.0 of *AVRO*, SRs capture all the bug-prone files (100%). We discovered multiple instances (452-771) of SRs. However, only a small portion (18.58%-28.66%) of them are related to bug-prone files. The structural relation is not efficient to hint bug-proneness.

2) Bug-prone files also present high similarity on source code lexicon. We employ information retrieval techniques to measure the textual similarity between source code lexicons including class name, method name, global varibale name, and comments. We normalize these data to obtain IR-based relation (IRR). As shown in Table II, from 1.1.0 to 1.3.0 of *AVRO*, the average bug rate of IRRs is 74%, which is 336% higher than SRs. On the contrary, the average number of IRRs is merely 48 instances, which is 8% of SRs. Although IRRs only cover 46 bugs (about 50% of 87 bugs in total) and 36 bug-related files(about 32% of 110 files), it is still a practical indicator to inform bug-proneness.

3) The combination of source code lexicon (IR-based relations) and source code syntax (structural relations) capture newly introduced bug-prone file during software evolution. We explore the interaction of IR-based relations and structural relations in Table III and discovered: From 1.1.0 to 1.3.0, the intersection of IRRs and SRs stably contain 10 instances and the average bug rate of them is 58.8%. On the contrary, the difference of IRRs and SRs contain 23 to 93 instances, and the average bug rate of them is as high as 80.2%. From 1.1.0 to 1.3.0, we observed the number of files and bugs in this system is continually increasing shown in Table I. Therefore, the difference of IRRs and SRs do capture newly introduced bug-prone files in time.

Based on these three observations, we discovered a typical pattern, the difference of IRRs and SRs, to hint PFG. This indicates, if two files are involved in IR-based relations but not in structural relations, they are likely to be a suspect file pair (SFP). According to the information hiding principle [7], the structurally isolated files should encapsulate implementation details and evolve independently. If they are related to IR-based relations at the same time, it has the possibility to be a design flaw. Based on this pattern, in this paper, we proposed a method, by combining IR-based relations (IRR) and structural relations (SR), to identify potential problematic file group (PFG) at early stages.

Our method detects PFG as follows: First, we extract structural relations and IR-based relations among files. Structural relations are extracted from the source code syntax using *Understand*[3], a commercial reverse-engineering tool. IR-based relations are extracted from the source code lexicon using information retrieval techniques. Second, we collect suspect file pairs (SFP) by calculating the difference of IRRs and

[3]https://scitools.com/

TABLE I: A Case of Problematic File Group (PFG)

System				Problematic File Group (PFG)			
Release	Fs[1]	Bugs[2]	DL	Fs[1]	Bugs[3]	BugChurn[4]	Hotspot
1.0.0	72	–	70	–	–	–	–
1.1.0	91	18	62	6	8	522	×
1.2.0	102	12	61	6	2	80	×
1.3.0	156	57	74	6	16	618	✓

[1] The number of files in system or PFG
[2] The number of bugs in system
[3] The number of bugs involved at least one file in PFG
[4] The number of lines of code spent on fixing bugs in PFG

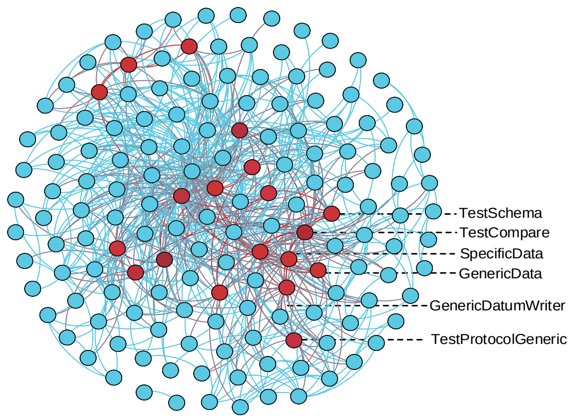


Fig. 1: The PFGs in Avro 1.3.0

SRs. Next,for each SFP, we search its common design rule to generate suspect file unit (SFU). We collect SFPs and SFUs by employing design structure matrix (DSM), a state-of-the-art tool to manage multiple types of software relations. Finally, we recursively merge suspect file units (SFU) to construct suspect file groups (SFG) considered as problematic file group (PFG) candidates.

Compared with measuring-based approaches, our method can accurately locate the potential problematic file groups. Compared with history-based approaches, we can find these file groups at early stages. We evaluate our method on 838 versions of 15 projects. The experiment results showed that: 1) The identified groups only use 14% of all the files 2) Almost 70% of the top 10% bug-prone files are covered by the identified groups; 3) The precision of identified groups is as high as 92%; 4) The identified file groups will incur 957% and 1050% bugs and changes in future versions than average.

The rest of the paper is organized as follows: Section II gives a running example to illustrate the idea of our approach. Section III presents the details of our approach. Section IV evaluates experimental results. Section V presents some discussions. Section VI shows the related work. Section VII finally concludes the paper.

II. Illustrative Example

In this section, we use the PFG in Table I as an example to illustrate related concepts and the procedure of our approach.

TABLE II: Structural Relation via IR-based Relation

System				Structural Relation (SR)					IR-based Relation (IRR)				
Release	Fs	Bugs[1]	BugFs[2]	Bugs[1]	BugFs[2]	SR	BugSR[3]	Bugrate[4]	Bugs[1]	BugFs[2]	IRR	BugIRR[3]	Bugrate[4]
1.0.0	72	–	–	–	–	319	–	–	–	–	18	–	–
1.1.0	91	18	46	18	46	452	84	18.58%	12	16	23	19	82.61%
1.2.0	102	12	7	12	7	496	102	20.56%	6	4	29	14	48.28%
1.3.0	156	57	57	57	57	771	221	28.66%	27	16	93	84	90.32%

[1] The number of bugs in system and the number of bugs involved at least one file in SRs or IRRs
[2] The number of bug-prone files in system, SRs or IRRs
[3] The number of SRs or IRRs in which both of subordinate files are bug-prone.
[4] The rate of the BugSRs or BugIRRs in all SRs or IRRs .

TABLE III: The Interaction of IR-based Relation and Structural Relation

System	$IR \cap SR$[1]			$IR \cap \neg SR$[2]		
Release	Rs[3]	BugRs[4]	Bugrate[5]	Rs[3]	BugRs[4]	Bugrate[5]
1.0.0	10	–	–	8	–	–
1.1.0	10	7	70.00%	13	12	92.31%
1.2.0	11	4	36.36%	18	10	55.56%
1.3.0	10	7	70.00%	83	77	92.77%

[1] The intersection of IRRs and SRs.
[2] The difference of IRRs and SRs.
[3] The number of software relations.
[4] The number of Rs or BugIRRs in which both of subordinate files are bug-prone.
[5] The rate of the BugRs in all Rs.

A. Concepts based on Design Structure Matrix

Design structure matrix (DSM) is a state-of-the-art tool to visualize multiple software relations. Based on DSM, design rule hierarchy (DRH) is designed for understanding the structure of software relations according to design rule theory (DR). We introduce these concepts as follows:

Design Structure Matrix (DSM). A DSM is a square matrix. Each element in DSM is a source file. The rows and columns of a DSM are labeled with the same set of source files in the same order. Each cell in DSM represents the relations between the file in row and the file in column. A marked cell in row x and column y, cell (x, y), means that the file in row x depends on the file in column y. The marks in the cell are refined to represent different types of relations. We use the DSM shown in Figure 3.(b) as an example. In Figure 3.(b), cell(2,1) is labeled with “dp”, which means *GenericDatumReader* structurally depends on *GenericData*. Cell (2,1) is also labeled with ”,7”, which represents *GenericData* and *GenericDatumReader* changed together with 7 times in the revision history. Cell(7,5) is marked with blue color, which indicates *GenericDatumReader* and *SpecificData* are connected with IR-based relations. In this paper, we employ DSM to manage structural relations and IR-based relations.

Design Rule Theory (DR). Baldwin and Clark proposed the design rule theory (DR) [5] to explains the power of modularity in the form of options. Sullivan [8] first introduced design theory to software design. According to this theory, source files in the software system can be classified into two roles: design rules and independent modules. Design rules usually represents interfaces or abstract classes that decouple the rest of system into independent modules. The independent modules usually represent implementation classes. They depend on design rules but evolve independently with each other.

For example, in an Abstract Factory Pattern [9], the abstract factory interface is considered as a design rule, which decouples concrete factory classes as independent modules. If the interface remains stable, the system is not affected by the change of concrete factory classes.

Design Rule Hierarchy (DRH). To automatically identify design rules and independent modules in the software system, design rule hierarchy (DRH) is proposed [10–12]. DRH is a clustering algorithm to recursively identify the tree structure in the system. DRH clusters a DSM into a hierarchy structure with several layers, and each layer is decoupled into a few modules. Files in low layers only depend on the files in higher layers. Files in the different modules of the same layer are mutually independent from each other. The files in the top layer represent the most influential files, such as key abstract classes and interfaces. The files in the bottom layer represent the truly independent modules. They can be improved or replaced without influencing other parts of the system. For a source file, the responsible design rule can be found in a higher layer.

As shown in Figure 3.(b), 14 files are clustered into a DRH with 5 layers: l_1:(rc_1-rc_2), l_2:(rc_3-rc_4), l_3:(rc_5-rc_7), l_4:(rc_8), and l_5:(rc_9-rc_{14}). rc_n represents the file in row n. l and m are short for layer and module. The top layer l_1 contains one module: m:(rc_1-rc_2). The files in this layer are the most influential design rules. The bottom layer contains 5 modules: m_1:(rc_9), m_2:(rc_{10}), m_3:(rc_{11}), m_4:(rc_{12}), and m_5:(rc_{13}-rc_{14}). The files in this layer are truly independent modules. The $l2$-$l4$ contain 1, 2, and 1 module respectively.

In this paper, we collect SFPs by calculating the difference of SRs and IRRs in DSM. We generate SFU by searching the nearest common design rule for a SFP in DRH. For convenience, we also summarize all the terminologies in Table IV.

B. A Running Example

Problem Definition. A problematic file group (PFG) is a structurally-connected file group, where the contained files are frequently involved in bug fixes and change commits. It is defined as:

$$PFG = \{v_i, F, SR, Cost\,(v_i, v_n)\} \quad (1)$$

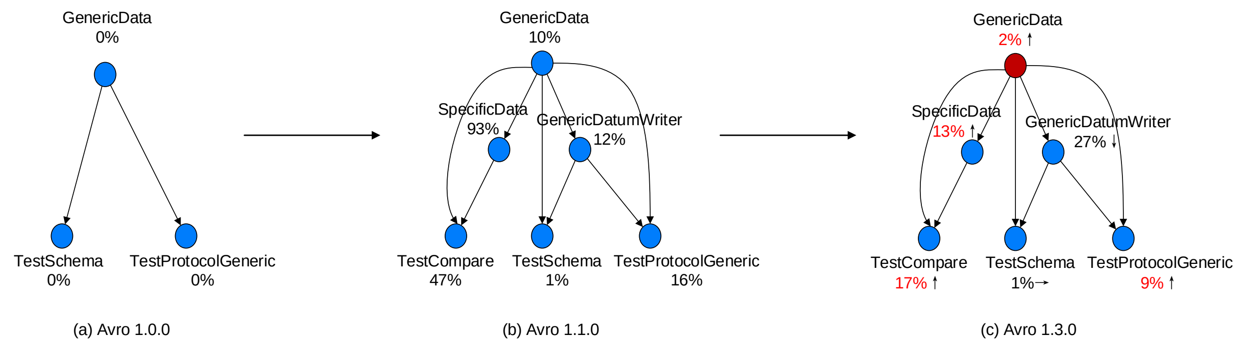


Fig. 2: The Evolution of A PFG (Each node represents a source file. Each edge represents a structural relation. The description behind node represents the name of a source file. The number behind node represents the bug ranking of a source file)

TABLE IV: Terminology Table

Abbreviation	Full Name
PFG	Problematic File Group
SFP	Suspect File Pair
SFU	Suspect File Unit
SFG	Suspect File Group
SR	Structural Relation
IRR	IR-based Relation
DSM	Design Structure Matrix
DRH	Design Structure Hierarchy

where v_i represents the detected version i. F represents the contained source files. *SR* represents structural relations among these involved source files, including inheritance, implementation, method call, field access, type reference, and instance creation. $Cost\,(v_i, v_n)$ represents, from version i to version n, the increased maintenance cost (bug frequency and churn) was spent on this group .

A PFG is not a co-change file group. In each related bug fix/change commit, at least one file in PFG is involved. A PFG is also not a collection of most bug-prone/change-prone files. It is connected with software structural relations. We use the PFG shown in Table I as a illustrated example:

Illustrated Example. The PFG shown in Table I contains six source files including *TestCompare*, *TestSchema*, *TestProtocolGeneric*, *SpecificData*, *GenericDatumWriter*, and *GenericData*. Figure 2.(a)-2.(c) demonstrate the evolution of the structure of this PFG. We can observe that, in *Avro* 1.0.0, the structure of this PFG is relatively simple. Only three of them are involved. However, in *Avro* 1.1.0, the PFG organizes contained six source files with a pyramid structure. This group is consuming maintenance efforts. In *Avro* 1.3.0, by tracing the bug report and revision history of this group, we discover that the average bug ranking of involved files increases 20%. More importantly, the total change spent on this file group increases almost 82%. We represent the structure of this PFG using DRH and DSM. We surprisingly find that all of involved files are decoupled from the same design rule: *GenericData* in DRH. Figure 3.(a)-3.(c) present the evolution of the design structure led by *GenericData* from 1.0.0 to 1.3.0. It is obvious that the PFG also has a structural impact on other files. According to the theory of code smell, this PFG is a typical flawed structure informally referred as “spaghetti code” or “big ball of mud”. The leading file of this structure: *GenericData* is gradually evolving into the ‘God interface’, propagating defects on more source files and consuming significant maintenance costs.

To early detect this PFG, we employ hotspot detection and our approach as follows:

Hotspot Detection. Mo et.al [6] proposed a suite of problematic patterns related to high error-proneness and change-proneness named hotspot. This PFG was detected by hotspot as unstable interface in 1.3.0, shown in Figure 3(c) marked with red color. According to the definition of unstable interface, if a file is structurally depended by many files and also changes with them frequently, this file and its subordinate files are considered to be an Unstable Interface. By default, For a file, if there are more than 10 files structurally depending on it ($dp > 10$) and more than 5 files change together with it more than 5 times ($cdp > 5$), hotspot will identify this file group as an Unstable Interface. According to the definition, the file group is identified by hotspot as Unstable Interface only when they have been revised for enough times. In this case, we also find that the PFG is detected until 1.3.0. However, at that time, it has accumulated significant maintenance costs.

The procedure of our approach is as follows:

Our Approach. First, we collect the suspect file pair (SFP) by calculating the difference of SRs and IRRs using DSM. In this case, there is no SFP in 1.0.0. On the contrary, both in 1.1.0 and 1.3.0, three instances of SFPs are found marked with blue background color shown in Figure 3(b) and 3(c), including (*SpecificData*, *GenericDatumWriter*), (*TestCompare*, *TestSchema*), and (*TestCompare*, *TestProtocolGeneric*).

Second, we obtain the suspect file unit (SFU) by searching the nearest common design rule of the SFP in DRH. In our paper, SFU is formally defined as a triple including a pair of suspect files (SFP) and their nearest common design rule. For example, in 1.1.0 shown in Figure 3(b), the nearest common design rule of the SFP (*TestCompare*, *TestSchema*) is *GenericDatumWriter*. Therefore, the SFU of them is (*GenericDatumWriter*: *TestSchema*, *TestCompare*). In 1.1.0 and 1.3.0, we both generate three instances of SFUs:

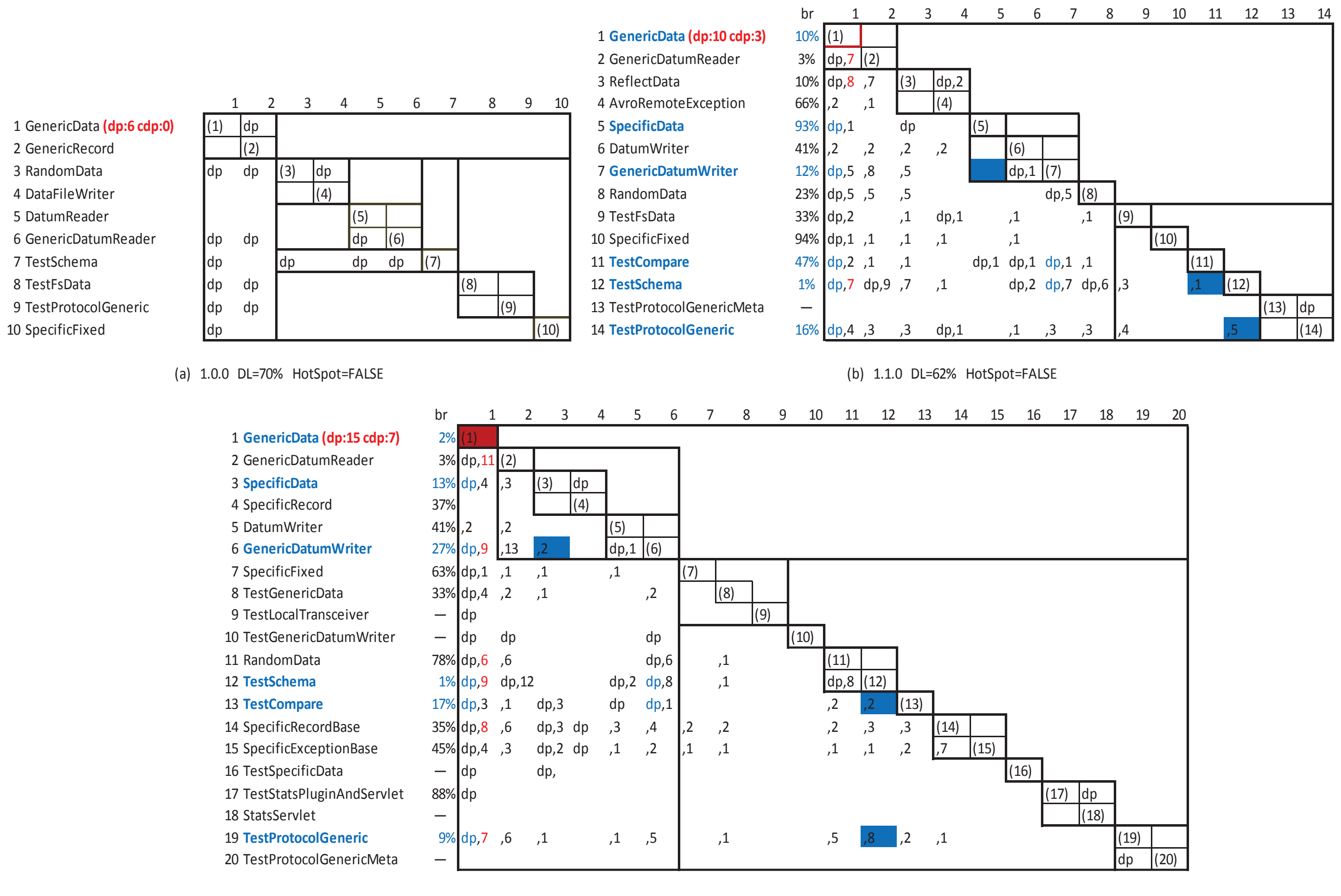

(a) 1.0.0 DL=70% HotSpot=FALSE (b) 1.1.0 DL=62% HotSpot=FALSE (c) 1.3.0 DL=74% HotSpot=TRUE

Fig. 3: A Running Example to Detect PFG

(*GenericDatumWriter*: *TestSchema*, *TestCompare*), (*GenericData*: *GenericDatumWriter*, *SpecificData*), and (*GenericData*: *TestProtocolGeneric*, *TestCompare*).

Finally, we recursively merge SFUs to generate suspect file groups (SFG) as PFG candidates. Both in 1.1.0 and 1.3.0, we merge the preceding three SFUs as a SFG marked with the light blue color including {*TestCompare*, *TestSchema*, *TestProtocolGeneric*, *SpecificData*, *GenericDatumWriter*, *GenericData*}. According to our algorithm, (*GenericData*: *GenericDatumWriter*, *SpecificData*) and (*GenericData*: *TestProtocolGeneric*, *TestCompare*) are first merged because they share the same design rule. Next, (*GenericDatumWriter*: *TestSchema*, *TestCompare*) is also merged into this group because the design rule: *GenericDatumWriter* is also the member of SFP of this group.

In summary, the detected SFG is exactly the fore-mentioned PFG. Compared with hotspot detection, our approach identifies this group in 1.1.0 as soon as it first emerges.

III. Methodology

Our approach takes extracted structural relations (SR) and IR-based relations (IRR) as the input, and follow three steps: collecting suspect file pairs (SFP), generating suspect file unit (SFU), and constructing suspect file group (SFG).

A. Extracting Structural and IR-based Relations

In this step, we obtain structural relations and IR-based relations as follows:

Extracting Structural Relations. Structural relation, derived from source code syntax, is the most common software relation for program comprehension and software maintenance. In our paper, we select six types of syntactic relations as structural relations including software inheritance, implementation, method call, field access, type reference and instance creation. We employ *Understand*, a commercial reverse-engineering tool, to extract these relations among files. For a subject, we denote all the collected structural relation as *SR*.

Extracting IR-based Relations. IR-based relation, derived from source code lexicon, is designed for exploiting the textual similarity among files using information retrieval techniques. In this paper, we extract IR-based relations using the following five steps:

1) Crawling lexical data. In this step, we crawl four kinds of source code lexicons as lexical features including class name,

method name (function and function parameter name), global variable name, and comment. Supported by our implemented lexical parser with Eclipse JDT[4], we collect all the lexical data as a $n \times 4$ matrix:

$$\begin{aligned} LeicalData &= \{LS_1, LS_2, LS_3, LS_4\} \\ &= \{LF_1, LF_2, \cdots, LF_n\}^{\mathrm{T}} \\ &= \left\{ \begin{array}{cccc} WS_1^1 & WS_1^2 & WS_1^3 & WS_1^4 \\ WS_2^1 & WS_2^2 & WS_2^3 & WS_2^4 \\ \cdots & \cdots & \cdots & \cdots \\ WS_n^1 & WS_n^2 & WS_n^3 & WS_n^4 \end{array} \right\} \end{aligned} \tag{2}$$

where n is the number of source files and each element W_i^k represents the crawled word set of $file_i$ on $feature_k$, which is defined as follows:

$$WS_i^k = \{w_1^k, w_2^k, \cdots, w_m^k\} \tag{3}$$

where w^k represents the extracted word and m represents the number of them. The i ranges from 1 to n. The k ranges from 1 to 4 and the $feature_k$ corresponds to class name, method name, global variable name, and comment respectively.

Based on the definition of lexical data, for each $file_i$, the collected lexical data: LF_i, is defined:

$$LF_i = \{WS_i^1, WS_i^2, WS_i^3, WS_i^4\} \tag{4}$$

We use *GenericDatumWriter*, one of files within the preceding PFG, as an example. Figure 4 presents part of its source code, which is iteratively parsed with our tool line by line shown in column type. The collected lexical data of *GenericDatumWriter* is demonstrated in the first row of Table V. Each cell corresponds to the WS_i where repeated words are expressed with brackets.

2) Preprocessing lexical data. In this step, we preprocess each word of the word set with filtering, decoupling, and stemming. The filtering process removes the 345 stop words for the natural language [13]. The decoupling process separates each word according to the naming convention, such as camel casing and snake casing [14]. The stemming process searches the root for each word using Porter algorithm [13]. Thus, for each WS_i^k, the preprocessed word set is defined as follows:

$$PWS_i^k = \{pw_1^k, pw_2^k, \cdots, pw_m^k\} \tag{5}$$

where pw represents the preprocessed word and m represents the number of words. Table V presents the preprocessing of lexical data of *GenericDatumWriter* step by step. For convenience, we mark the diff part between rows in bold. For instance, the "out(2)", in row: *Raw* and column: *Global Variable*, is marked in bold, indicating it will be filtered in the next row. The "writeRecord", in row: *Filtered* and column: *Global Variable*, is marked in bold, indicating it will be decoupled into write and Record in the next row. The "Record", in row: *Decoupled* and column: *Global Variable*, is marked in bold, indicating it will be stemmed in the next row.

[4]http://www.eclipse.org/jdt/

3) Generating lexical space using TF-IDF weighting. In this step, we generate feature space using one of the most popular information retrieval model: TF-IDF weighting [15]. Based on the definition of equation 1, for each $feature_k$, we generate its lexical space as LS_k:

$$LS_k = \{WS_1^k, WS_2^k, \cdots, WS_n^k\} \tag{6}$$

where n is the number of files. For each LS_k, we generate its feature space using TF-IDF as E^k, which is a $m \times n$ matrix (file-by-word). m represents the number of distinct words in feature k.

A generic entry $e_{i,j}^k$ of this matrix denotes the relevance of the i^{th} word in the j^{th} file.

$$e_{i,j}^k = \frac{t_{i,j}}{\sum_o t_{o,j}} \times \log \frac{n}{|\{j : pw_i \in file_j\}|} \tag{7}$$

where $t_{i,j}$ represents the occurrence frequency of word pw_i in $file_j$. $\sum_o t_{o,j}$ represents the occurrence frequency of word pw_i in all of files. $|\{j : pw_i \in file_j\}|$ represents the number of files containing pw_i. In summary, the $e_{i,j}^k$ represents the term frequency and inverse document frequency (*TF-IDF*) for the word: pw_i and the file: $file_j$.

4) Computing textual similarity. In this step, we compute textual similarity matrix as S_k for each lexical feature: LS_k based on the obtained TF-IDF matrix: E_k. The constructed textual similarity matrix is a $n \times n$ matrix (file-by-file) where n is the number of files. A generic entry $s_{i,j}^k$ of S^k denotes the cosine semantic similarity between i^{th} file and j^{th} file of E^k:

$$s_{i,j}^k = \frac{\sum_{l=1}^m e_{l,i}^k \times e_{l,j}^k}{\sqrt{\sum_{l=1}^m (e_{l,i}^k)^2} \times \sqrt{\sum_{l=1}^m (e_{l,j}^k)^2}} \tag{8}$$

where the $e_{l,i}^k$ and $e_{l,j}^k$ represent the entries of E^k.

5) Obtaining IR-based relations. In this step, we obtain IR-based relations by normalizing the textual similarity matrix and fusing them. We first normalize the S^k as $\overline{S}^k$. The entry of $\overline{s}_{i,j}^k$ in $\overline{S}^k$ is defined as:

$$\overline{s}_{i,j}^k = \begin{cases} 1, & s_{i,j}^k > \theta \\ 0, & other \end{cases} \tag{9}$$

where θ is empirically set as 0.8.

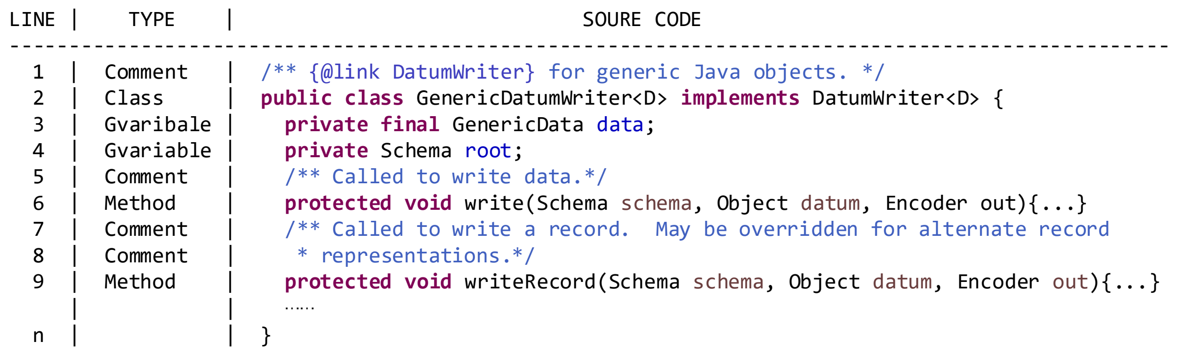
The normalized four similarity matrices are fused as *SIM*, which is defined as:

$$SIM = \sum_{k=1}^{4} \overline{S}^k \tag{10}$$

Based on the definition of *SIM*, we formally define the IR-based relation as *IRR*:

$$IRR = \{(file_i, file_j) \mid SIM_{i,j} > th\} \; (i, j \in \{1, 2, \cdots, n\}) \tag{11}$$

where th is set as 2. In IR-based relation: *IRR*, $(file_i, file_j)$ is equivalent to $(file_j, file_i)$.



```
LINE |    TYPE     |                              SOURE CODE
---------------------------------------------------------------------------------------------------
  1  |  Comment    |  /** {@link DatumWriter} for generic Java objects. */
  2  |  Class      |  public class GenericDatumWriter<D> implements DatumWriter<D> {
  3  |  Gvaribale  |    private final GenericData data;
  4  |  Gvariable  |    private Schema root;
  5  |  Comment    |    /** Called to write data.*/
  6  |  Method     |    protected void write(Schema schema, Object datum, Encoder out){...}
  7  |  Comment    |    /** Called to write a record.  May be overridden for alternate record
  8  |  Comment    |     * representations.*/
  9  |  Method     |    protected void writeRecord(Schema schema, Object datum, Encoder out){...}
     |             |    ……
  n  |             |  }
```

Fig. 4: Crawling Lexical Data of GenericDatumWriter (Gvarible is short for global variable name)

TABLE V: Preprocessing Lexical Data of GenericDatumWriter (The Diff is marked in Bold)

Steps	Class	Method	Global Variable	Comment
Raw	GenericDatumWriter	Data,root	write,schema(2),datum(2), **out**(2),writeRecord	Link,generic,Java,objects,data,overridden,alternate,representation, **a**,**may**,**be**,DatumWriter,**for**(2),**to**(2),called(2),write(2),record(2)
Filtered	**GenericDatumWriter**	Data,root	write,schema(2),datum(2), **writeRecord**	Link,generic,Java,objects,data,overridden,alternate,representation, **DatumWriter**,called(2),write(2),record(2)
Decoupled	**Generic**,**Datum**,**Writer**	**Data**,root	write(2),schema(2),datum(2), **Record**	**Link**,**generic**,**Java**,**objects**,data,**overridden**,**alternate**,**representation**, **Datum**,**Writer**,**called**(2),write(2),record(2)
Stemmed	generic,datum,write	data,root	write(2),schema(2),datum(2), record	link,gener,java,object,data,overrid,altern,repres,datum,call(2), record(2),write(3)

B. Collecting Suspect File Pairs (SFP)

SFP is defined as a pair of file: x and y. They are involved in IR-based relation but not structurally connected.

$$SFP = \{\ (x,y) \mid (x,y) \notin SR \wedge \ (x,y) \in IRR\} \qquad (12)$$

where SR represents the structural relation and IRR represents the IR-based relation.

For a subject, We map all the structural relation and IR-based relation in a DSM. By iteratively inspecting the cells of this DSM, all the SPFs are collected.

```
Algorithm 1: Merge SFU
Input: $SFU = \{(c_1 : a_1, b_1), \cdots, (c_n : a_n, b_n)\}$
$SFG \longleftarrow \{\}$
while $SFU \neq \emptyset$ do
    $(c_t : a_t, b_t) \longleftarrow$ selected and removed from $SFU$
    for $(z_k : x_k, y_k) \in SFG$ do
        # case1: same design rule
        if $c_t = z_k$ then
            Merge $(c_t : a_t, b_t)$ and $(z_k : x_k, y_k)$ like case1
        end
        # case2: correlated design rule and $SFP$
        if $a_t = z_k \vee b_t = z_k$ then
            Merge $(c_t : a_t, b_t)$ and $(z_k : x_k, y_k)$ like case2
        end
        # case3: not case1 or case2
        $SFG \longleftarrow SFG \cup (c_t : a_t, b_t)$
    end
end
return $SFG$
```

C. Generating Suspect File Units (SFU)

Although SFP can reveal maintenance problem, its involved files are actually structurally isolated. However, we aim at detecting problematic file group connected with structural relations (PFG). As a consequence, SFU is designed as the minimal structurally-connected unit to cover SFP.

We generate SFU by searching the nearest common design rule of SFP. The reasons are: 1) In DRH, the design rule and its subordinate files construct the minimal unit of design structure. 2) According to design rule theory, if the subordinate files are bug-prone, their common design rules are also likely to be bug-prone [2].

Therefore, a SFU is defined as a triple including a pair of suspect files (SFP) and their nearest common design rule:

$$SFU = \{(z : x, y) \mid z = dr(x, y) \ \wedge \ (x, y) \in SFP\} \qquad (13)$$

where dr represents the nearest common design rules of x and y in DRH.

We generate SFUs for all the obtained SFPs. For example, in Figure 3(c), the common design rules of SFP: (*TestSchema*, *TestCompare*) are *GenericDatumWriter* and *GenericData*. The nearest common design rule is *GenericDatumWriter*. Thus, the SFU is constructed as (*GenericDatumWriter*: *TestSchema*, *TestCompare*)

D. Constructing Suspect File Groups (SFG)

The generated SFUs actually are not isolated. Thus, we recursively merge SFUs to construct suspect file groups (SFG) considered as PFG candidates.

Given two SFUs, there are two typical cases to merge them:

Case 1: $(c_1 : a_1, b_1)$ and $(c_1 : a_2, b_2)$. If two *SFU*s share with same design rule, they should be merged like:

$$(c_1 : a_1, b_1) + (c_1 : a_2, b_2) => (c_1 : a_1, b_1, a_2, b_2) \tag{14}$$

For example, in Figure 3(c), the *SFU*s: (*GenericData*: *GenericDatumWriter*, *SpecificData*) and (*GenericData*: *TestProtocolGeneric*, *TestCompare*) should be merged as: (*GenericData*: *GenericDatumWriter*, *SpecificData*, *TestCompare*, *TestProtocolGeneric*).

Case 2: $(c_1 : c_2, c_3)$ and $(c_2 : a_2, b_2)$. If the design rule of a SFU participate in the *SFP* of another SSU, they should be merged by joining the design rule and SFP:

$$(c_1 : c_2, c_3) + (c_2 : a_2, b_2) => (c_1 : c_2, a_2, b_2, c_3) \tag{15}$$

For example, the *SFU*s: (*GenericData*: *GenericDatumWriter*, *SpecificData*) and (*GenericDatumWriter*: *TestSchema*, *TestCompare*) in Figure 3(c) should be merged as: (*GenericData*: *GenericDatumWriter*, *TestSchema*, *TestCompare*, *SpecificData*).

Algorithm I shows the procedure of merging SFUs. From Line 2 to line 15, we iteratively select the element from SFU and merge it into SFG. Line 3 selects $(c_t : a_t, b_t)$ from SFU. Line 4 iteratively selects $(z_t : x_t, y_t)$ from SFG. If $(c_t : a_t, b_t)$ and $(z_t : x_t, y_t)$ share the same design rule, it will be merged like case1 in line 7. If $(c_t : a_t, b_t)$ and $(z_t : x_t, y_t)$ are correlated with the connection of design rule and subordinate files, it will be merged like case2 in line 11. If it does follow neither case1 nor case2, $(c_t : a_t, b_t)$ will be directly put into SFG. Eventually, all the SFUs are merged into suspect file group (SFG) as PFG candidates.

IV. Evaluation

To evaluate the effectiveness of our approach, we investigated 838 versions of 15 Apache open source projects, including 33353 bug reports and 86690 revision commits, as our evaluation subjects shown in Table VI. These projects differ in domain, scale, and other characteristics. The bug reports and revision commits are extracted from their version control (Git) and issue-tracking systems (JIRA). We only study bug reports that have a resolution of Fixed. We extracted structural relations and IR-based relations in each version, and obtained 2230 SFPs and 461 SFGs in total. In this section, we investigate the following research questions:

RQ1: Whether the files identified in SFGs will co-change together?

RQ2: Whether the files identified in SFGs will incur high maintenance costs in the subsequent versions?

RQ3: What is the accuracy of SFGs to predict PFGs?

RQ4: How early can our approach discover PFGs?

A. The Co-Change of SFG

To answer **RQ1**, we iteratively analyzed all the detected SFGs, and explored whether the involved files change together in revision history. For a SFG, we measure it using two metrics: *RC* and *CO*. For a project including m commits: $\{C_1, C_2, \cdots, C_m\}$ and a detected file group: G,

RC measures the number of related commits in which files of G are involved. It is defined as follows:

$$RC(x) = \frac{|\{C_i \mid |C_i \cap G| \geq x\}\,(i = 1, 2, \cdots, m)\,|}{m} \tag{16}$$

where x is the threshold. Specifically, $RC(1)$ represents the number of commits involving at least one files of G. Similarly, $RC(2)$ represents the number of commits involving at least two files of G. $RC(2)$ denotes the number of commits where files in G change together.

OC measures the distribution of the overlapping of involved files in G for co-change commits. It is defined as follows:

$$OC(x) = \frac{\left\{C_i \mid \frac{|C_i \cap G|}{|G|} \geq x \wedge C_i \in RC(2)\right\}}{RC(2)} \tag{17}$$

where x represents the threshold. Specifically, in this paper, we select 10%, 30% and 50%. For example, $OC(10\%)$ represents the number of co-change commits where the overlapping of involved files in G is greater than 10%.

Table VII demonstrates the result of *RC* and *OC*. For each project, we calculate the average value of *RC* and *OC* with different thresholds for all detected SFGs.

We discovered that the detected SFG participates in a significant amount of revision commits (almost 7.0% of all the commits). However, most of these commits (80%) are involved only one file of SFG. Averagely, the co-change commit of SFG are only 1.4% of all commits. More importantly, merely a small part of SFG change together in these revision history. For all the co-change commits of a SFG, almost 50% of them are involving less than 10% of the files in SFG. Only 7.3% of them are involving more than 50% of the files in SFG. In summary, SFG may participate in a large amount of commits but they do not frequently and completely change together. In most cases, only a small part of SFG will co-change with each other.

B. The Maintenance Cost of SFG

To answer **RQ2**, we iteratively analyzed all the SFGs of each version, and explore whether the involved files will incur significant maintenance costs in subsequent versions. We design two metrics to measure the impact of SFGs on software maintenance: Future Bug Frequency (FBF) and Future Change Bug Frequency (FCF).

For a detected file group: G in version: v_i, FBF and FCF calculate the average bug/change frequency of each file involved in this group from version: v_i to version: v_n.

To measure FBF and FCF, we first collect the related bug fixing/change information as two matrices. Both of them are $m \times n$ (File-by-Version), where rows represent files in system and columns represent versions. An element: bf_{ji}/cf_{ji} represents the number of bug fixes/code changes involved in F_j in version v_i. For a software system containing n consecutive versions and m source files, Table VIII demonstrates the related bug fixing information as a matrix (*Bug Fixing Matrix*). Similarly, Table IX demonstrates the related code change information as a matrix (*Code Change Matrix*).

TABLE VI: The Basic Information of 15 Subjects

Subject	Version Range (Vers)[1]	Description	Iss[2]	Cmts[2]	Fs[3]	LOC[3]	SFPs[4]	SFGs[4]
Avro	1.0.0 to 1.8.2 (65)	Serialization system	574	1490	540	66K	31	7
Cassandra	0.4.1 to 3.9.0 (170)	Distributed database	4848	20850	2012	362k	67	15
Flume	1.3.0 to 1.7.0 (8)	Tool for collecting high throughput data	862	1599	624	81K	11	3
Hadoop	0.1.0 to 2.6.3 (105)	Tool for distributed big data processor	2320	13615	7197	1115K	240	48
Hbase	0.1.0 to 1.2.4 (71)	Distributed database for Hadoop	7647	12974	3334	1108K	222	45
Log4j	2.0.0 to 2.7.0 (43)	Java-based logging utility	772	8906	1630	122K	82	17
Mahout	0.1.0 to 0.9.0 (17)	Scalable machine learning libraries	878	3588	1215	109K	57	12
Mina	0.8.3 to 3.0.0 (49)	Network application framework	323	2400	319	23K	5	2
Openjpa	1.0.0 to 2.4.1 (25)	Java persistence project	1200	4798	4542	427K	91	18
Pdfbox	1.1.0 to 2.0.4 (23)	Library for manipulating PDF documents	1782	5814	1168	135K	19	4
Pig	0.1.0 to 0.16.0 (43)	Platform for analyzing big data	1843	2935	1617	251K	86	17
Tika	0.2.0 to 1.8.0 (37)	Content analyzer	682	3306	875	89K	73	15
Zookeeper	3.0.0 to 3.5.1 (57)	Tool for providing centralized services	639	1435	572	71K	31	6
Cxf	2.1.0 to 3.1.12 (62)	Service framework	3384	12538	6662	621K	89	18
Camel	1.1.0 to 2.9.8 (62)	Integration framework	6504	24163	13609	809K	87	17

[1] The number of version and its range
[2] The total number of bug issues and revision commits
[3] The average number of files and LOC of the project per version
[4] The average number of detected SFPs and SFGs of the project per version

TABLE VII: The Co-Change of SFG

Project	RC[1]		OC[2]		
	1	2	10%	30%	50%
Avro	1.6%	0.5%	50.0%	16.7%	6.7%
Cassandra	9.2%	1.6%	33.4%	3.1%	0.9%
Flume	3.9%	1.1%	100.0%	43.6%	17.2%
Hadoop	6.9%	1.7%	38.6%	24.2%	13.6%
Hbase	7.1%	1.7%	49.6%	11.9%	4.7%
Log4j	3.2%	0.4%	61.5%	17.6%	1.0%
Mahout	2.8%	1.2%	10.0%	0.0%	0.0%
Mina	4.6%	0.2%	100.0%	100.0%	0.0%
Openjpa	8.1%	1.2%	66.0%	31.7%	13.4%
Pdfbox	12.2%	1.1%	34.5%	33.3%	6.1%
Pig	12.6%	2.2%	38.5%	17.4%	6.5%
Tika	4.2%	0.3%	40.4%	31.4%	21.3%
Zookeeper	17.5%	5.8%	46.1%	9.3%	0.4%
Cxf	3.2%	0.5%	44.5%	22.2%	14.7%
Camel	7.3%	2.0%	47.3%	17.5%	3.1%
Avg	7.0%	1.4%	50.7%	25.3%	7.3%

[1] 1, 2 represent RC(1), RC(2)
[2] 10%, 30%, 50% represent OC(10%), OC(30%), OC(50%)

TABLE VIII: Bug Fixing Matrix

$$\begin{array}{c|ccccccc|}
 & v_1 & v_2 & v_3 & \cdots & v_i & \cdots & v_n \\ \hline
F_1 & bf_{11} & bf_{12} & bf_{13} & \cdots & bf_{1i} & \cdots & bf_{1n} \\
F_2 & bf_{21} & bf_{22} & bf_{23} & \cdots & bf_{2i} & \cdots & bf_{2n} \\
F_3 & bf_{31} & bf_{32} & bf_{33} & \cdots & bf_{3i} & \cdots & bf_{3n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
F_j & bf_{j1} & bf_{j2} & bf_{j3} & \cdots & bf_{ji} & \cdots & bf_{jn} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
F_m & bf_{m1} & bf_{m2} & bf_{m3} & \cdots & bf_{mi} & \cdots & bf_{mn} \\ \hline
\end{array}$$

TABLE IX: Code Change Matrix

$$\begin{array}{c|ccccccc|}
 & v_1 & v_2 & v_3 & \cdots & v_i & \cdots & v_n \\ \hline
F_1 & cf_{11} & cf_{12} & cf_{13} & \cdots & cf_{1i} & \cdots & cf_{1n} \\
F_2 & cf_{21} & cf_{22} & cf_{23} & \cdots & cf_{2i} & \cdots & cf_{2n} \\
F_3 & cf_{31} & cf_{32} & cf_{33} & \cdots & cf_{3i} & \cdots & cf_{3n} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
F_j & cf_{j1} & cf_{j2} & cf_{j3} & \cdots & cf_{ji} & \cdots & cf_{jn} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
F_m & cf_{m1} & cf_{m2} & cf_{m3} & \cdots & cf_{mi} & \cdots & cf_{mn} \\ \hline
\end{array}$$

Thus, for a detected file group: G in version: v_i, the FBF: $FBF(G, v_i)$ and FCF: $FCF(G, v_i)$ are defined as follows:

$$FBF(G, v_i) = \frac{\sum_{F_j \in G} \sum_{k=i+1}^{n} bf_{jk}}{|G|} \tag{18}$$

$$FCF(G, v_i) = \frac{\sum_{F_j \in G} \sum_{k=i+1}^{n} cf_{jk}}{|G|} \tag{19}$$

where $\sum_{k=i+1}^{n} bf_{jk}/\sum_{k=i+1}^{n} cf_{jk}$ represents the total number of bug fixes/changes involved in F_j from v_i to v_n. $FBF(G, v_i)/FCF(G, v_i)$ represents the average number of bug fixes/code changes of each file in file group:G from v_i to v_n.

For 838 versions of these 15 projects, in each version: v_i, we collect SFGs and measure its FBF and FCF from v_i to v_n (v_n represents the latest version of subject). For comparison, we also measure the FBF and FCF of all the files (AFG) and all the bug-prone files (BFG) in each version.

The results are demonstrated in Figure 5. Due to limit space, we only exhibit the measurement of eight projects and the full data is also available [16]. In each figure, x-axis presents the version and y-axis presents the FBF/FCF. The red line represents the FBF/FCF of SFG, the blue line represents the FBF/FCF of BFG and the black line represents the FBF/FCF of AFG. Some points on SFG/BFG are also highlighted. The percentage marked with red/blue color demonstrates the increase rate of SFG/BFG compared with AFG.

We observed that, in these 838 versions of 15 projects, the average FBF of SFG is 6.7, which is 957% higher than the value of AFG (0.7). The average FCF of SFG (FCF_{SFG}) is 8.4, which is also 1050% higher than the value of AFG (0.8). Meanwhile, the average FBF of BFG is 4.0, which is 571% higher than AFG. The average FCF of BFG is 6.2, which is 775% higher than AFG. According to our observations, files within SFGs do incur much higher maintenance costs, including bug fixing and code change, in subsequent versions. Compared with AFG, SFG presents enough high bug rate. Compared with BFG, SFG concentrate on locating the most

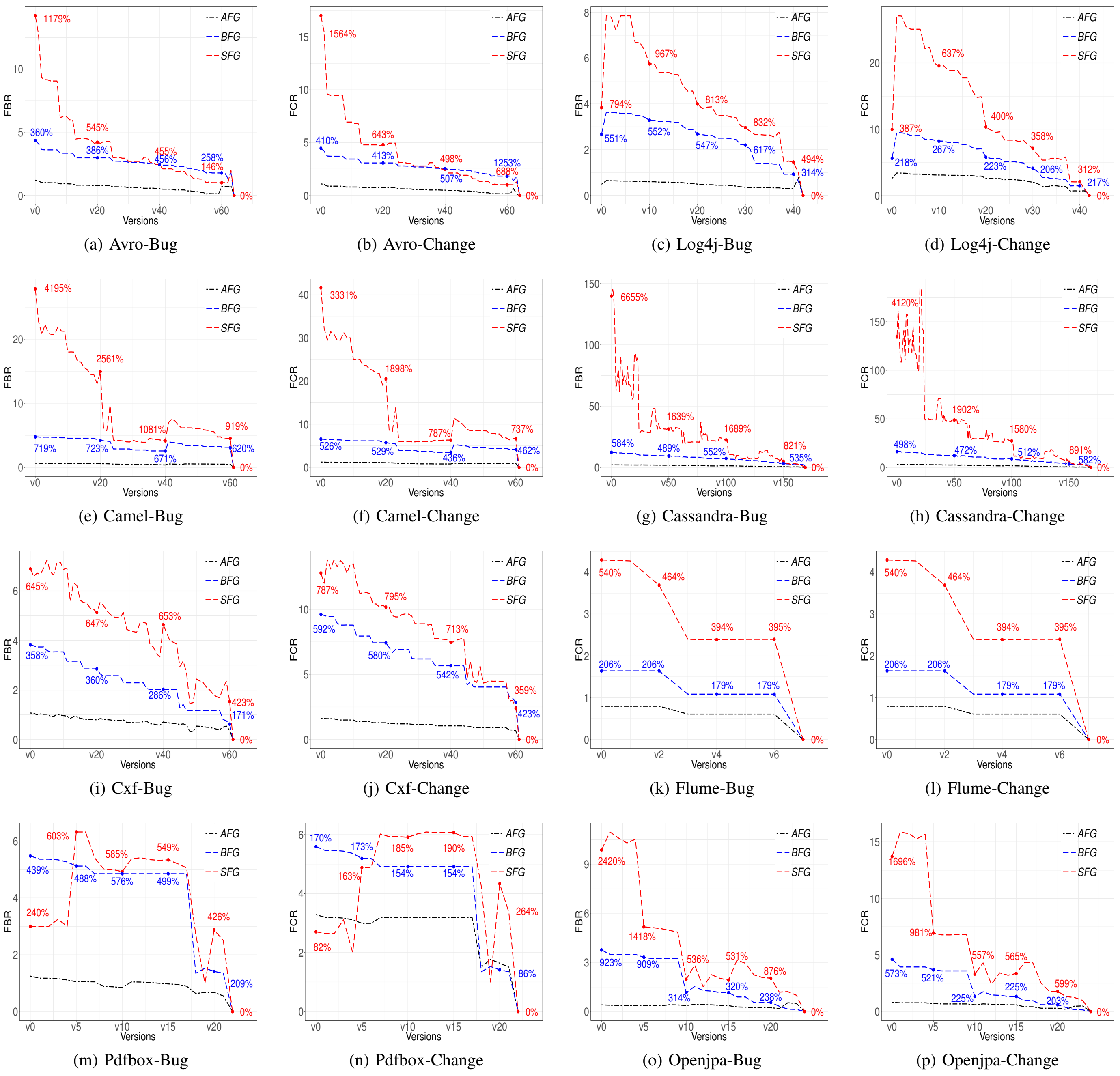


Fig. 5: The Future Bug/Change Rate of SFG (*AFG* represents all the files. *BFG* represents all the bug-prone files involved in bug fixing at least once. *SFG* represents our detected file)

bug-prone file group.

In particular, the FBF/FCF of SFGs shows best results in two subjects: Cassandra (with largest number of versions, 170 versions) and Camel (with largest number of revisions: 27547 bug and change commits). In Cassandra, the average FBF of SFG is as high as 27.4. More importantly, the average FCF of SFG is nearly 40.5. In Camel, the average FBF of SFG is 9.8 and the average FC is 14.1. The results indicate that files within SFG of these two projects incur extremely frequent and repeated bug fixes and changes.

Obviously, all the curves of SFG in Figure 5 is decreasing with the increase of versions. It is clear that, the decrease is caused by the definition of FBF and FCF. In the last version of each project, the FBF and FCF are all equal to 0 for the lack of subsequent versions. Additionally, we also discovered that, some file groups reported by SFG are not frequent enough in bug fixes and changes as we consider. Actually, these files may contain potential bugs and will cause maintenance costs

TABLE X: Four File Rankings

ID	Label	Name	Description
1	FR(1,x%)	$BF_{x\%}$	$BugFrequency_{x\%}$
2	FR(2,x%)	$BC_{x\%}$	$BugChurn_{x\%}$
3	FR(3,x%)	$CF_{x\%}$	$ChangeFrequency_{x\%}$
4	FR(4,x%)	$CC_{x\%}$	$ChangeChurn_{x\%}$

later than our observed versions. We are still tracking these issues now. Moreover, there are 9 versions in which the FBF of SFG is lower than AFG. Meanwhile, there are also 7 versions in which the FCF of SFG is lower than AFG. We manually inspect files within the detected group and find that most of them are actually unstable. Some files are even directly removed in later versions. As a consequence, the FBF/FCF decreases because these removed files do not participate in subsequent revision commits anymore.

In summary, files within SFG do incur frequent bug fixes and code changes. The identified SFGs can be considered as practical PFG candidates.

C. The Accuracy of SFG

To answer **RQ3**, we evaluate the accuracy of SFGs from four perspectives because there is no specific ground truth for PFG. These four perspectives are: (1) BugFrequency: the number of times involved in bug commits. (2) BugChurn: the number of lines of code to fix bugs. (3) ChangeFrequency: the number of times involved in change commits. (4) ChangeChurn: the number of lines of changed code. All of them are derived from collected bug reports and change commits.

In total, for each project, we generate four file rankings using the preceding four measures including *BF*, *BC*, *CF* and *CC* shown in Table X. For each ranking list, we also capture its top 10%, 30% and 100%. For short, we use $FR(i, x\%)$ to represent the top $x\%$ of the file ranking according to the measure i. For example, $FR(1, 30\%)$ represents the top 30% of bug-prone files ranked by bug frequencies: $BugFrequency_{30\%}$.

For a project of n versions, we obtain SFGs in each version: $\{SFG_1, SFG_1, \cdots, SFG_n\}$. Thus, the file group detected in SFG is defined as SFG_{all}:

$$SFG_{all} = \{SFG_1 \cup SFG_1 \cup \cdots \cup SFG_n\} \tag{20}$$

We calculate the precision and coverage of SFG_{all}. First, we calculate the precision of SFG_{all} as *BugRate*:

$$BugRate = \frac{SFG_{all} \cap FR\,(1, 100\%)}{SFG_{all}} \tag{21}$$

where $FR\,(1, 100\%)$ represents all the bug-prone files in project.

Then we calculate the coverage of SFG_{all} as $Cov(i, x\%)$:

$$Cov(i, x\%) = \frac{SFG_{all} \cap FR\,(i, x\%)}{FR\,(i, x\%)} \tag{22}$$

where $FR\,(i, x\%)$ represents the top $x\%$ of the file ranking using measure i. To calculate the coverage rate of a SFG, we exhaustively explored all the possible combinations and obtain 12 coverage values. For example, $Cov(1, 10\%)$ indicates the percentage of the top 10% of bug-prone files in terms of bug frequency: $BugFrequency_{10\%}$, covered by SFGs.

Table XI shows the precision and coverage metrics of these projects. Based on the collected data, we make the following observations:

1) The file identified in SFGs is limited (3%-32% of the system). The average proportion of the file within SFGs is only 14%. The identification results is easy for developers to follow and focus.

2) The average bug rate of SFGs is as high as 92% in all the projects. The highest bug rate of SFGs is 100% in flume. The lowest bug rate of SFGs is 77% in Mina. The true positive rate is high enough to guide developer to locate problematic file groups.

3) SFGs participate in a significant percentage (61%-68%) of the most bug-prone and change-prone files (top10%). The coverage rate of the top 10% bug-prone files in terms of bug frequency is 68%. The coverage rate of the top 10% bug-prone files in terms of bug churn is 63%. Similar results can also be observed in change-proneness, the coverage rate of the top 10% change-prone files in terms of change frequency is 68%. The coverage rate of the top 10% change-prone files in terms of change churn is 61%. Thus, our method can discover problematic file groups with high maintenance costs.

4) SFGs only identify a small portion (37% and 35%) of all the bug-prone or change-prone file sets. The coverage rate of all the bug-prone files is 37%. The coverage rate of all the change-prone files is 35%. Actually, SFG is not design to cover all the problematic files, such as the work of defect prediction. Our approach identifies structurally-connected file groups frequently contributing to the bug-proneness and change-proneness that deserves special attention.

Taking *AVRO* as an example, our approach averagely identified 7 SFGs over 65 versions of this project, including 91 distinct files in total. There are 19 and 18 files covering the top 10% bug-prone files in terms of bug frequency and bug churn. The coverage rate is 82% and 71%. Meanwhile, There are 19 and 18 files covering the top 10% change-prone files in terms of bug frequency and bug churn. The coverage rate is 82% and 71%. The bug rate of SFGs is as high as 94%. Our approach do identify file groups consuming expensive maintenance costs.

In summary, the result shows that our method presents a high accuracy in discovering problematic file groups.

D. The Timeliness of SFG

To answer **RQ4**, we employ our method in a specific project of multiple versions. By conducting fine-grained analysis of the detection results, we illustrate the timeliness of our approach. Flume is selected as our subject.

Flume is an Apache open source project for processing high throughput data. We intensively analyzed its eight versions from 1.2.0 (released on July 26, 2012) to 1.7.0 (released on October 17, 2016). There are 624 files and 81k lines of code in each version on average. In total, this project consumes 25944 lines of code to fix 862 bugs.

TABLE XI: The Top Frequency and Churn Coverage in Bug and Change

Subjects	FS	Precision	Recall											
		BugRate[1]	BugFrequency[2]			ChangeFrequency[2]			BugChurn[3]			ChangeChurn[3]		
			10%	30%	100%	10%	30%	100%	10%	30%	100%	10%	30%	100%
Avro	14%	0.94	0.82	0.61	0.47	0.82	0.59	0.47	0.71	0.62	0.47	0.71	0.63	0.47
Cassandra	12%	0.98	0.55	0.27	0.18	0.55	0.27	0.17	0.50	0.34	0.18	0.48	0.33	0.17
Flume	24%	1.00	0.44	0.30	0.28	0.44	0.28	0.21	0.55	0.40	0.28	0.54	0.39	0.21
Hadoop	32%	0.65	0.64	0.53	0.46	0.64	0.52	0.39	0.53	0.50	0.46	0.46	0.47	0.39
Hbase	19%	0.99	0.93	0.77	0.61	0.91	0.75	0.60	0.77	0.75	0.61	0.78	0.74	0.60
Log4j	4%	0.94	0.78	0.61	0.48	0.80	0.60	0.46	0.69	0.63	0.48	0.66	0.59	0.46
Mahout	3%	0.98	0.74	0.49	0.36	0.75	0.50	0.36	0.55	0.46	0.36	0.51	0.45	0.36
Mina	23%	0.77	0.72	0.42	0.31	0.67	0.35	0.26	0.58	0.44	0.31	0.55	0.41	0.26
Openjpa	3%	0.91	0.65	0.38	0.27	0.72	0.40	0.27	0.71	0.50	0.27	0.66	0.42	0.27
Pdfbox	9%	0.97	0.36	0.20	0.13	0.36	0.20	0.13	0.38	0.26	0.13	0.38	0.24	0.13
Pig	5%	0.97	0.79	0.62	0.48	0.79	0.62	0.48	0.80	0.69	0.48	0.80	0.68	0.48
Tika	25%	0.96	0.90	0.68	0.53	0.91	0.68	0.52	0.81	0.73	0.53	0.80	0.70	0.52
Zookeeper	19%	0.99	0.74	0.52	0.37	0.72	0.51	0.37	0.66	0.53	0.37	0.64	0.53	0.37
Cxf	7%	0.98	0.60	0.38	0.31	0.62	0.40	0.29	0.61	0.48	0.31	0.60	0.47	0.29
Camel	13%	0.80	0.56	0.37	0.25	0.57	0.39	0.28	0.56	0.46	0.25	0.54	0.47	0.28
Avg	14%	0.92	0.68	0.48	0.37	0.68	0.47	0.35	0.63	0.52	0.37	0.61	0.50	0.35

[1] The bug rate of bug-prone files in the identified file group
[2] The coverage of top problematic files (10%, 50% and 100%) according to the bug or change frequency
[3] The coverage of top problematic files (10%, 50% and 100%) according to the lines of code to fix bug or make change

TABLE XII: Early Detection of PFG in Flume

Flume		PFG[1]											
		Context (8)			Configurable (11)			Event (20)			Sink (9)		
Version	Date	BC[2]	SFG	Hotspot	BC[2]	SFG	Hotspot	BC[2]	SFG	Hotspot	BC[2]	SFG	Hotspot
1.3.0	Dec 2, 2012	–	✓	×	–	✓	×	–	×	×	–	×	×
1.3.1	Jan 2, 2013	340	✓	×	570	✓	×	–	×	×	–	×	×
1.4.0	July 2, 2013	875	✓	×	641	✓	×	–	×	×	–	×	×
1.5.0	May 20, 2014	944	✓	×	392	✓	×	–	×	×	–	×	×
1.5.1	July 16, 2014	745	✓	×	0	×	×	130	✓	×	–	×	×
1.5.2	Nov 18, 2014	655	✓	✓	0	×	×	267	✓	×	–	×	×
1.6.0	May 20, 2015	660	✓	✓	0	×	×	0	✓	×	–	✓	×
1.7.0	Oct 17, 2016	55	✓	✓	248	✓	✓	915	✓	×	819	✓	×
Saving Cost[3]		4274	100%	16%	1851	86%	0	1312	90%	0	819	100%	0

[1] The problem file group. Each element includes the leading file and number of files in this group
[2] The bug churn, the number of lines of code spent to fix bugs.
[3] The saving maintenace cost. Each element includes the total bug churn in this group, the percentage of saving lines of code using our approach and hotspot.

We detect PFG using Mo et.al's tool [4] and our method in Flume of 8 versions. From 1.3.0 to 1.7.0, we compare the results of Hotspot and SFGs in each version. In total, there are four identified PFG shown in Table XII:

Two PFGs have been detected both by Hotspot and SFG, which are led by *Context* and *Configurable*. These two groups include 8 and 11 files. The revision history records that 4274 and 1851 lines of code have been spent to fix bugs in these two groups, which are 17% and 7% of the entire maintenance cost.

The PFG of *Context* is a long-lived problematic group, contributing to significant maintenance costs from 1.3.0 to 1.7.0. Hotspot identifies this group in 1.5.2 while our method identifies in 1.3.0. The PFG of *Configurable* also causes maintenance problem from 1.3.0. Some files in this group are removed in 1.5.1 for refactoring but they are soon introduced again in 1.7.0. Hotspot identifies this file group only in 1.7.0. On the contrary, our method identifies this group from 1.3.0 to 1.5.0. In 1.7.0, our method also identifies this group again. In total, our method present 5 and 7 versions earlier than hotspot detection.

The other two PFGs are detected by our method but not hotspot. These two groups are led by *Event* and *Sink*, including 8 and 11 files. The revision history records that they consume 1312 and 819 lines of code to fix bugs, which are 5% and 3% of all the maintenance cost. The PFG of *Event* and *Sink* are both generated for introducing new features in 1.5.1 and 1.7.0. Our method do identify them in 1.5.1 and 1.7.0. On the contrary, they are not detected by Hotspot.

We assume that the maintenance costs of a file group can be saved if this group is discovered in the previous version. Based on this hypothesis, we observed that the detection of these four file groups using our method would save as many as 6740 lines of code to fix bugs, taking up 81% of all the maintenance costs in this group. However, the hotspot only saves 683 lines of code, taking 8% of all the maintenance costs.

In summary, the results show that our method can detect the PFGs as soon as they were firstly introduced, which is much earlier than hotspot detection. Our method would help developers save large number of lines of code to fix bugs by early detecting these problematic file groups. Moreover, our method can also detect some PFG missed by Mo et.al's tool. This is due to the fact that their tool requires enough long history. In contrast, our method do not rely on history.

V. Discussion

In this section, we discuss the rationale of our approach and the threats to validity.

A. The Root Cause of SFP

Based on our observations, we discover that the SFP is a typical pattern to hint maintenance problem. It seems plausible that high code lexicon similarity incurs repeated bugs and change. However, we are still not clear why it works. To understand the root cause of SFP, we manually inspect 300 file pairs and summarize four reasons:

1) **Code Clone.** One common reason of SFP is code clone. There exist several similar code fragments between two files in code clone. when the code in one file is modified, the other file also need similar change to resynchronize the clone part. For example, in Cassandra 3.6.0, we detect almost 40 lines of clone fragments between *DateTieredCompactionStrategy* and *TimeWindowCompactionStrategy*. These two file are also found the repeated change in the subsequent versions. It suggests the clone part should be extracted as the common interface or utility method in time during evolution.

2) **Poor Inheritance.** Poor inheritance is also a significant factor for SFP. There always exist more than two subclasses redundantly extended from one base class. Thus, these subclasses are incurring similar bugs and changes for overload inheritance. For example, in Log4j 1.3.0, *LiteralPatternConverter* and *NamedPatternConverter* are two subclasses from the base class: *PatternConverter*. However, these two files share more than half of the inherited variables and methods. It suggests that the base class should be decoupled into a few simple interfaces. Thus, the subclasses can flexibly implement them. In Log4j 1.3.8, *LiteralPatternConverter* and *NamedPatternConverter* are refactored by implementing a lightweight interface: LoggingEventPatternConverter.

3) **Implicit Relation.** e.g. reflection, multi-thread, and etc. Implicit relation is a reason for SFP. There exist some hidden relations not captured by our tool. As a consequence, some identified SFPs are actually connected with these implicit relations because they are not detected by our tool. Since these relations are unconscious to discover, it causes underlying maintenance problems. For example, in PDFBox 2.0.1, *XMPBasicSchema* and *XMPSchemaFactory* have the relevance of reflectionm, which is not detected by our tool *Understand*. It suggests these files should be highlighted in the comments.

4) **Developer Preference.** Developer preference is a human reason for SFP. SFPs in this category are neither code clone nor implicitly related. The involved files always have a higher complexity. In the modification requests and comments, these files are discussed together by developers. It seems that these file pairs have the similar function. From the view of program semantic, they should be refactored together in time. For example, in Hadoop 0.4.0, *DataNodeInfor* and *DataNodeReport* are a SFP which are often discussed by developers in comments. As a consequence, they are also soon to be reconstructed.

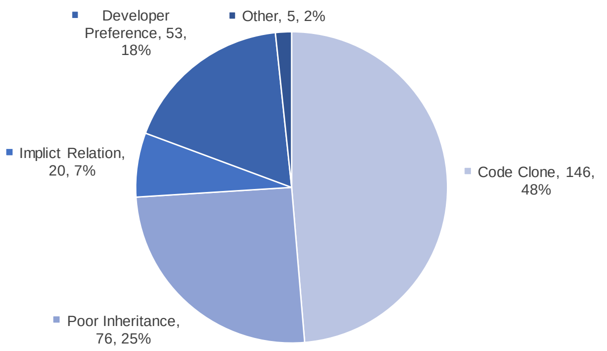


Fig. 6: Four reasons for SFP (Each reason contains three items: the name, the number of instances and its percentage)

In summary, we manually inspect 300 instances of SFPs and summarize four reasons for explaining the rationale. The classified results are demonstrated in Figure 6. We observed that code clone is the most common reason (48.7%). A large amount of these SFPs are introduced by the poor inheritance (25.3%) and developer preference (10.6%). Only 6.7% of them are caused by the implicit relation. By employing the summarized four reasons, we can explain 98.3% of the sampled SFPs.

B. Limitations and Threats to Validity

First, we conducted a large-scale study of PFG candidates detection in more than 800 versions of the 15 open source projects. Because all of selected subjects are Apache projects, it is unclear whether our approach will generalize to other open projects in difference community or closed industrial projects. In order to minimize this bias as much as possible, we choose projects of different sizes and domains. In our future work, we will apply our approach to more projects.

Second, it is difficult to define what is a problematic file. We evaluate it from the maintenance cost including bugs and changes spent during evolution. In our paper, we consider that if a file is frequently involved in bug fixes, frequently introduced new features or updated, it has a high possibility to be problematic. Additionally, it is also challenging for evaluating whether our method can identify PFGs in future versions. Thus, we select long-lived projects with adequate revision commits and bug reports. For example, in Cassandra, we collect 170 versions, more than 120k bug reports and change commits in total. We are also employing our approach in some young projects and reporting detected issues to the community.

Third, our approach might be sensitive to the selection of thresholds. We employ two thresholds in extracting IR-based relations. These thresholds are empirically set based on the work of Bavota et.al [17]. We also manually inspect the obtained relations to ensure its quality. The full results about thresholds in our experiments are available [16]. However, the best thresholds for various projects may be different.

Therefore, automatically determining the best thresholds is also part of our future work.

VI. Related Work

Coupling Metrics: Software coupling was first introduced by Steve et.al [18]. Based on the concept of software coupling, Chidamber and Kemerer proposed a suite of coupling metrics to evaluate the quality of object-oriented programs [19]. Briand et.al [20] also employed coupling metrics to inform the software high-level design. There are rich literature in various coupling metrics from multiple prospectives including software syntactic [19], revision history [21, 22], source code lexicon [23], and dynamic execution [24]. These coupling metrics are designed to capture various aspects of software quality. Our work exploits the combination of structural coupling metric and IR-based coupling metric to inform software quality in time.

Defect Prediction: Over the past decades, a plethora of research efforts focus on helping practitioners predict defects [25–30]. For example, Selby and Basili [31] employed dependency structure to improve the accuracy of predicting defects. Nagappan et.al [32] made a comparison of complexity metrics and leveraged them in defect prediction. Qu et.al [33] studied the structure of method invocation and their impact on defect prediction. Tan et.al [34] predict defects by considering the deep learning algorithm. All of these works focus on improving the prediction of individual bug-prone files. However, our study focuses on detecting a group of structurally-connected files.

Software Textual Analysis: Software textual analysis is widely used in refactoring [35], reverse-engineering [36, 37], bug localization [38, 39], and code search [40, 41]. By extracting textual features of the source file, software textual analysis gain new insight into software projects. For instance, Lo et.al [38] leverage textural features in locating the relevant files for a bug report. Zhang et.al [40] employ software textural analysis to improve the efficiency of code search. In our work, we derive textual features from source code lexicon and measure its similarity using information retrieval techniques. We discover the issues by combing the structural analysis and textual analysis.

Code Smell: The concept of code smell, also known as code anomaly, was first introduced by Fowler et.al [42]. Code smell is also a effective method to study problematic file group [43–49]. For example, Macia et.al [43] studied problematic file group by extending the definition of code smell in architectural level. Oizumi et.al [49] studied problematic file group by clustering the classic code smells. In this paper, our work also has the intersection of some classical code smells such as code clone. We find part of the results can also be detected by the clone detection tool: CCFinderx [50] and CloneDetection [51]. However, most of our results can not be identified by state-of-the-art code smell detection tool such as Sonar[5] and JDeodorant[6].

[5]https://www.sonarqube.org/
[6]http://www.jdeodorant.com/

VII. Conclusion

In this paper, we proposed a method by combining structural relations (SR) and IR-based relations (IRR) to identify problematic file groups (PFG) at early stage in software evolution. We evaluated our approach using 838 versions of 15 Apache open source projects, involving 33353 bug reports and 86690 revision commits. The results suggest that our method can discover the PFG candidates effectively and timely. The detected file groups incurred 957% bug frequencies and 1050% change frequencies than average in the subsequent versions. We also find that our method merely use 14% of the system files to cover 70% of the top 10% bug-prone files with a high precision (92%). These observations indicate the file groups, detected by our approach, are shown to have the potential risk in causing maintenance problem. By early detecting and refactoring them in time, Our method can assist architects and developers to save significant maintenance costs.

Acknowledgement

This work was supported by National Science Foundation of China (61532004, 61532015, 61472318, 61672419, 61602369, 61702414), Ministry of Education Innovation Research Team (IRT13035), Key Projects in National Science and Technology Pillar Program (2013BAK09B01), Natural Science Basic Research Program of Shaanxi (2018JQ6078) and the Fundamental Research Funds for the Central Universities.

References

[1] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 179–188.

[2] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 967–977.

[3] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 488–498.

[4] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 499–510.

[5] C. Y. Baldwin and K. B. Clark, "Design rules, vol. 1: The power of modularity," vol. 1, 2000.

[6] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. IEEE, 2015, pp. 51–60.

[7] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm Acm*, vol. 15, no. 3, pp. 1–50, 1972.

[8] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," *Acm Sigsoft Software Engineering Notes*, vol. 26, no. 5, pp. 99–108, 2001.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of reusable object-oriented software," *Addison-Wesley Professional Computing Series, Reading, Mass.: Addison-Wesley, —c1995*, vol. 49, no. 2, p. 241–276, 1995.

[10] Y. Cai and K. J. Sullivan, "Modularity analysis of logical design models," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 91–102.

[11] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 197–208.

[12] Y. Cai, H. Wang, S. Wong, and L. Wang, "Leveraging design rules to improve software architecture recovery," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. ACM, 2013, pp. 133–142.

[13] N. Kambhatla, "Combining lexical, syntactic, and semantic features with maximum entropy models for extracting relations," in *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, 2004, p. 22.

[14] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 2009, pp. 158–167.

[15] M. Yamamoto and K. W. Church, "Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus," *Computational Linguistics*, vol. 27, no. 1, pp. 1–30, 2001.

[16] https://github.com/cuidi34/ToRData/.

[17] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[18] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *Ibm Systems Journal*, vol. 38, no. 2.3, pp. 231–256, 1974.

[19] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 11, pp. 197–211, 1994.

[20] L. C. Briand, S. Morasca, and V. R. Basili, *Defining and Validating Measures for Object-Based High-Level Design*. IEEE Press, 1999.

[21] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 190–198.

[22] S. Wong and Y. Cai, "Generalizing evolutionary coupling with stochastic dependencies," in *Ieee/acm International Conference on Automated Software Engineering*, 2011, pp. 293–302.

[23] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *IEEE International Conference on Software Maintenance*, 2006, pp. 469–478.

[24] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, *Predicting the Probability of Change in Object-Oriented Systems*. IEEE Press, 2005.

[25] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[26] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *International Conference on Software Engineering*, 2005, pp. 284–292.

[27] A. E. Hassan, "Predicting faults using the complexity of code changes," *Proc.intl Conf.on Softw.eng*, pp. 78–88, 2009.

[28] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: a study of breakage and surprise defects," in *ACM Sigsoft Symposium and the European Conference on Foundations of Software Engineering*, 2011, pp. 300–310.

[29] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding the impact of code and process metrics on post-release defects:a case study on the eclipse project," in *Acm-Ieee International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 1–10.

[30] T. H. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan, "Explaining software defects using topic models," in *IEEE Working Conference on Mining Software Repositories*, 2012, pp. 189–198.

[31] R. W. Selby and V. R. Basili, "Analyzing error-prone system structure," *IEEE Transactions on Software Engineering*, vol. 17, no. 2, pp. 141–152, 1991.

[32] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

[33] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *Journal of Systems and Software*, vol. 108, pp. 193–210, 2015.

[34] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Ieee/acm International Conference on Software Engineering*, 2017, pp. 297–308.

[35] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Software re-modularization based on structural and semantic metrics," in *Working Conference on Reverse Engineering*, 2010, pp. 195–204.

[36] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic,

and Y. Cai, “Enhancing architectural recovery using concerns,” in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 552–555.

[37] J. Garcia, I. Ivkovic, and N. Medvidovic, “A comparative analysis of software architecture recovery techniques,” in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 486–496.

[38] T. D. B. Le, R. J. Oentaryo, and D. Lo, “Information retrieval and spectrum based bug localization: better together,” in *Joint Meeting on Foundations of Software Engineering*, 2015, pp. 579–590.

[39] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports,” in *International Conference on Software Engineering*, 2012, pp. 14–24.

[40] F. Lv, H. Zhang, J. G. Lou, S. Wang, D. Zhang, and J. Zhao, “Codehow: Effective code search based on api understanding and extended boolean model (e),” in *Ieee/acm International Conference on Automated Software Engineering*, 2015, pp. 260–270.

[41] S. P. Reiss, “Semantics-based code search,” in *International Conference on Software Engineering*, 2009, pp. 243–253.

[42] M. Fowler, *Refactoring: improving the design of existing code*. Springer Berlin Heidelberg, 2002.

[43] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. V. Staa, “On the relevance of code anomalies for identifying architecture degradation symptoms,” in *European Conference on Software Maintenance and Reengineering*, 2012, pp. 277–286.

[44] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. V. Staa, “Are automatically-detected code anomalies relevant to architectural modularity?:an exploratory analysis of evolving systems,” in *International Conference on Aspect-Oriented Software Development*, 2012, pp. 167–178.

[45] A. Von Staa, A. Garcia, E. Cirilo, I. Macia, and R. Arcoverde, “Supporting the identification of architecturally-relevant code anomalies,” in *IEEE International Conference on Software Maintenance*, 2012, pp. 662–665.

[46] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, “Comparing four approaches for technical debt identification,” *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.

[47] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, “Identifying architectural bad smells,” in *Software Maintenance and Reengineering, 2009. CSMR’09. 13th European Conference on*. IEEE, 2009, pp. 255–258.

[48] W. Oizumi, A. Garcia, M. Ferreira, and A. Von Staa, “When code-anomaly agglomerations represent architectural problems? an exploratory study,” in *Software Engineering*, 2014, pp. 91–100.

[49] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, “Code anomalies flock together: exploring code anomaly agglomerations for locating design problems,” in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 440–451.

[50] “Ccfinder: A multi-linguistic token based code clone detection system for large scale source code,” in *IEEE Transactions on Software Engineering*, 2002.

[51] V. Wahler, D. Seipel, J. W. V. Gudenberg, and G. Fischer, “Clone detection in source code by frequent itemset techniques,” in *IEEE International Workshop on Source Code Analysis and Manipulation*, 2004, pp. 128–135.