

Received December 9, 2020, accepted January 4, 2021, date of publication January 25, 2021, date of current version February 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3054472

# Early Detection of Flawed Structural Dependencies During Software Evolution

DI CUI 

School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China

e-mail: cuidi@stu.xjtu.edu.cn


This work was supported in part by the National Natural Science Foundation of China under Grant 61632015, Grant 61772408, Grant U1766215, Grant 61721002, Grant 61532015, and Grant 61833015; in part by the Ministry of Education Innovation Research Team under Grant IRT\_17R86; and in part by the Project of China Knowledge Centre for Engineering Science and Technology.

**ABSTRACT** During software evolution, complex structural dependencies between source files pose a great challenge on maintenance activities. Some of these dependencies propagate defects among files, incurring frequent bugs or changes, and consuming significant maintenance costs. They can be referred to as flawed structural dependencies. In this paper, we proposed a method to identify these potential problematic dependencies at an early stage during software evolution, by combing structural and semantic dependencies, so that developers can save maintenance costs by fixing these issues in time. Our method works as follows: First, we extract structural dependencies from the source code syntax and semantic dependencies from the source code lexicon. Second, we collect suspect file pairs by calculating the difference between structural and semantic dependencies. Next, we exhaustively examine each source file in the system and locate the interaction of its impacted subordinated files and suspect file pairs (SFP) as suspect dependencies. Finally, we gather all the suspect dependencies as flawed structural dependencies candidates. We evaluate our method using 838 releases of 15 open source projects, including 33353 bug reports and 86690 revision commits. The detection result shows that our identified dependencies use 14% of all the files to capture almost 70% of top 10% bug-prone files or change-prone files with enough high precision: 92%. Moreover, our identified dependencies also incur 957% of bug frequencies and 1050% of change frequencies than average in future versions. In summary, our method can effectively and efficiently detect flawed structural dependencies in time during software evolution.

**INDEX TERMS** Software quality, software maintenance, software evolution, architectural design, code dependencies.

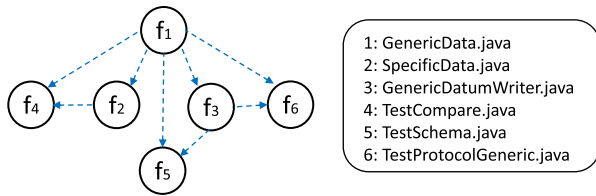
## I. INTRODUCTION

As software evolves, an increasing amount of maintenance efforts were spent on assuring software quality [30]. However, the rapid growth of complex structural dependencies in software systems poses a great challenge to maintenance activities [39]. Recent research has shown that a significant proportion of bug-prone files are often connected by structural dependencies [33]. Our recent industrial collaboration [26] revealed that bug-proneness may propagate through these flawed structural dependencies among files. The accumulation of them forms a great source of technical debt [37], which will gradually make software system harder to maintain, debug, evolve, and eventually cause a ripple effect. For

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Pietrantuono .

a software system, there always exists a large proportion of flawed structural dependencies frequently incurring bugs and changes, which consumes significant maintenance costs during software evolution [37]. If these dependencies can be discovered early and fixed via refactoring in time, it is possible to avoid the increasing maintenance cost.

For example, Figure 1 shows a group of 6 source files architecturally connected with 8 instances of flawed structural dependencies in Avro 1.3.0 [7]. These files and related dependencies are organized with a pyramid structure and *GenericData* is the top-most file dominating the other five files. Table 1 shows the evolution of this flawed structure from Avro 1.0.0 to Avro 1.3.0. Combined with the revision history, bug reports, and source code of this group, we discovered these flawed dependencies were first introduced when fixing *AVRO-110* by implementing *Comparable*



**FIGURE 1.** The flawed structural dependencies in Avro 1.3.0 (Each node represents a source file. Each edge represents a structural dependency).

**TABLE 1.** A case of the flawed structural dependencies in a group of files. (Ver is short for versions).

System			Flawed Structural Dependencies				
Ver	#Files	#Bugs <sup>1</sup>	#Files	#Bugs <sup>2</sup>	#Churn <sup>3</sup>	DL	Hotspot
1.0.0	72	—	—	—	—	70	—
1.1.0	91	18	6	8	522	62	×
1.2.0	102	12	6	2	80	61	×
1.3.0	156	57	6	16	618	74	✓

<sup>1</sup> The number of bugs in system.

<sup>2</sup> The number of bugs in which patched files are involved in flawed structural dependencies.

<sup>3</sup> The number of lines of code spent on patched files in flawed structural dependencies.

interface and employing a singleton design pattern on *GenericData* in 1.1.0. This is a typical design flaw caused by a quick and dirty implementation that may frequently roll back later. We found that, in subsequent versions, these involved 6 source files consume significant maintenance costs from 1.2.0 to 1.3.0. Developers spent nearly 700 lines of code related to 18 bugs within this structure. If these dependencies can be detected early and fixed in time, the extra costs can be saved.

To the best of our knowledge, state-of-the-art related techniques on diagnosing flawed structural dependencies can be divided into two lines. The first line is the metric-based techniques. One representative work is Decoupling Level Metric (DL) [32], which evaluates the maintainability of structural dependencies based on the design rule theory [48]. We employ DL in detecting the preceding structural dependencies. As shown in Table 1, DL does monitor AVRO's structural health from 1.0.0 to 1.2.0. However, this high-level approach cannot locate the problematic dependencies directly.

The second line is history-based techniques. One of the most recent work is hotspot detection [38]. It can leverage revision history to examine structural dependencies correlated with bugs and changes. We also employ this technique in detecting the preceding dependencies. As shown in Table 1, they can be identified in 1.3.0. However, hotspot incurs the following two issues: First, flawed structural dependencies cannot be identified as soon as they were introduced. In the preceding example, these dependencies are detected after significant maintenance costs were paid. By 1.3.0, developers have committed nearly 700 lines of code. Second and most importantly, hotspot may misdiagnoses dependencies already fixed. In the preceding example again, these dependencies are patched in 1.3.0 and will not contribute to bug-proneness in a

long time. However, it will be continually reported by hotspot in subsequent versions according to the revision history [38].

Our previous work [27] systematically investigated the relationship between three types of dependencies (structural dependencies, history dependencies, and semantic dependencies) and software bug-proneness. Our work revealed that semantic dependencies capture a significant proportion of bug-prone files with higher accuracy and efficiency compared with structural dependencies and history dependencies. This finding inspires us that semantic dependencies can assist us in detecting flawed structural dependencies. Our work also revealed that the combination of structural dependencies and semantic dependencies can further improve the performance of bug prediction. The difference between semantic dependencies and structural dependencies, which means that two files are connected with the semantic dependency but not structural dependency, presents the highest efficiency to capture bug-proneness from these combinations. According to the information hiding principle [39], the structurally isolated files should evolve independently for they encapsulate implementation details separately. If they are related to semantic dependencies at the same time, it has the possibility to be a design flaw. This finding can hint us to further locate flawed structural dependencies.

Based on these findings, in this paper, we proposed a method, by combining semantic dependencies and structural dependencies, to identify flawed structural dependencies at early stages during software evolution and maintenance. Once a dependency problem is induced during evolution, our method can achieve just-in-time detection for not requiring revision history. The history-based technique may report it after a few versions until some patches and changes are accumulated on this issue. Our method detects flawed structural dependencies as follows: First, we extract structural dependencies and semantic dependencies among files. Structural dependencies are extracted from the source code syntax using *Depends* [1], a state-of-the-art dependency analysis tool. semantic dependencies are extracted from the source code lexicon using information retrieval techniques. Second, we collect suspect file pairs by calculating the difference between semantic dependencies and structural dependencies. Next, we exhaustively examine each source file in the system. We obtain its impacted subordinated file by calculating the transitive closure of the graph constructed from structural dependencies and locate their interactions with suspect file pairs as suspect dependencies. Finally, we gather all these suspect dependencies as flawed structural dependencies candidates.

Compared with metric-based techniques, our method can accurately locate the potential problematic structural dependencies. Compared with history-based techniques, we can find these flawed structural dependencies at the early stages. We collected 33353 bug reports and 86690 revision commits from 838 versions of 15 open source projects, and systematically evaluated our method. The experiment results showed that: (1) The identified structural dependencies use merely

**TABLE 2. Structural Dependencies versus Semantic Dependencies.**

System			Structural Dependencies (Str)				Semantic Dependencies (Sem)			
Version	#Bugs <sup>1</sup>	#BuggyFiles <sup>2</sup>	#Bugs <sup>1</sup>	#BuggyFiles <sup>2</sup>	#Deps <sup>3</sup>	#BugDeps <sup>4</sup>	#Bugs <sup>1</sup>	#BuggyFiles <sup>2</sup>	#Deps <sup>3</sup>	#BugDeps <sup>4</sup>
1.0.0	–	–	–	–	319	–	–	–	18	–
1.1.0	18	46	18	46	452	84 (18.6%)	12	16	23	19 (82.6%)
1.2.0	12	7	12	7	496	102 (20.6%)	6	4	29	14 (48.3%)
1.3.0	57	57	57	57	771	221 (28.7%)	27	16	93	84 (90.3%)

<sup>1</sup> The number of involved bugs in system, structural dependencies or semantic dependencies.

<sup>2</sup> The number of involved patched files during fixing bugs in system, structural dependencies or semantic dependencies.

<sup>3</sup> The number of structural dependencies or semantic dependencies.

<sup>4</sup> The number of dependencies in which both of the connected files are patched files.

14% of all the files; (2) The precision of identified structural dependencies is as high as 92%; (3) Almost 70% of the top 10% bug-prone and change-prone files are covered by the identified structural dependencies; (4) The involved files in identified structural dependencies will incur 957% and 1050% frequencies of bugs and changes in subsequent versions than average.

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

## II. PROBLEM DEFINITION AND MOTIVATION

### A. PROBLEM DEFINITION

Flawed structural dependencies (FSD) is a set of structural dependencies. A instance of flawed structural dependency is a pair of files and both of them are frequently involved in bug fixes and change commits. It is formally defined as:

$$FSD = \{(f_i, f_j, type, Cost(v_m, v_n)) \mid i \neq j, m \neq n\} \quad (1)$$

where  $f_i$  and  $f_j$  represent the source files.  $v_m$  and  $v_n$  represent the versions.  $type$  represents the type of structural dependency between  $f_i$  and  $f_j$ , including inheritance, implementation, method call, field access, type reference, and instance creation.  $Cost(v_m, v_n)$  represents, from version  $m$  to version  $n$ , the increased maintenance cost (bug frequency and churn) was spent on this dependency.

Recent study revealed [78] that flawed structural dependencies may not capture all co-change dependencies. The involved files may be patched separately in multiple bug fixes. Previous study [33] have shown that flawed structural dependency cover 70% of the most bug-prone/change-prone files on average. That is, 30% of them are not connected with structural dependencies. A more precise and efficient approach is deserved to detect flawed structural dependencies.

### B. MOTIVATION

From the findings in our previous work, we assume that semantic dependencies and their interaction with structural dependencies can assist us to early detect flawed structural

**TABLE 3. The interaction of structural dependencies (Str) and semantic dependencies (Sem).**

System Version	Str $\cap$ Sem <sup>1</sup>		Str $\cap \neg$ Sem <sup>2</sup>	
	#Deps <sup>3</sup>	#BugDeps <sup>4</sup>	#Deps <sup>3</sup>	#BugDeps <sup>4</sup>
1.0.0	10	–	8	–
1.1.0	10	7 (70.0%)	13	12 (92.3%)
1.2.0	11	4 (36.4%)	18	10 (55.6%)
1.3.0	10	7 (70.0%)	83	77 (92.8%)

<sup>1</sup> The intersection of structural dependencies and semantic dependencies.

<sup>2</sup> The dependencies contained in structural dependencies but not in semantic dependencies.

<sup>3</sup> The number of dependencies.

<sup>4</sup> The number of dependencies in which both of the connected files are patched files.

dependencies. In this part, we first validate our assumptions in Avro. Table 2 and Table 3 present the correlation between these dependencies and software bug-proneness from 1.0.0 to 1.3.0. We observed that:

1) Most of the bug-prone files are relevant to structural dependencies. This observation is consistent with Lu et al's work [33] and our previous work [27]: there exists the correlation between bug locations and dependencies, which is more significant in security bugs [80] and performance bugs [79]. As shown in Table 2, from 1.1.0 to 1.3.0 of Avro, structural dependencies capture all the buggy/bug-prone files (100%). This validates the existence of flawed structural dependencies that propagate bug-proneness among files. In total, we discovered 452-771 instances of structural dependencies. However, only a small portion (18.58%-28.66%) of them are related to bug dependencies. It means that deriving flawed dependencies from structural dependencies directly is challenging.

2) Semantic dependencies present a higher efficiency in capturing software bug-proneness compared with structural dependencies. As shown in Table 2, from 1.1.0 to 1.3.0 of Avro, semantic dependencies cover 46 bugs (50% of 87 bugs in total) and 36 buggy/bug-prone files (32% of 110 files). The proportion of flawed/bug dependencies from semantic dependencies is 74%, which is 336% higher than structural dependencies. On the contrary, the average number of semantic dependencies is merely 48 instances, which is 8% of structural dependencies. It implies that semantic dependencies can be used as hints to further detect flawed structural dependencies.

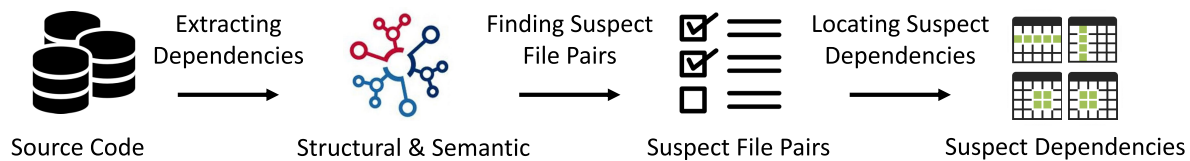


FIGURE 2. The overview of our approach to detect flawed structural dependencies.

3) The combination of semantic dependencies and structural dependencies captures newly introduced bug-prone files during software evolution. We explore the interaction of semantic dependencies and structural dependencies in Table 3 and discovered: From 1.1.0 to 1.3.0, their intersection stably contains 10 instances and the average proportion of flawed dependencies from them is 58.8%. On the contrary, the difference between semantic dependencies and structural dependencies contains 23 to 93 instances, and the average proportion of flawed dependencies from them is as high as 80.2%. From 1.1.0 to 1.3.0, the number of files and bugs in this system is continually increasing shown in Table 1. Therefore, we observed the difference between semantic dependencies and structural dependencies indeed captures newly introduced bug-prone files in time.

These observations validate our assumption of semantic dependencies and motivate us to design our method to early detect flawed structural dependencies.

### III. METHODOLOGY

In this section, we describe our methodology. Figure 2 presents an overview of our approach to detect flawed structural dependencies. Our approach takes the source as input and follows three steps: extracting structural dependencies and semantic dependencies, collecting suspect file pairs, and generating suspect dependencies as flawed structural dependencies candidates.

#### A. EXTRACTING STRUCTURAL AND SEMANTIC DEPENDENCIES

In this step, we extract dependencies including structural dependencies and semantic dependencies as follows:

##### 1) EXTRACTING STRUCTURAL DEPENDENCIES

Structural dependency, derived from the source code syntax, is one of the most common code dependencies for program comprehension and software maintenance. In our paper, we select six types of syntax, including software inheritance, implementation, method call, field access, type reference, and instance creation, as the source to extract structural dependencies. We employ *Depends* [1], a state-of-the-art dependency analysis tool, to extract these dependencies among files. For a subject, we denote all the collected structural dependencies as *Str*.

##### 2) EXTRACTING SEMANTIC DEPENDENCIES

Semantic dependencies, derived from the source code lexicon, explore the textual similarity among files using

information retrieval techniques. In this paper, we extract semantic dependencies using the five steps following the previous work [36]:

1) Crawling raw lexical data. In this step, we crawl four types of source code lexicons as features including class name ( $feature_1$ ), method name (function and function parameter name,  $feature_2$ ), global variable name ( $feature_3$ ), and comment ( $feature_4$ ). Supported by our implemented lexical parser based on srcML [2], a state-of-the-art lightweight and scalable parsing tool, we summarize all the collected data as a  $n \times 4$  matrix:

$$RawLexicalData == \begin{Bmatrix} WS_1^1 & WS_1^2 & WS_1^3 & WS_1^4 \\ WS_2^1 & WS_2^2 & WS_2^3 & WS_2^4 \\ \dots & \dots & \dots & \dots \\ WS_n^1 & WS_n^2 & WS_n^3 & WS_n^4 \end{Bmatrix} \quad (2)$$

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

$$WS_i^k = \{w_1^k, w_2^k, \dots, w_m^k\} \quad (3)$$

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  $feature_k$  is correlated with class name, method name, global variable name, and comment respectively.

For each  $file_i$ ,  $\{WS_i^1, WS_i^2, WS_i^3, WS_i^4\}$  represents the collected lexical data set. We use *GenericDatumWriter*, one of the preceding files with flawed structural dependencies, as an example. Figure 3 presents a part of its source code. The column type presents the results parsed by our tool line by line. The collected lexical data of *GenericDatumWriter* is demonstrated in the first row of Table 4. 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 set with three processes including filtering, decoupling, and stemming. The filtering process removes the 345 stop words for the natural language [62]. The decoupling process separates each word according to the naming convention, such as camel casing and snake casing [63]. The stemming process searches the root for each word using Porter algorithm [62]. Thus, for each  $WS_i^k$ , the preprocessed word set is defined as follows:

$$PWS_i^k = \{pw_1^k, pw_2^k, \dots, pw_m^k\} \quad (4)$$

where  $pw$  represents the preprocessed word and  $m$  represents the number of words. Table 4 presents the preprocessing

LINE	TYPE	SOURCE CODE
1	Comment	/** {@link DatumWriter} for generic Java objects. */
2	Class	<b>public class</b> GenericDatumWriter<D> <b>implements</b> DatumWriter<D> {
3	Gvaribale	<b>private final</b> GenericData <b>data</b> ;
4	Gvariable	<b>private</b> Schema <b>root</b> ;
5	Comment	/** Called to write data.*/
6	Method	<b>protected void</b> 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	<b>protected void</b> writeRecord(Schema schema, Object datum, Encoder out){...}
n		..... }

FIGURE 3. Crawling Lexical Data of GenericDatumWriter (Gvarible is short for global variable name).

TABLE 4. Preprocessing Lexical Data of GenericDatumWriter (The difference is marked in Bold).

Steps	Class	Method	Global Variable	Comment
Raw	GenericDatumWriter	Data,root	write, schema(2), datum(2), <b>out(2)</b> , writeRecord	Link, generic, Java, objects, data, overridden, alternate, representation, <b>a, may, be, DatumWriter, for(2), to(2), called(2), write(2), record(2)</b>
Filtered	<b>GenericDatumWriter</b>	Data,root	write, schema(2), datum(2), <b>writeRecord</b>	Link, generic, Java, objects, data, overridden, alternate, representation, <b>DatumWriter, called(2), write(2), record(2)</b>
Decoupled	<b>Generic, Datum, Writer</b>	<b>Data, root</b>	write(2), schema(2), datum(2), <b>Record</b>	<b>Link, generic, Java, objects, data, overridden, alternate, representation, Datum, Writer, called(2), write(2), record(2)</b>
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)

of lexical data of *GenericDatumWriter* step by step. For convenience, we mark the difference 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.

3) Generating TF-IDF weighting matrix from preprocessed lexical data. In this step, we generate weighting matrix using one of the most popular information retrieval model: TF-IDF [31]. Based on the definition of equation 2, for each *feature<sub>k</sub>*, we gather the involved preprocessed lexical data as lexical space: *LS<sub>k</sub>*, which is defined as follows:

$$LS_k = \{PWS_1^k, PWS_2^k, \dots, PWS_n^k\} \quad (5)$$

where *n* is the number of files. For each *LS<sub>k</sub>*, we generate its weighting matrix using TF-IDF as *E<sup>k</sup>*, which is a *m × n* matrix (file-by-word). *m* represents the number of distinct words in feature *k*.

A generic entry *e<sub>i,j</sub><sup>k</sup>* of this matrix denotes the relevance of the *i<sup>th</sup>* word in the *j<sup>th</sup>* file.

$$e_{i,j}^k = \frac{t_{i,j}}{\sum_o t_{o,j}} \times \log \frac{n}{|\{j : pw_i \in file_j\}|} \quad (6)$$

where *t<sub>i,j</sub>* represents the occurrence frequency of word *pw<sub>i</sub>* in *file<sub>j</sub>*.  $\sum_o t_{o,j}$  represents the occurrence frequency of word *pw<sub>i</sub>* in all of files.  $|\{j : pw_i \in file_j\}|$  represents the number of files containing *pw<sub>i</sub>*. In summary, the *e<sub>i,j</sub><sup>k</sup>* represents the

term frequency and inverse document frequency (TF-IDF) for the word: *pw<sub>i</sub>* and the file: *file<sub>j</sub>*. For the collected TF-IDF weighting matrices, we further use principle component analysis (PCA) to reduce the dimension of features to compute textual similarity as: *RE<sub>k</sub>*. We implemented the TF-IDF and PCA using the state-of-the-art machine learning library: scikit-learn [23].

4) Computing textual similarity. In this step, we compute textual similarity matrix as *S<sub>k</sub>* for each lexical feature: *LS<sub>k</sub>* based on the reduced TF-IDF matrix: *RE<sup>k</sup>*. The constructed textual similarity matrix is a *n × n* matrix (file-by-file) where *n* is the number of files. A generic entry *s<sub>i,j</sub><sup>k</sup>* of *S<sup>k</sup>* denotes the cosine semantic similarity between *i<sup>th</sup>* file and *j<sup>th</sup>* file of *RE<sup>k</sup>*:

$$s_{i,j}^k = \frac{\sum_{l=1}^m re_{l,i}^k \times re_{l,j}^k}{\sqrt{\sum_{l=1}^m (re_{l,i}^k)^2} \times \sqrt{\sum_{l=1}^m (re_{l,j}^k)^2}} \quad (7)$$

where the *re<sub>l,i</sub><sup>k</sup>* and *re<sub>l,j</sub><sup>k</sup>* represent the entries of *RE<sup>k</sup>*.

5) Obtaining semantic dependencies. In this step, we obtain semantic dependencies by normalizing the textual similarity matrix and fusing them. We first normalize the *S<sup>k</sup>* as  $\bar{S}^k$ . The entry of  $\bar{s}_{i,j}^k$  in  $\bar{S}^k$  is defined as:

$$\bar{s}_{i,j}^k = \begin{cases} 1, & s_{i,j}^k > \theta \\ 0, & \text{other} \end{cases} \quad (8)$$

where  $\theta$  is empirically set as 0.8.

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

$$SIM = \sum_{k=1}^4 \bar{S}^k \quad (9)$$

Based on the definition of  $SIM$ , we formally define the semantic dependencies as  $Sem$ :

$$Sem = \{(file_i, file_j) \mid SIM_{i,j} > th, i \neq j\} \quad (10)$$

where  $th$  is set as 2. In semantic dependencies:  $Sem$ ,  $(file_i, file_j)$  is equivalent to  $(file_j, file_i)$ .

### B. COLLECTING SUSPECT FILE PAIRS

The suspect file pair (SFP) is defined as a pair of file:  $x$  and  $y$ . They are involved in semantic dependencies but not structural dependencies.

$$SFP = \{(x, y) \mid (x, y) \notin Str \wedge (x, y) \in Sem\} \quad (11)$$

where  $Str$  represents the structural dependencies and  $Sem$  represents the semantic dependencies.

For a subject, we exhaustively inspect each pair of files and collect all the suspect file pairs (SFP).

### C. GENERATING SUSPECT DEPENDENCIES

Although suspect file pairs (SFP) can reveal maintenance problem, each pair of files are actually structurally isolated. However, we aim at detecting flawed structural dependencies. As a consequence, we design the following algorithm to capture the interaction between structural dependencies and suspect file pairs as suspect dependencies.

Before describing the detection algorithm, we first introduce two notions: structural dependency graph:  $G$  and its transitive closure graph:  $G+$ . The structural dependency graph is defined as follows:

$$G = \{F, Str\} \quad (12)$$

where  $F$  represents the collection of all the source files in system.  $Str$  represents the structural dependencies among these source files. The transitive closure of structural dependency graph is defined as:

$$G+ = \{F, Str+\} \quad (13)$$

where  $Str+$  represents the reachable structural dependencies among these source files. For an edge:  $(a, b)$  in  $Str+$ , it means that there is a path from file  $a$  to file  $b$ . That is, file  $a$  has the structural impact on file  $b$ .

Algorithm 1 shows the procedure of detecting suspect dependencies. The idea is based on the finding unveiled by recent work [33]: if the subordinate files are bug-prone, the leading files are also likely to be bug-prone. The input of this algorithm is the collection of suspect file pairs and transitive closure of the structural dependency graph. The output is a set of suspect dependencies. This algorithm inspects all the source files in the system iteratively. For each file, we use it as the leading file:  $v$ . Line 2 to 9 select a set of subordinate

### Algorithm 1 SusDepsDetection( $G+$ , $SFP$ )

---

```

1:  $V, E \leftarrow \{G+\}.F, \{G+\}.Str+$  % initialization
2:  $SusDepSet \leftarrow \emptyset$  % detection target
3: for  $v$  in  $V$  do
4:    $Sub(v) \leftarrow \emptyset$  % the subordinate files influenced by  $v$ 
5:   for  $v_s$  in  $V$  do
6:     if  $(v, v_s) \in E$  then
7:        $Sub(v).append(v_s)$ 
8:     end if
9:   end for
10:   $SusFileSet(v) \leftarrow \{v\}$  % the subset of suspect files
11:  for  $a$  in  $Sub(v)$  do
12:    for  $b$  in  $Sub(v)$  do
13:      if  $a \neq b$  and  $(a, b) \in SFP$  then
14:         $SusFileSet(v).append(a)$ 
15:         $SusFileSet(v).append(b)$ 
16:      end if
17:    end for
18:  end for
19:   $SusDepSet(v) \leftarrow \emptyset$  % the subset of suspect deps
20:  for  $i$  in  $SusDepSet(v)$  do
21:    for  $j$  in  $SusDepSet(v)$  do
22:      if  $i \neq j$  and  $(i, j) \in E$  then
23:         $SusDepSet(v).append((i, j))$ 
24:      end if
25:    end for
26:  end for
27:   $SusDepSet \leftarrow SusDepSet \cup SusDepSet(v)$ 
28: end for

```

---

files having structural dependencies with the leading file:  $v$ . It means that these selected files are structurally impacted by the file:  $v$ . We further analyze them and related dependencies as follows: Line 10 to 18 discover the overlap of these impacted files and suspect file pairs (SFP) as the suspect file set because our observation in Section II revealed that the suspect file pair is an efficient pattern to characterize bug-proneness. We collect their intersection as the suspect files. Line 19 to 26 further distill its involved dependencies for they are involved in suspect files. Line 27 iteratively finally gathers all the suspect dependencies into the suspect dependency set as flawed structural dependencies candidates.

### IV. RUNNING EXAMPLE

In this section, we use the case of flawed structural dependencies presented in Table 1 as an example to illustrate the procedure of our approach. We use the design structure matrix (DSM), a state-of-the-art tool, to visualize multiple software dependencies. We introduce the related concept as follows:

**Design Structure Matrix (DSM).** A DSM is a square matrix. Each element in DSM represents 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 dependencies between the file in a row and the file in the column. A marked cell in row  $x$  and column  $y$ , cell  $(x, y)$ ,

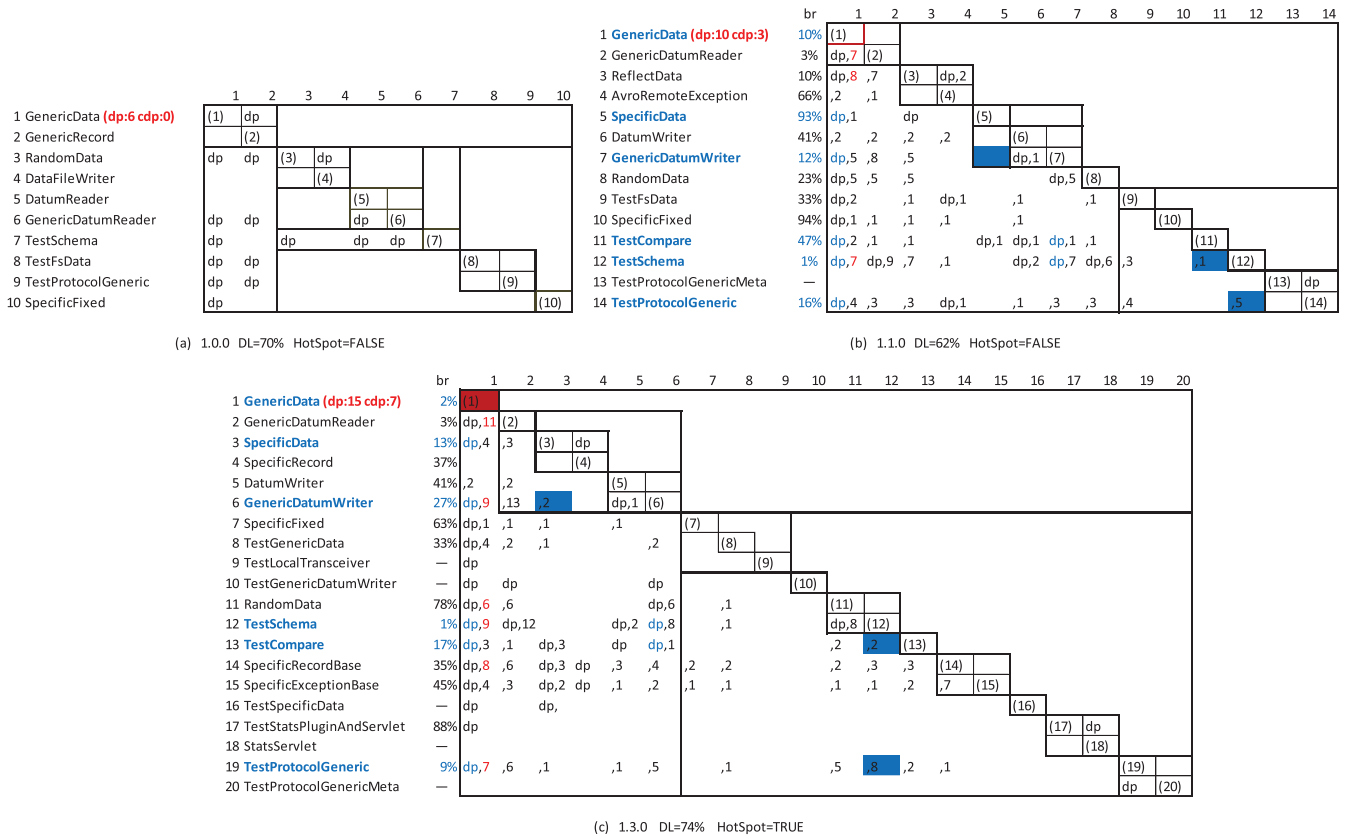


FIGURE 4. A running example to detect flawed structural dependencies.

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 dependencies such as structural or semantic. We use the DSM shown in Figure 4.(b) as an example. In Figure 4.(b), cell(2,1) is labeled with “dp”, which means *GenericDatumReader* have structural dependencies with *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 filled with blue color, which indicates *GenericDatumReader* and *SpecificData* are connected with semantic dependencies. In this paper, we employ DSM to manage structural and semantic dependencies.

**Case Study.** The case of flawed structural dependencies shown in Table 1 and Figure 1 involves six source files including *TestCompare*, *TestSchema*, *TestProtocolGeneric*, *SpecificData*, *GenericDatumWriter*, and *GenericData*, which is organized with a pyramid structure. We found that *GenericData* is the top leading file structurally connected with other files. By tracing the source code, bug report, and revision history of these files, we represent the evolution of these files and related dependencies structurally influenced by *GenericData* using the design structure matrix (DSM). Figure 4.(a)–4.(c) present the results from 1.0.0 to 1.3.0. For each file of each version in presented DSMs, we label its bug ranking on the left. We observed that, by Avro 1.3.0, the average

bug ranking of the six files in the previous case increases almost 20%. According to the concept of code smell, this flawed structure is a typical case informally referred to 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 increasingly significant maintenance costs. We believe that the poor dependency/architectural design can be the root cause for this case, which violates the OO design principle: the single responsibility principle (SRP). Once a change is made, several modifications will also be induced accordingly. We can fix this issue by applying the state-of-the-art interactive architectural refactoring tool [24] to guide our redesign step by step.

To early detect flawed structural dependencies in the above case, we employ DL measurement, Hotspot detection, and our method as follows:

**DL Measurement.** Mo *et al.* [32] proposed a new metric: decoupling named DL metric based on design rule theory to measure the maintenance complexity of software dependencies. The DL opens the possibility of quantifying canonical principles of single responsibility and separation of concerns, aiding the comparison of various projects and variation of the individual project. In this case, for an individual project, we intuitively believe that the variation of DL metric may capture the

inducing of flawed structural dependencies. However, from Figure 4.(a)-4.(c), we observed that the variation of DL reports the presence of flawed structural dependencies. However, this method failed to concentrate on related files and dependencies.

**Hotspot Detection.** Mo *et al.* [38] proposed a suite of problematic patterns related to high error-proneness and change-proneness named hotspot. The studied case was detected by hotspot as an instance of the unstable interface pattern in 1.3.0, shown in Figure 4(c) marked with the red color. According to the definition of unstable interface pattern, 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 instance of an unstable interface pattern. By default, if there are more than 10 files structurally depending on this file ( $dp > 10$ ) and more than 5 files change together with it more than 5 times ( $\#cochange > 5$ ), hotspot will identify them. According to the definition, these files are identified by Hotspot only when they have been revised enough times. In this case, we also found that the studied case of flawed structural dependencies is detected until Avro 1.3.0. However, at that time, it has accumulated significant maintenance costs.

**Our Method.** The procedure of our method is as follows:

First, we extract all the structural and semantic dependencies from the source code for each version. Second, we collect all the suspect file pairs (SFP) by calculating the difference of structural and semantic dependencies. In this studied case, there are no suspect file pairs (SFP) in 1.0.0. On the contrary, both in 1.1.0 and 1.3.0, three instances of suspect file pairs (SFP) within are found filled with blue background color shown in Figure 4(b) and 4(c), including (*SpecificData*, *GenericDatumWriter*), (*TestCompare*, *TestSchema*), and (*TestCompare*, *TestProtocolGeneric*). Finally, we determine the suspect file set by calculating the overlap of structurally influenced files and suspect file pairs. For example, in 1.1.0 and 1.3.0 shown in Figure 4.(b)-Figure 4.(c), we obtained a set of six files:  $\{TestCompare, TestSchema, TestProtocolGeneric, SpecificData, GenericDatumWriter, and GenericData\}$ , as the suspect file set. We further discover dependencies among them as suspect dependencies, which are identified as flawed structural dependencies candidates. For example, in 1.1.0 and 1.3.0 shown in Figure 4.(b)-Figure 4.(c), the following 7 instance of suspect dependencies are detected by our approach: (*GenericData*, *SpecificData*), (*GenericData*, *GenericDatumWriter*), (*GenericData*, *TestCompare*), (*GenericData*, *TestSchema*), (*GenericData*, *TestProtocolGeneric*), (*GenericDatumWriter*, *TestSchema*), and (*GenericDatumWriter*, *TestProtocolGeneric*),

**Summary.** The detected suspect dependencies cover the fore-mentioned flawed structural dependencies. Compared with DL measurement, our method identifies them accurately. Compared with hotspot detection, our approach identifies them in 1.1.0 as they first emerge.

## V. EVALUATION

To evaluate the effectiveness of our approach, we investigated 838 versions of 15 Apache open source projects (Avro [7], Cassandra [8], Flume [9], Hadoop [10], Hbase [11], Log4j [12], Mahout [13], Mina [14], Openjpa [15], Pdfbox [16], Pig [17], Tika [18], Zookeeper [19], Cxf [20], and Camel [21]) as our evaluation subjects. They are involved with 33353 bug reports and 86690 revision commits shown in Table 5. These projects differ in domain, scale, and other characteristics. The bug reports and revision commits are extracted from their version control system: Git [5] and issue tracking systems: JIRA [6]. We only study bug reports having a resolution of fixed. We extracted structural and semantic dependencies for each version, and identify suspect dependencies. For these detected results, we investigated the following research questions:

**RQ1:** Whether the files in our identified dependencies will incur high maintenance costs in the subsequent versions?

**RQ2:** What is the accuracy of our method to detect flawed structural dependencies?

**RQ3:** Can our method discover flawed structural dependencies in time?

### A. THE MAINTENANCE COST OF SUSPECT DEPENDENCIES

To answer **RQ1**, we iteratively analyzed all the detected suspect dependencies 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 these detected dependencies on subsequent maintenance efforts: future bug frequency (FBF) and future change frequency (FCF).

For files in suspect dependencies: *SD*, future bug frequency (FBF) and future change frequency (FCF) calculate the average bug fixing/code change frequency of each involved file from the version:  $v_i$  to the version:  $v_n$ .

To measure future bug frequency (FBF) and future change frequency (FCF), we first collect the related bug fixing/code 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:  $bf_{ji}$  or  $cf_{ji}$  represents the number of frequencies of bug fixes/code changes for  $F_j$  in version  $v_i$ . For a software system with  $n$  consecutive versions and  $m$  source files, Table 6 demonstrates the related bug fixing information as a matrix (*Bug fixing frequency matrix*). Similarly, Table 7 demonstrates the related code change information as a matrix (*Code change frequency matrix*).

Thus, for a detected file group: *SD* from version:  $v_i$  to version:  $v_n$ , the future bug frequency (FBF):  $FBF(SD, v_i, v_n)$  and future change frequency (FCF):  $FCF(SD, v_i, v_n)$  are defined as follows:

$$FBF(SD, v_i, v_n) = \frac{\sum_{F_j \in SD} \sum_{k=i+1}^n bf_{jk}}{|SD|} \quad (14)$$

$$FCF(SD, v_i, v_n) = \frac{\sum_{F_j \in SD} \sum_{k=i+1}^n cf_{jk}}{|SD|} \quad (15)$$



TABLE 5. The basic information of 15 studied subjects.

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

TABLE 6. Bug fixing frequency matrix.

	$v_1$	$v_2$	$v_3$	...	$v_i$	...	$v_n$
$F_1$	$bf_{11}$	$bf_{12}$	$bf_{13}$	...	$bf_{1i}$	...	$bf_{1n}$
$F_2$	$bf_{21}$	$bf_{22}$	$bf_{23}$	...	$bf_{2i}$	...	$bf_{2n}$
$F_3$	$bf_{31}$	$bf_{32}$	$bf_{33}$	...	$bf_{3i}$	...	$bf_{3n}$
...	...	...	...	...	...	...	...
$F_j$	$bf_{j1}$	$bf_{j2}$	$bf_{j3}$	...	$bf_{ji}$	...	$bf_{jn}$
...	...	...	...	...	...	...	...
$F_m$	$bf_{m1}$	$bf_{m2}$	$bf_{m3}$	...	$bf_{mi}$	...	$bf_{mn}$

TABLE 7. Code change frequency matrix.

	$v_1$	$v_2$	$v_3$	...	$v_i$	...	$v_n$
$F_1$	$cf_{11}$	$cf_{12}$	$cf_{13}$	...	$cf_{1i}$	...	$cf_{1n}$
$F_2$	$cf_{21}$	$cf_{22}$	$cf_{23}$	...	$cf_{2i}$	...	$cf_{2n}$
$F_3$	$cf_{31}$	$cf_{32}$	$cf_{33}$	...	$cf_{3i}$	...	$cf_{3n}$
...	...	...	...	...	...	...	...
$F_j$	$cf_{j1}$	$cf_{j2}$	$cf_{j3}$	...	$cf_{ji}$	...	$cf_{jn}$
...	...	...	...	...	...	...	...
$F_m$	$cf_{m1}$	$cf_{m2}$	$cf_{m3}$	...	$cf_{mi}$	...	$cf_{mn}$

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

For each version:  $v_i$  of each studied 15 projects, we measure its future bug frequency (FBF) and future change frequency (FCF) of collected suspect dependencies from the current version:  $v_i$  to the latest version:  $v_n$ . For comparison, we also measure the future bug frequency (FBF) and future change frequency (FCF) of all the files (AF) and all the bug-prone files (ABF) for each version.

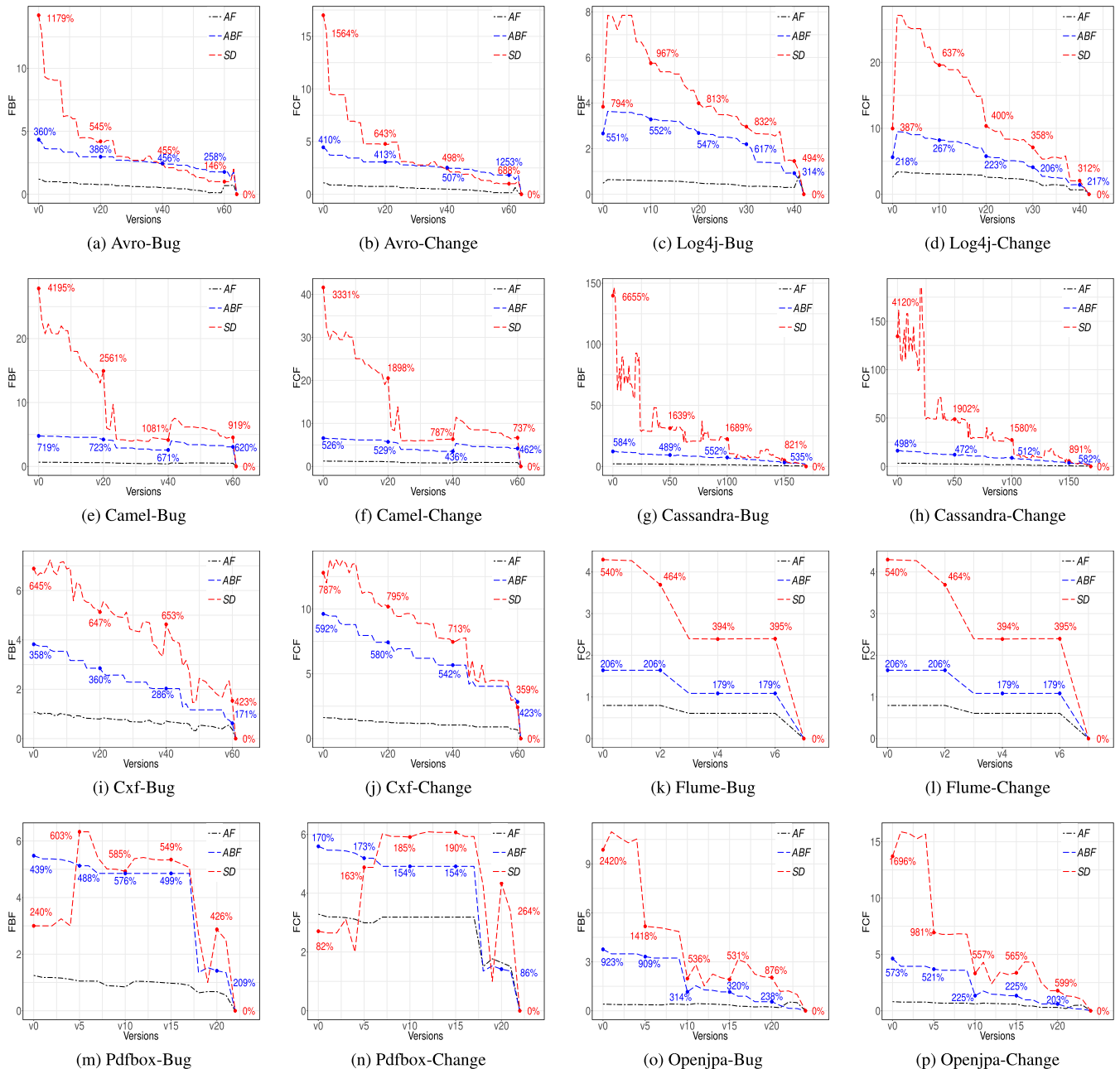
The results are demonstrated in Figure 5. We only present the measurement of eight projects and the full data is also available online [22]. In each figure, the x-axis presents the version and the y-axis presents the future bug frequency (FBF) or the future change frequency (FCF). The red line represents the results of suspect dependencies (SD), the blue line represents the results of all the bug-prone files (ABF) and the black line represents the results of all the files (AF). Some points on suspect dependencies (SD) or bug-prone files (ABF) are also highlighted. The percentage marked with

red or blue color demonstrates the increase rate of the result of FBF/FCF on suspect dependencies (SD) or the bug-prone files (ABF) compared with the result of FBF/FCF on all the files (AF).

For each project, we sum up the future bug frequency (FBF) or future change frequency (FCF) and calculate the average value of them. We observed that, from these 15 selected projects, the average FBF of suspect dependencies: SD is 6.7, which is 957% higher than the value of all the files: AF (0.7). The average FCF of suspect dependencies: SD is 8.4, which is also 1050% higher than the value of all the file: AF (0.8). Meanwhile, the average FBF of all the bug-prone files: ABF is 4.0, which is 571% higher than all the files: AF. The average FCF of all the bug-prone files: ABF is 6.2, which is 775% higher than all the files: AF. According to our observations, files within suspect dependencies do incur much higher maintenance costs, including bug fixing and code change, in subsequent versions. Compared with all the files: AF and all the bug-prone files: ABF, suspect dependencies (SD) present enough high bug and change frequency for they may concentrate on the most bug-prone files.

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

Obviously, all the curves of suspect dependencies: SD in Figure 5 is decreasing with the increase of versions. A possible explanation is 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 files reported by suspect dependencies: SD are not



**FIGURE 5.** The future bug frequency (FBF) or future change frequency (FCF) of suspect dependencies (SD) (AF represents all the files. ABF represents all the bug-prone files which are patched in fixing bugs. SD represents files in our detected suspect dependencies).

frequent enough in bug fixes and changes as we consider. Actually, these files may contain potential bugs and will cause maintenance costs later than our observed versions. We are still tracking these issues now. Moreover, there are 9 versions in which the FBF of suspect dependencies: SD is lower than all the files: AF. Meanwhile, there are also 7 versions in which the FCF of suspect dependencies: SD is lower than all the files: AF. We manually inspect these files and find that most of them are actually unstable. Some files are even directly removed in later versions. As a consequence, the FBF or FCF decreases because these removed files do not participate in subsequent revision commits anymore.

**Answer to RQ1:** Files within detected suspect dependencies do incur frequent bug fixes and code changes in subsequent versions, which are 957% and 1050% frequent than average. Thus, the identified suspect dependencies can be considered as flawed structural dependencies candidates.

**B. THE ACCURACY OF SUSPECT DEPENDENCIES**

To answer RQ2, we evaluate the accuracy of suspect dependencies from the following four perspectives because there is no specific ground truth for flawed structural dependencies. These perspectives are:

- **BugFrequency**: the number of times involved in bug fixing commits.
- **BugChurn**: the number of lines of code to in bug fixing commits.
- **ChangeFrequency**: the number of times involved in change commits.
- **ChangeChurn**: the number of lines of changed code in change commits.

These data are derived from collected bug reports and revision history. Thus, for files in each project, we generate four types of rankings based on the preceding four measures. They are:

- $BF(x\%)$ : the top  $x\%$  of the files ranked with the bug frequency.
- $BC(x\%)$ : the top  $x\%$  of the files ranked with the bug churn.
- $CF(x\%)$ : the top  $x\%$  of the files ranked with the change frequency.
- $CC(x\%)$ : the top  $x\%$  of the files ranked with the change churn.

Specifically,  $BF(100\%)$  is equivalent to  $BC(100\%)$ , which represents the files involved in bug fixing commits.  $CF(100\%)$  is equivalent to  $CC(100\%)$ , which represents the files involved in change commits.

As for suspect dependencies, for a project of  $n$  versions, we obtain suspect dependencies (SD) in each version:  $\{SD_1, SD_2, \dots, SD_n\}$ . Thus, all the detected dependencies are defined as  $SD_{all}$ :

$$SD_{all} = \{SD_1 \cup SD_2 \cup \dots \cup SD_n\} \quad (16)$$

We calculate the precision and recall of  $SD_{all}$ . First, we calculate the precision of  $SD_{all}$  as  $BugRate$ :

$$BugRate = \frac{|\{(i, j) | (i, j) \in SD_{all} \wedge i, j \in ABF\}|}{|SD_{all}|} \quad (17)$$

where  $ABF$  represents all the bug-prone files in project, which can be represented with  $BF(100\%)$  or  $BC(100\%)$ .  $BugRate$  measures the proportion of detected dependencies in which both of subordinate files are involved in bug fixing commits.

Then we calculate the recall of  $SD_{all}$  using its coverage on different subsets of top  $x\%$  bug-prone or change-prone files as:

$$CoverageRate(BCF, x\%) = \frac{|SD_{all} \cap BCF(x\%)|}{|BCF(x\%)|} \quad (18)$$

where  $\{SD_{all}\}$  represents all the involved files with suspect dependencies.  $BCF(x\%)$  represents the top  $x\%$  of files ranked using the above four measure, which can be the most bug-prone files with frequency:  $BF(x\%)$ , the most bug-prone files with churn:  $BC(x\%)$ , the most change-prone files with frequency:  $CF(x\%)$ , the most change-prone files with churn:  $CC(x\%)$ .

Selecting the top 10%, 30% and 100% of the above four measures, we calculate the coverage of  $\{SD_{all}\}$  on these different combinations. Table 8 shows the  $Bugrate$  as precision and the  $Coverage$  as recall of these projects. Based on the collected data, we make the following observations:

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

2) The average bug rate of suspect dependencies is as high as 92% of all the projects. From these subjects, the bug rate of suspect dependencies ranges from 77% to 100%. The true positive rate is high enough to guide the developer to locate problematic dependencies.

3) Suspect dependencies participate in a significant proportion (61%-68%) of the most bug-prone and change-prone files in terms of frequency or churn (top 10%). From this subject, the average coverage rate of the top 10% bug-prone files in terms of frequency is 68%. The average coverage rate of the top 10% bug-prone files in terms of churn is 63%. Similar results can also be observed in most change-prone files, the average coverage rate of the top 10% change-prone files in terms of frequency is 68%. The average coverage rate of the top 10% change-prone files in terms of churn is 61%. Thus, our method can discover problematic dependencies with the most maintenance costs.

4) Suspect dependencies only identify a small proportion (37% and 35%) of all the bug-prone or change-prone files. The average coverage rate of all the bug-prone files is 37%. The average coverage rate of all the change-prone files is 35%. In actual, our method is not designed to inspect all the bug-prone or change-prone files like state-of-the-art techniques of bug prediction or change prediction. Our approach identifies problematic dependencies and related files frequently contributing to the bug-proneness and change-proneness, which deserves the software practitioner's special attention.

Taking Avro as an example, our approach identified 91 distinct files with suspect dependencies in total over 65 versions of this project. There are 19 and 18 files capturing the top 10% bug-prone files in terms of frequency and churn. The coverage rate is 82% and 71%. Meanwhile, there are also 19 and 18 files covering the top 10% change-prone files in terms of frequency and churn. The coverage rate is 82% and 71%. At the same time, the bug rate of suspect dependencies is as high as 94%. Our approach does identify flawed structural dependencies consuming expensive maintenance costs.

**Answer to RQ2:** Our method presents a high accuracy in discovering flawed structural dependencies. Our detection results capture almost 70% of the most bug-prone or change-prone files (10%) in terms of frequency or churn with over 92% precision, merely using 14% of files in the system. Our reported results can be easily followed and focused on by developers.

### C. THE TIMELINESS OF SUSPECT DEPENDENCIES

To answer RQ3, we employ our method in an Apache open project: Flume [9], which is a framework for processing high throughput data. By conducting a fine-grained analysis of the detection results, we illustrate the timeliness of our method.

TABLE 8. The precision and recall results of suspect dependencies.

Subject	%File <sup>1</sup>	BugRate <sup>2</sup>	CoverageRate <sup>3</sup>									
			Top 10%				Top 30%				All <sup>4</sup>	
			BF <sub>10%</sub>	BC <sub>10%</sub>	CF <sub>10%</sub>	CC <sub>10%</sub>	BF <sub>30%</sub>	BC <sub>30%</sub>	CF <sub>30%</sub>	CC <sub>30%</sub>	Bug	Change
Avro	14%	94%	82%	71%	82%	71%	61%	62%	59%	63%	47%	47%
Cassandra	12%	98%	55%	50%	55%	48%	27%	34%	27%	33%	18%	17%
Flume	24%	100%	44%	55%	44%	54%	30%	40%	28%	39%	28%	21%
Hadoop	32%	65%	64%	53%	64%	46%	53%	50%	52%	47%	46%	39%
Hbase	19%	99%	93%	77%	91%	78%	77%	75%	75%	74%	61%	60%
Log4j	4%	94%	78%	69%	80%	66%	61%	63%	60%	59%	48%	46%
Mahout	3%	98%	74%	55%	75%	51%	49%	46%	50%	45%	36%	36%
Mina	23%	77%	72%	58%	67%	55%	42%	44%	35%	41%	31%	26%
Openjpa	3%	91%	65%	71%	72%	66%	38%	50%	40%	42%	27%	27%
Pdfbox	9%	97%	36%	38%	36%	38%	20%	26%	20%	24%	13%	13%
Pig	5%	97%	79%	80%	79%	80%	62%	69%	62%	68%	48%	48%
Tika	25%	96%	90%	81%	91%	80%	68%	73%	68%	70%	53%	52%
Zookeeper	19%	99%	74%	66%	72%	64%	52%	53%	51%	53%	37%	37%
Cxf	7%	98%	60%	61%	62%	60%	38%	48%	40%	47%	31%	29%
Camel	13%	80%	56%	56%	57%	54%	37%	46%	39%	47%	25%	28%
<b>Avg</b>	14%	92%	68%	63%	68%	61%	48%	52%	47%	50%	37%	35%

<sup>1</sup> The proportion of files involved in suspect dependencies in the system.  
<sup>2</sup> The bug rate of suspect dependencies as the precision.  
<sup>3</sup> The coverage of files within suspect dependencies on top x% bug-prone files or change-prone files in terms of frequency or churn. The x% can be 10%, 30%, or 100% (All).  
<sup>4</sup> 'All' represents the results of all the bug-prone or change-prone files. Specifically, 'bug' represents the coverage results on all the bug-prone files (BF<sub>100%</sub> or BC<sub>100%</sub>). 'change' represents the coverage results on all the change-prone files (CF<sub>100%</sub> or CC<sub>100%</sub>).

TABLE 9. Early detection of groups of flawed structural dependencies in Flume.

Flume			Flawed Structural Dependencies <sup>1</sup>											
			Context (8)			Configurable (11)			Event (20)			Sink (9)		
Version	Year	Mon	BC <sup>2</sup>	Our	Hotspot	BC <sup>2</sup>	Our	Hotspot	BC <sup>2</sup>	Our	Hotspot	BC <sup>2</sup>	Our	Hotspot
1.3.0	2012	Dec	-	✓	×	-	✓	×	-	×	×	-	×	×
1.3.1	2013	Jan	340	✓	×	570	✓	×	-	×	×	-	×	×
1.4.0	2013	July	875	✓	×	641	✓	×	-	×	×	-	×	×
1.5.0	2014	May	944	✓	×	392	✓	×	-	×	×	-	×	×
1.5.1	2014	July	745	✓	×	0	×	×	130	✓	×	-	×	×
1.5.2	2014	Nov	655	✓	✓	0	×	×	267	✓	×	-	×	×
1.6.0	2015	May	660	✓	✓	0	×	×	0	✓	×	-	✓	×
1.7.0	2016	Oct	55	✓	✓	248	✓	✓	915	✓	×	819	✓	×
Saving Bug Churn <sup>3</sup>			4274	4274	715	1851	1603	0	1312	1182	0	819	819	0

<sup>1</sup> We studied four groups of flawed structure dependencies. Each group presents the leading file and number of involved files.  
<sup>2</sup> The bug churn. The number of lines of code spent on fixing bugs.  
<sup>3</sup> The saving bug churn. Each element presents the possible saving lines of code on fixing bugs through our method or hotspot detection.

We intensively analyzed its 8 versions from 1.2.0 (released on July 26, 2012) to 1.7.0 (released on October 17, 2016). As presented in Table 5, there are 624 files and 81k lines of code involving 862 bugs. We further gather the information of each report and found that 25944 lines of code are consumed. We collected all the flawed structural dependencies and constructed its graph. We further use strongly connected component analysis to automatically detect several groups of dependencies to assist our exploration. We employed Networkx [25], a state-of-the-art graph analysis tool, to support strongly connected component analysis. Thus, we automatically identified four groups of flawed structural dependencies shown in Table 9. Each group has only a leading file. We iteratively employ hotspot [38] and our method in these 8 versions. From 1.3.0 to 1.7.0, we compare the detected results of these two approaches, which are illustrated in Table 9.

Two groups of flawed structural dependencies have been both detected by hotspot and our method, 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 within these two groups, which are 17% and 7% of the entire maintenance cost.

The group of flawed structural dependencies led by *Context* is long-lived, 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 it in 1.3.0. The group of flawed structural dependencies of *Configurable* also causes a maintenance problem from 1.3.0. Some files in this group are removed temporarily in 1.5.1 for some reason but they have soon been introduced again in 1.7.0. Hotspot only identifies this group in 1.7.0. On the contrary, our method identifies this group from 1.3.0. In total, our method presents 5 and 7 versions earlier than hotspot in these two groups of flawed structural dependencies.

The other two groups of flawed structural dependencies are detected by our method but missed by 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. Both of these two groups of *Event* and *Sink* are generated for introducing new features in 1.5.1 and 1.7.0 separately. All of them are missed by Hotspot. On the contrary, our method does identify them as they first emerge.

We find that there are 8256 lines of code on maintaining these four groups of dependencies (4274+1851+1312+819). We assume that the maintenance costs of dependencies and related files can be saved if they are discovered in the previous version. Based on this hypothesis, we observed that our method would save as many as 7878 lines of code to fix bugs, taking up 95% of all the maintenance costs in this group (7878/8256). However, hotspot only saves 715 lines of code, taking 8% of all the maintenance costs (715/8256).

**Answer to RQ3:** The results show that our method can detect flawed structural dependencies as soon as they were first introduced, which is much earlier than the state-of-the-art technique: hotspot. Our method would help developers save a large number of lines of code to fix bugs by early detecting these dependencies.

## VI. DISCUSSION

In this section, we discuss the rationale of our method, the application of our detection result, and the threats to validity.

### A. THE RATIONALE OF OUR METHOD

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

1) **Code Clone.** One common reason for the suspect file pair (SFP) is the code clone. There exist several similar pieces of code fragments between two files with code clones. When the code in one file is modified, the other file also needs similar changes to re-synchronize the cloned code. For example, in Cassandra 3.6.0, we detect almost 40 lines of cloned code fragments between *DateTieredCompactionStrategy* and *TimeWindowCompactionStrategy*. These two files incur repeated changes in the subsequent versions. It suggests the cloned code should be extracted as the common interface or utility method. Refactoring and managing code clones is an always challenging task in the software community. To the best of our knowledge, the state-of-the-art technique to refactor clones is using lambda expression [82], which is proved to be simple and efficient. If developers use version control system like Git and SVN, this tool can be easy to be integrated as the plugin to implement just-in-time refactoring before committing code.

2) **Poor Inheritance.** Poor inheritance is also a significant reason for suspect file pairs. There always exist more than two subclasses redundantly extended from one base class. Thus, these subclasses may frequently incur similar bugs and changes for the overload inheritance. For example, in Log4j 1.3.0, *LiteralPatternConverter* and *NamedPatternConverter* are two subclasses inherited from the base class: *PatternConverter*. However, these two subclasses share more than half of the inherited variables and methods. It implies that the base class should be decoupled into a few simple interfaces. Thus, the subclasses can flexibly implement

28868

inheritance from them. Thus, in Log4j 1.3.8, these two subclasses, *LiteralPatternConverter* and *NamedPatternConverter*, are refactored by implementing a lightweight interface: *LoggingEventPatternConverter*.

3) **Implicit Dependency.** The implicit dependency, e.g. reflection and multi-thread, is also a reason for suspect file pair (SFP). In actual, some identified suspect file pairs (SFP) are connected with the implicit dependencies, which are missed by our employed detection tool: *Depends*. Since these dependencies are unconscious to discover, it may frequently cause underlying maintenance problems. For example, in PDFBox 2.0.1, *XMPBasicSchema* and *XMPSchemaFactory* have the relevance of reflection (a type of implicit dependencies), which are not captured by *Depends*. The revision history demonstrates these files consuming increasing costs.

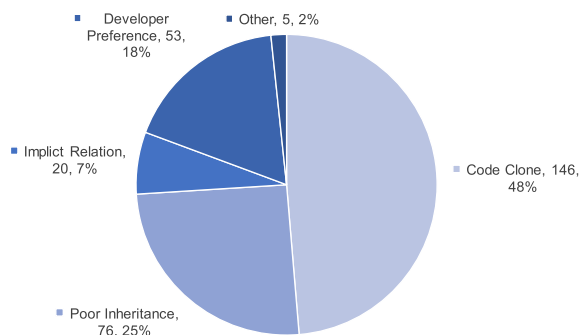
4) **Developer Preference.** Developer preference is a human reason for suspect file pairs (SFP). Suspect file pairs in this category are caused by neither code clone nor implicit dependencies. The involved files always have a higher complexity. In the modification requests and comments, these files are discussed together by developers. It seems that these file pairs share similar functionality. 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 suspect file pair (SFP) which are often discussed by developers in comments. As a consequence, they are also soon to be reconstructed.

In summary, we manually inspect 300 instances of suspect file pairs (SFP) and summarize four possible explanations of suspect file pairs (SFP). The proportion of each reason is demonstrated in Figure 6. We observed that the code clone captures the most cases (48.7%). A large amount of these suspect file pairs (SFP) are triggered by poor inheritance (25.3%) and developer preference (10.6%). Only 6.7% of them are caused by implicit dependencies. With the assistance of these four summarized reasons, we can explain the 98.3% of the sampled suspect file pairs (SFP). Automatically classifying reasons for SFP with machine learning techniques is our ongoing work. We design this workflow as follows: First, we collected the dataset including suspect file pairs and related reasons. Then, we employ state-of-the-art code embedding techniques such as PathMiner [83] to extract features for a file pair. Finally, we use classification techniques like CNN and RNN to implement automatic classification of reasons for suspect file pairs.

### B. APPLICATIONS OF OUR DETECTION RESULTS

Although there exist several differences between our method and DL measurement or hotspot detection, our detection results can be combined to improve these techniques.

For the DL measurement, since our method is not designing a metric to improve measurement, our method cannot support the comparison of various projects like DL. When we use the DL measurement and our method to keep observing a project, our detection results can be combined to diagnose. Our results can be combined with the variation of DL



**FIGURE 6.** Four possible explanations for suspect file pairs (SFP) (Each explanation contains three items: the name, the number of instances and its proportion).

measurement to illustrate and locate the dependency problem. Our results can also be combined with the stability of DL measurement to implement early warning.

For hotspot detection, when we use our method and hotspot detection to keep monitoring a project, our detection results can further assist to prioritize the hotspot results. We classify the interaction of our method and hotspot into three categories: 1) detection results captured by both hotspot and our method. These results should be assigned to the highest priority. It implies that these results have caused severe consequences and may have a persistent impact on software maintenance in the future; 2) detection results captured by our method not hotspot. These results should be added and assigned with moderate priority. It means that these results should be further observed and can be fixed if possible; 3) detection results captured by hotspot not our method. These results should be assigned to the lowest priority. A possible reason is that they have been already fixed. We should keep tracing them and prioritize them in case. In summary, leveraging our method to further improve the existing technique: hotspot is our ongoing work.

### C. RUNTIME ANALYSIS

We conducted a competitive analysis of hotspot and our method in run time. The experiment environment is a 3.2GHz i7-3930K desktop with 12 logical cores, 6 physical cores, and 32GB of memory. We measure the run time of 8 versions of Flume [9] in Section V.C using hotspot and our method separately. The results suggest that, on average, for a version of Flume, hotspot consumes 69.6 seconds to finish the complete detection while our method merely takes 30.4 seconds. The performance of our method improved by 128% compared with hotspot. One possible reason is that hotspot relies on the design rule hierarchy [40] technique to analyze all the dependencies while our method uses the efficient graph analysis. To further investigate the impact of different components in our method on performance is our ongoing work.

### D. LIMITATIONS AND THREATS TO VALIDITY

First, we conducted a large-scale detection of flawed structural dependencies candidates in more than 800 versions of the 15 open source projects. Because all of the selected subjects are Apache projects, it is unclear whether our

approach will generalize to other open source projects in a different community or closed source 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 method to more subjects.

Second, it is difficult to define what is the flawed structural dependencies. To limit this threat, we evaluate it from the maintenance cost including bugs and changes during evolution. In our paper, we consider that if a pair of files with dependencies are frequently involved in bug fixes, introduced with new features or updated, it has a high possibility to be problematic. Additionally, it is also challenging to evaluate whether our method can identify flawed structural dependencies 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. Besides these projects, we are also employing our approach in some young projects and reporting detected flawed structural dependencies to the community.

Third, our approach might be sensitive to the selection of thresholds. We employ two thresholds in extracting semantic dependencies. These thresholds are empirically set based on the work of Bavota *et al.* [58]. We also manually sampled and inspected the obtained dependencies to ensure their quality. The full results of thresholds in our experiments are available [22]. However, the best thresholds for various projects may be different. Therefore, automatically determining the best thresholds is also part of our future work.

Last, we currently evaluate our method to predict future bugs or changes using past bug fixes and changes. We detect issues in the past version and evaluate its impact on bugs and changes in the current version. We are considering using machine learning prediction techniques to further improve our evaluation. We also evaluate our method at the file level. It is imperative to further evaluate at the function level. The detection results can be validated and combined with cover coverage tools to report concrete bugs. It is our ongoing work to improve the effectiveness of our method.

## VII. RELATED WORK

**Coupling Metrics:** The concept of software coupling was first introduced by Steve *et al.* [28]. Based on this concept, Chidamber and Kemerer proposed a suite of related coupling metrics to evaluate the quality of object-oriented programs [64]. Briand *et al.* [51] also employed coupling metrics to inform the software high-level design. There are rich literature in various coupling metrics from multiple perspectives including software syntactic [64], revision history [76], source code lexicon [65], and dynamic execution [52]. These coupling metrics are designed to capture various aspects of software quality. Our work explores the combination of structural and semantic dependencies (coupling) between files to inform software quality in time.

**Bug Prediction:** Over the past decades, a plethora of research efforts focus on helping practitioners predict

bugs [29], [43], [66], [67], [71], [72]. For example, Selby and Basili [41] employed the dependency structure to improve the accuracy of predicting bugs. Nagappan *et al.* [44] made a comparison of complexity metrics and leveraged them in bug prediction techniques. Qu *et al.* [45] studied the structure of method invocations and their impact on bug prediction. Wang *et al.* [77] predict bugs 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 problematic structural dependencies.

**Software Textual Analysis:** Software textual analysis is widely used in refactoring [57], reverse-engineering [35], [36], bug localization [68], [69], and code search [70], [73]. By extracting textual features of the source file, software textual analysis gains new insight into software projects. For instance, Le *et al.* [68] leveraged textural features in locating the relevant files for a bug report. Zhang *et al.* [70] employed software textual analysis to improve the efficiency of the code search. In our work, we derive textual features from the source code lexicon and measure its similarity using information retrieval techniques as semantic dependencies.

**Code Smell:** The concept of code smell, also known as code anomaly, was firstly introduced by Fowler [50]. The code smell is an effective and efficient method to hint files with potential issues [46], [47], [53]–[56], [59]. Macia *et al.* [54] studied flawed dependencies among files by extending the definition of code smell at the architectural level. Oizumi *et al.* [47] studied flawed dependencies by clustering numerous code smells. In this paper, our detection results also are overlapped with some typical code smells such as code clones. We find that part of our results can also be detected by the clone detection tool: CCFindexr [74] and CloneDetection [75] for code clone is a reason for suspect file pairs (SFP). However, compared with some state-of-the-art code smell detection tools such as Sonar [3] and JDeodorant [4], most of our detected suspect dependencies cannot be captured.

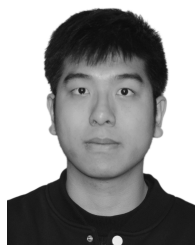
## VIII. CONCLUSION

In this paper, we proposed a method by combining structural and semantic dependencies to identify flawed structural dependencies candidates at an early stage during software evolution. We evaluated our method 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 flawed structural dependencies candidates effectively and timely. The detected results incurred 957% of bug frequencies and 1050% of change frequencies than average in the subsequent versions. We also find that our method merely uses 14% of files in the system to cover 70% of the top 10% bug-prone files with high precision (92%). These observations indicate the suspect dependencies, detected by our method, are shown to have the potential risk of causing maintenance problems. By early detecting and refactoring these dependencies in time, our method can assist software practitioners to save significant maintenance costs.

## REFERENCES

- [1] *Depends*. Accessed: 2020. [Online]. Available: <https://github.com/multilang-depends/depends>
- [2] *Srcml*. Accessed: 2020. [Online]. Available: <https://www.srcml.org>
- [3] *Sonarqube*. Accessed: 2020. [Online]. Available: <https://www.sonarqube.org>
- [4] *JDeodorant*. Accessed: 2020. [Online]. Available: <http://www.jdeodorant.com>
- [5] *Git*. Accessed: 2020. [Online]. Available: <https://git-scm.com>
- [6] *JIRA*. Accessed: 2020. [Online]. Available: <https://issues.apache.org>
- [7] *Avro*. Accessed: 2020. [Online]. Available: <http://avro.apache.org>
- [8] *Cassandra*. Accessed: 2020. [Online]. Available: <http://cassandra.apache.org>
- [9] *Flume*. Accessed: 2020. [Online]. Available: <http://flume.apache.org>
- [10] *Hadoop*. Accessed: 2020. [Online]. Available: <http://hadoop.apache.org>
- [11] *HBase*. Accessed: 2020. [Online]. Available: <http://hbase.apache.org>
- [12] *Log4j*. Accessed: 2020. [Online]. Available: <http://logging.apache.org>
- [13] *Mahout*. Accessed: 2020. [Online]. Available: <http://mahout.apache.org>
- [14] *Mina*. Accessed: 2020. [Online]. Available: <http://mina.apache.org>
- [15] *OpenJPA*. Accessed: 2020. [Online]. Available: <http://openjpa.apache.org>
- [16] *Pdftbox*. Accessed: 2020. [Online]. Available: <http://pdftbox.apache.org>
- [17] *Pig*. Accessed: 2020. [Online]. Available: <http://pig.apache.org>
- [18] *Tika*. Accessed: 2020. [Online]. Available: <http://tika.apache.org>
- [19] *Zookeeper*. Accessed: 2020. [Online]. Available: <http://zookeeper.apache.org>
- [20] *Cxf*. Accessed: 2020. [Online]. Available: <http://cxfr.apache.org>
- [21] *Camel*. Accessed: 2020. [Online]. Available: <http://camel.apache.org>
- [22] *Dataset*. Accessed: 2020. [Online]. Available: <https://github.com/cuidi34/EarlyDetectionData>
- [23] *Scikit*. Accessed: 2020. [Online]. Available: <https://scikit-learn.org>
- [24] *RefactoringNavigator*. Accessed: 2020. [Online]. Available: <https://github.com/llmhyy/Refactoring-Navigator>
- [25] *Networkx*. Accessed: 2020. [Online]. Available: <https://networkx.org>
- [26] *Siemens*. Accessed: 2020. [Online]. Available: <https://siemens.com>
- [27] D. Cui, T. Liu, Y. Cai, Q. Zheng, Q. Feng, W. Jin, J. Guo, and Y. Qu, "Investigating the impact of multiple dependency structures on software defects," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 584–595.
- [28] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 38, nos. 2–3, pp. 231–256, 1974.
- [29] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, 2005, pp. 284–292.
- [30] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyeve, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2. Firenze, Italy: IEEE Press, May 2015, pp. 179–188.
- [31] M. Yamamoto and K. W. Church, "Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus," *Comput. Linguistics*, vol. 27, no. 1, pp. 1–30, Mar. 2001.
- [32] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: A new metric for architectural maintenance complexity," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 499–510.
- [33] L. Xiao, Y. Cai, and R. Kazman, "Design rule spaces: A new form of architecture insight," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 967–977.
- [34] Y. Cai, H. Wang, S. Wong, and L. Wang, "Leveraging design rules to improve software architecture recovery," in *Proc. 9th Int. ACM Sigsoft Conf. Qual. Softw. Archit. (QoSA)*, 2013, pp. 133–142.
- [35] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 486–496.
- [36] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2011, pp. 552–555.
- [37] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 488–498.
- [38] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Proc. 12th Work. IEEE/IFIP Conf. Softw. Archit.*, May 2015, pp. 51–60.
- [39] Y. Cai and K. Sullivan, "Modularity analysis of logical design models," in *Proc. 21st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2006, pp. 91–102.

- [40] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2009, pp. 197–208.
- [41] R. W. Selby and V. R. Basili, "Analyzing error-prone system structure," *IEEE Trans. Softw. Eng.*, vol. 17, no. 2, pp. 141–152, Feb. 1991.
- [42] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. Int. Conf. Softw. Maintenance*, 1998, pp. 190–198.
- [43] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.
- [44] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th Int. Conf. Softw. Eng. (ICSE)*, 2006, pp. 452–461.
- [45] 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," *J. Syst. Softw.*, vol. 108, pp. 193–210, Oct. 2015.
- [46] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Proc. 13th Eur. Conf. Softw. Maintenance Reeng.*, 2009, pp. 255–258.
- [47] 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 *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 440–451.
- [48] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*, vol. 1. Cambridge, MA, USA: MIT Press, 2000.
- [49] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm. ACM*, vol. 15, no. 3, pp. 1–50, 1972.
- [50] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Berlin, Germany: Springer, 2002.
- [51] L. C. Briand, S. Morasca, and V. R. Basili, *Defining and Validating Measures for Object-Based High-Level Design*. Piscataway, NJ, USA: IEEE Press, 1999.
- [52] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, *Predicting the Probability of Change in Object-Oriented Systems*. Piscataway, NJ, USA: IEEE Press, 2005.
- [53] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2012, pp. 662–665.
- [54] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2012, pp. 277–286.
- [55] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems," in *Proc. 11th Annu. Int. Conf. Aspect-Oriented Softw. Develop. (AOSD)*, 2012, pp. 167–178.
- [56] W. Oizumi, A. Garcia, M. Ferreira, and A. Von Staa, "When code-anomaly agglomerations represent architectural problems? An exploratory study," in *Proc. Softw. Eng.*, 2014, pp. 91–100.
- [57] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Software re-modularization based on structural and semantic metrics," in *Proc. 17th Work. Conf. Reverse Eng.*, Oct. 2010, pp. 195–204.
- [58] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Softw. Eng.*, vol. 18, no. 5, pp. 901–932, Oct. 2013.
- [59] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing four approaches for technical debt identification," *Softw. Qual. J.*, vol. 22, no. 3, pp. 403–426, Sep. 2014.
- [60] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," *ACM SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 99–108, Sep. 2001.
- [61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Professional Computing Series), vol. 49, no. 2. Addison-Wesley, 1995, pp. 241–276.
- [62] N. Kambhatla, "Combining lexical, syntactic, and semantic features with maximum entropy models for extracting relations," in *Proc. ACL Interact. Poster Demonstration Sessions*, 2004, p. 22.
- [63] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under\_score," in *Proc. IEEE 17th Int. Conf. Program Comprehension*, May 2009, pp. 158–167.
- [64] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 11, pp. 197–211, Jun. 1994.
- [65] D. Poshyvanik and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proc. 22nd IEEE Int. Conf. Softw. Maintenance*, Sep. 2006, pp. 469–478.
- [66] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 78–88.
- [67] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan, "High-impact defects: A study of breakage and surprise defects," in *Proc. ACM SIGSOFT Symp. Eur. Conf. Found. Softw. Eng.*, 2011, pp. 300–310.
- [68] T.-D.-B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, Aug. 2015, pp. 579–590.
- [69] 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 *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 14–24.
- [70] 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 *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 260–270.
- [71] 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 *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Sep. 2010, pp. 1–10.
- [72] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. E. Hassan, "Explaining software defects using topic models," in *Proc. 9th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, Jun. 2012, pp. 189–198.
- [73] S. P. Reiss, "Semantics-based code search," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 243–253.
- [74] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.
- [75] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Proc. Source Code Anal. Manipulation, 4th IEEE Int. Workshop*, 2004, pp. 128–135.
- [76] S. Wong and Y. Cai, "Generalizing evolutionary coupling with stochastic dependencies," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2011, pp. 293–302.
- [77] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 297–308.
- [78] R. Mo and M. Zhan, "History coupling space: A new model to represent evolutionary relations," in *Proc. 26th Asia-Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2019, pp. 126–133.
- [79] Y. Zhao, L. Xiao, X. Wang, L. Sun, B. Chen, Y. Liu, and A. B. Bondi, "How are performance issues caused and resolved?—An empirical study from a design perspective," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, Apr. 2020, pp. 181–192.
- [80] Q. Feng, R. Kazman, Y. Cai, R. Mo, and L. Xiao, "Towards an architecture-centric approach to security analysis," in *Proc. 13th Work. IEEE/IFIP Conf. Softw. Archit. (WICSA)*, Apr. 2016, pp. 221–230.
- [81] W. Jin, Y. Cai, R. Kazman, Q. Zheng, D. Cui, and T. Liu, "ENRE: A tool framework for extensible eNtity relation extraction," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Engineering: Companion Proc. (ICSE-Companion)*, May 2019, pp. 67–70.
- [82] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone refactoring with lambda expressions," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 60–70.
- [83] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, "PathMiner: A library for mining of path-based representations of code," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, May 2019, pp. 13–17.



**DI CUI** is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. His research interests include software maintenance and evolution, and architecture recovery of software systems.

...