2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Investigating the Impact of Multiple Dependency Structures on Software Defects

Di Cui*, Ting Liu*, Yuanfang Cai†, Qinghua Zheng*, Qiong Feng†, Wuxia Jin*, Jiaqi Guo*, Yu Qu*
*School of Electronic and Information Engineering, Xian Jiaotong University, Xian 710049, China
{cuidi, wx_jin, jasperguo2013}@stu.xjtu.edu.cn; {tingliu, qhzheng. quyuxjtu}@mail.xjtu.edu.cn
†Department of Computer Science, Drexel University, Philadelphia, USA
{yc349, qf28}@drexel.edu

***Abstract*—Over the past decades, numerous approaches were proposed to help practitioner to predict or locate defective files. These techniques often use syntactic dependency, history co-change relation, or semantic similarity. The problem is that, it remains unclear whether these different dependency relations will present similar accuracy in terms of defect prediction and localization. In this paper, we present our systematic investigation of this question from the perspective of software architecture. Considering files involved in each dependency type as an individual design space, we model such a design space using one *DRSpace*. We derived 3 DRSpaces for each of the 117 Apache open source projects, with 643,079 revision commits and 101,364 bug reports in total, and calculated their interactions with defective files. The experiment results are surprising: the three dependency types present significantly different architectural views, and their interactions with defective files are also drastically different. Intuitively, they play completely different roles when used for defect prediction/localization. The good news is that the combination of these structures has the potential to improve the accuracy of defect prediction/localization. In summary, our work provides a new perspective regarding to which type(s) of relations should be used for the task of defect prediction/localization. These quantitative and qualitative results also advance our knowledge of the relationship between software quality and architectural views formed using different dependency types.**

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

I. Introduction

Over the past decades, numerous approaches were proposed to help practitioner to predict or locate defective files [1–7], and various dependency structures were used as the basis for bug prediction/localization algorithms. To the best of our knowledge, three types of dependency relations have been frequently and intensively used by researchers: syntactic relation (derived from source code, such as inheritance and method call), history relation (derived from the commit history of a project, mainly co-change frequency among files), and semantic relation (derived from identifiers and comments, and calculate the similarity among software entities). All of these three relations have been used as features to predict or locate software defect. The problem is that, it is not clear if and to extent these dependency relations differ from each other. Prior research focused on prediction/localization algorithms, but not on the underlying dependency types, a more fundamental question.

Ting Liu is the corresponding author.

In this paper, we explore the following research questions:

For these three dependency relations, to what extent are their structures similar to each other? There are a large number of publications on using one of them to predict/locate defects and reported reasonably performance. It is intuitive that these dependency structures should be highly similar. We are interested in quantifying the similarity among these dependency structures to test this intuition.

If and to what extent will these three dependency relations present similar performance in defect prediction and localization tasks? It is unclear whether they have similar relations with error-prone files. If the answer is yes, it means that there is no fundamental difference among these dependency relations. However, if the answer is no, we need to further investigate the next question.

Whether the combination of these three dependency structures has the potential to improve the performance of defect prediction and localization? If the answer is no, it implies that these different structures can not be effectively and efficiently used together, and we should consider which one to choose. If yes, we further explore how the intersection and union of these three relations contribute to software defects prediction/localization.

To answer these questions, we need to systematically investigate the impact of these dependency structures and their relations with defective files. The ideal approach would be using these dependency types to conduct bug prediction/location analyses and make a comparison. However, it is not realistic given the large number of existing tools and algorithms. Moreover, these prediction/classification models are black boxes, hiding the intrinsic relations between these software structures and defective files. Instead, we try to reveal the relations by reversely explaining defects from the perspective of software architecture views (white box), as shown in Fig. 1.

We employ the *DRSpace* [8, 9] model, a state-of-the-art reverse engineering and architecture modeling technique, to support our study. A *DRSpace* can be used to model the overall software architecture as multiple overlapping design spaces. Each design space can only capture one of many possible relations. The *DRSpace* model is also an effective approach to analyzing software structures in a fine-grained way.

In this paper, we report our comprehensive empirical study to investigate the relation of three dependency types (syntactic, history, and semantic) with software defects. We intensively studied 117 Apache open source projects with 643,079 revision commits and 101,364 bug reports. Using *DRSpaces*, for each dependency type, we calculate the interaction between its design space and defective files. Based on these data, we answer the three research questions as follows:

First, comparing the structures formed by syntactic relation, history relation, and semantic relation, only 25% of these structures are similar. This result implies that these three dependency relations form drastically different architecture views. Intuitively, their effectiveness on defect prediction/localization should be very different and it is imperative to explore their influences on software defects.

Second, the syntactic, history, and semantic relations capture different subsets of defective files. The syntactic structure captures the largest number of defective files but with the lowest accuracy. By contrast, the history structure covers the least number of defective files but with high accuracy. The semantic structure is in-between. It implies that, although there are rich literature on using one dependency structure to predict/locate software defects, we should be aware of their significant differences. We need a comprehensive understanding regarding to the relation between dependency types and defects.

Third, the combination of syntactic structure, history structure, and semantic structure has the potential to effectively improve the performance of defect prediction/localization: the union of them can cover most of the defective files, and their intersections can capture files with most severe problems. We present the detailed results in Section III. The design of strategies to flexibly combine these three structures to improve the prediction/localization accuracy is future work.

The contribution of our work is:

- A systematic comparative analysis of different types of dependencies and their impact on defects. Our empirical study has revealed, for the first time, their drastically different relations with defective files, which advanced our understanding of three dependency types.
- A new perspective to analyze the software defects. Our empirical study revealed that different dependency structure captures drastically different subsets of defective files. When a bug prediction/localization algorithm is devised, the designer should take their differences into account. The union of all three relations can capture more buggy files, but their intersection may capture most severe bugs. Each option comes with costs and benefits.
- A benchmark to investigate the relation between software defects and various dependency structures. We collected 117 Apache open projects involving 643,079 revision commits, and 101,364 bug reports. All original data and extracted dependencies are publically available[1].

The rest of the paper is organized as follows: Section II presents the related work. Section III and IV introduce our methodology and experiment results. Section V presents the discussion. Section VI presents the threats to validity. Section VII concludes the paper.

[1]https://github.com/cuidi34/ICSE19-Data.git

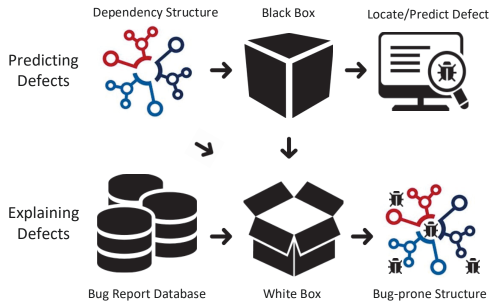


Fig. 1: Predicting Defects vs Explaining Defects

II. Related Work

A. Using Dependency Structures to Predict Defects

Over the past decades, using various dependency structures to predict software defects has been widely studied [1–7]. We summarized classical defect prediction framework in Fig. 2, which contains three steps: 1) generating dependency structures, 2) deriving structural measurements, and 3) training prediction models. In each step, researchers designed various strategies to improve the accuracy of defect prediction. For example, within the first two steps, Selby and Basili [1] first studied syntactic metrics to predict bug-prone files. Nagappan et al. [2] also derived complexity metrics based on syntactic structures to improve prediction accuracy. Zimmermann et al. [3] reported that syntactic-based network measures could be used to construct successful defect predictor. Furthermore, Cataldo et al. [4] derived change density metrics to measure history structures for defect prediction. Lin et al. [6] used advanced machine learning algorithms to automatically measure semantic structures to improve prediction accuracy. For the third step, researchers also employed several machine learning algorithms to train classification models to improve the accuracy of defect prediction. For instance, Li et al. [10] leveraged ensemble and kernel learning algorithms in defect prediction to improve the training model. Di Nucci [11] proposed an adaptive method to dynamically select classifiers for defect prediction tasks. Our work aims to advance our understanding regarding to the impact of different software dependency types on defect prediction/localization, but not to propose a new prediction/localization approach.

B. Reverse Engineering Techniques

Reverse-engineering is a technique to recover high-level design from source code. Over the past decades, there are rich literature about reverse engineering techniques [12–15]. These techniques first aggregate program entities, such as files or classes, into modules based on different rationales. For example, *Algorithm for Comprehension-driven Clustering* (*ACDC*) [12] is a pattern-driven technique proposed by Tzperpos and Holt, which clusters entities based on naming conventions and syntactic structures. *Architecture Recovery Using Concerns*

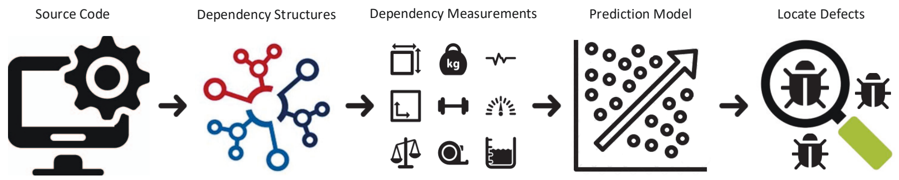


Fig. 2: Classic Defect Prediction Framework

(*ARC*) [13] is a NLP-based techniques proposed by Joshua et al. that leverages semantic structures. Back et al. [16] first leveraged the combination of history and syntactic structures to improve the quality of recovered design. Mitchell et al. also proposed a genetic method named *Bunch*, which improves reverse engineering results by optimizing objective functions [17]. Researchers also made systematic comparisons of these techniques: Nenad et al. [18] conducted a comparative study of 10 reverse engineering techniques and claimed that *ARC* and *ACDC* outperformed others in terms of accuracy. Luttellier et al. [19] conducted another comparative study and made similar observations. Our work uses reverse engineering methods to understand different dependency types but not to recover architecture.

C. Architectural Smells

Architectural smells were proposed to describe problematic relations among files that may have negative impact on software quality. Researchers proposed architectural smells based on different dependency types. For instance, Garcia et al. [20] first defined a suite of code smells based on syntactical structures. Ran et al. [21] integrated history and syntactical relations and proposed a suite of architectural smells called *Hotspot Patterns*. Oizumi [22] studied architectural smells by clustering classical code smells using syntactic and semantic structures. Fontana et al. [23, 24] summarized the previous work and developed a tool named ARCAN to support the automatic detection of these smells. Our work explores the independence and combination of different dependency types but not summarizing architectural anti-patterns.

III. Methodology

In this section, we present our systematical study regarding to the impact of different dependency structures on software defects. It is not possible for us to directly answer this question by executing defect prediction tools given the large number of them. Instead, we leverage DRSpace, a state-of-the-art reverse engineering technique, to investigate different types of relations among files, and their interaction with the set of buggy files. Fig. 3 shows the overview of our research framework that includes three components: *Data Collection*, *DRSpaces Generation* and *Issue Analysis*.

A. Data Collection

Here we introduce the subjects we studied, the three types of relation we extracted, and the defect information collected from these subjects.

*1) **Subjects**:* We studied the latest versions of 117 projects from the Apache community[2]. These projects vary in sizes, domains and functionalities, and most of them are widely used in defect prediction research. The number of files in these projects varies from 236 to 7216. The number of lines of code (LOC) in these projects varies from 11K to 1M. We additionally crawled the revision history of these projects from the version control system (Git[3]), and bug reports from the issue tracking system (JIRA[4]). We only study bug reports that have been fixed. In total, we collected 643,079 commits and 101,364 valid bug reports.

*2) **Software Relation Extraction**:* For each subject, we extract its structural relation, history relation, and semantic relation as follows:

a) Syntactic Relation: Syntactic dependency is the most commonly used relation among source files, including inheritance, implementation, method call, field access, type reference, instance creation, etc. We employed *Understand*[5], a commercial reverse-engineering tool, to extract structural relations among source files, and denote the collected syntactic relation as E_1.

b) History Relation: History relation, also known as *evolutionary coupling*, is derived from the revision history of a project, modeling the co-change probability among files. In our study, we employed the history coupling probability (*HCP*) matrix [25], a conditional probability model to manifest how likely a change to a file may influence other files. We implemented this algorithm and configure the *HCP* parameters following the work of Xiao et al. [25]. We denote the collected history relation as E_2.

c) Semantic Relation: Semantic relation is derived from the source code lexicon to capture the textual similarity between files. Here we employed *relation topic model* (*RTM*) [26, 27], a probabilistic topic modeling technique, to capture the semantic relation among files based on source code identifiers and comments. We crawled the lexical information using the lexicon parser in *Understand*, and implemented *RTM* using the *lda*[6] package of R. We configure the *RTM* parameters following the work of Bavota et al. [28], and denote the collected semantic relation as E_3.

We can thus model a software system as a directed multigraph: (V, E_1, E_2, E_3), where V represents the set of files within the system, and E_1, E_2, E_3 represent the syntactic

[2]http://www.apache.org/
[3]https://git-scm.com/
[4]https://www.atlassian.com/software/jira
[5]https://scitools.com/
[6]http://cran.rproject.org/web/packages/lda/

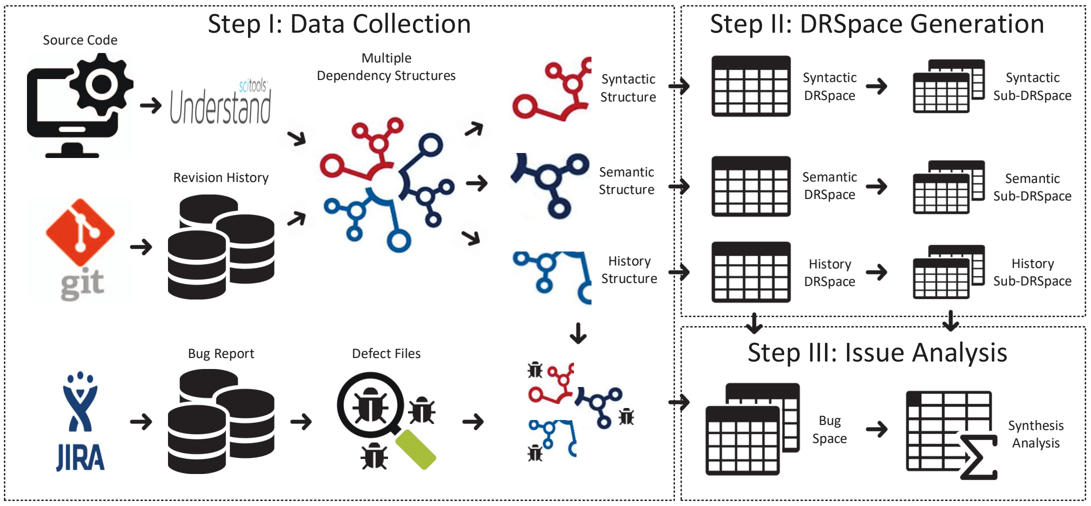


Fig. 3: Research Framework

relation, history relation and semantic relation among these files respectively.

*3) **Defect Information Extraction**:* From each project, we also extracted defect information from bug reports recorded in its issue tracking system. For each bug report (BR_i), we collect the set of files changed to fix it, and the lines of code (LOC) spent on these fixes. We model these data as follows:

$$BR_i = \{(F_j, Churn(BR_i, F_j)) \mid j = 1, 2, ..., m\} \qquad (1)$$

where m represents the number of files involved in the bug (BR_i). Here each bug is modeled as a set of pairs, each containing two elements: F_j and $Churn(BR_i, F_j)$, representing the fixed file and the number of lines of code spent on it to fix the bug.

We also model the relation between bugs and source files as follows:

$$F_j = \{(BR_i, Churn(BR_i, F_j)) \mid i = 1, 2, ..., n\} \qquad (2)$$

where n represents the number of bugs a file (F_j) is involved. Here each file is mapped to a set of pairs, each containing two elements: BR_i and $Churn(BR_i, F_j)$, representing an involving bug and the number of lines of code spent to fix it.

B. DRSpace Generation

In this section, we introduce our method of managing various dependency structures using *Design Rule Spaces* (DRSpaces) [8], a new architecture model.

*1) **Generating DRSpace**:* Xiao et al. proposed that a software system can and should be viewed as multiple overlapping design spaces that can be reverse-engineered from source code. For example, a feature implemented, or a pattern applied, can be viewed as an individual design space that contains all the files participating in the feature/pattern [8]. They also mentioned that different dependency types, such as inheritance and method call, can form their own design spaces.

Based on this rationale, they proposed a *Design Rule Space* (DRSpace) [8] model, each capturing one design space of the system. In a DRSpace, the files are clustered into hierarchical layers, and each layer is decoupled into independent modules. Files in lower layers only depend on the files in higher layers. Files in different modules within the same layer are mutually independent from each other. Moreover, each DRSpace has one or a few *leading files*, and all other files are directly or indirectly depend on the leading files. For example, an observer pattern DRSpace must contain an observer interface, upon which all other pattern participants depend. The files within a DRSpace can have one or more types of dependencies. This architecture model is consistent with the definition proposed by Bass et al. [29]: the architecture of a software system is a set of structures, and each DRSpace is one of many module structures within the system.

In our study, we examine a software structure through the lenses of DRSpaces. For each subject, using the three types of relations extracted, we model the project using a suite of DRSpaces: *Syntactic DRSpaces* (*SynDR*), *History DRSpaces* (*HisDR*), and *Semantic DRSpaces* (*SemDR*), in which source files only have syntactic relation, history coupling relation, and semantic coupling relation respectively.

*2) **Generating Sub-DRSpace**:* Using each file as a leading file, we further split each DRSpace into a set of sub-DRSpaces:

$$DRSpace = \left\{sub\text{-}DRSpace_j \mid j = 1, 2, ..., m\right\} \qquad (3)$$

where m is the total number of sub-DRSpaces. Each sub-DRSpace ($sub\text{-}DRSpace_j$) consists of two elements:

$$sub\text{-}DRSpace_j = (F_j, Suboridnate(F_j)) \qquad (4)$$

where F_j represents the leading file of the sub-DRSpace and $Suboridnate(F_j)$ represents the files that directly or indirectly depend on the leading file.

For each subject with x files that have syntactic relation with other files, y files having history coupling with other files, and z files having semantic relation with other files, we will generate x *syntactic sub-DRSpaces*, y *history sub-DRSpaces*, and z *semantic sub-DRSpaces*.

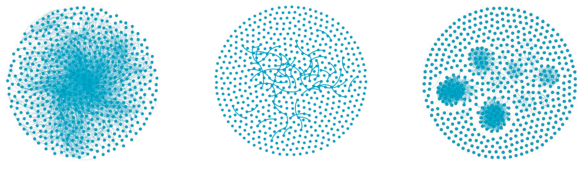

(a) Syntactic (b) History (c) Semantic

Fig. 4: Three Dependency Structures of Log4j

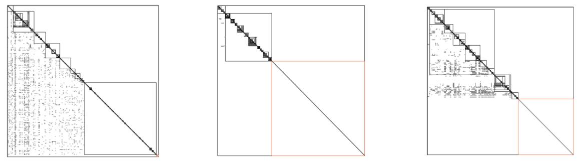

(a) Syntactic (b) History (c) Semantic

Fig. 5: Three DRSpaces of Log4j

C. Issue Analysis

For each subject, given the three DRSpaces and their sub-DRSpaces, we now present our method of analyzing their relations with defective, i.e., error-prone files.

Following the work of Xiao et al. [8], we first use a *BugSpace* to model the set of files involved in bug fixing. We use *BugFre_x%ile* to model the set of files within the top *xth* percentile of a *BugSpace*, ranked based on the number of times they were changed for bug-fixing (bug frequency). We also use *BugChurn_x%ile* to model the most expensive files, ranked within the top *xth* percentile based on their lines of code (LOC) spent on bug fixing (bug churn).

ArchRoot [8, 30] is a greedy algorithm designed to extract a set of sub-DRSpaces that covers a target set of error-prone files, that is, files within a bug space *BugFre_x%ile*, or *BugChurn_x%ile*. The input of this algorithm is a DRSpace containing all the files within the system, and a target bug space. *ArchRoot* iteratively inspects each sub-DRSpace and calculates its intersection with the target bug space, and return a set of sub-DRSpaces. In our study, we denote the union of all the files within the sub-DRSpaces calcuated from *ArchRoot* as a *RootSpace*.

D. Running Example

In this section, we demonstrate these concepts and our method using a running example, Log4j 2.8.2[7], a distributed logging system.

Data Collection. We extracted the syntactic relation and semantic relation from Log4j, version 2.8.2. In this snapshot, there are 1702 files, and 8823 pairs of them have structural relations, and 2678 pairs have at least 70% semantic similarity.

To extract history coupling relation, we collect 9425 revision commits from May 2010 to Aug 2017. Within this time period, there are 137 file pairs having at least 30% co-change probability. Within the same history period, we collected 634 bug reports that have been resolved, and there are 136 files changed to fix these bugs.

[7]https://logging.apache.org/log4j/2.x/

We employ Gephi[8], an open source graph visualization platform, to model these three relations as graphs, as shown in Fig. 4 (a)-(c). These graphs depict the *syntactic structure*, *history structure*, and *semantic structure* among Log4j source files, in which vertexes represent files and edges represent different relations among them. As we can see from these graphs, the overall structure formed by these relations are very different.

DRSpace and Sub-DRSpace Extraction. For each relation, we generate its *DRSpace* using Titan [31], a visualization tool that present file structures using *Design Structure Matrices* (DSMs). A DSM is a square matrix in which rows and columns are labeled by the same set of files in the same order. 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 cells can be used to denote different types of relations. Fig. 5.(a)-(c) depicts the *Syntactic DRSpace*, *History DRSpace* and *Semantic DRSpace* of Log4j. Similar to Fig. 4, we can observe that these structures are drastically different.

For each *DRSpaces*, we decouple it into a set of *sub-DRSpaces*, each led by one file within the DRSpace. Fig. 6-8 depicts the syntactic sub-DRSpace, history sub-DRSpace and semantic sub-DRSubpaces led by the same leading file: *FileManager* of Log4j. We explain these DSMs as follows:

Fig. 6 shows the syntactic sub-DRSpace led by *FileManager*. *Cell*$(5, 1)$ is labeled with "*dp*", which means that *DefaultRolloverStrategy* syntactically depends on *FileManager*. Fig. 7 shows its history sub-DRSpace. *Cell*$(5, 1)$ is labeled with "$(hc, 30\%), \#7$". "$(hc, 30\%)$" means that *DefaultRolloverStrategy* is historically coupled with *FileManager* and the co-change probability is 30%. "#7" means that these two files changed together 7 times as recorded in the commit history. Fig. 8 shows the semantic sub-DRSpace led by the same file. *Cell*$(5, 1)$ is labeled with "$(sc, 82\%)$", which means that *DefaultRolloverStrategy* has 82% lexical similarity with *FileManager*.

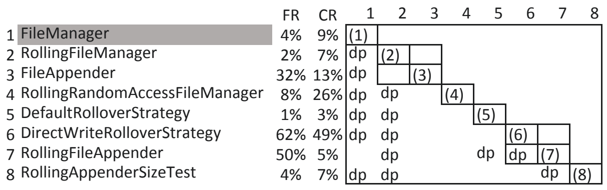
Issue and Root Analyses. The sub-DRSpaces shown in Fig. 6-8 are among the output sub-DRSpaces of the ArchRoot algorithm. The input of this algorithm includes each DRSpace and all the files involved in bug fixing, e.g. *BugFeq_*100%*ile*. This means that these sub-DRSpaces cover all the error-prone files and reveal their relations.

For example, the columns *FR* and *CR* illustrate the rankings of each file based on its bug frequency and bug churn. As we can see, in addition to the leading file, *DefaultRolloverStrategy* and *RollingFileManager* are ranked within top 1%ile and top 2%ile resprectively, and are captured by all three sub-DRSpaces. Other than that, however, the files captured by these sub-DRSpaces are very different. Next we explore if and to what extent their ability of capturing buggy files are different.

IV. Experiment

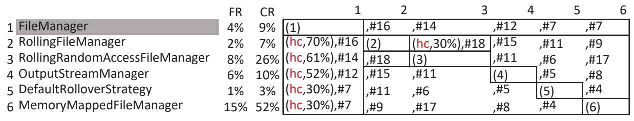
In this section, we present our exploration of the following three research questions to understand the impact of different

[8]https://gephi.org/



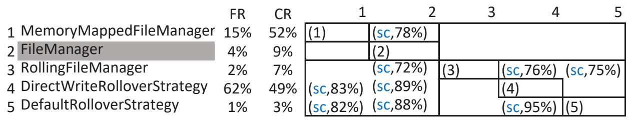
		FR	CR	1	2	3	4	5	6	7	8
1	FileManager	4%	9%	(1)							
2	RollingFileManager	2%	7%	dp	(2)						
3	FileAppender	32%	13%	dp		(3)					
4	RollingRandomAccessFileManager	8%	26%	dp	dp		(4)				
5	DefaultRolloverStrategy	1%	3%	dp	dp			(5)			
6	DirectWriteRolloverStrategy	62%	49%	dp	dp				(6)		
7	RollingFileAppender	50%	5%		dp			dp	dp	(7)	
8	RollingAppenderSizeTest	4%	7%	dp	dp					dp	(8)

Fig. 6: Syntactic sub-DRSpace of FileManager



		FR	CR	1	2	3	4	5	6
1	FileManager	4%	9%	(1)	,#16	,#14	,#12	,#7	,#7
2	RollingFileManager	2%	7%	(hc,70%),#16	(2)	(hc,30%),#18	,#15	,#11	,#9
3	RollingRandomAccessFileManager	8%	26%	(hc,61%),#14	,#18	(3)	,#11	,#6	,#17
4	OutputStreamManager	6%	10%	(hc,52%),#12	,#15	,#11	(4)	,#5	,#8
5	DefaultRolloverStrategy	1%	3%	(hc,30%),#7	,#11	,#6	,#5	(5)	,#4
6	MemoryMappedFileManager	15%	52%	(hc,30%),#7	,#9	,#17	,#8	,#4	(6)

Fig. 7: History sub-DRSpace of FileManager



		FR	CR	1	2	3	4	5
1	MemoryMappedFileManager	15%	52%	(1)	(sc,78%)			
2	FileManager	4%	9%		(2)			
3	RollingFileManager	2%	7%		(sc,72%)	(3)	(sc,76%)	(sc,75%)
4	DirectWriteRolloverStrategy	62%	49%	(sc,83%)	(sc,89%)		(4)	
5	DefaultRolloverStrategy	1%	3%	(sc,82%)	(sc,88%)		(sc,95%)	(5)

Fig. 8: Semantic sub-DRSpace of FileManager

TABLE I: The *MoJoFM* Matrix

Technique	SynDR	HisDR	SemDR	ACDC	ARC
SynDR	-	$8\%^*$	$22\%^*$	27%	26%
HisDR	-	-	$24\%^*$	16%	12%
SemDR	-	-	-	19%	22%
ACDC	-	-	-	-	39%
ARC	-	-	-	-	-

dependency types, syntactic, history and semantic, on software defect prediction/localization:

RQ1: *What is the structural similarity among these different types of relations?*

The answer to this question will advance our understanding regarding to the overall differences of these three relations.

RQ2: *Does each dependency structure present similar performance in terms of capturing defective files?*

The answer to this question will shed lights on their differences in terms of locating or predicting bugs.

RQ3: *Can the combination of these three structures improve the accuracy of predicting/locating defects? What about their intersection and union?*

The answer to this question will provide suggestions regarding to how different relations can be better leveraged.

A. Structural Similarity

*1) **Approach**:* To answer *RQ1*, we measure the overall similarity among these three different relations using *MoJoFM* [32], and use *C2C* [18] to measure the local similarity among sub-DRSpaces lead by the same file. As references, we employed several state-of-the-art reverse-engineering techniques from the *ARCADE* [33] toolkit for comparison. The related concepts are listed as follows:

MoJoFM [32] is a distance score measuring between two structures, expressed as a percentage, computed as follows:

$$MoJoFM\,(A,B) = 1 - \left(\frac{mno\,(A,B)}{max\,(mno\,(\forall A,B))}\right) \times 100\% \quad (5)$$

where $mno\,(A,B)$ is the minimum number of *Move* or *Join* operations needed to transform structure A into structure B. *MoJoFM* returns 0% if the two architectures are completely different. While *MoJoFM* returns 100% if the two architectures are same. The higher the score, the more similar the two structures are.

C2C [18] measures to what extent two sub-DRSpaces with the same leading file overlap. Given two *sub-DRSpaces*: SA and SB with the same leading file but different substructures:

$$C2C(A,B) = \begin{cases} \frac{SA \cap SB}{SA \cup SB} & |SA|, |SB| \neq 0 \\ 0 & other \end{cases} \quad (6)$$

where $|SA|$ and $|SB|$ represent the number of files contained in SA and SB respectively.

ARCADE [33] is a software toolkit for reverse-engineering architectural design from source code. We use two state-of-the-art reverse-engineering techniques, *ACDC* [12] and *ARC* [13] from *ARCADE* as references.

*2) **Results**:* Table I demonstrates the *MoJoFM* results as a square matrix, where rows and columns are labeled by the same set of recovery techniques in the same order. The cells contain the average *MoJoFM* scores. Since the matrix is symmetric, only half of the cells are labeled with scores. For instance, the cell (2,3) is marked with "$24^*\%$", meaning that the average *MoJoFM* score between history (*HisDR*) and semantic (*SemDR*) DRSpaces of the 117 projects is 24%.

Fig 9 demonstrates the distribution of *C2C* scores among three dependency structures. The x-axis in Fig. 9 represents the *C2C* results, which is divided into 10 intervals ranging from 0 to 100%. The y-axis represents the percentage of sub-DRSpaces pairs whose $C2C$ scores fall into the given interval. For example, the first bar in Fig. 9 (a) shows that there are 28.4% of syntactic-history sub-DRSpace pairs have less than 10% similarity.

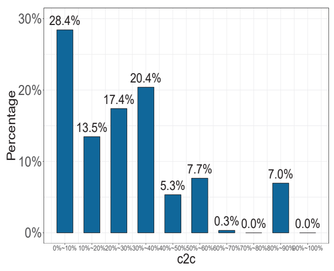
*3) **Observations**:* **The syntactic relation, history relation, and semantic relation present drastically different overall structures (<25%)**: the cells in Table I marked with '*' denotes the average *MoJoFM* scores among the syntactic, history and semantic structures of the 117 projects (8%-24%).

The three DRSpaces, *SynDR*, *HisDR*, and *SemDR*, are also different from the structures derived from the state-of-the-art reverse engineering techniques: their similarity scores with ACDC and ARC are less than 30%: the cells in Table I not marked with '*' denote the average *MoJoFM* score between DRSpaces vs. ACDC/ARC (5%-27%).

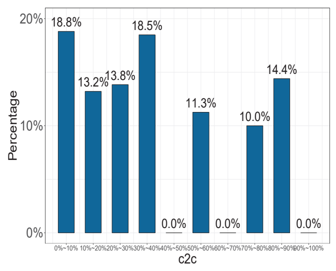
The syntactic structure, history structure, and semantic structure also appear to be significantly different when comparing sub-DRSpaces. Fig. 9 demonstrates that 42%-58% of sub-DRSpace pairs present relative low similarities (<30%), and only 7%-24% of sub-DRSpace pairs present higher similarities (>70%).

B. The Relation between Dependency and Software Defects

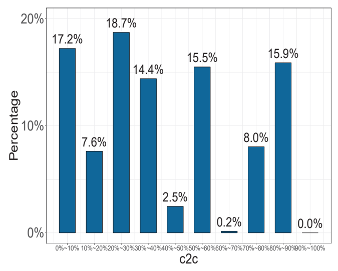
*1) **Approach**:* Here we investigate the relation between each dependency type and software defects, to answer *RQ2*.



(a) Syntactic vs History sub-DRSpace

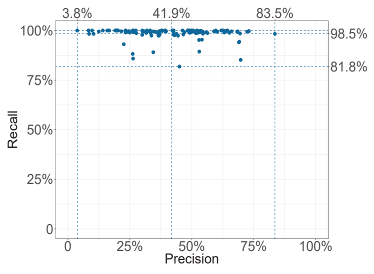


(b) Syntactic vs Semantic sub-DRSpace

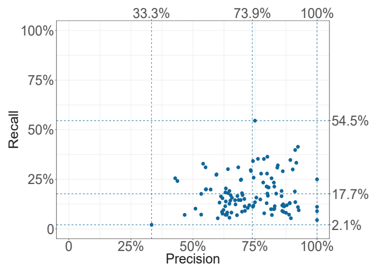


(c) History vs Semantic sub-DRSpace

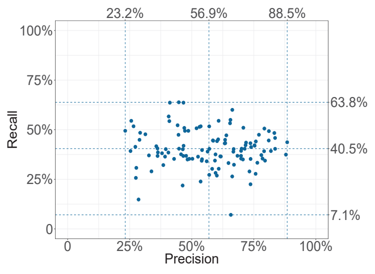
Fig. 9: The distribution of *C2C*



(a) Syntactic Structure



(b) History Structure



(c) Semantic Structure

Fig. 10: The Distribution of Precision/Recall to Cover *BugSpace* (The x-axis represents the precision and the y-axis represents the recall)

TABLE II: The Precision/Recall to Cover *BugFre_x%ile* and *BugChurn_x%ile*

Percentage	BugFre_x%ile						BugChurn_x%ile					
	Syntactic		History		Semantic		Syntactic		History		Semantic	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
10%ile	9.92%	99.29%	43.77%	31.12%	33.32%	43.49%	10.42%	99.78%	43.72%	32.32%	35.81%	41.98%
20%ile	14.43%	99.36%	48.36%	26.36%	35.63%	43.11%	14.39%	99.67%	46.98%	26.57%	35.99%	42.80%
30%ile	18.34%	99.40%	52.12%	23.86%	38.09%	42.65%	18.77%	99.31%	49.52%	23.06%	38.39%	42.54%
40%ile	22.35%	99.41%	55.55%	22.55%	40.23%	42.47%	22.39%	99.17%	52.61%	21.50%	40.15%	42.48%
50%ile	26.17%	99.39%	58.04%	21.17%	43.52%	42.15%	25.75%	99.05%	55.47%	20.13%	42.50%	42.24%
60%ile	29.47%	99.30%	61.13%	19.96%	46.37%	41.94%	29.18%	98.73%	58.81%	19.28%	45.50%	41.85%
70%ile	32.68%	99.22%	64.35%	19.14%	49.21%	41.64%	32.30%	98.71%	62.24%	18.76%	48.19%	41.46%
80%ile	35.79%	99.03%	67.47%	18.72%	51.99%	41.12%	35.55%	98.67%	66.13%	18.34%	50.80%	41.20%
90%ile	39.07%	98.80%	70.35%	18.02%	54.40%	40.69%	38.63%	98.52%	70.05%	17.88%	53.92%	40.89%
100%ile	41.87%	98.52%	73.91%	17.69%	56.92%	40.46%	41.87%	98.52%	73.91%	17.69%	56.92%	40.46%

We first define a specific set of error-prone files as *TargetSpace*, which can be either files frequently involved in bug fixing— *BugFre_x%ile*, or files that are most expensive to fix—*BugFre_x%ile*. To answer *RQ2*, we employed multiple *TargetSpace*s, from *BugFre_10%ile* to *BugFre_100%ile*, and from *BugChurn_10%ile* to *BugChurn_100%ile*. For each *TargetSpace*, we use the *ArchRoot* algorithm to locate its corresponding *RootSpace* and calculate its *Precision* and *Recall* as follows:

$$Precision = \frac{|RootSpace \cap TargetSpace|}{|RootSpace|} \tag{7}$$

$$Recall = \frac{|TargetSpace \cap BugSpace|}{|BugSpace|} \tag{8}$$

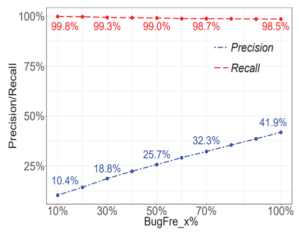
For each DRSpace formed using syntactic, history, or semantic relation, and each *TargetSpace*, we run the ArchRoot algorithm and calculate their precision and recall scores.

*2) **Results**:* Table II presents the average precision and recall scores of the 117 subjects to cover from *BugFre_10%ile* to *BugFre_100%ile*, and from *BugChurn_10%ile* to *BugChurn_100%ile* using syntactic structure, history structure, and semantic structure respectively. Fig. 10 depicts the distribution of precision/recall scores of these subjects, where each point represents a subject. Fig. 11 demonstrates the trend of these scores in Table II.

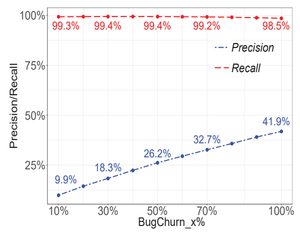
*3) **Observations**:* **The syntactic structure, history structure, and semantic structure present completely different coverage over bug-prone files (*BugSpace*).** The results in Table II and Fig. 10 demonstrate that syntactic structures present the highest recall (98.5%) but the lowest precision (41.9%) on average. This is not surprising because the syntactic structure contains the largest number of files.

On the contrary, history structures present the highest precision (73.9%) but the lowest recall (17.7%), which is understandable since not all buggy files have to be historically coupled with other files. The result of semantic structure is in-between, 56.9% precision and 40.5% recall on average.

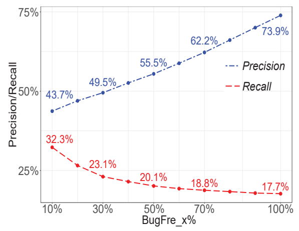
In summary, the three relations present significantly different capabilities in term of covering error-prone files. The



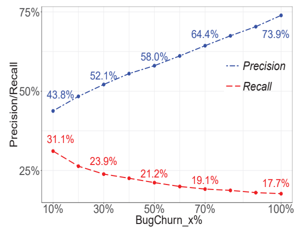
(a) *BugFre_x%ile* Covered by Syntactic Structure



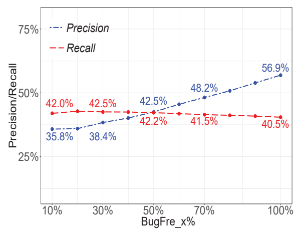
(b) *BugChurn_x%ile* Covered bySyntactic Structure



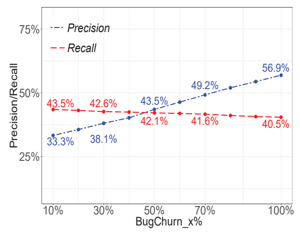
(c) *BugFre_x%ile* Covered by History Structure



(d) *BugChurn_x%ile* Covered by History Structure



(e) *BugFre_x%ile* Covered by Semantic Structure



(f) *BugChurn_x%ile* Covered by Semantic Structure

Fig. 11: The Trend of Precision/Recall results of Covering *BugFre_x%ile* and *BugChurn_X%ile* (The x-axis represents the *BugFre_x%ile/BugChurn_x%ile* and the y-axis represent the precision/recall)

implication is that, using just one of these relations to conduct bug prediction/localization will not be sufficient.

C. Dependency Interaction and the Impact on Software defects

*1) **Approach**:* To answer *RQ3*, we exhaustively studied the intersection and union of syntactic structure, history structure, and semantic structure. In total, we generated 8 combinations of these three dependency structures, measured their interaction with two representative *TargetSpaces*, *BugFre_10%ile* and *BugFre_100%ile*, and made a systematic comparison.

Generating Combinations: The 8 combinations include: *Syn* ∩ *His*, *Syn* ∩ *Sem*, *His* ∩ *Sem*, *Syn* ∩ *His* ∩ *Sem*, *Syn* ∪ *His*, *Syn* ∪ *Sem*, *His* ∪ *Sem*, and *Syn* ∪ *His* ∪ *Sem*, where *Syn*, *His* and *Sem* are the abbreviation for syntactic structure, history structure, and semantic structure. The notation, ∪, represents the union operator and ∩ represents the intersection operator.

Combination measures: For each combination, we generated its DRSpace, and calculated sub-DRSpaces covering two *TargetSpaces*: *BugFre_10%ile* and *BugFre_100%ile* using the *ArchRoot* algorithm to generate a *RootSpace*. As an example, consider the union of the three structures: (*Syn* ∪ *His* ∪ *Sem*). Fig. 12 depicts its sub-DRSpace led by *FileManager* in Log4j.

To measure these results, in addition to precision and recall, we also employ three additional metrics as follows:

File% measures the percentage of files involved in each combination.

BFR, following the work of Mo et al. [21], measures the average bug frequency of each file in a calculated *RootSpace* versus the average bug frequency of all the files within the *BugSpace*, defined as:

$$BFR = \frac{BugFre\,(RootSpace)}{BugFre\,(BugSpace)} \times \frac{|BugSpace|}{|RootSpace|} \times 100\% \quad (9)$$

where $|RootSpace|$ and $|BugSpace|$ represent the number of files involved in them. $BugFre\,(RootSpace)$ represents the sum of the number of bug fixes for each file involved in *RootSpace*.

BCR, similarly, measures the average bug churn of each file in a calculated *RootSpace* versus the average bug churn of all the files within the *BugSpace*, defined as

$$BCR = \frac{BugChurn\,(RootSpace)}{BugChurn\,(BugSpace)} \times \frac{|BugSpace|}{|RootSpace|} \times 100\% \quad (10)$$

where $|RootSpace|$ and $|BugSpace|$ represent the number of files involved in them. $BugChurn\,(RootSpace)$ represents the sum of the lines of code in bug fixes for each file involved in *RootSpace*.

We use an example to illustrate *BFR* and *BCR*. For a BugSpace with three files: $\{A,B,C\}$, its involved bug frequencies and churn are listed as follows: (3,100), (2,200) and (1,300). For a generated *RootSpace*: $\{A,B\}$, to calculate *BFR*, the results of $BugFre\,(RootSpace)$ and $BugFre\,(BugSpace)$ are counted as 5 and 6 respectively. According to formula (9), the result of *BFR* should be (5/6)*(3/2) = 125%, meaning that the average bug frequency of the files within the *RootSpace* is higher than the average file within the bug space. The result of *BCR* should be (300/600)*(3/2) = 75%, meaning that the average bug churn of files within the *RootSpace* is lower than the average file within the bug space.

In summary, *BFR* and *BCR* are used to measure the average maintenance cost of the *RootSpaces* calculated from these combinations, using the average scores of the *BugSpace* as a baseline.

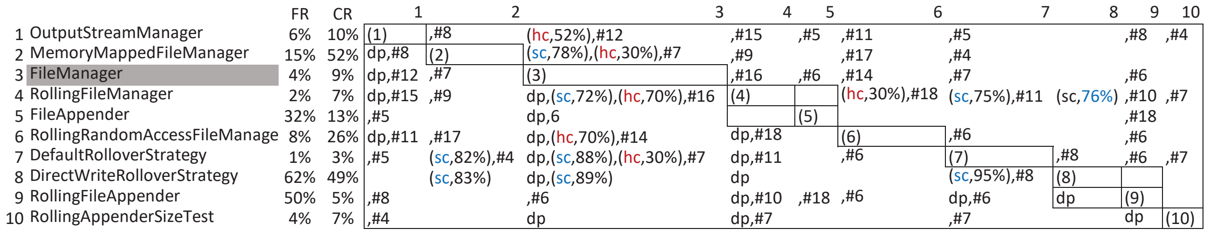
*2) **Results**:* Table III presents the results. The columns *Type* and *Formula* present the details of each combination. For each combination, the other columns present the average scores of the 117 subjects to cover *BugFre_10%ile* and *BugFre_100%ile* (i.e. the overall *BugSpace*) respectively.

*3) **Observations**:* **The intersection of syntactic, history, and semantic relations merely covers 0.82% of all the bug-prone files (*BugSpace*) but shows nearly 80% precision and over 500% bug frequency/churn rate.** Table III shows that the intersection of these three structures only occupy 0.28% of all the files on average. It can also capture the 2.63% of the most bug-prone files *BugFre_10%ile* with a relative high precision (nearly 50%).

The union of syntactic structure, history structure and semantic structure covers the 99.3% of all the bug-prone files(*BugSpace*) with 43.4% precision and almost 80% bug

TABLE III: The Interaction of Dependency Structures

Type	Formula	File%	BugFre_10%ile				BugSpace			
			Precision	Recall	BFR	BCR	Precision	Recall	BFR	BCR
Syn	(V, E_1)	96.38%	10.42%	99.78%	262.50%	210.84%	41.90%	98.50%	70.97%	71.14%
His	(V, E_2)	8.41%	43.72%	32.32%	2111.37%	1314.45%	73.90%	17.70%	918.09%	560.08%
Sem	(V, E_3)	35.11%	36.36%	44.45%	1294.51%	862.57%	56.90%	40.50%	284.20%	188.26%
Syn ∩ His	$(V, E_1 \cap E_2)$	3.77%	49.12%	15.94%	2802.91%	1332.68%	76.06%	8.13%	1236.42%	602.38%
Syn ∩ Sem	$(V, E_1 \cap E_3)$	16.25%	46.52%	31.38%	1732.65%	1057.02%	68.45%	22.05%	551.94%	337.46%
His ∩ Sem	$(V, E_2 \cap E_3)$	0.71%	50.97%	4.13%	2846.15%	1566.86%	79.62%	1.88%	1197.66%	562.91%
Syn ∩ His ∩ Sem	$(V, E_1 \cap E_2 \cap E_3)$	0.28%	49.94%	2.63%	3043.47%	1536.33%	79.46%	0.82%	1210.64%	524.09%
Syn ∪ His	$(V, E_1 \cup E_2)$	96.52%	10.54%	99.84%	267.19%	214.68%	42.11%	98.76%	72.55%	72.06%
Syn ∪ Sem	$(V, E_1 \cup E_3)$	96.71%	10.52%	99.79%	263.60%	205.87%	41.94%	98.76%	74.25%	73.53%
His ∪ Sem	$(V, E_2 \cup E_3)$	40.52%	34.85%	62.72%	1161.97%	831.25%	59.46%	52.96%	298.47%	243.59%
Syn ∪ His ∪ Sem	$(V, E_1 \cup E_2 \cup E_3)$	97.50%	14.28%	100.0%	424.21%	339.84%	43.45%	99.37%	84.81%	83.29%



	FR	CR	1	2	3	4	5	6	7	8	9	10
1 OutputStreamManager	6%	10%	(1)	,#8	(hc,52%),#12	,#15	,#5	,#11	,#5		,#8	,#4
2 MemoryMappedFileManager	15%	52%	dp,#8	(2)	(sc,78%),(hc,30%),#7	,#9		,#17	,#4			
3 FileManager	4%	9%	dp,#12	,#7	(3)	,#16	,#6	,#14	,#7		,#6	
4 RollingFileManager	2%	7%	dp,#15	,#9	dp,(sc,72%),(hc,70%),#16	(4)		(hc,30%),#18	(sc,75%),#11	(sc,76%)	,#10	,#7
5 FileAppender	32%	13%	,#5		dp,6		(5)				,#18	
6 RollingRandomAccessFileManage	8%	26%	dp,#11	,#17	dp,(hc,70%),#14	dp,#18		(6)	,#6		,#6	
7 DefaultRolloverStrategy	1%	3%	,#5	(sc,82%),#4	dp,(sc,88%),(hc,30%),#7	dp,#11		,#6	(7)	,#8	,#6	,#7
8 DirectWriteRolloverStrategy	62%	49%		(sc,83%)	dp,(sc,89%)	dp			(sc,95%),#8	(8)		
9 RollingFileAppender	50%	5%	,#8		,#6	dp,#10	,#18	,#6	dp,#6	dp	(9)	
10 RollingAppenderSizeTest	4%	7%	,#4		dp	dp,#7			,#7		dp	(10)

Fig. 12: The Combination of Syntactic, History, and Semantic sub-DRSpaces

frequency/churn rate. Table III shows that the union of these three structures occupy 97.5% of all the files on average, and can capture all the most bug-prone files ($BugFre\_10\%ile$) with a relative low precision (14.28%).

For intersections of two dependency structures, the intersection of syntactic structure and semantic structure (*Syn* ∩ *Sem*) shows a higher recall rate (22.05%) to cover all the bug-prone files (*BugSpace*); the intersection of history structure and semantic structure (*His* ∩ *Sem*) shows a higher precision rate (79.62%) to cover all the bug-prone files (*BugSpace*). The intersection of syntactic structure and semantic structure (*Syn* ∩ *Sem*) captures 16.25% of all the files. It also covers the most bug-prone files (*BugFre*_10%*ile*) with 46.52% precision rate and 31.38% recall rate. The intersection of history structure and semantic structure (*His* ∩ *Sem*) only captures 0.71% of all the files, which covers the most bug-prone files (*BugFre*_10%*ile*) with a higher precision rate (50.97%) and a lower recall rate (4.13%). The intersections involving history relation can improve the precision by 23%-38%. The intersections involving semantic structure can also improve the precision by 12%-38%.

For unions of two dependency relations, the union of syntactic structure and history structure (*Syn* ∪ *His*) covers 98.76% of all the bug-prone files (*BugSpace*); the union of history structure and semantic structure (*His* ∪ *Sem*) shows a higher precision rate (59.46%). The union of syntactic structure and history structure (*Syn* ∪ *His*) captures 96.52% of all the files, which also covers the most bug-prone files (*BugFre*_10%*ile*) with 10.54% precision rate and 99.84% recall rate. The union of history structure and semantic structure (*His* ∪ *Sem*) captures 40.52% of all the files, which covers the most bug-prone files (*BugFre*_10%*ile*) with a higher precision rate (34.85%) and a lower recall rate (62.72%). The unions involving syntactic structure can improve the recall by 58%-81%.

V. Result Discussion

In this section, we discuss the results and answer the three research questions.

RQ1: The result of *RQ1* reveals that the syntactic relation, history relation, and semantic relation present significantly different structures. Intuitively, this implies that these three relations will perform drastically different when they are used to predict or locate software defects. The result of *RQ1* also reveals that these three dependency relations present different architecture views from the state-of-the-art reverse engineering techniques. This may provide a new perspective to recover architecture using these three dependency structures directly.

RQ2: The result of *RQ2* demonstrates that the syntactic relation, history relation, and semantic relations are significantly different in terms of their ability to capture error-prone files. The syntactic structure shows the advantage of covering most defective files, while the history structure and semantic structure show the advantage of improving the coverage accuracy. The indication is that when performing bug location, prediction, and prioritization tasks, these relations should be considered and compared separately.

RQ3: The result of *RQ3* is unexpected. The intersection of these relations only covers a rather small portion of defects (0.82%) but with high precision scores. However, the union of them can almost cover all the defects (99.37%). These results demonstrate that these three dependency structures are independent and complementary. The high recall rate implies that these three dependency structures are enough for defect prediction/localization. However, the precision of their union is merely 43.45%. Thus, we can focus on how to improve the precision using these three structures. The results of intersections/unions for two structures provide us an intuitive way.

Compared with using one dependency structure, the combinations involving syntactic structure could further improve the coverage recall. The combinations involving history structure and semantic structure can improve the coverage precision. This inspires us to design proper strategies to combine these three structures together to improve the accuracy of defect prediction/localization.

VI. Threat to Validity

In this section, we discuss the threats to validity and limitation of our study.

1) ***Internal threats****:* First, for each subject, we collect commit data, generate history relation, and derive bug reports from the beginning of its revision history to the latest release. Prior research [34] suggested that if only recent history is used, the result could be different. To validate our work, we recalculate the data, extracting history relation and bug reports from the most recent 3 releases of each subject. The results showed that specific bug frequency and churn ranking orders are different, but the general conclusions are exactly the same.

Second, we set the thresholds for determining history coupling and semantic coupling according to previous studies. It is unclear whether these threshold settings are generalizable. To eliminate this threat, we conducted a large scale study using 117 open source projects, and manually inspected the generated relations for 5 projects. We confirmed that the extracted software relations are reasonable.

Third, for syntactic structure, history structure and semantic structure, we employed *DRSpaces* to understand the architectural design. In the future, we will use various reverse-engineering techniques and make a systematical comparison of their influence on software quality.

Fourth, for each dependency structure, we employ the *ArchRoot* algorithm to calculate its interactions with buggy files. *ArchRoot* is a greedy algorithm whose effectiveness and efficiency can be further improved. In the future, we will also design pruning strategies to make more detailed and comparative analyses.

Finally, our research aims at investigating the *correlation* between dependency relations and file error-proneness, but not *causality*. We do not have evidence how dependency causes the presence of defects. In our future work, we will explore the causality relation by studying each bug report and its introduced change.

2) ***External threats****:* The first threat comes from locating buggy files using bug reports. Following prior history-based bug prediction work [4, 8], we link these bug reports with its fixing commits by heuristically searching BugIDs from revision messages. However, researchers have [35] pointed out that, developers may commit bug fixes using wrong bugID or even without reporting bugID. In these cases, our approach may be biased. To eliminate this threat, we select the Apache Open Projects and study bug reports from its issue tracking system: JIRA. The bug report information in JIRA for Apache projects is manually entered by experts, containing less noise. In our future work, we plan to study the impact of missing links to bug reports.

The second threat comes from the chosen subjects. We intensively studied 117 Apache open source projects. It is still unclear whether our observations will generalize to closed-source industrial projects and open-source projects from other communities. Studying more subjects is our ongoing work.

The final threat comes from the imbalance problem, which is a common problem in machine learning and defect prediction. However, our objective is to investigate the differences of the three major dependency types, not to propose a new defect prediction/localization algorithm.

VII. Conclusion

In this paper, we presented our systematic study on the relation between file error-proneness and syntactic dependency, history dependency, and semantic dependency. We conducted our study on 117 Apach open source projects involving 643,079 revision commits and 101,364 bug reports. Supported by DRSpaces, for each dependency type, we created its overall DRSpace, split it into a set of sub-DRSpaces, and calculated their interactions with bug spaces. We investigated three research questions using these data. The results demonstrated the independence and complementary nature of these three dependency types, and their drastically different impact on file error-proneness. We also presented a suite of qualitative and quantitative results, which provide new insights that may benefit defect prediction, localization, and prioritization.

Acknowledgement

This work was supported by National Key R&D Program of China (2016YFB1000903), National Natural Science Foundation of China (61632015, 61772408, U1766215, 61532015, 61833015, 61702414), Ministry of Education Innovation Research Team (IRT_17R86), Innovative Research Group of the National Natural Science Foundation of China (61721002), Project of China Knowledge Centre for Engineering Science and Technology. This work was also supported by the United States National Sciences Foundation grants 1514561, 1514315 and 1816594. T. Liu is the corresponding author.

References

[1] 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.

[2] 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.

[3] T. Zimmermann and N. Nagappan, “Predicting defects using network analysis on dependency graphs,” in *ACM/IEEE International Conference on Software Engineering*, 2008, pp. 531–540.

[4] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, “Software dependencies, work dependencies, and

their impact on failures," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, 2009.

[5] 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.

[6] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Ieee/acm International Conference on Software Engineering*, 2016, pp. 297–308.

[7] 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.

[8] 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.

[9] 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.

[10] Z. Li, X. Y. Jing, X. Zhu, and H. Zhang, "Heterogeneous defect prediction through multiple kernel learning and ensemble learning," in *IEEE International Conference on Software Maintenance and Evolution*, 2017, pp. 91–102.

[11] D. D. Nucci, F. Palomba, R. Oliveto, and A. D. Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 202–212, 2017.

[12] V. Tzerpos and R. C. Holt, "Acdc: An algorithm for comprehension-driven clustering." in *wcre*, 2000, pp. 258–267.

[13] 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.

[14] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 35–44.

[15] B. S. Mitchell, "A heuristic approach to solving the software clustering problem," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 285–288.

[16] F. Beck and S. Diehl, "Evaluating the impact of software evolution on software clustering," in *Reverse Engineering*, 2010, pp. 99–108.

[17] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[18] 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.

[19] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 69–78.

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

[21] 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.

[22] 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.

[23] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *IEEE International Conference on Software Maintenance and Evolution*, 2017, pp. 433–437.

[24] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A tool for architectural smells detection," in *IEEE International Conference on Software Architecture Workshops*, 2017, pp. 282–285.

[25] 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.

[26] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[27] J. Chang and D. M. Blei, "Hierarchical relational models for document networks," *Annals of Applied Statistics*, vol. 4, no. 1, pp. 124–150, 2010.

[28] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2014.

[29] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., 2012.

[30] 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.

[31] L. Xiao, Y. Cai, and R. Kazman, "Titan: A toolset that connects software architecture with quality analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 763–766.

[32] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *IEEE International Workshop on Program Comprehension*, 2004, p. 194.

[33] D. M. Le, P. Behnamghader, J. Garcia, and D. Link, "An empirical study of architectural change in open-source software systems," in *MSR*, 2015, pp. 235–245.

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

[35] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links:bugs and bug-fix commits," in *ACM Sigsoft International Symposium on Foundations of Software Engineering, 2010, Santa Fe, Nm, Usa, November*, 2010, pp. 97–106.