Received: 31 January 2017 | Revised: 27 January 2018 | Accepted: 6 May 2018

DOI: 10.1002/smr.1960

RESEARCH ARTICLE - STANDARDS

WILEY Journal of Software: Evolution and Process

Do code data sharing dependencies support an early prediction of software actual change impact set?

Xiaoyu Liu[1] | LiGuo Huang[1] | Alexander Egyed[2] | Jidong Ge[3]

[1] Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas 75275-0122, USA

[2] Institute of Software Systems Engineering, Johannes Kepler University, Linz 4040, Austria

[3] State Key Laboratory for Novel Software and Technology, Nanjing University, 22 Hankou Road, Nanjing, China

Correspondence
LiGuo Huang, Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas 75275-0122, USA.
Email: lghuang@engr.smu.edu

Abstract

Existing studies have shown that structural dependencies within code are good predictors for code actual change impact set—a set of entities that repeatedly changing together to ensure a consistent and complete change. However, the result is far from ideal, particularly when insufficient historical data are available at an early stage of software development. This paper demonstrates that a better understanding of data dependencies in addition to call dependencies greatly improves actual change impact set prediction. We propose a new approach and tool (namely, CHIP) to predict software actual change impact sets leveraging both call and data sharing dependencies. For this purpose, CHIP employs novel extensions (dependency frequency filtering and shared data type idf filtering) to reduce false positives. CHIP assumes that developers know initial places where to start making changes in the source code even though they may not know all changes. This approach has been empirically evaluated on 4 large-scale open source systems. Our evaluation demonstrates that data sharing dependencies have a complementary impact on software actual change impact set prediction as compared with predictions based on call dependencies only. CHIP improves the F2-score compared with the predictors using both Program Dependence Graph and evolutionary couplings.

KEYWORDS

data sharing dependency, software impact analysis, source code dependency

1 | INTRODUCTION

Software systems are composed of entities such as classes, methods, and variables. These entities depend on one another, for example, by 2 methods sharing data passed through parameters. Problems arise when systems evolve, and developers change these entities to add new features, retire existing features, or fix bugs. During such evolutions, developers must ensure that dependent entities are changed consistently.[1] Since developers find it hard to manually identify such dependent entities, they benefit from automated support. We thus speak of the change impact analysis (CIA) that guides developers[1] to make changes. It identifies sets of entities that repeatedly change together to ensure a consistent and complete change. This set is called the software actual change impact set.[2] Traditional CIA, which aims at estimating the actual change impact set of a system due to a proposed change,[3] is performed manually by developers. Due to the known complexity of the manual identification of actual change impact sets, developers would greatly benefit from automated predictions (aka, automated CIA).

State-of-the-art approaches to predicting the actual change impact sets employ 2 main types of automated CIA approaches, namely, static and dynamic analysis. Dynamic analysis[4-15] heavily relies on a complete system execution profile, which is usually difficult to acquire and only available late during development (after a majority of the system has been implemented) due to the unavailability of complete tasks. Moreover, most of these techniques trace individual system execution paths and focus on method calls while ignoring data dependencies (eg, fields or variables) shared across different execution paths. Static analysis[1,16-26] predicts actual change impact sets by mining the code change

repositories from previous changes in software repositories.[27] Some static analysis approaches require access to a long history of code changes to capture the extent to which software artifacts were changed together, namely, evolutionary couplings. However, as software evolves, older impact set changes could become outdated and possibly misleading. Besides, these approaches often require learning-based impact set predictors that assume that actual change impact sets follow the same patterns as documented in the history. However, this assumption is not always true because software changes do not always impact the rest of the system following the same patterns. Other static analysis, approaches based on textual analysis,[1,18-20,22-25] avoid this problem by extracting conceptual dependencies (coupling) via the analysis of comments and/or identifiers in source code.[27] They thus do not rely on the history of a software system but rather on its current state. However, it requires developers to encode the implicit actual change impact sets from the comments and/or identifiers, and hence, the quality of the change prediction depends on the quality of the encoding. To counter this, another kind of static analysis method is structural analysis, [1,3,18-20,28-37] which leverages call dependencies among entities (most notably method calls) as indicators for the actual change impact set. For example, if method A calls method B, and B is changed, then method A may likely need changing also. In turn, if method A is changed then all methods calling A may also need changing. This ripple effect of change propagation progresses until no more changes are required to source code.[38] A variety of these structural analysis approaches leverage program dependence graphs (PDGs)[39,40] or employ program slicing.[30,31,41] Program dependence graph incorporates 2 kinds of code dependence other than call dependency: (1) control dependence representing the control flow relationships of the program, and (2) data flow dependence representing the data flow relationships of the program.[42,43] Program slicing,[30,31,41] similar to PDG, addresses the computation of effects among program points by traversing data flow and control flow. It is interesting to note that very few approaches consider data sharing in predicting actual change impact sets. We believe data sharing is another important indicator for actual change impact sets, which is overlooked by the state-of-the-art CIA. Our working assumption is that class/method data sharing dependencies are a vital complement to call dependencies in fully understanding code entities and how they are affected by changes.

To improve the state-of-the-art, this paper investigates the role of data sharing dependencies in actual change impact set prediction. A method-level data sharing dependency is defined as 2 methods reading or manipulating variables that point to the same data stored in the same (physical) memory location no matter whether the variables holding the pointers are identical or not. A class-level data sharing dependency is the aggregation of method-level dependencies. Both kinds of data sharing dependencies are useful. Since shared data are often accessed through references or chains of references, unlike other code structural dependencies (ie, call dependencies and control dependencies), data sharing dependencies can only be captured through runtime profiling analysis. To demonstrate that more expressive data sharing dependencies are useful for the actual change impact set prediction, we compare our approach with PDG in Section 7. In addition, we compare our approach with the state-of-the-art approach using evolutionary couplings extracted from association rules.

As a prerequisite, our approach requires an initial set of changes, which are the initial changes made by developers. This initial set of classes is likely a subset of classes that need changing because the developer may not yet understand the complete impact of the changes. The initial set of changes is thus a subset of all changes needed. Regardless, our approach will analyze call and data sharing dependencies on the initially changed class(es) to identify additional, dependent classes that likely need changing as well. Our approach then recommends these dependent classes that likely need changing as well. Our approach then recommends these dependent classes to the developer who decides on whether or not additional changes are necessary. Our approach then repeats the dependency analysis with every change iteration, thus refining the change prediction.

The main contributions of this paper are the following:

- A dynamic mechanism to capture both call and data sharing dependencies across software classes and methods, which is then used to frame a novel basic CHange Impact set Predictor (CHIP). Our experiment results show that code dependencies complement each other in the actual change impact set prediction. Furthermore, data sharing dependencies are particularly useful in specific change impact scenarios including "move refactoring," "remove classes and statement," "bug fixing," "functional improvement," and "code replacement."
- A change impact set predictor with dependency frequency filter and a novel data type inverse document frequency (*idf*) filter to reduce false positives in prediction (ie, false change impact sets) while maintaining the recall. We show that these filters significantly improve F2-score of CHIP compared with state-of-the-art approaches.
- An automated tool that provide developers with change impact set predictions.
- An empirical evaluation on hundreds of changes taken from 4 open source systems that span across different domains. The evaluation results not only demonstrate usefulness of our approach but also help to better understand the benefit of data sharing dependencies with respect to call dependencies for change impact set prediction

The empirical evaluation also aims to answer the following 3 research questions:

RQ1. *Do data sharing dependencies complement call dependencies in actual change impact set prediction?*

RQ2. *How effective is the data sharing dependencies compared with traditional program dependence graphs (PDGs) and evolutionary couplings in actual change impact set prediction?*

RQ3. *The combined call and data sharing dependencies improve predictions in different change impact scenarios as compared with standalone call dependencies?*

The rest of the paper is organized as follows. Section 2 presents the preliminaries and motivating example. Section 3 introduces the proposed actual change impact set prediction framework. Section 4 describes our data sharing dependency generation approach. Section 5 elaborates the proposed actual change impact set prediction algorithm. Sections 6 and 7 present the experiment setup and results, followed by a discussion of threats to validity in Sections 8. Section 9 summarizes the related work. Section 10 concludes and envisages our future work.

2 | PRELIMINARIES

This section first introduces the state-of-the-art call dependencies generation method that we employ in this study, specifically the PDG generation algorithm and evolutionary coupling generation algorithm. Then, we present an example to motivate our CHange Impact set Prediction (CHIP) framework.

2.1 | Dynamically capturing call dependencies

Sound static analysis approaches for capturing call dependencies can guarantee correctness through overestimation but typically produce a large number of false positives (wrong predictions)[44] and consume a significant amount of time or resources when the analysis is performed on the whole software system. To avoid such problems, we use dynamic analysis to capture actually observed call dependencies. This eliminates false positives and a reasonable test coverage ensures a high degree of completeness. Next, we describe how System Runtime Profiling is used to capture call dependencies and what are the differences between call dependency in CHIP and PDG.

1. *System Runtime Profiling*: To capture high quality execution traces in a Java system, we leverage a tool built on JVMTI (Java Virtual Machine Tool Interface), which was developed in our previous work.[45] This tool provides a way to inspect the state of the Java system and control its execution while the system runs on the Java Virtual Machine (JVM). Our tool can query and record the special events that are generated by JVM including "method entry" and "method exit." To ensure the correctness and completeness of captured dependencies, all functions according to requirements and use cases documents of each subject system must be executed with our instrumented runtime profiler. The execution traces are stored to be analyzed in the subsequent steps for dependencies generation.
2. *Call Dependency Generator*: Note that our approach can be generalized to any language platform in which execution traces can be captured. We use Java as an example to illustrate the process of capturing call dependencies. Call dependencies are captured by traversing all records in "method entry" and "method exit" records among the JVM events. This generator traces the events of each thread separately. Method X calls method Y if both classes are registered in the callback functions of "method entry" and "method exit."
3. *Differences between call dependency and PDG*: The main difference between call dependencies extracted by CHIP (CHIP-Call) and PDG is: PDG is extracted using static analysis approach, while CHIP-Call is extracted using dynamic analysis approach. PDG incorporates transitive calling queries by define and use the same value. For example, PDG considers a dependency relation between method A and method B if a value defined by A is used by B. However, PDG may not include method calls induced by aggregation of multiple JVM events analyzed by the JVM runtime profiling tool. To explain the difference between CHIP-Call and PDG, we use jEdit system as an example. Table 1 shows a partial execution trace log with 2 execution trace records captured by CHIP. In Table 1, we found a calling dependency between methods *OptionsDialog.init()* and *StyleTableModel.getTableCellRender()* induced by aggregation of 2 execution records in 2 different JVM events (eg, method exit and method entry). Furthermore, Figure 1 shows a Venn diagram that shows the overlap and distinction between class-level CHIP-Call and class-level PDG. Both dependencies are captured from the jEdit system that we will discuss in Section 7. As we can see both approaches share 471 dependencies. However, CHIP-Call includes 1866 additional call dependencies not captured by PDG, and PDG has 999 data/control flow dependencies not included in CHIP.

2.2 | Program dependence graph (PDG) and evolutionary couplings capturing

In this study, we generate the Program Dependence Graph (PDG) and capture evolutionary couplings to replicate the state-of-the-art actual change impact set predictors for comparison with our approach.

TABLE 1 Partial execution trace log of jEdit

Order	Method	JVM Events	Thread ID
1	*OptionsDialog.init()*	method exit	27058272
2	*StyleTableModel.getTableCellRender()*	method entry	27058272

Abbreviation: JVM, Java Virtual Machine.

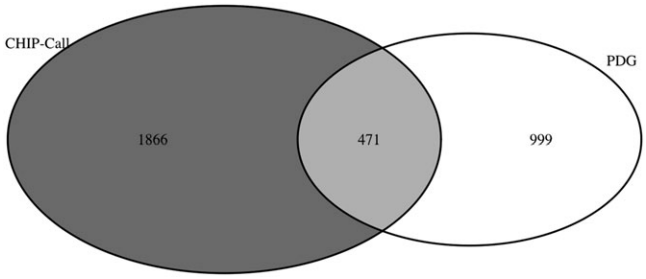


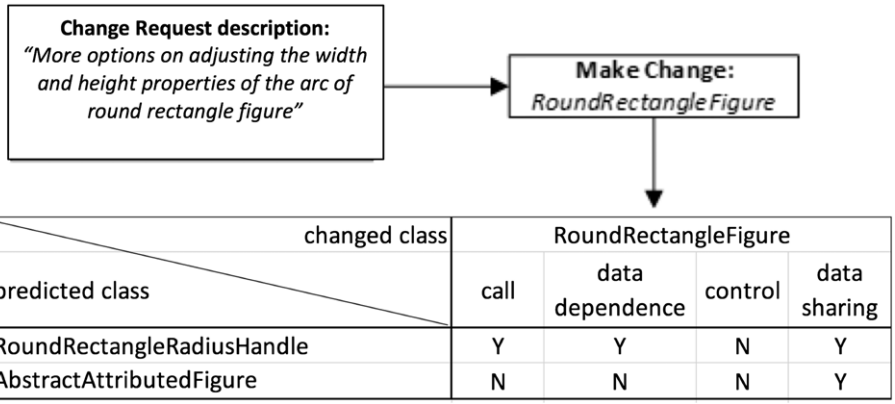
FIGURE 1 Venn diagram of class-level call dependencies in CHIP (CHIP-Call) versus class-level PDG in jEdit. CHIP, CHange Impact set Prediction; PDG, Program Dependence Graph

To generate PDG, we apply a tool called "DUA-Forensics" leveraged in Santelices et al.[38] For point-to analysis, "DUA-Forensics" finds a set of classes or methods in locations pointed to by a known class or method variable in context- and flow- insensitive way.[46] To ensure the correctness and completeness of dependencies, it exploits not only interprocedural (ie, across methods) but also intraprocedural dependencies such as exception control dependencies.

To capture evolutionary couplings, we apply association rules based on itemset mined from historical change commits. We mine the SVN logs that were generated before the commits in test set. Details are shown in Table 6. We choose the same support value (=1) as in Zimmermann et al[16] to capture a more comprehensive set of evolutionary couplings.

2.3 | Motivating example

Figure 2 shows an example of an actual change impact set prediction on an early version jHotDraw[47] following the CIA process. A developer starts to change the code based on a change request (ie, an item that can be either a bug or a request for enhancement[48]). This change request starts with a short description: *More options on adjusting the width and height properties of the arc of round rectangle figure*. The developer first changes the class *RoundRectangleFigure* as it seems an obvious choice. However, the developer does not know whether the change is limited to this class or also propagated to other dependent classes that require changes as well. Since this is an early version generated at the early lifecycle of the software development, the developer does not have the access to sufficient historical data either. Table 2 shows the execution traces of 3 classes (*RoundRectangleFigure*, *RoundRectangleRadiusHandle*, and *AbstractAttributedFigure*) in jHotDraw captured by our system runtime profiling tool. If we merely consider call dependencies and PDG, the system would predict that class *RoundRectangleRadiusHandle* would be the only affected class because in Figure 3 and Table 2 it is obvious that the constructor method (ie, *init()*) in class *RoundRectangleRadiusHandle* is called in method *createHandles()* of class *RoundRectangleFigure*. According to other studies,[39,49]



changed class / predicted class	RoundRectangleFigure			
	call	data dependence	control	data sharing
RoundRectangleRadiusHandle	Y	Y	N	Y
AbstractAttributedFigure	N	N	N	Y

FIGURE 2 An example of actual change impact set prediction

TABLE 2 Partial execution trace log of jHotDraw

Methods	Accessed Variable Name	Accessed Variable Reference ID	JVM Events
RoundRectangleFigure.createHandles()	...	...	Method exit
RoundRectangleRadiusHandle.init()	...	...	Method entry
RoundRectangleFigure.setArc()	*roundrect*	19330724	...
AbstractAttributedFigure.setAttribute()	*attributes*	19330724	...

Abbreviation: JVM, Java Virtual Machine.

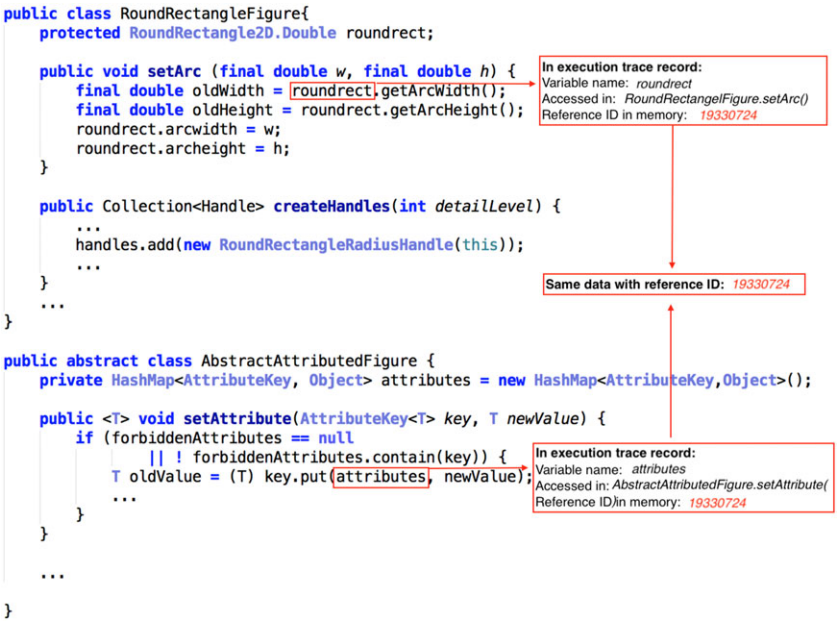


FIGURE 3 An example of code snippet

data dependencies in PDG are extracted by transitive calling of method queries by value define and use. For example, if method M0 is required to be changed. Since M0 calls M1 and M4, and also transitively M2, M3 (both via M1), and M5 (further via M2), the change impact set will not only include M0 but also M1, M4, M2, M3, and M5. Class *AbstractAttributedFigure* would not be predicted as in impact set since there is neither a call dependency nor data dependency captured by PDG between classes *RoundRectangleFigure* and *AbstractAttributedFigure*. However, in the version control history of jHotDraw, we find that class *AbstractAttributedFigure*, often changed together with *RoundRectangleFigure*, is also listed in the actual change impact set submitted by developer in the commit history. The reason is that there was data sharing between *RoundRectangleFigure* and *AbstractAttributedFigure*. The difference between the data sharing dependencies and data dependencies in PDGs is that data sharing dependencies can occur by aggregation of multiple JVM events, no matter whether they are transitive or not. For example, in the code snippet shown in Figure 3, there is a data sharing dependency between *RoundRectangleFigure* and *AbstractAttributedFigure* since the field *roundrect* accessed in *RoundRectangleFigure.setArc()* is eventually accessed by *AbstractAttributedFigure.setAttribute()* as Object value in the *HashMap* typed field *attributes*. Both fields (ie, *roundrect* and *attributes*) use the same data indicated by sharing the same reference ID (19330724) recorded in the execution trace. This execution trace record is shown in Table 2. It is harder to manually observe that in code snippet that there is a data sharing occurs between classes *RoundRectangleFigure* and *AbstractAttributedFigure* because a field *roundrect* accessed in *RoundRectangleFigure.setArc()* is also accessed in *RoundRectangleRadiusHandle.setAttribute()* as a field defined as *attributes*. Fields *roundrect* and *attributes* point to the same piece of data in the memory identified by reference ID 19330724, even though they are declared different variable names. This example tells us actual change impact set prediction probably depends not only on call, data, and control flow relations between classes but also their data sharing relations. Therefore, we will investigate whether and how much data sharing dependencies can support software actual change impact set prediction as a complement to call and control flow dependencies.

3 | PROPOSED ACTUAL CHANGE IMPACT SET PREDICTION (CHIP) FRAMEWORK

3.1 | Usage scenario

Let us consider the following scenario described in Section 2.3. A developer starts by changing identified initial set of classes. No automation supports this first step. Automation comes into play to find out if the developer needs to change other dependent classes (the actual change impact set) and to prevent changes like relocating methods inside a class from affecting the actual change impact set prediction. For this, we are only interested in analyzing and making use of class-level dependencies to predict class-level actual change impact set. Specifically, our approach leverages not only existing code dependencies such as call dependencies but also dependencies based on data sharing, which enhances the existing data flow and control dependencies. Further, we find that the proposed CHange Impact set Prediction (CHIP) framework is particularly useful in predicting actual change impact set in the following commonly observed change activities.[50] We speak of actual impact scenarios (discussed in Section 7):

- "Moving Refactoring": A piece of code is moved from one class to another class.
- "Remove Class or Statement": A class or statement in a class is removed.

- "Bug Fixing": Initial changes are made to fix bugs.
- "Functional Improvement": Initial changes are made to add new features for the system.
- "Code Replacement": A piece of code is replaced.

3.2 | CHange Impact set Prediction framework

Figure 4 provides an overview of CHIP framework to automatically predict the actual change impact set in source code with an initial set of changes made by developers. In particular, we will explore whether dependencies based on data sharing captured by our data sharing dependency generator can complement call dependencies in actual change impact set prediction. A supporting tool for all components in the framework has been implemented.

Our framework leverages both call and data sharing dependencies. Figure 4 shows the overview of CHIP. The preparation of dependencies is implemented in the "Preprocessing" component (B). In this component, we capture all execution data through an execution trace profiling tool (B-1) and extract both call and data sharing dependencies using the corresponding algorithms (B-2). The "Change Impact set Prediction" component automates the prediction of actual change impact set (C-1), dependency frequency filtering (C-2) and shared data type *idf* filtering (C-3) with thresholds, in which both thresholds is selected by applying adaptive learning approach. In this step, CHIP makes actual change impact set prediction based on various combinations of code dependencies, as shown in Section 6.

4 | DATA SHARING DEPENDENCY GENERATOR

We capture data sharing dependencies via dynamic analysis from execution traces. Since data sharing dependencies are captured between 2 methods, the class-level data sharing dependencies are simply the aggregation of method-level dependencies between 2 classes. This step is realized by 2 components in the CHIP framework: (a) system runtime profiling (Section 2.1-1) and (b) data sharing dependency generator (discussed next).

Similar to call dependencies, data sharing dependencies can be automatically generated via analyzing the events in the execution trace database. Among all JVM events, we find that "method entry," "method exit," "field access," and "field modification" are the ones closely related to data sharing relations. We have developed our own data sharing dependency generation method and tool to capture class-level data sharing dependencies. This implementation is currently limited to Java systems but should be easily extensible to other languages. Our tool incorporates 4 variants of data sharing dependency generator for analyzing different JVM events: "field access" events, "field modification" events, "method entry," and "method exit" events, as well as the situation of crossing over different events. The algorithms for capturing data sharing dependencies (Sections 4.1 to 4.4) are available at https://www.dropbox.com/s/wp9pfcptubw9g7p/DataDep.pdf?dl=0.

4.1 | Field access

Data sharing dependency in a "field access" (FA) event is captured when 2 classes access the same field object (variables sharing the same reference ID) through a "field access" event. In JVM, each object is assigned with a unique hash code as an identifier (note: this is different from the objects' nonunique hash code). If different parameters refer to the same object, then they are assigned with the identical identifier.

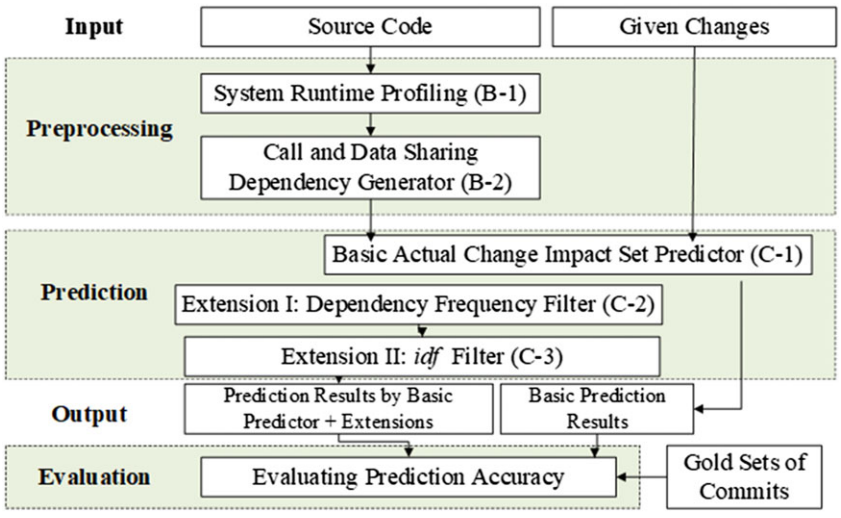


FIGURE 4 Overview of software actual CHange Impact Set Prediction framework

4.2 | Field modification

For data sharing dependency in "field modification" (FM), the process is almost the same as in FA except that this object is modified in "field modification" event. For the object being modified, its unique identifier shall be updated by a new one.

4.3 | Parameter passing

Data sharing dependency for Parameter Passing (PP) is captured through "method entry" and "method exit" events. Both events provide a way to inspect variables (including parameters and return value) created or received by a method. Similarly, if the objects share the same unique identifier, they are thus identical. The return value is differentiated by its attribute "CurMReturnValue" as a unique identifier. The class that initially accesses this return value is the receiver.

4.4 | Cross events

In this case, data sharing dependency is captured when it is observed across any of the 4 different JVM events: "field access," "field modification," "method entry," and "method exit." If affected data are identical based on their unique identifiers, both classes are considered data sharing dependent on each other even though they may reach objects in different JVM events.

Figure 5 shows 7 examples involving the data sharing dependencies captured in 4 different situations in jHotDraw. Based on Figure 5 CHIP constructs a data sharing dependency set as shown in Figure 6 simply by adding the dependencies as lines between the classes. Specifically, a data sharing dependency link between 2 classes is added when any method in each class accesses, modifies, or sends/receives the same piece of data as identified by its unique identifier.

5 | ACTUAL CHANGE IMPACT SET PREDICTOR (CHIP)

In general, CHIP takes 2 inputs: (1) a set of methods/classes that the developer already made the change and (2) The call and data sharing dependencies captured using the approaches described in Sections 2 and 4. We have developed 2 variants of the predictor: (1) basic actual change impact set predictor and (2) two augmented basic predictors with extensions, one with dependency frequency filter and the other with a data type inverse document frequency (*idf*) filter in order to improve the precision of the basic predictor, shown as Steps C-1, C-2, and C-3 in Figure 4.

5.1 | Basic actual change impact set predictor

The basic predictor can predict the actual change impact sets at both method- and class-levels based on the call and/or data sharing dependencies in code. For the empirical evaluation in Section 7, we have developed 4 variants of the basic predictor based on (1) call dependencies only (P_c), (2) data sharing dependencies only (P_d), (3) both call and data sharing dependencies (P_{cd}), (4) PDG (P_{pdg}), and (5) evolutionary couplings (P_e). P_{pdg} and P_e are included for comparison with the other CHIP variants (P_d, P_{cd}) based on data sharing and/or call dependencies. Meanwhile, at both method- and

- **Parameter Passing:** ***RoundRectangleRadiusHandle.locate()*** receives the same data (reference ID=19330724) in "**method entry**" event returned by ***RoundRectangleFigure.createHandles()*** in "**method exit**" event
- **Parameter Passing:** ***DiamondFigure.drawStroke()*** receives the same data (reference ID=2642538) in "**method entry**" event returned by ***RoundRectangleFigure.drawFill()*** in "**method exit**" event
- **Field Access:** ***AbstractAttributedFigure.setAttribute()*** and ***RoundRectangleFigure.setArc()*** accesses the data (reference ID=19330724) in "**field access**" event
- **Parameter Passing:** ***RoundRectangleFigure.drawFill()*** receives the same data (reference ID=26904053) in "**method entry**" event returned by ***TextAreaFigure.drawParagraph()*** in "**method exit**" event
- **Field Modification:** ***ConnectionTool.mouseReleased()*** modifies the same data that is also modified by ***RoundRectangleFigure.clone()*** (old reference ID=19330724, updated reference ID=5634481)
- **Cross Events:** ***RoundRectangleFigure.findConnector()*** accesses the same data (reference ID=20773904) in "**field access**" event that is also accessed by ***LineConnectionFigure.setNode()*** in "**method exit**" event
- **Cross Events:** ***RoundRectangleFigure.drawFill()*** access the same data (reference ID=11270298) in "**method exit**" event that is also accessed by ***TriangleFigure.drawFill()*** in "**field access**" event

FIGURE 5 Examples of data sharing dependencies

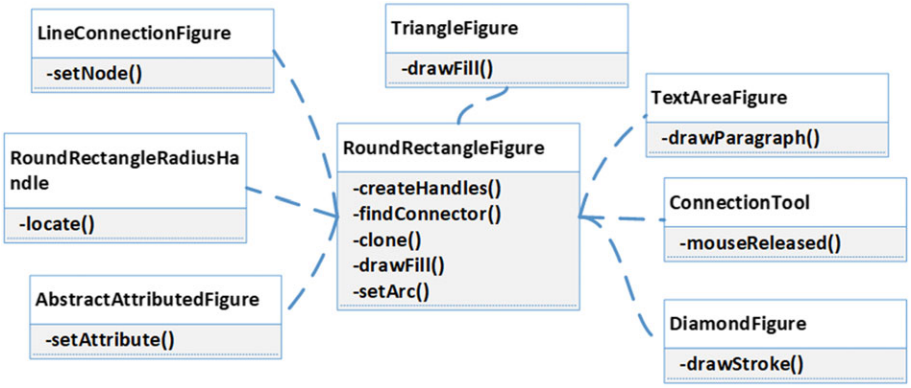


FIGURE 6 An example data sharing dependency set generated from Figure 5

class-levels, we consider the effect of the order of changes. When developers make a set of relevant changes in order, we believe that the very first change he/she makes have a leading impact on the subsequent changes in the actual change impact set, while the impacts of subsequent changes are diminishing. Therefore, in our algorithm, we assign a higher weight on the very first change and start actual change impact set predictions based on code dependencies with the very first change. Then, for other methods/classes after the very first change, only shared predictions based on a pair of methods/classes are included in the predicted change impact set. Equation 1 formalizes the computation of the prediction taking the order of changes into account:

$$P = A_1 + \sum_{i,j=2}^{n} A_i \cap A_j. \tag{1}$$

A_1 denotes the prediction made based on the code dependencies with the very first change in an actual change impact set. A_i and A_j are prediction set of ith/jth changed method/class in the given set of changes of size n and $A_i \neq A_j$.

However, that was the very first one in the actual change impact set from the commit history is enclosed and unknown to others except developers themselves. Therefore, in our experiment, to determine the first changed method/class in a a given change set, we greedily run through all possible first changed method/class.

5.1.1 | Method-level prediction

The algorithm predicts the methods that need to be changed together with the method being changed by developer. The following example explains how the basic predictor predicts actual change impact sets at method-level. The CIA process starts with an initial set of method-level changes that are made by the developer. With an initial set of changes {m1, m2, m3}, if m1 is the first method being changed by developer, the predictor (P_c or P_d) will search the call or data sharing dependencies to determine which method has the call or data sharing dependency on m1. If P_c detects that m4 has call dependency with m1, it will predict m4 as in the actual change impact set. Then, for m2 and m3, because of order of changes effect, only their shared predication are considered. For instance, if P_c detects that m5 has call dependencies on both m2 and m3, it will predict m5 in the actual change impact set. So the actual change impact set of {m1, m2, m3} are {m4, m5}. Similarly, P_d detects that m6 has data sharing dependency on m1 and m7 has data sharing dependencies on both m2 and m3, it will predict {m6, m7} as the change impact set. P_{cd} make prediction based on both call and data sharing dependencies. For the aforementioned 2 examples, P_{cd} will predict {m4, m5, m6, m7} as the change impact set. In method-level prediction, besides direct dependencies, 2-step transitive dependencies are also considered taken into account because many changes are caused by 2-step transitive impact of initial changes. For instance, if {m1, m2, m3} are given methods being changed, {m4, m5, m6, m7} are method-level change impact set based on direct dependencies with the given change set. If m8 is detected to have call/data sharing dependency with m4, m8 will also be predicted in the change impact set since m8 has 2-step transitive dependency with given changed method via m4.

5.1.2 | Class-level prediction

The class-level actual change impact set can be derived from the method-level change impact set prediction results. It is a simple aggregation of the method-level results by their owner classes. For example, with the same initial set of method-level changes {m1, m2, and m3}, P_{cd} will predict {m4, m5, m6, and m7}. Assume m1 belongs to class c1, m2, and m3 belong to class c2, m4, and m5 belong to class c3, m6, and m7 belong to c4 and c5, respectively. We can aggregate the method-level results and make the change impact set prediction at the class-level accordingly. If classes {c1,

c2} are the given set of changes being made by developer, P_{cd} will predict its actual change impact set as {c3, c4, c5}. Algorithm 1 illustrates how the class-level change impact set is predicted.

Algorithm 1. Basic Actual Change Impact Set Predictor

1: **Input:**
2: $\{comb_1, comb_2, \ldots, comb_n\}$: Given changed classes combinations from historical change commits for evaluation
3: Call dependencies
4: Data sharing dependencies
5: PDG includes all PDG couples
6. Evolutionary includes all evolutionary couples
7: **Output**: co-change prediction results
8: **Method:**
9: **for all** $comb_i \subseteq \{comb_1, comb_2, \ldots, comb_n\}$ **do:**
10: **for each call dependency (*cd*):**
11: **if (*comb*$_i$ contains one class in *cd* and this class is first changed class) or the other class predicted by multiple classes in *comb*$_i$ with call dependencies:**
12: add the other class in *cd* into prediction result of call;
13: add the other class in *cd* into prediction result of call+data;
14: **end for**
15: **for each data sharing dependency (*dd*):**
16: **if (*comb*$_i$ contains one class in *dd* and this class is first changed class) or the other class predicted by multiple classes in *comb*$_i$ with data sharing dependencies:**
17: add the other class in *dd* into prediction result of data sharing;
18: add the other class in *dd* into prediction result of call+data;
19: **end for**
20: **for each dependency (*pdg*) in PDG:**
21: **if (*comb*$_i$ contains one class in *pdg* and this class is first changed class) or the other class predicted by multiple classes in *comb*$_i$ with PDG:**
22: add the other class in *pdg* into PDG prediction result;
23: **end for**
24: **for each evolutionary couple (e):**
25: **if (*comb*$_i$ contains one class in *e* and this class is first changed class) or the other class predicted by multiple classes in *comb*$_i$ with evolutionary couples:**
26: add the other class in *e* into evolutionary prediction result;
27: **end for**
28: **end for**
29: **Output** co-change prediction results based on call dependencies, data dependencies, call+data dependencies, PDG and evolutionary with duplicated results filtered

Example: Figure 7 shows a portion of combined call and data sharing dependencies from jHotdraw, based on which the basic predictor P_{cd} is constructed to predict the actual change impact set at the class-level based on both call and data sharing dependencies. The solid links represent call dependencies and dashed links represent data sharing dependencies. Based on Figure 7, the basic predictor P_{cd} is able to predict the actual change impact set with class *RoundRectangleFigure*, which are listed in Table 3. "Y" indicates a dependency between 2 classes while "N" indicates no dependency is detected. In this example, 3 classes *BoundsOutlineHandle*, *ResizeHandleKit*, and *AbstractFigure* are predicted in the actual change impact set based on call dependencies, while 6 classes *AbstractAttributedFigure*, *TextAreaFigure*, *ConnectionTool*, *LineConnectionFigure*, *DiamondFigure*, and *TriangleFigure* can only be predicted through data sharing dependencies. The class *RoundRectangleRadiusHandle* is predicted by both call and data sharing dependencies.

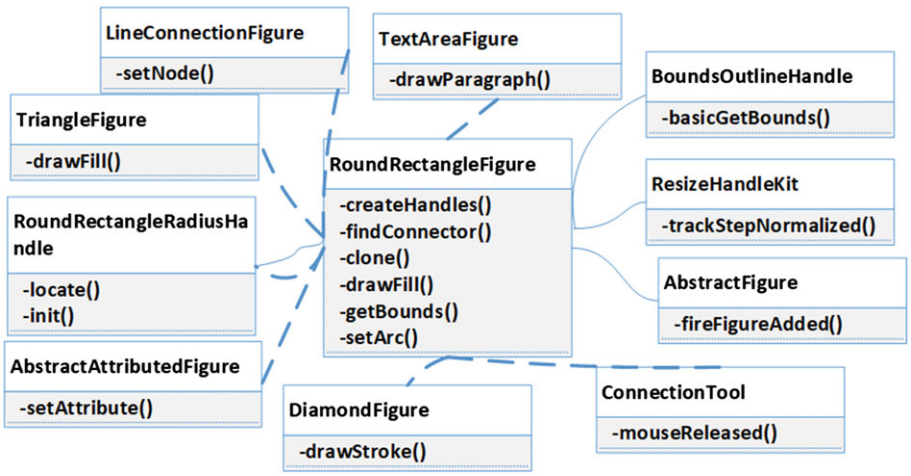


FIGURE 7 An example of comprehensive set of dependencies with call dependencies (solid links) and data sharing dependencies (dash links) for the basic predictor

TABLE 3 Predicted actual change impact set with class *RoundRectangleFigure* by basic predictor

		RoundRectangeFigure	
		Call	Data
1	*RoundRectangleRadiusHandle*	Y	Y
2	*BoundsOutlineHandle*	Y	N
3	*ResizeHandleKit*	Y	N
4	*AbstractFigure*	Y	N
5	*AbstractAttributedFigure*	N	Y
6	*TextAreaFigure*	N	Y
7	*ConnectionTool*	N	Y
8	*LineConnectionFigure*	N	Y
9	*DiamondFigure*	N	Y
10	*TriangleFigure*	N	Y

Section 7 evaluates the predictability of CHIP at both method- and class-levels in order to compare with different state-of-the-art approaches, which investigate actual change impacts at either method-level or class-level.

5.2 | Extension I: Dependency frequency filter

Nevertheless, we notice that both call and data sharing dependencies introduce heavy noise (false positives), which in turn compromises the precision of prediction. One reason is that the methods/classes, which have less dependencies on the given change set, are treated equally important with the ones which have more dependencies on the given change set. To resolve this issue, we introduce a novel Dependency Frequency Filter extension to the basic predictor using combined call and data sharing dependencies (P_{cd}). This extension is devised based on how frequent a method/class is predicted with the given change set. Here is an example. We start from predicting the actual change impact set at the method-level, with an initial set of methods {m1, m2, m3} being changed (ie, given change set) and the dependency frequency filter threshold set as 2, if P_{cd} detects that m4 only has one data sharing dependency with m1, while m5 has totally 3 call and/or data sharing dependencies with m2 and m3, then m5 will be predicted in the actual change impact set of {m1, m2, m3} rather than m4 since m4 only has one dependency link with the given change set, which is below the dependency frequency filter threshold 2. Next, we will make actual change impact set prediction at the class-level with dependency frequency filter threshold as 2 as well. Assume m1 belongs to class c1 and both m2 and m3 belong to class c2. If m4 is owned by class c3, then c3 will have one data sharing dependency with c1. If m5 is owned by class c4, c4 will have 6 call and/or data sharing dependencies with c2. In this case, c3 will be filtered out and not be predicted in the actual change impact set because it only has one dependency link with the given change set {c1, c2}, which is below the dependency frequency filter threshold of 2.

Here, we employ adaptive learning to determine the value of the threshold. First, we randomly split the commits of each system fivefolds. We reserve onefold as an evaluation set and use the other fourfolds as training sets. We then train on the training sets and test the learnt threshold on the held-out evaluation set. We compute the F2-score—a measure that combines precision (fraction of actual changes in prediction) and recall (fraction of actual changes that are predicted) of the prediction results of evaluation set. If a better F2-score is achieved, the threshold is adjusted. This process is repeated. Finally, the optimal threshold corresponding to the highest value of F2-score is chosen.

5.3 | Extension II: Inverse data frequency (*idf*) filter

We also notice that data sharing dependencies themselves introduce heavy noise (false positives), which also compromises the precision of predictions. The reason for this kind of noise is that all shared data types are considered equally important for predicting the actual change impact set. To tackle this problem, we extend the basic predictor relying on combined call and data sharing dependencies (P_{cd}) with a novel shared data type inverse data frequency (*idf*) filter.

For a data sharing dependency across classes, methods in 2 distinct classes must share data. This data sharing may involve one or multiple variables or parameters and may cover multiple data types. However, not all the data types provide equally useful implication for the actual change impact set. The column "Occur" in Table 4 shows how often a data type occurs in all data sharing dependencies generated from the iTrust system. The maximum occurrence of a data type can be the total number of data sharing dependencies (for iTrust, the total number of data sharing dependencies is 92 285) meaning that a data type is shared in every dependencies. The minimum occurrence is one meaning that this data type is only shared once. For example, data type *java.lang.String* is shared by classes much more frequently than other data types. A reasonable conjecture is that *java.lang.String* is a commonly shared data type to pass string data across many methods in the iTrust system, which means that this kind of data types is thus too "general." If a data sharing dependency between 2 classes is upon a number of "general" data types, this data sharing dependency is probably too "general" to imply an actual change impact set in practice, which should be excluded from our actual change impact set prediction results.

TABLE 4 Top 5 objects with the lowest *idf*

	Object Combination	Occur	*idf*	Normalized *idf*
1	*java.lang.String*	73024	0.23	0.0
2	*java.sql.ResultSet*	51178	0.59	0.032
3	*java.lang.Class*	45030	0.72	0.043
4	*java.util.List*	41469	0.80	0.051
5	*edu.ncsu.csc.itrust.beans.PersonnelBean*	38645	0.87	0.057

Hence, we borrow the idea of Inverse Document Frequency[51] from information retrieval to define our Inverse Data Frequency to weigh the importance of each data type for actual change impact set prediction. Inverse Data Frequency (*idf*) is the measure of occurrence of a data type across all data sharing dependencies in a system. Specifically, it is defined as follows:

$$idf = log\left(\frac{N}{n_d}\right), \tag{2}$$

where N is the total amount of data sharing dependencies and n_d is the occurrence of a data type across all data sharing dependencies. For the purpose of generalizing the *idf* across different systems, we normalize all the *idfs* in a system as follows:

$$idf_{norm} = \frac{idf - idf_{min}}{idf_{max} - idf_{min}}, \tag{3}$$

idf_{min} and idf_{max} denote the lowest and highest *idf* in a system, respectively. The normalization ensures that idf_{norm} falls between 0 and 1. Since each data sharing dependency could have more than one shared data type, the *idf* for each data sharing dependency is calculated as idf_{accum}, which is normalized accumulation of idf_{norm} of all shared data types in each data sharing dependency. In principle, *idf* will value rare data types higher than common data types (eg, *java.lang.String*). The threshold for idf_{accum} on each system is selected using adaptive learning similar to Extension I (Dependency Frequency Filter) to determine that data sharing dependences are too "general" to be included in the actual change impact set prediction.

Figure 8 shows the trimmed call and data sharing dependencies after applying the *idf* filter on the call and data sharing dependencies of jHotDraw in Figure 7. Table 5 lists the predicted actual change impact set with class *RoundRectangleFigure* by CHIP with *idf* extension. In this example, there are multiple method-level data sharing dependencies between classes *ConnectionTool* and *RoundRectangleFigure*. If we set the threshold of *idf* as 0.03, *ConnectionTool* is eliminated since among all method-level data sharing dependencies between *ConnectionTool* and *RoundRectangleFigure* the highest *idf* falls below 0.03. Besides, although *RoundRectangleFigure* and *RoundRectangleRadiusHandle* are linked by both call and data sharing dependencies, *RoundRectangleRadiusHandle* is still eliminated because the highest *idf* of data sharing dependencies between *RoundRectangleFigure* and *RoundRectangleRadiusHandle* is also below the *idf* threshold.

6 | EXPERIMENT DESIGN

Our experiments use CHIP to predict actual change impact sets in source code with an initial set of changes. To investigate the effects of data sharing dependencies on actual change impact set prediction, we compare the performance among the CHIP variants including the predictors

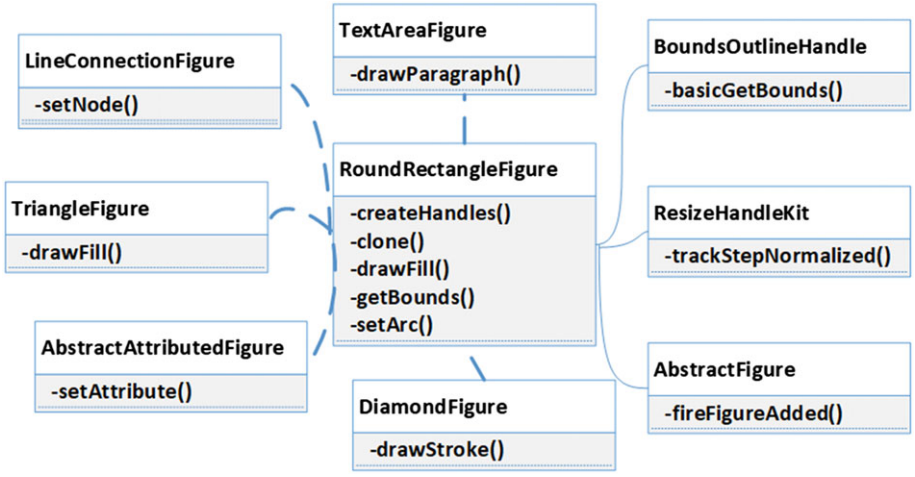


FIGURE 8 Trimmed call dependencies (solid links) and data sharing dependencies (dash links) after applying the *idf* filter on Figure 7

TABLE 5 Predicted actual change impact set with class *RoundRectangleFigure* by basic predictor + *idf* extension

		RoundRectangeFigure	
		Call	**Data**
1	*BoundsOutlineHandle*	Y	N
2	*ResizeHandleKit*	Y	N
3	*AbstractFigure*	Y	N
4	*AbstractAttributedFigure*	N	Y
5	*TextAreaFigure*	N	Y
6	*LineConnectionFigure*	N	Y
7	*DiamondFigure*	N	Y
8	*TriangleFigure*	N	Y

based on call dependencies (P_c), data sharing dependencies (P_d), combined call and data sharing dependencies (P_{cd}), and P_{cd} with frequency and *idf* extensions (P_{cd}+ext). We also compare the performance of CHIP variants built on data sharing dependencies and the combined call and data sharing dependencies (P_d, P_{cd}) with PDG (P_{pdg}) and evolutionary couplings (P_e).

6.1 | Datasets

Our experiments are conducted on 4 open source Java systems, which have been developed and evolved for a total of over 40 years by hundreds of developers all over the world: jEdit 4.3, a mature programmer's text editor; iTrust 13.0, a medical management system; jHotDraw 7.2, a Java GUI framework for graphics; GanttProject 2.0.9, a cross-platform project scheduling and management system. The choice of these 4 systems is motivated by the need of (1) historical change commits (ie, gold sets of change commits) for evaluation; (2) systems belonging to different problem domain; (3) systems of different sizes that are neither too small nor too large to allow developers to assess dependencies among methods/classes of an entire system; and (4) possibility of capturing high quality execution trace by runtime profiling. Table 6 summarizes the characteristics of the 4 systems.

In class-level prediction, for a commit of size *n*, which contains *n* changed classes, we randomly pick *i* classes ($1 \leq i \leq n$) from the commit as the initial set of changes. Similarly, in method-level, we randomly pick *i* methods ($1 \leq i \leq n$) from the commit with *n* changed methods as the initial set of changes. Then, we ask CHIP to predict the *n-i* actual change impact set in the commit. We exhaustively explore all combinations of size *i* before increasing the size of the initial change set by 1 and repeat the process. We permutate all subsets in a commit as the given change sets greedily since developers could start by changing any subset for a change task. Hence, our evaluation tests all possible scenarios how changes may unfold. Meanwhile, since the order of changes are taken into consideration, we present "best case" defined as the change order with the most optimal recall achieved. "Best case" imitates the actual scenario that developer makes the change. And "overall" permutates all possible scenarios of changes by summarizing results of all possible orders of changes.

6.1.1 | Data preprocessing

The evaluation excludes the testing code (eg, jUnit tests) because change impact between testing code and system code are usually expected and for this reason are less important.[52] Furthermore, we do not consider changed classes in the commit if the changes are merely (1) independent formatting changes (eg, removing an empty line), or (2) API changes without reporting the changed classes. Although many existing impact set prediction studies exclude merging commits or large classes, our study incorporates them all for a more comprehensive study purpose.

To investigate the effects of data sharing dependencies on predicting various change impact scenarios, we classify the commits in 4 systems based on their change activities documented in commit messages into "move refactoring", "remove class or statement", "bug fixing", "functional improvement" and "code replacement," which dominate in those systems. Section 3 has defined these change impact scenarios.

TABLE 6 Characteristics of 4 open source systems

System Version	iTrust 13.0	GanttProject 2.0.9	jHotDraw 7.2	jEdit 4.3
Size (KLOC)	43	45	72	109
# of classes	461	475	546	503
Evaluated commits	c216–c256	Most recent 76 commits to 2.0.9	c518–c798	c7998–c8340
Mined SVN logs for evolutionary couplings	Since 2009-08-18	Since 2010-12-08	Since 2004-02-01	Since 2006-09-17
# of call dependencies	5954	5055	4550	6463
# of data sharing dependencies	92285	108779	112531	137370

6.2 | Evaluation metrics

We measure precision (*P*) recall (*R*) and F2-score of actual change impact set prediction. Here, F2-score instead of F-score is used because recall is more important than precision in actual change impact set prediction. We here use an example to explain how the pair of metrics are measured for an individual prediction experiment. For a commit C containing a set of changed classes {c1, c2, c3} and a starting set of changes {c1}, which is known being changed by developer, if the predicted change impact set are classes {c1, c2, c4, c5}, then c1 and c2 are "true positive," c4 and c5 are "false positive," c3 is "false negative." Thus, recall is measured as 2/3=66.7%, precision is 2/4=50% and F2-score is 5*66.7%*50%/(66.7%+4*50%)=62.5%. To measure the overall prediction accuracy for an entire system, we sum up the "true positive" (TP_{total}), "false negative" (FN_{total}), and "false positive" (FP_{total}) from all the prediction experiments on individual commits in a system and calculate the overall recall, precision and F2-score as follows:

$$recall_{total} = \frac{TP_{total}}{TP_{total} + FN_{total}}, \tag{4}$$

$$precision_{total} = \frac{TP_{total}}{TP_{total} + FP_{total}}, \tag{5}$$

$$f2\text{-}sccore_{total} = \frac{5{*}recall_{total}{*}precision_{total}}{recall_{total} + 4{*}precision_{total}}. \tag{6}$$

6.3 | Time efficiency compared with PDG

Table 7 shows the time efficiency of CHIP built on data sharing dependencies compared with PDG. Experiments were performed on a computer with Intel Core i5 2.8GHz (configured with one thread and 8GB RAM). In terms of the dependency extraction time, as we expected, extracting all 4 types of data sharing dependencies costs more time than PDG since data sharing dependencies contains much finer-grained information than the PDG. In all cases, the time of extracting data sharing dependencies from execution traces generated in software testing phase are within affordable 82 minutes. Compared with PDG, it takes almost the same amount of time for CHIP to make each prediction of actual change impact set.

7 | EXPERIMENT RESULTS

This section presents the experiment results to answer our research questions.

7.1 | RQ1. Do data sharing dependencies complement call dependencies in actual change impact set prediction?

Results: The results of performance metrics of call, data sharing dependencies and combined call and data sharing dependencies on the 4 systems are shown in Table 8 (class-level) and Table 9 (method-level). Performance metrics are measured in 2 modes: *best case* and *overall*. As described in Section 5.1, since the order of changes is considered. The order is known by developers who made those changes, but it is unknown to researchers. *Best case* is the most likely change order according to the most optimal recall. *Overall* is the measure of all possible orders. Comparing the prediction results of P_{cd} with P_c, in class-level predictions recall of P_{cd} outperforms P_c by 3.1% to 17.4% overall and 3.4% to 7.4% in best case over the 4 systems. In method-level predictions, recall improves by 3.4% to 32.1% overall and 3.7% to 7.1% in best case by P_{cd} compared with P_c. Adding extension I (Dependency Frequency filter) and extension II (*idf* filter) to P_{cd}, prediction precision is also greatly improved. Table 10 shows that for class-level predictions in iTrust, GanttProject, jHotDraw, and jEdit false positives are reduced by 97.2%, 93.6%, 72.0%, and 85% using P_{cd}+ext compared with P_{cd}, while true positives are only compromised by only 3.4%, 8.9%, 5.6%, and 5.8% correspondingly. Comparing the prediction results of P_{cd} with P_c, Table 8 shows that F2-score of P_{cd}+ext outperforms P_c by 2.3% to 35.8% overall and as much as 38.3% in best case over 4 systems. In class-level prediction, best case P_{cd}+ext ensures recall over 90% and precision over 20% while in overall P_{cd}+ext achieves recall larger than 75% while keeping precision greater than 10%. However, in jHotDraw, we found that P_c achieves better F2-score than P_{cd}+ext, since intra-class dependencies are not

TABLE 7 Time efficiency of P_{cd}+ext and P_{pdg} on 4 systems

	Dependency Extraction Time		CHIP Prediction Time	
Systems	Data Sharing Dependency	PDG	Data Sharing Dependency	PDG
iTrust	27m21s	5m18s	44s	44s
GanttProject	27m54s	2m37s	1m4s	1m3s
jHotDraw	35m34s	4m20s	57s	1m7s
jEdit	81m16s	1m13s	47s	47s

Abbreviation: PDG, Program Dependence Graph.

TABLE 8 Four systems at class-level: precision (P(%)), recall (R(%)), and F2-score (F2(%)) by P_{cd}+ext, P_{cd}, P_d, P_c, P_{pdg}, and P_e

Systems	CHIP Variants	Best Case			Overall			Overall-Single		
		R	P	F2	R	P	F2	R	P	F2
iTrust	P_{cd}+ext	96.6	40.0	75.3	90.6	40.7	72.7	82.8	25.0	56.6
	P_{cd}	100.0	1.9	8.8	100.0	2.4	11.0	100.0	1.7	7.8
	P_d	100.0	2.1	9.5	100.0	2.5	11.5	100.0	1.8	8.2
	P_c	96.6	10.7	37.0	82.6	11.5	36.9	72.4	5.9	22.2
	P_{pdg}	96.6	7.9	29.9	82.6	7.1	26.5	72.4	4.6	18.4
	P_e	89.7	9.7	33.8	81.2	7.4	27.0	65.5	7.3	25.2
GanttProject	P_{cd}+ext	90.6	53.5	79.6	80.5	19.3	49.3	60.1	10.0	29.1
	P_{cd}	99.5	7.5	29.0	91.1	5.3	21.6	77.3	3.5	14.9
	P_d	99.0	7.6	29.2	89.4	5.5	22.0	74.4	3.5	14.7
	P_c	92.1	23.3	57.9	76.2	18.6	47.0	45.8	11.7	28.9
	P_{pdg}	97.0	7.1	27.5	93.0	5.6	22.5	58.6	5.9	21.0
	P_e	85.2	23.5	55.9	63.8	22.4	46.6	30.0	9.4	20.1
jHotDrw	P_{cd}+ext	93.1	22.6	57.4	75.4	11.1	34.9	50.7	9.5	27.1
	P_{cd}	98.6	8.0	30.1	78.7	8.5	29.7	66.4	4.4	17.4
	P_d	96.2	7.8	29.4	76.7	8.5	29.4	61.9	4.2	16.5
	P_c	93.1	21.4	55.7	65.8	25.6	50.1	46.0	12.5	30.0
	P_{pdg}	96.3	9.5	34.0	72.4	9.8	31.7	50.9	6.1	20.7
	P_e	69.7	48.0	63.9	57.8	57.6	57.8	25.5	22.0	24.7
jEdit	P_{cd}+ext	93.3	24.0	59.2	87.7	13.8	42.3	77.1	12.4	37.8
	P_{cd}	99.0	4.8	20.2	92.5	3.4	14.9	90.5	3.0	13.1
	P_d	94.3	4.4	18.6	84.1	3.5	15.1	77.1	2.8	12.3
	P_c	95.2	14.0	44.1	89.4	7.2	27.3	82.9	6.8	25.5
	P_{pdg}	91.4	10.7	36.5	87.2	7.0	26.6	73.3	6.3	23.5
	P_e	78.1	55.4	72.2	70.9	43.7	63.1	46.7	34.5	43.6

Abbreviation: CHIP, CHange Impact set Prediction.

TABLE 9 Four systems at method-level: precision (P(%)), recall (R(%)), and F2-score (F2(%)) by P_{cd}+ext, P_{cd}, P_d, P_c, P_{pdg}, and P_e

Systems	CHIP Variants	Best Case			Overall			Overall-Single		
		R	P	F2	R	P	F2	R	P	F2
iTrust	P_{cd}+ext	96.3	100.0	97.0	89.2	84.2	86.0	81.5	100.0	84.6
	P_{cd}	100.0	0.8	3.7	97.3	0.7	3.3	96.3	0.7	3.5
	P_d	100.0	0.8	3.7	97.3	0.7	3.5	96.3	0.8	3.8
	P_c	96.3	3.7	16.0	86.5	3.0	13.2	81.5	2.7	11.8
	P_{pdg}	96.3	2.3	10.6	89.2	1.7	8.0	85.2	1.6	7.2
	P_e	96.3	29.5	66.3	86.5	14.7	43.7	81.5	15.1	43.3
GanttProject	P_{cd}+ext	91.5	63.1	84.0	82.1	58.4	76.0	60.6	54.1	59.2
	P_{cd}	99.2	3.2	14.0	92.9	2.8	12.3	79.6	2.3	10.3
	P_d	99.2	3.3	14.4	92.6	2.8	12.3	79.1	2.3	10.4
	P_c	92.1	18.1	50.7	60.9	15.8	38.7	22.0	7.5	15.9
	P_{pdg}	96.6	4.7	19.5	73.0	4.8	18.9	39.4	2.8	10.8
	P_e	98.1	11.4	38.9	90.0	11.2	37.3	79.9	9.1	31.4
jHotDrw	P_{cd}+ext	91.1	52.8	79.6	81.5	12.5	38.8	68.6	12.1	35.4
	P_{cd}	97.8	0.8	4.0	95.3	0.8	3.9	91.8	0.8	3.7
	P_d	97.5	0.8	4.0	95.2	0.8	3.9	91.5	0.8	3.7
	P_c	92.8	10.1	35.2	81.3	9.0	31.1	68.4	7.5	26.1
	P_{pdg}	94.5	2.6	11.7	85.3	2.5	11.1	75.4	2.0	9.2
	P_e	90.3	9.9	34.4	85.3	8.5	30.3	76.0	7.4	26.7
jEdit	P_{cd}+ext	92.9	57.0	82.5	82.9	70.8	80.2	62.1	43.3	57.2
	P_{cd}	94.2	2.3	10.4	87.8	2.5	11.1	74.3	1.5	7.0
	P_d	94.2	2.4	10.7	87.3	2.5	11.2	72.9	1.5	7.1
	P_c	94.2	15.5	46.8	84.5	23.2	55.3	65.0	10.3	31.6
	P_{pdg}	96.1	9.7	34.6	85.2	15.8	45.4	65.0	6.8	23.9
	P_e	92.9	42.3	74.9	81.3	54.2	74.0	57.9	29.3	48.4

Abbreviation: CHIP, CHange Impact set Prediction.

employed in class-level change impact set prediction. An intra-class dependency means code dependency between 2 methods in the same class. Intra-class dependencies within the same class are not leveraged in class-level actual change impact set prediction across different classes. However, for method-level predictions, intra-class dependencies are fully employed. Therefore, Table 9 shows that at method-level F2-score of P_{cd}+ext outperforms P_c by 7.7% to 72.8% overall and 33.3% to 81.0% in best case over the 4 systems. In method-level prediction best case P_{cd}+ext achieves recall over 90% and precision over 50% while in overall mode, P_{cd}+ext achieves recall larger than 81% with precision greater than 10%. In general, data sharing dependency complements call dependency in actual change impact set prediction.

TABLE 10 Prediction count before and after extensions applied on P_{cd}

Systems	CHIP Variants	True Positives(Reduced by)	False Positives(Reduced by)
iTrust	P_{cd}	29	1495
	P_{cd}+ext	28(−3.4%)	42(−97.2%)
GanttProject	P_{cd}	202	2476
	P_{cd}+ext	184(−8.9%)	158(−93.6%)
jHotDrw	P_{cd}	1096	12662
	P_{cd}+ext	1035(5.6%)	3541(−72.0%)
jEdit	P_{cd}	104	2055
	P_{cd}+ext	98(5.8%)	310(84.9%)

Abbreviation: CHIP, CHange Impact set Prediction.

Meanwhile, it is also interesting to learn the performance of CHIP when a single entity (method or class) is changed initially. For class-level prediction, Table 8 ("Overall-single" column) shows that the recall of P_{cd} outperforms recall of P_c by 7.6% to 31.5% in all 4 systems. For method-level prediction, Table 9 ("Overall-single" column) shows that P_{cd} improves recall by 9.3% to 57.6%. After the extensions was added, the F2-score is improved in the range of 0.2% to 34.4% by P_{cd}+ext in class-level and 9.3% to 72.8% in the method-level compared with P_c. Because of the same reason previously mentioned, in jHotDraw P_c achieves better F2-scores than P_{cd}+ext in class-level actual change impact set prediction.

Statistical Testing:To determine whether leveraging call and data sharing dependencies with extensions significantly improves the prediction accuracy over standalone call dependencies, we apply 2-tailed paired *t* test to determine whether the improvement of F2-score by P_{cd}+ext over F2-score of P_c is significant using class-level predictions as an example. Method-level statistical test results are similar. Our null hypothesis is as follows: *There is no difference between F2-score of P_{cd}+ext and F2-score of P_c*. Tables 11 and 12 show that in all 4 systems $P < .0001$, which suggests that F2-scores of P_{cd}+ext are significantly different than F2-scores of P_c in those systems in both class-level and method-level predictions. Since the mean of F2-scores of P_{cd}+ext is greater than the mean of F2-scores of P_c, the statistical test results suggest that F2-score of P_{cd}+ext are significantly larger than F2-score of P_c in all 3 systems (except jHotDraw) with the confidence interval (CI) of 95% under the mean of prediction. For jHotDraw, the mean of F2-score of P_{cd}+ext appears to be less than F2-score of P_c since intra-class dependencies are not employed as discussed earlier. In method-level predictions, taking advantage of the intra-class call and data sharing dependencies, the mean of F2-scores

TABLE 11 Four systems at class-level: paired *t*-test results, mean, and confidence interval (CI) of F2-score by P_{cd}+ext vs P_c, P_{cd}+ext vs P_{pdg}, and P_{cd}+ext vs P_e

	P Value			Mean of F2-score and CI			
	P_{cd}+ext vs P_c	P_{cd}+ext vs P_{pdg}	P_{cd}+ext vs P_e	P_{cd}+ext, %	P_c, %	P_{pdg}, %	P_e, %
iTrust	<.0001	<.0001	.0001352	72.2	48.0	35.4	46.1
				(66.7-77.6)	(41.0-55.0)	(29.1-41.6)	(36.5-55.7)
GanttProject	<.0001	<.0001	<.0001	54.1	52.2	26.2	50.1
				(53.9-54.4)	(52.0-52.5)	(26.0-26.4)	(49.8-50.3)
jHotDraw	<.0001	<.0001	<.0001	40.8	54.3	37.9	58.3
				(40.6-41.1)	(54.1-54.6)	(37.7-38.2)	(58.0-58.5)
jEdit	<.0001	<.0001	.0014272	55.6	41.7	38.2	64.6
				(51.6-59.5)	(37.7-45.8)	(34.8-41.6)	(61.4-67.9)

TABLE 12 Four systems at method-level: paired *t*-test results, mean, and confidence interval (CI) of F2-score by P_{cd}+ext vs P_c, P_{cd}+ext vs P_{pdg}, and P_{cd}+ext vs P_e

	P Value			Mean of F2-score and CI			
	P_{cd}+ext vs P_c	P_{cd}+ext vs P_{pdg}	P_{cd}+ext vs P_e	P_{cd}+ext, %	P_c, %	P_{pdg}, %	P_e, %
iTrust	<.0001	<.0001	<.0001	65.1 (56.9-71.8)	12.6 (10.4-14.9)	8.3 (5.8-10.8)	35.0 (27.7-42.4)
GanttProject	<.0001	<.0001	<.0001	58.5 (53.9-63.0)	37.3 (33.9-40.7)	20.3 (18.9-21.6)	38.6 (36.6-40.6)
jHotDraw	<.0001	<.0001	.0008341	73.3 (67.2-79.4)	32.5 (27.8-37.1)	14.7 (11.1-18.3)	52.6 (44.0-61.2)
jEdit	<.0001	<.0001	.0029973	81.7 (81.4-82.1)	65.5 (65.1-65.9)	59.4 (58.9-59.9)	77.2 (76.8-77.7)

of P_{cd}+ext are greater than the mean of F2-scores of P_c in all 4 systems, which suggest that F2-score of P_{cd}+ext are significantly larger than F2-score of P_c in all 4 systems. Thus, the actual change impact set predictor built on combined call and data sharing dependencies outperforms the predictor built on call dependencies only.

Summary:With an initial given set of changes, in overall and best cases, in average CHIP with combined code dependencies (P_{cd}) can predict more than 90% of the actual change impact sets in both class- and method-levels on the 4 systems, which significantly outperforms CHIP with only call dependencies (P_c). Extended with Dependency Frequency and *idf* filters, precision of CHIP with combined code dependencies (P_{cd}) are largely improved with little compromise of recall. P_{cd}+ext achieves significantly better F2-score than P_c with call dependencies only. We conclude that data sharing dependencies complement call dependencies in actual change impact set prediction.

7.2 | RQ2. How effective is the data sharing dependencies compared with traditional program dependence graphs and evolutionary couplings in actual change impact set prediction?

Results: Table 8 (class-level) and Table 9 (method-level) show that CHIP based on combined call and data sharing dependencies with both Dependency Frequency and *idf* filter extensions (P_{cd}+ext) genrerate significantly better prediction results as compared with that built upon PDG (P_{pdg}). F2-score is improved by 22.7% to 52.1% in best case and 3.2% to 46.2% overall in class-level. F2-score is improved even further by 47.9% to 86.4% in best case and 27.7% to 78.0% overall in method-level.

When comparing with P_e based on evolutionary couplings in class-level predictions, P_{cd}+ext generate better prediction results than P_e in iTrust and GanttProject. F2-score is improved by 41.5% and 23.7% in best case and 45.7% and 2.7% overall in class-level predictions. At method-level prediction, P_{cd}+ext achieves even better prediction results than P_e in all 4 systems, where F2-scores are improved by 7.6% to 45.1% in best case and 6.2% to 42.3% overall. Figures 9 and 10 also show that in all 4 systems, by setting dependency frequency threshold as none, the actual change impact sets predictions made by P_{cd}+ext can achieve increasingly better F2-scores than P_{pdg} at both class- and method-levels with increasing *idf* threshold. Figures 9 and 10 also show that as *idf* threshold increases, P_d+ext can achieve even better F2-score than P_{cd}+ext. However, our goal is to achieve an optimal precision while maintaining recall at high level (above 75% at class-level prediction, above 80% at method-level). In this case, we think P_{cd}+ext is more valuable than P_d+ext.

For the case that only a single class is given as the initial given set of changes, P_{cd}+ext outperforms P_{pdg} in class-level by improving F2-score by 6.4% to 38.2%. Furthermore, in method-level prediction, F2-score is improved by 26.2% to 77.4%. Compared with P_e, P_{cd}+ext achieves better F2-scores for all systems except jEdit in class-level prediction with the F2-score improved by 2.4% to 31.4% (class-level) and 8.8% to 41.3% (method-level).

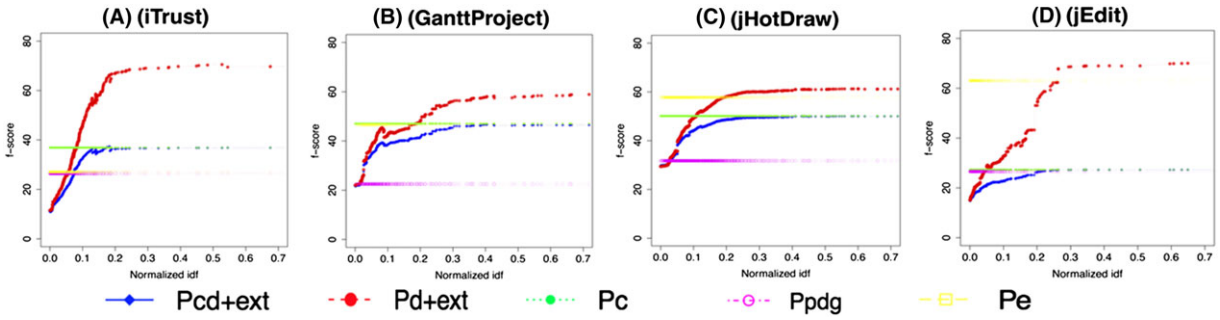


FIGURE 9 A-D. Trend analysis on 4 systems in class-level prediction: F2-score (%) of prediction by P_{cd}+ext, P_d+ext, P_c, P_{pdg}, and P_e under difference *idf* threshold

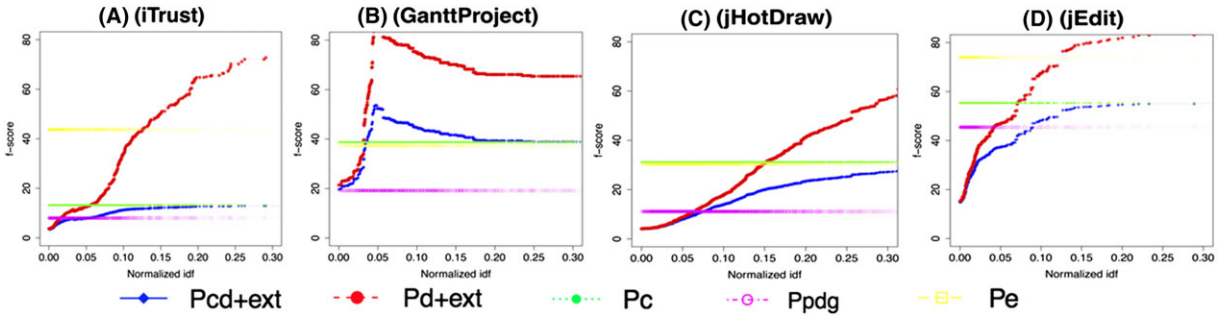


FIGURE 10 A-D. Trend analysis on 4 systems in method-level prediction: F2-score (%) of prediction by P_{cd} + ext, P_d+ext, P_c, P_{pdg}, and P_e under difference *idf* threshold

Statistical Testing: To determine whether combined call and data sharing dependencies with extensions significantly improves the prediction over CHIP with PDG and evolutionary couplings, we apply 2-tailed paired *t* test to determine whether the improvement of F2-score by P_{cd}+ext over F2-score of P_{pdg} and P_e is significant. Our null hypothesis is as follows: (1) *There is no difference between F2-score of* P_{cd}*+ext and F2-score of* P_{pdg}; (2) *There is no difference between F2-score of* P_{cd}*+ext and F2-score of* P_e. Tables 11 and 12 show that in all systems $P < .01$, which suggests that F2-scores of P_{cd}+ext are significantly different than F2-scores of P_{pdg} and P_e in those systems in both class-level and method-level predictions. Moreover, since the mean of F2-scores of P_{cd}+ext are greater than the mean of F2-scores of P_{pdg} and P_e, the statistical test results suggest that F2-score of P_{cd}+ext are significantly larger than F2-score of P_{pdg} and P_e in all 4 systems with the confidence interval (CI) of 95% under the mean of predictions at both class- and method-levels. Our actual change impact set predictor built on combined call and data sharing dependencies outperforms the state-of-the-art approaches using PDG and evolutionary couplings.

Summary: Extended with Dependency Frequency and *idf* filters, given an initial set of changes, CHIP with combined code dependencies (P_{cd}+ext) can achieve significantly better F2-score than that with PDG only over all 4 systems. When comparing with CHIP with evolutionary couplings only in method-level predictions, by employing intra-class code dependencies, P_{cd}+ext can achieve significantly better predictions with F2-score improved in all subject systems as well.

7.3 | RQ3. The combined call and data sharing dependencies (P_{cd}+ext) improve predictions in different change impact scenarios as compared with standalone call dependencies (P_c)?

Results: Tables 13 and 14 show how P_{cd}+ext with specific kind of data sharing dependencies only (FA, FM, PP, CA) or with all 4 kinds combined (overall) can contribute to 5 commonly encountered change impact scenarios. In general, they all achieve significantly better F2-scores than P_c. The results also show that for each of the 5 change impact scenarios, data sharing dependencies captured in specific JVM events better contribute to the F2-score of prediction than others. For example, in remove class or statement scenario, removal of code is sometimes caused by the removal of software features. Thus, in data sharing dependencies from field access JVM events (FA) related to the removed code, the accessed data probably needs to be changed as well. This change will then propagate to the classes that need to access that piece of data. Table 13 shows that P_{cd}+ext with data sharing dependencies captured in FA (P_{cd}+ext-FA) improves F2-score by 20.9% in best case and 21.1% overall compared with P_c in predicting the actual change impact sets due to class or statement removal. Also, in move refactoring scenario,

TABLE 13 Five actual change impact scenarios at class-level: precision (P), recall (R), and F2-score (F2) by P_{cd}+ext-overall, P_{cd}+ext-FA, P_{cd}+ext-FM, P_{cd}+ext-PP, P_{cd}+ext-CE, and P_c

		Best Case			Overall		
Systems	CHIP Variants	R, %	P, %	F2, %	R, %	P, %	F2, %
Move refactoring	P_{cd}+ext-overall	97.8	10.0	35.5	68.8	20.2	46.5
	P_{cd}+ext-FA	83.7	14.6	42.9	62.9	22.9	46.6
	P_{cd}+ext-FM	84.8	16.0	45.5	65.8	20.6	45.8
	P_{cd}+ext-PP	94.6	10.0	35.2	67.3	21.1	46.8
	P_{cd}+ext-CE	97.8	10.6	36.9	64.2	22.9	47.2
	P_c	89.1	8.8	31.5	61.3	14.6	37.3
Remove class or statement	P_{cd}+ext-overall	77.8	10.6	34.3	67.3	10.2	31.8
	P_{cd}+ext-FA	77.8	14.8	42.0	67.3	18.3	43.8
	P_{cd}+ext-FM	77.8	13.0	39.0	67.3	17.1	42.4
	P_{cd}+ext-PP	77.8	10.2	33.4	67.3	10.0	31.4
	P_{cd}+ext-CE	74.1	13.1	38.3	66.0	10.7	32.4
	P_c	77.8	5.4	21.1	67.3	6.2	22.7
Bug fixing	P_{cd}+ext-overall	85.4	25.2	57.8	62.2	33.9	53.3
	P_{cd}+ext-FA	90.0	24.4	58.5	66.9	30.1	53.8
	P_{cd}+ext-FM	68.2	32.3	55.8	62.0	34.2	53.3
	P_{cd}+ext-PP	84.0	27.2	59.2	63.4	30.2	52.0
	P_{cd}+ext-CE	89.4	24.9	58.9	65.7	30.2	53.2
	P_c	96.3	21.8	57.3	76.7	18.4	47.0
Functional improvement	P_{cd}+ext-oveall	81.7	26.8	58.0	60.1	50.3	57.9
	P_{cd}+ext-FA	82.2	31.5	62.2	61.3	50.2	58.7
	P_{cd}+ext-FM	72.1	32.6	58.0	59.0	50.3	57.0
	P_{cd}+ext-PP	76.6	29.3	57.9	58.9	50.3	56.9
	P_{cd}+ext-CE	80.7	27.8	58.5	60.1	51.5	58.1
	P_c	92.9	23.0	57.7	69.1	26.4	52.2
Code replacement	P_{cd}+ext-oveall	88.3	41.4	72.0	61.7	50.3	59.0
	P_{cd}+ext-FA	84.4	50.9	**74.6**	65.4	50.5	61.8
	P_{cd}+ext-FM	70.3	34.1	58.0	57.8	50.7	56.2
	P_{cd}+ext-PP	70.3	81.1	72.2	60.9	50.2	58.4
	P_{cd}+ext-CE	89.8	39.9	71.9	61.7	50.5	59.1
	P_c	99.2	34.0	71.8	73.9	27.3	55.1

Abbreviation: CHIP, CHange Impact set Prediction.

TABLE 14 Five actual change impact scenarios at method-level: precision (P), recall (R), and F2-score (F2) by P_{cd}+ext-overall, P_{cd}+ext-FA, P_{cd}+ext-FM, P_{cd}+ext-PP, P_{cd}+ext-CE, and P_c

		Best Case			Overall		
Systems	CHIP Variants	R, %	P, %	F2, %	R, %	P, %	F2, %
Move refactoring	P_{cd}+ext-overall	95.9	11.6	39.2	80.8	20.1	50.3
	P_{cd}+ext-FA	92.5	12.5	40.6	78.7	8.8	30.3
	P_{cd}+ext-FM	92.5	11.8	39.2	78.0	20.1	49.5
	P_{cd}+ext-PP	96.6	11.6	39.2	81.4	20.0	50.5
	P_{cd}+ext-CE	96.6	11.7	39.3	78.7	20.0	49.6
	P_c	96.6	11.6	39.2	80.8	9.9	33.2
Remove class or statement	P_{cd}+ext-overall	99.0	12.6	41.7	87.0	10.0	34.3
	P_{cd}+ext-FA	98.1	14.9	46.4	77.2	12.5	37.9
	P_{cd}+ext-FM	99.0	17.0	50.3	82.4	13.5	40.8
	P_{cd}+ext-PP	99.0	12.7	42.0	88.6	10.1	34.7
	P_{cd}+ext-CE	99.0	14.2	45.2	85.5	10.6	35.3
	P_c	98.1	8.3	30.9	77.2	8.0	28.4
Bug fixing	P_{cd}+ext-overall	87.5	10.3	35.1	84.0	10.2	34.3
	P_{cd}+ext-FA	91.3	10.8	36.6	87.0	10.1	34.6
	P_{cd}+ext-FM	87.5	10.3	35.1	84.0	10.4	34.8
	P_{cd}+ext-PP	87.5	10.3	35.0	84.0	10.2	34.4
	P_{cd}+ext-CE	87.5	10.4	35.1	84.0	10.1	34.1
	P_c	88.8	8.3	30.2	85.0	7.9	28.8
Functional improvement	P_{cd}+ext-overall	70.0	11.9	35.4	67.6	10.4	32.1
	P_{cd}+ext-FA	90.0	10.2	35.2	78.4	10.1	33.4
	P_{cd}+ext-FM	70.0	10.0	31.8	67.6	10.0	31.4
	P_{cd}+ext-PP	70.0	10.4	32.7	67.6	10.0	31.5
	P_{cd}+ext-CE	70.0	10.9	33.7	67.6	10.4	32.1
	P_c	90.0	6.9	26.3	78.4	4.7	18.9
Code replacement	P_{cd}+ext-overall	72.2	21.7	49.2	56.3	50.0	54.9
	P_{cd}+ext-FA	73.3	21.5	49.4	57.4	22.7	43.9
	P_{cd}+ext-FM	72.2	22.8	50.4	56.3	23.4	43.9
	P_{cd}+ext-PP	72.2	21.5	49.1	56.5	23.2	43.9
	P_{cd}+ext-CE	74.4	20.8	49.1	56.8	23.1	43.9
	P_c	87.5	17.8	49.0	64.2	19.4	43.9

Abbreviation: CHIP, CHange Impact set Prediction.

the moved code may be used in new features at a new location other than the old location where it was moved from.[53] The data that accessed or manipulated by the moved code are also very likely to be changed and used in those new features, which will propagate changes to other classes which the moved code has data sharing dependency with. Tables 13 and 14 show that under move refactoring scenario, P_{cd}+ext with field modification data sharing dependencies (P_{cd}+ext-FM) improves F2-score the most by 14% in best case compared with P_c. In each kind of scenario, we can find at least one specific kind of data sharing dependencies achieve significantly better F2-scores than P_c. Figures 11 and 12 also show that by setting Dependency Frequency threshold as 0, with an increasing *idf* threshold, P_{cd}+ext built on all 4 types of data sharing dependencies can achieve increasingly better F2-scores than P_c.

Statistical Testing: The null hypothesis for RQ3 is as follows: *There is no difference between F2-score of P_{cd}+ext with selected type of data sharing dependencies only and F2-score of P_c*. Tables 15 and 16 show that in all 5 scenarios, *P* is less than or around .01, which suggests that in specific type of data sharing dependencies, F2-score of P_{cd}+ext is significantly different than F2-score of P_c in those scenarios in both class-level and method-level predictions. Moreover, since the mean of F2-scores of P_{cd}+ext are greater than the mean of F2-scores of P_c, the statistical test results suggest that F2-score of P_{cd}+ext is significantly larger than F2-score of P_c in all those scenarios with the confidence interval (CI) of 95% under the mean of predictions.

Summary: Under each of the 5 change impact scenarios, P_{cd}+ext with data sharing dependencies extracted from specific JVM events can achieve better F2-score than P_c.

8 | THREATS TO VALIDITY

8.1 | Dataset quality and completeness

Like all related works, one limitation of our evaluation is the potential incompleteness of the gold set of change commits for evaluation since the quality and number of commits may influence the accuracy of prediction. Besides, we apply the dynamic analysis to capture call dependencies and data sharing dependencies. Therefore, the quality and completeness of both dependencies may be influenced by the profiling data. But we believe

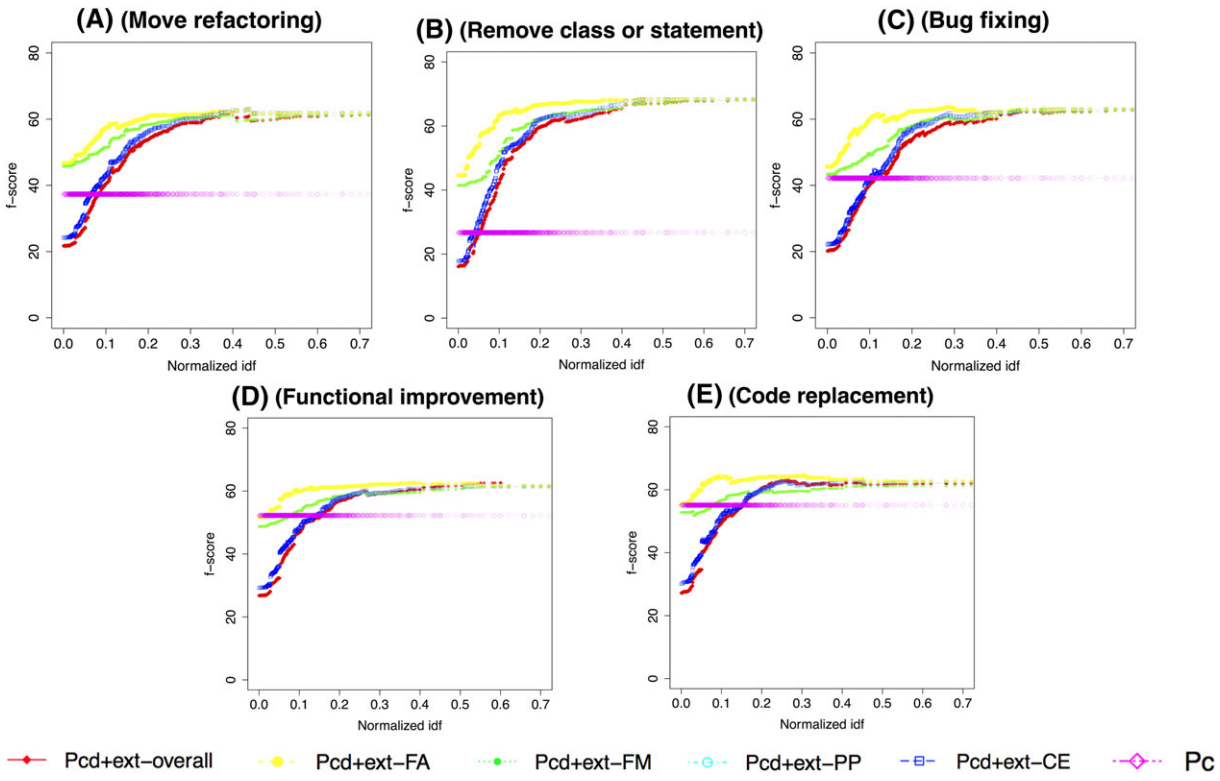


FIGURE 11 A-E. Trend analysis in class-level prediction: F2-score(%) of prediction by different kinds of P_{cd}+ext vs P_c under difference *idf* threshold

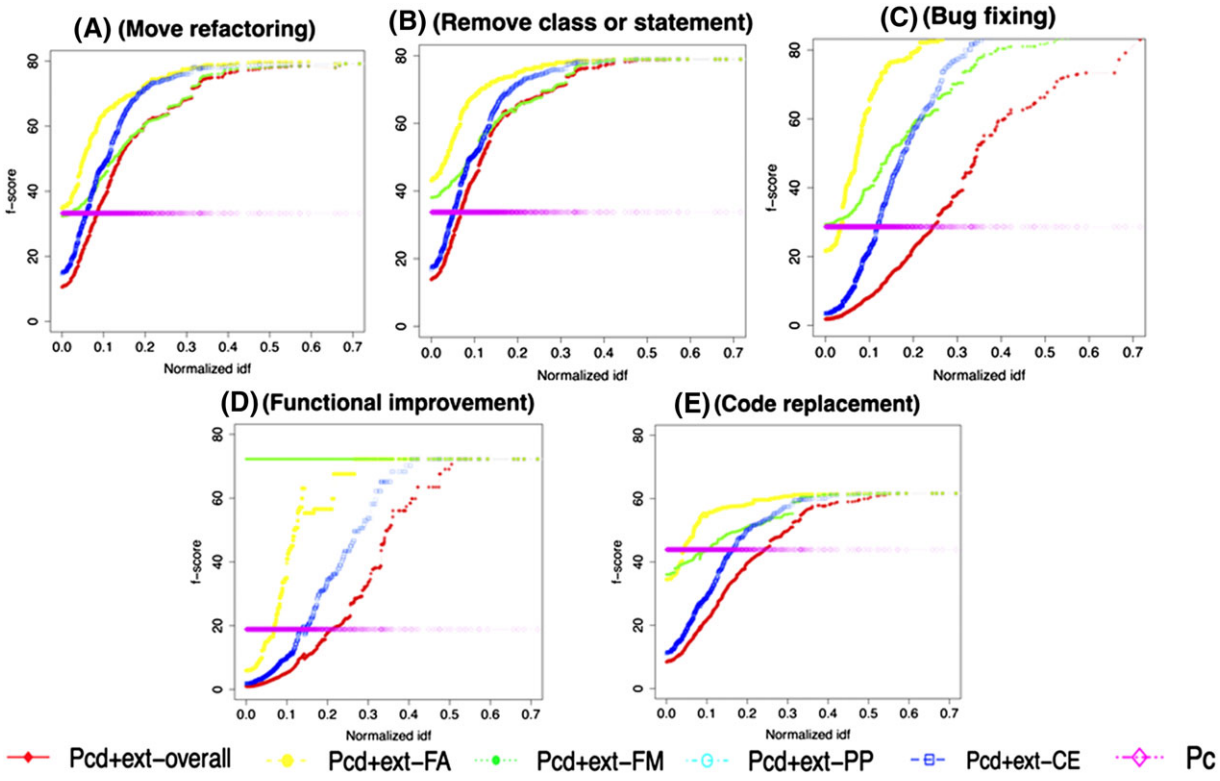


FIGURE 12 A-E. Trend analysis in method-level prediction: F2-score(%) of prediction by different kinds of P_{cd}+ext vs P_c under difference *idf* threshold

TABLE 15 Five actual change impact scenarios at class-level: paired *t*-test results, mean, and confidence interval (CI) of F2-score by P_{cd}+ext vs P_c

	P Value P_{cd}+ext vs P_c	Mean of F2-score and CI	
		P_{cd}+ext, %	P_c, %
Move refactoring	<.0001	58.9 (57.9-59.9)	49.4 (48.2-50.6)
Remove class or statement	<.0001	47.3 (44.0-50.5)	28.6 (26.5-30.7)
Bug fixing	<.0001	55.9 (54.9-57.0)	49.4 (48.2-50.6)
Functional improvement	.000670832	58.7 (58.4-59.0)	57.5 (57.2-57.9)
Code replacement	.00035209	59.4 (59.1-59.7)	59.3 (58.9-59.6)

TABLE 16 Five actual change impact scenarios at method-level: paired *t*-test results, mean, and confidence interval (CI) of F2-score by P_{cd}+ext vs P_c

	P Value P_{cd}+ext vs P_c	Mean of F2-score and CI	
		P_{cd}+ext, %	P_c, %
Move refactoring	.013787341	48.6	38.1
		(41.7-55.5)	(31.3-44.9)
Remove class or statement	<.0001	45.0	25.9
		(35.0-55.0)	(15.9-35.9)
Bug fixing	.000118944	46.8	29.3
		(41.3-52.3)	(26.9-31.7)
Functional improvement	<.0001	43.8	21.4
		(37.6-50.0)	(19.1-23.6)
Code replacement	<.0001	60.8	44.2
		(60.2-61.3)	(43.9-44.5)

completeness of profiling data is not a serious threat since we capture complete execution traces by running through all function per requirements and use case documents of each subject system.

8.2 | Execution traces without unique identifier

As other dynamic analysis research leveraging JVMTI to profile systems, we face the same problem of handling data records, including static field and primitive local variables, which do not have a unique identifier.[45] To solve this problem, we simply use the owner class's identifier.

8.3 | Generalization of experiment results

Our empirical results are based on 4 large open source Java systems. Although these software systems are diversified in application domains, it still requires further empirical evaluation on systems implemented in other programming languages and development paradigms. In the evaluation, we use gold sets of commits from the SVN repositories. Similar to other research adopting the same strategy, we are aware that all classes in the same commits might not be related to one another.

9 | RELATED WORK

9.1 | Mining software repositories based analysis

This class of research uses data mining approaches on historical code change repositories to detect frequent impact set patterns.[1,16-21,26,54-63] For instance, Zimmerman et al,[16] Mondal et al,[54] and Moonen et al[64] all use association rules based mining on CVS logs for detecting evolutionary coupling among source code entities. Ying et al[17] use the similar approach to identify files that frequently change together. Steff and Russo[55] perform CIA by analyzing historical change couplings. CHIANTI and its application[61-63] do impact analysis by analyzing 2 versions of an application and decomposes their difference into a set of atomic change. This kind of approach relies on the quality of software historical repositories. If insufficient historical data are available (such as new project or project with incomplete repository), mining software repository based techniques are

inapplicable. Moreover, there are changes that contradict the frequent impact set patterns. Our approach, however, does not need to keep track of or learn from a long code change history data, and our work does not reply on frequent impact set patterns, either.

9.2 | Textual analysis techniques

Some research work applies information retrieval (IR) techniques on textual data such as comments and/or identifiers in the source code. Gethers et al[65] present a CIA technique on textual change request. Kagdi et al[66] extract conceptual coupling by comparing the similarity in source code using IR-based techniques. Hassan and Holt,[1] Hassan and Gall,[18] and Bavota et al[20] predict change propagation on code textual semantic measures. Some recent works also study change recommendations for bug fixing. Park et al[67] present their approach not only uses commit history data but also uses the data of supplementary patch to investigate the omission errors reduction in change recommendation. Different from our approach, the main purpose of their approach is for bug fixing, and sufficient amount of supplementary patch data is required. But our approach does not rely on any supplementary patch or textual data. And in their work presented in Park et al,[68] researchers also indicate that change recommendation for multi-bug fixing has a higher level of severity and tend to be harder to recommend. Xia and Lo[69] propose an approach called SUPLOCATOR to recommend methods that need to be changed for bug fixing based on relationships among code such as method invocation, containment, inheritance, historical changes, content similarities, and name similarities. Different from our approach, their major purpose is for bug fixing and their approach relies on code inline text such as method invocation and name. Most of those techniques require developers to encode the implicit relations in the comments and/or identifiers and hence the quality of the change prediction depends on the quality of the encoding. When such kind of data are unavailable or in low quality, those approaches are limited. However, CHIP is not affected by such problem because it does not rely on any textual data.

9.3 | Dynamic analysis techniques

Major dynamic impact analysis techniques include Dynamic Slicing, CoverageImpact, and PathImpact. Dynamic Slicing[12-15] analyzes the change impact by extracting slice from an execution trace. CoverageImpact[4,6,31,70] leverages field data to perform CIA. PathImpact or similar approaches, on the other hand, performs impact analysis on whole profiling,[5,7-9,71,72] which, however, is difficult to acquire. Moreover, most of these techniques focus on tracing individual system execution paths and focus on method calls[73] while ignoring data (eg, fields or variables) shared across execution paths. In our work, instead, we extracts a more comprehensive data sharing dependencies by referencing their IDs in memory, unveiling some hidden relations in execution traces.

9.4 | Structural analysis techniques

A number of previous works[3,29-37,39,40,49,74-77] focuses on the analysis of structural dependence, most of which leverage the call dependencies among entities (most notably methods and classes) and Program Dependence Graph (PDG) as indicators for change impact set. Our work belongs to this category. Other studies, such as Hassan et al,[78] perform CIA for software architecture evolution relying on architecture models described by architecture description languages (ADLs). Our work does not reply on any architecture models or ADLs. We have compared CHIP variants based on data sharing dependencies and combined call and data sharing dependencies with existing predictors built on other structural dependencies including call dependencies, PDG, and evolutionary couplings.

10 | CONCLUSIONS AND FUTURE WORK

The paper presents an automated approach and tool (CHIP) for software actual change impact set prediction built upon combined code structural dependencies. It exploits not only call dependencies but also data sharing dependencies, a more comprehensive data dependency by referencing shared data's ID in memory. To improve the precision of the basic predictor relying on data sharing dependencies (P_d and P_{cd}), we introduce 2 novel Dependency Frequency and data type *idf* filter extensions to the basic predictor. We have empirically evaluated CHIP in both method-level and class-level actual change impact set predictions on 4 open source systems. The evaluation results support our hypothesis that data sharing dependencies complement call dependencies in actual change impact set predictions with consistently improved recall and F-score by as much as 19.2% (class-level) and 48.6% (method-level) in average. The results also indicate that after applying data sharing dependencies with both extension, CHIP consistently improves F2-scores of prediction as compared with the ones built on Program Dependence Graph (PDG), which includes dependencies based on data flow and control flow, as well as the ones built on evolutionary couplings in all 4 systems. By adding both extensions to CHIP, the combined call and data sharing dependencies improve F2- score of prediction by as much as 46.2% (class-level), 86.4% (method-level) compared with the predictors built on Program Dependence Graph (PDG), and 41.5% (class-level), 45.1% (method-level) compared with the ones built on evolutionary couplings. Moreover, our empirical experiment results show that specific type of data sharing dependencies are particularly useful on predicting certain change impact scenarios. Future work will further experiment our approach on other software systems and exploring additional types and granularity of dependencies among various levels of code modules for software actual change impact set prediction.

ORCID

Xiaoyu Liu http://orcid.org/0000-0003-3219-1211

REFERENCES

1. Hassan AE, Holt RC. Predicting change propagation in software systems. In: Proceedings of the International Conference on Software Maintenance(ICSM); 2004; Chicago, USA:284-293.
2. Arnold RS, Bohner SA. Impact analysis - towards a framework for comparison. In: Proceedings of the Conference on Software Maintenance (CSM 93); 1993; Montreal, Que., Canada:292-301.
3. Buckner J, Buchta J, Petrenko M, Rajlich V. JRipples: A tool for program comprehension during incremental change. In: Proceedings of the International Workshop on Program Comprehension; 2005; St. Louis, MO, USA:149-152.
4. Orso A, Harrold MJ. Leveraging field data for impact analysis and regression testing. In: Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering; 2003; Helsinki, Finland:128-137.
5. Orso A, Apiwattanapong T, Law J, Rothermel G, Harrold MJ. An empirical comparison of dynamic impact analysis algorithms. In: Proceedings of the International Conference on Software Engineering; 2004; Edinburgh, Scotland, UK:491-500.
6. Breech B, Tegtmeyer M, Pollock L. A comparison of online and dynamic impact analysis algorithms. In: Proceedings of the European Conference on Software Maintenance and Reengineering; 2005; Manchester, UK:143-152.
7. Law J, Rothernel G. Whole program path-based dynamic impact analysis. In: Proceedings of the International Conference on Software Engineering; 2003; Portland, Oregon, USA:308-318.
8. Law J, Rothermel G. Incremental dynamic impact analysis for evolving software systems. In: Proceedings of theInternational Symposium on Software Reliability Engineering; 2003; Denver, CO, USA:430-441.
9. Breech B, Danalis A, Shindo S, Pollock L. Online impact analysis via dynamic compilation technology. In: Proceedings of the International Conference on Software Maintenance; 2004; Chicago Illinois, USA:453-457.
10. Apiwattanapong T, Orso A, Harrold MJ. Efficient and precise dynamic impact analysis using execute-after sequences. In: Proceedings of the International Conference on Software Engineering; 2005; St. Louis, Missouri, USA:432-441.
11. Ramanathan MK, Grama A, Jagannathan S. Sieve: A tool for automatically detecting variations across program versions. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. Tokyo; 2006:241-252.
12. Agrawal H, Horgan JR. Dynamic program slicing. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI '90). New York, NY, USA: ACM; 1990:246-256.
13. Korel B, Laski J. Dynamic program slicing. *Inf Process Lett*. 1988;29(3):155-163.
14. Zhang X, Gupta R, Zhang Y. Precise dynamic slicing algorithms. *Proceedings of International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society; 2003:319-329.
15. Masri W, Nahas N, Podgurski A. Memoized forward computation of dynamic slices. *2006 17th International Symposium on Software Reliability Engineering*. Raleigh, NC; 2006:23-32.
16. Zimmermann T, Zeller A, Weigerber P, Diehl S. Mining version histories to guide software changes. *IEEE Trans Softw Eng*. 2005;31(6):429-445.
17. Ying ATT, Murphy GC, Ng R, Chu Carroll MC. Predicting source code changes by mining change history. *IEEE Trans Softw Eng*. 2004;30(9):574-586.
18. Hassan AE, Gall RC. Replaying development history to assess the effectiveness of change propagation tools. *Empir Softw Eng*. 2006;11(3):335-367.
19. Malik H, Hassan AE. Supporting software evolution using adaptive change propagation heuristics. In: Proceedings of the International Conference on Software Maintenance; 2008; Beijing, China:177-186.
20. Bavota G, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A. An empirical study on the developers' perception of software coupling. *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. Piscataway, NJ, USA: IEEE Press; 2013:692-701.
21. McIntosh S, Adams B, Nagappan M, Hassan AE. Mining Co-Change Information to Understand when Build Changes are Necessary. *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*: IEEE; 2014:241-250.
22. Canfora G, Cerulo L. Impact analysis by mining software and change request repositories. *11th IEEE International Software Metrics Symposium (METRICS'05)*. Como; 2005:20-29.
23. Hill E, Pollock L, Vijay-Shanker K. Exploring the neighborhood with dora to expedite software maintenance. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07). New York, NY, USA: ACM; 2007:14-23.
24. Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T. Using information retrieval based coupling measures for impact analysis. *Empir Softw Eng*. 2009;14(1):5-32.
25. Gethers M, Poshyvanyk D. Using relational topic models to capture coupling among classes in object- oriented software systems. In: 26th IEEE International Conference on Software Maintenance (ICSM'10); 2010; Timioara, Romania:1-10.
26. Kagdi H, Yusuf S, Maletic JI. Mining sequences of changed-files from version histories. *3rd Interna- tional Workshop on Mining Software Repositories (MSR'06) Shanghai, China*. New York: ACM Press; 2006:47-53.
27. Li B, Sun X, Leung H, Zhang S. A survey of code-based change impact analysis techniques. *Soft Testing, Verification and Reliability*. 2013;23(8):613-646.
28. Gwizdala S, Jiang Y, Rajlich V. Jtrackera Tool for Change Propagation in Java; 2003:223-229.
29. Yazdanshenas AR, Moonen L. Fine-grained change impact analysis for component-based product families. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Trento; 2012:119-128.
30. Rungta N, Person S, Branchaud J. A change impact analysis to characterize evolving program behaviors. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Trento; 2012:109-118.
31. Mithun A, Robinson B. Practical change impact analysis based on static program slicing for industrial software systems. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). New York, NY, USA: ACM; 2011:746-755.

32. Weiser M. Program slicing. In: Proceedings of the 5th international conference on Software engineering (ICSE '81). Piscataway, NJ, USA: IEEE Press; 1981:439-449.
33. Arzt S, Bodden E. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). New York, NY, USA: ACM; 2014:288-298.
34. Kim M, Notkin D, Grossman D, Wilson G. Identifying and summarizing systematic code changes via rule inference. *IEEE Trans Softw Eng*. 2013;39(1):45-62.
35. Sun X, Li B, Zhang S, Tao C, Chen X, Wen W. Using lattice of class and method dependence for change impact analysis of object oriented programs. In: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11). New York, NY, USA: ACM; 2011:1439-1444.
36. Huang LL, Song YT. Precise dynamic impact analysis with dependency analysis for object-oriented programs. In: Proceedings of the International Conference on Advanced Software Engineering and Its Applications; 2008; Hainan Island, China:217-220.
37. Geipel MM, Schweitzer F. The link between dependency and cochange: Empirical evidence. *Software Engineering, IEEE Transactions on*. 2012;38(6):1432-1444.
38. Santelices R, Zhang Y, Cai H, Jiang S. DUA-forensics: A fine-grained dependence analysis and instrumentation framework based on Soot. *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis (SOAP '13)*. New York, NY, USA: ACM; 2013:13-18.
39. Cai H, Santelices R. Diver: Precise dynamic impact analysis using dependence-based trace pruning. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). New York, NY, USA: ACM; 2014:343-348.
40. Korpi J, Koskinen J. Supporting impact analysis by program dependence graph based forward slicing. In: Advances and Innovations in Systems, Computing Sciences and Software Engineering; 2007; Springer Netherlands:197-202.
41. Gallagher K, Lyle J. Using program slicing in software maintenance. *Transact Softw Eng*. 1991;17(8):751-762.
42. Yau S, Nicholl R, Tsai J, Liu S. An integrated life- cycle model for software maintenance. *IEEE Trans Softw Eng*. 1988;15(7):58-95.
43. Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. *ACM Trans Program Lang Syst*. 1987;9(3):319-349.
44. Lhoták O. Comparing call graphs. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '07). New York, NY, USA: ACM; 2007:37-42.
45. Kuang H, Mader P, Hu H, et al. Do data dependencies in source code complement call dependencies for understanding requirements traceability? *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Trento; 2012:181-190.
46. Andersen LO. Program analysis and specialization for the C programming language. *Ph.D. Thesis*: DIKU, U. of Copenhagen; 1994.
47. jHotDraw. http://www.jhotdraw.org
48. Canfora G, Cerulo L. Impact analysis by mining software and change request repositories. *11th IEEE International Software Metrics Symposium (METRICS'05)*. Como; 2005:9-29.
49. Cai H, Santelices R. Abstracting Program Dependencies Using the Method Dependence Graph. In: 2015 IEEE International Conference on Software Quality, Reliability and Security; 2015; Vancouver, BC:49-58.
50. Chapin N, Hale JE, Khan KM, Ramil JF, Tan WG. Types of software evolution and software maintenance. *J Softw Maint Evol Res Pract*. 2001;13(1):3-30.
51. Baeza-Yates R, Ribeiro-Neto B. *Modern information retrieval*. New York: ACM press; 1999.
52. Silva LL, Valente MT, de A Maia M, Anquetil N. Developers' perception of co-change patterns: An empirical study. *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*: IEEE; 2015:21-30.
53. Fowler M. *Refactoring: Improving the design of existing code*. India: Pearson Education; 2009.
54. Mondal M, Roy CK, Schneider KA. Improving the detection accuracy of evolutionary coupling by measuring change correspondence. *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Antwerp; 2014:358-362.
55. Steff M, Russo B. Co-evolution of logical couplings and commits for defect estimation. *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*: IEEE Press; 2012:213-216.
56. Xia X, Lo D, McIntosh S, Shihab E, Hassan AE. Cross-project build co-change prediction. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Montreal, QC; 2015:311-320.
57. Servant F, Jones JA. History slicing: Assisting code-evolution tasks. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). New York, NY, USA: ACM; 2012:11. Article 43.
58. Negara S, Codoban M, Dig D, Johnson RE. Mining fine-grained code changes to detect unknown change patterns. In: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). New York, NY, USA: ACM; 2014:803-813.
59. Padhye R, Mani S, Sinha VS. NeedFeed: Taming change notifications by modeling code relevance. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*: ACM; 2014.
60. Jiang Q, Peng X, Wang H, Xing Z, Zhao W. Summarizing Evolutionary Trajectory by Grouping and Aggregating relevant code changes. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Montreal, QC; 2015:361-370.
61. Ren X, Chesley OC, Ryder BG. Identifying failure causes in java pro- grams: An application of change impact analysis. *IEEE Trans Softw Eng*. 2006;32(9):718-732.
62. Ren X, Shah F, Tip F, Ryder BG, Chesley O. Chianti: A tool for change impact analysis of java programs. In: Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications; 2004; Vancouver, BC, Canada:432-448.
63. Stoerzer M, Ryder BG, Ren X, Tip F. Finding failure-inducing changes in java programs using change classification. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2006; Portland, OR, USA:57-68.
64. Moonen L, Di Alesio S, Binkley D, Rolfsnes T. Practical guidelines for change recommendation using association rule mining. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Singapore; 2016:732-743.
65. Gethers M, Dit B, Kagdi H, Poshyvanyk D. Integrated impact analysis for managing software changes. *Software Engineering (ICSE), 2012 34th International Conference on*: IEEE; 2012:430-440.

66. Kagdi H, Gethers M, Poshyvanyk D. Integrating conceptual and logical couplings for change impact analysis in software. *Empir Softw Eng*. 2013;18(5):933-969.

67. Park J, Kim M, Bae DH. An empirical study of supplementary patches in open source projects. *Empir Softw Eng*. 2017;22(1):436-473.

68. Park J, Kim M, Bae DH. An empirical study on reducing omission errors in practice. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). New York, NY, USA: ACM; 2014:121-126.

69. Xia X, Lo D. An effective change recommendation approach for supplementary bug fixes. *Autom Softw Eng*. 2017;24(2):455-498.

70. Breech B, Tegtmeyer M, Pollock L. Integrating influence mechanisms into impact analysis for increased precision. *Proceedings of the International Conference on Software Maintenance*. Philadelphia, Pennsylvania, USA; 2006:55-65.

71. Silva LL, Paixão KR, de Amo S, de A Maia M. On the Use of Execution Trace Alignment for Driving Perfective Changes. *2011 15th European Conference on Software Maintenance and Reengineering*. Oldenburg; 2011:221-230.

72. Santelices R, Harrold MJ, Orso A. Precisely detecting runtime change interactions for evolving software. *Proceedings of International Conference on Software Testing, Verification and Validation*: IEEE; 2010:429-438.

73. Beszedes A, Gergely T, Farago S, Gyimothy T, Fischer F. The dynamic function coupling metric and its use in software evolution. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering; 2007; Amsterdam, Netherlands:103-112.

74. Maia MCO, Bittencourt RA, de Figueiredo JCA, Guerrero DDS. The hybrid technique for object-oriented software change impact analysis. *2010 14th European Conference on Software Maintenance and Reengineering*. Madrid; 2010:252-255.

75. Cai H, Santelices R. A framework for cost-effective dependence-based dynamic impact analysis. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Montreal, QC; 2015:231-240.

76. Hattori L, Guerrero D, Figueiredo J, Brunet J, Damásio J. On the precision and accuracy of impact analysis techniques. *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*. Portland, OR; 2008:513-518.

77. Cai H, Santelices R. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. *J Syst Softw*. 2015;103:248-265.

78. Hassan MO, Deruelle L, Basson H. A knowledge-based system for change impact analysis on software architecture. *2010 Fourth International Conference on Research Challenges in Information Science (RCIS)*. Nice, France; 2010:545-556.

How to cite this article: Liu X, Huang L, Egyed A, Ge J. Do code data sharing dependencies support an early prediction of software actual change impact set? *J Softw Evol Proc*. 2018;e1960. https://doi.org/10.1002/smr.1960