The Journal of Systems and Software 208 (2024) 111912

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Revealing code change propagation channels by evolution history mining ☆

Daihong Zhou, Yijian Wu *, Xin Peng, Jiyue Zhang, Ziliang Li

School of Computer Science, Fudan University, 2005 Songhu Road, Shanghai, China
Shanghai Key Laboratory of Data Science, 2005 Songhu Road, Shanghai, China

ARTICLE INFO

Dataset link: https://github.com/FudanSELab/cpcminer

Keywords:
Code change propagation
Change impact analysis
Evolutionary coupling
Long-term impact
Frequent subgraph mining

ABSTRACT

Changes on source code may propagate to distant code entities through various kinds of relationships, which may form up change propagation channels. It is however difficult for developers to reveal code change propagate channels due to sophisticated interrelationships among code entities. In this work, we propose a novel graph representation for the changed code entities and related code entities changed within a range of space and time so that the types of relationships along which the changes are propagated can be explicitly presented. Then a subgraph mining technique is used to find the frequent change propagation channels.

We finally reveal 40 types of frequent change propagation channels that cover over 98% cases of code change propagation in five well-known open-source Java projects. We find evidence that the code changes propagated through an *unchanged* intermediate code entity consume more time than those through a *changed* one, indicating the difficulties in maintaining code entities that related through *indirect* relationships. We find that a small proportion of code entities frequently appear in the FCPCs, and confirm the semantic relationships between code entities covered by 50 instances of FCPCs, indicating potential usefulness for developers to explain the range of change impact from given source code changes.

1. Introduction

Software is typically implemented by multiple interrelated source code entities. The difficulty of software maintenance typically comes from the complexity of the interrelationships that prevents developers from easily figuring out how to make changes to source code. When developers change some code entities, such as classes, methods, or variables, some other code entities linked directly or indirectly by different relationships could be impacted and should be correspondingly changed. This phenomenon is usually recognized as code change propagation (Hassan and Holt, 2004; Ren et al., 2004). The relationships propagating changes could be of various kinds, such as method calls, inheritance structure, code clones or other implicit ones, and act as a *path* of propagation (Han, 1997). The paths consist of the combinations of the relationship types, linking up multiple code entities. We identify the paths as code change propagation *channels* if they have repeatedly propagated changes in the software maintenance history. A typical change propagation channel is depicted in Fig. 1, where the changes originating from the code entity e_3 frequently cause changes in another related code entity e_2 and further in a third code entity e_1. If such change propagation is commonly found in code entities connected by the same types of relationships, we say these links of specific relationship types are change propagation channels.

Revealing the change propagation channels is important for software maintenance. First, developers may be unaware of the implicit *paths* that propagate changes to a distant code entity as they make changes. Missing change propagation might lead to bugs or other quality issues (Wang et al., 2018b). Identification of the channels could be beneficial for finding potentially missing changes and preventing bugs. Second, due to the complexity in software design, the number of relationships among code entities could be too large for developers to fully comprehend the structure of the software. Unveiling the channels helps developers to put more focus on the relationships that frequently play the *path* role and possibly to identify opportunities to optimize the design of the software.

However, traditional approaches such as change impact analysis (CIA) or co-change analysis do not suffice in revealing the change propagation channels. Traditional CIA approaches typically rely on the snapshot of software and find the code entities related to changed ones by structural dependencies (Oliva and Gerosa, 2015; Ren et al., 2004), code clone relationship (Mondal et al., 2019, 2020a), or some other relationships (Hassan and Holt, 2004). The large number and various

☆ Editor: Alexander Chatzigeorgiou.

* Corresponding author at: School of Computer Science, Fudan University, 2005 Songhu Road, Shanghai, China.
E-mail addresses: dhzhou17@fudan.edu.cn (D. Zhou), wuyijian@fudan.edu.cn (Y. Wu), pengxin@fudan.edu.cn (X. Peng), 20212010070@fudan.edu.cn (J. Zhang), 20210240204@fudan.edu.cn (Z. Li).

https://doi.org/10.1016/j.jss.2023.111912
Received 17 October 2022; Received in revised form 27 July 2023; Accepted 20 November 2023
Available online 22 November 2023

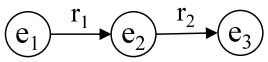

Fig. 1. An example of change propagation channel.

types of the relationships between code entities result in numerous potential paths to propagate changes. Most of the code entities on the paths do not frequently change. The purpose of revealing the change propagation channels is not only to identify the impacted code entities but to find the recurring patterns of how changes propagate through different types of relationships. Therefore, the frequently recurring links of relationship types that propagate changes, or code change propagation channels, could not be revealed by only considering the code entity links at single snapshots of the source code.

Co-change analysis provides a statistic-based way for revealing the patterns of change propagation from history. Two source code entities are said to be *co-change*d if they are changed in the same commit. The basic idea is that the code entities are usually closely related if the code entities often co-changed in the history. Therefore, the changes on one entity are very likely to propagate to the other (Agrawal et al., 1993; Zimmermann et al., 2005). However, co-changes analysis does not cover related changes in different commits. Some changes may have a long-term impact on other code entities, which result in the fact that the related code changes take place at a distant location several days or even weeks later (Brudaru and Zeller, 2008; Herzig, 2010). We call the related changes that spread in a period of time the *long-term changes*. Even if there has been research work that captures related code changes in different commits (Herzig and Zeller, 2011; Jaafar et al., 2014), the different types of relationships that link the code entities are not deeply investigated so that the change propagation is not explained in terms of the relationships. Therefore, the change propagation channels are not revealed.

In this work, we focus on revealing the change propagation channels and finding empirical evidence on how source code changes propagate on these channels in both *co-changes* and *long-term changes*. We consider the fine-grained relationships among code entities, including structure dependencies and code clone relationships. Specifically, the change propagation channel is defined as a path of two code entities linked by two relationships with one or two intermediate code entities, along which changes made to the code entity at one end are frequently propagated to the code entity at the other end. For this purpose, we propose a novel graph representation of the source code changes, which could capture the relevant code changes within limited time and space, while filtering out other irrelevant code changes. We then transform the graph representation into a new form which is independent of the concrete code entities so that the change propagation channels could be mined by existing graph mining techniques.

We try to answer the following five research questions.

- *RQ1: Do the long-term changes widely exist in software maintenance history in contrast to the co-changes?*
- *RQ2: Do change propagation channels exist? If so, which types of relationships between code entities frequently play the role of channels that propagate changes?*
- *RQ3: Are there significant differences in the change time intervals for changed code entities involved in the different types of frequent change propagation channels?*
- *RQ4: Are there significant differences in the change-proneness and bug-proneness of code entities involved in the different types of frequent change propagation channels?*
- *RQ5: Which code entities are frequently involved in the change propagation channels? Are the involved files for these code entities more bug-prone?*

RQ1 helps us to confirm the necessity of considering *long-term changes* as well as *co-changes* in revealing change propagation channels. RQ2 focuses on which types of relationships frequently propagate changes and play the role of change propagation channels. RQ3 and RQ4 enhance our understandings of the change propagation channels from the perspectives of the maintenance effort, in terms of the change time intervals, the change-proneness and the bug-proneness. RQ5 aims at the usefulness of the change propagation channels in identifying the code entities that are more likely to introduce bugs.

We implemented a tool called CPCMiner and conducted an empirical study with five medium-size and actively-maintained open-source Java projects from the Apache community to answer the research questions. The findings are as follows:

(1) Long-term changes widely exist just as co-changes do, implying the necessity of considering long-term changes for change propagation analysis.
(2) Only a small number (i.e. 40) of types of change propagation channels, consisting of three or four code entities, cover most (over 98%) cases of code change propagation, showing the existence of *frequent* change propagation channels (FCPCs). The FCPCs with an *unchanged* intermediate code entity (i.e., Type-0) are more common than those with a *changed* intermediate code entity (i.e., Type-1).
(3) The Type-0 FCPCs typically indicate more time spent on the involved changed code entities with comparison to the Type-1 FCPCs, in terms of significantly longer time intervals between the code changes in the channels.
(4) Code entities involved in the Type-0 FCPCs do not show more change-prone, in terms of the number of commits touching the code entities, than those in the Type-1 FCPCs. Moreover, a small proportion of code entities frequently appear in the FCPCs, and the involved files for these code entities contribute to a project's bug-proneness.
(5) We observe, in a case study on the changes of the code entities involved in the revealed FCPCs, that about 90% of the changes in the different commits covered by the FCPCs are relevant with regard to the semantics of the changes. This result indicates potential help for developers to understand the range and quality risks of the change impacts from given source code changes.

The findings in our study show that change propagation channels are helpful in program comprehension and software maintenance. The core contribution of this paper is (1) the concept of change propagation channel which can be used to represent the relationship types between code entities that frequently propagate changes within a period of time and a range of space. (2) a novel graph representation of the code changes, which can be used to capture the relevant code changes within both co-changes and long-term changes. (3) a graph-based approach to mine the change propagation channels. (4) a small number (i.e. 40) of types of change propagation channels which serve as an empirical evidence on how code changes propagate through these channels in both co-changes and long-term changes, and (5) a dataset that including entity-level software relationship graphs and change information from 5,032 versions of five well-maintained projects.

The rest of the paper is structured as follows. Section 2 describes an example. Section 3 formulates our research problem. Section 4 details our approach. Section 5 reports the answers of the research questions. Section 6 presents in-depth reflections based on the findings. Section 7 discusses the possible applications. Section 8 discusses the threats to validity. Section 9 summarizes related works. Section 10 concludes our work and presents possible research directions.

2. Running example

We present an example that the source code changes propagate in both *co-changes* and *long-term changes*. The maintenance task of this example was to *"add ALPN support for JSSE with Java 9"*.[1] in *Tomcat*

[1] ALPN is the abbreviation for Application Layer Protocol Negotiation, which is a new feature in protocol-based network communication.

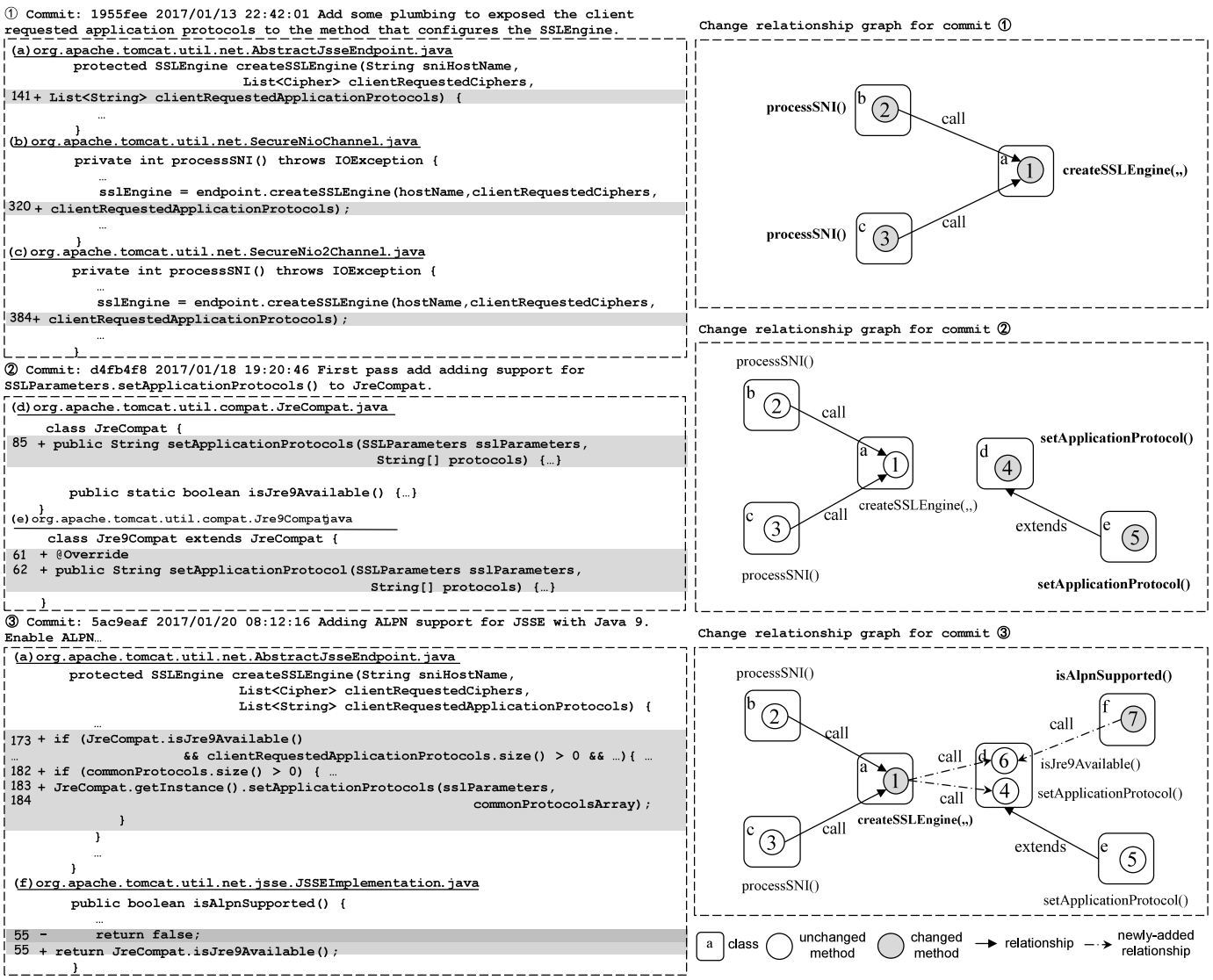

Fig. 2. An example of maintenance task in Tomcat.

Three of the commits serving for this task are depicted in Fig. 2, with a brief introduction of the changes in the source code (the left part) and a graph representing the relationships between the code entities (the right part). Note that there are other commits between these commits, which are not closely related to the given maintenance task.

In the first commit (*1955fee*), the developer added a new parameter `clientRequestedApplicationProtocols` in the signature of the method `createSSLEngine` in the file `AbstractJsseEndpoint.java`, and the two methods that call `createSSLEngine` from other two files were also co-changed. However, the developer did not handle the newly-added parameter in the method `createSSLEngine` in this commit.

In the second commit (*d4fb4f8*) five days later, the developer added a method named `setApplicationProtocols` in the class `JreCompat` and its subclass `Jre9Compat`, respectively. Yet we did not find any relations between the newly-added method `setApplicationProtocols` and the previously changed method `createSSLEngine`.

In the third commit (*5ac9eaf*) one and a half days after the second commit, the developer finally updated the method `createSSLEngine` to handle the parameter `clientRequestedApplicationProtocols` added in the first commit by calling the method `setApplicationProtocols` added in the second commit. Till then, the relationship path of method call was established.

In this example, we find that (1) the source code changes implementing the new feature are related but separated in different commits, and (2) the commits span a period of time in which some commits serving for other purposes exist, and (3) the propagation of code changes could occur in co-changes (e.g., in the first commit, the caller modified the way the method was called when the signature of the callee method was changed), or in long-term changes (e.g., the code changes in the third commit were obviously affected by the previous two commits), and (4) the related source code changes are linked by structural dependencies such as `Extend` and `Call`.

Based on the above observations, it can be concluded that capturing change propagation channels is not an easy task. We need to capture the relevant source code changes across multiple commits while filtering out other irrelevant code changes. Although the commit messages can serve as an important criterion for filtering relevant commits, a large amount of incomplete or ambiguous messages greatly disrupts the accuracy of filtering. In this example, the fine-grained relationships between changed code entities seem to indicate the possibility of capturing the relevant code changes. Therefore, we explore the approach for capturing change propagation channels based on the fine-grained relationships between changed code entities.

3. Change propagation channel

A ***change propagation channel (CPC)*** is a path of two code entities linked by two relationships with one or two intermediate code entities,[2] along which changes made to the code entity at one end are frequently propagated to the code entity at the other end. We use a ***code entity***[3] to represent a class, a method, or a variable (Ren et al., 2004). The relationship types include nine types of structural dependencies, the entity-contain relation, and code clone relation, which will be elaborated in Section 4.1.2.

In this work, we consider the minimal case of a CPC, in which the contained code entities cross *three* files with *two types* of relationships. This is a balanced decision on the generality and computational feasibility. If the CPC crosses only two files, the channel would be too short to effectively reveal the inherent characteristics of cross-file code change propagation. If the CPC crosses four or more files, the potential combinations of code entities and relationships could be vast, resulting in complex propagation paths that may become challenging to reveal. Moreover, we do not specifically emphasize the directionality of change propagation. The directionality of the relationships does not directly indicate the directionality of change propagation. It allows us focus on establishing connections between the changed code entities, making our mining work more feasible and explainable.

A *changed code entity* in the channel is denoted as `CE`; an *unchanged code entity* is denoted as `UE`. The two ends of a CPC are two changed code entities (represented by two `CE` placeholders), and the intermediate code entity is either changed or unchanged (represented by a `CE/UE` placeholder).[4] The types of the linking relationships in the channel are important because we assume that the relationship types could be a potential explanation of the change propagation.

We denote a CPC by the change status (`CE, UE`) of the involving entities as the nodes and the types of the linking relationships (i.e. `Call`) as the edges. For example, the CPC $CE \xrightarrow{\text{CALL}} UE \xleftarrow{\text{CALL}} CE$ means that the changes in two entities (the `CE`s at the two ends) are usually related typically for the reason that both of them call an intermediate entity that usually do not change.

Our aim is to find frequently recurring paths of code change propagation. To do this, we have to model the code changes and the relationships between the changed code entities as a relationship graph. Due to the complexity of the interrelationship among code entities, we limit the range of change impacts by *space*, in terms of k-hop relationships, and *time*, in terms of limited time. Thus, we propose a spatial–temporal window based approach for extracting the changes and the relations between the changed code entities, in the favor of computational feasibility. After that, we formulate the change propagation channels revealing problem as a frequent subgraph mining problem.

4. Approach

The overview of our approach is sketched in Fig. 3. The input is a Git repository containing the development history of a software project, and the output is a set of CPCs frequently recurring in the history. The approach consists of following three main steps:

Step 1: Data Preprocessing. We extract the changed code entities and construct the software relationship graphs (SRGs) for every commit to provide complete code changes and SRGs dataset (Section 4.1).

Step 2: Spatial–Temporal Window Processing. For every commit that satisfies certain criteria (i.e., the core commit), we first establish a spatial–temporal window (ST-Window), i.e., a limited time and space window, and then construct the Spatial–Temporal Change Relationship Graph (ST-CRG) by collecting the changed code entities and corresponding relationships within the ST-window (Section 4.2).

Step 3: Graph Transformation and Mining. After gaining all ST-CRGs, we transform every ST-CRG into a Change Propagation Channel Graph (CPC-G) by reverting the vertices and edges, and then mine frequent CPCs with a frequent subgraph mining algorithm (Section 4.3).

Although our approach is adaptable to other programming languages by applying corresponding language-specific parsers, we aim at software projects written in Java for our tool implementation in this paper.

4.1. Step 1: Data preprocessing

The goal of this step is to extract the changed code entities and construct a SRG for every commit (i.e., the code snapshot resulting from each commit). These individual SRGs collectively form the SRG dataset.

4.1.1. Step 1.1 extracting changed code entities

Code entities could be changed at a commit operation. As with prior work (Ren et al., 2004), we define the type of changes to the code entities, including *Addition*, *Deletion*, and *Modification*. Specifically, modifications to a class refer to changes in the class inheritance hierarchy, such as changing the definition of parent classes or interfaces. Modifications to a member method include changes in the accessibility modifier, method body or method signature. Modifications to a member variable include changes in the accessibility modifier, variable type or the initialization statement.

Although various ways for extracting changes of code entities exist (Xing and Stroulia, 2005; Fluri et al., 2007; Asaduzzaman et al., 2013; Falleri et al., 2014), we employ CLDiff (Huang et al., 2018), an abstract syntax tree (AST) based code differencing tool, to extract changed code entities in commits because CLDiff aggregates the fine-grained changes at the statement-level and is well-balanced in efficiency and precision. Given a commit, CLDiff produces the changes in the versions before and after the commit. The changes are fine-grained at the AST-node level, and then we group them to present the changes in classes, member methods, and member variables.

A list of the changed code entities (ces_i for $commit_i$) is then generated. Meanwhile, we also identify the list of changed files (cfs_i), where at least one code entity is changed. Note that the criteria of a changed file is stricter than that of the version control systems because changes *outside* of class are not considered since they are not relevant to the logic of the code. The changes of blanks, empty lines, and comments in the class are not considered, either.

4.1.2. Step 1.2 constructing software relationship graph dataset

In this work, we consider eleven types of code entity relationships, including nine types of structural dependencies (Pan et al., 2021), the structural containing relationship, and the code clone relationship, for the reason that these relationships are most likely to convey change or bug propagation as reported in previous works (Oliva and Gerosa, 2015; Cui et al., 2019; Mondal et al., 2019). A list of the relationship types is as follows. Note that the list is not exhaustive but could be extended with other types of relationships as required.

- **Ext**end (**EXT**): If class *A* inherits from class *B* via keyword "Extends", then there is a directed edge from class *A* to class *B* to denote their EXT relationship.
- **Impl**ements (**IMPL**): If class *A* realizes interface *B* via keyword "Implements", then there is a directed edge from class *A* to class *B* to denote their IMPL relationship.

[2] Two intermediate code entities mean that they are in a class or file. We formulate this situation with consideration of the possibility that changes in the same class or file may be transitive to change impacts (See Section 4.3).

[3] Interfaces and enumerations are all regarded as classes. Inner classes and anonymous classes are considered as part of their owning classes. The variable represents the member variable or class variable of the class.

[4] The intermediate code entities are denoted as `CE` if any of them are changed.

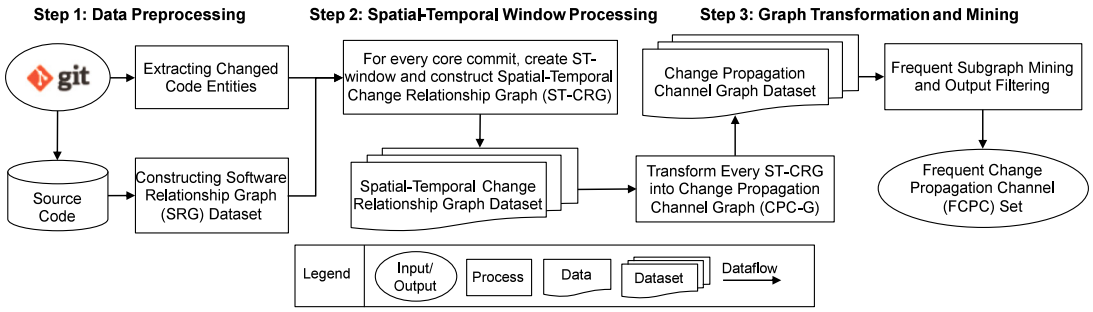

Fig. 3. An Overview of CPCMiner.

- **M**ember **Var**iable (**MVAR**): If class A declares a member variable b with type of class B, then there is a directed edge from the variable b to class B to denote their MVAR relationship.
- **Acc**ess (**ACC**): If one member method $A.m$ or member variable $A.a$ accesses another member variable $A.b$ *or* $B.b$, then there is a directed edge from method $A.m$ or variable $A.a$ to variable $A.b$ *or* $B.b$ to denote their ACC relationship.
- **Call** (**CALL**): If one member method m calls another member method n, then there is a directed edge from method m to method n to denote their CALL relationship.
- **Par**ameter (**PAR**): If one member method m has at least one parameter with type of class B, then there is a directed edge from method m to class B to denote their PAR relationship.
- **Cre**ate (**CRE**): If class B is created in member method m or member variable b, then there is a directed edge from method m or variable b to class B to denote their CRE relationship.
- **L**ocal **Var**iable (**LVAR**): If one member method m declares a local variable with type of class B, then there is a directed edge from method m to class B to denote their LVAR relationship.
- **Ret**urn Type (**RET**): If one member method m has a return type of class B, then there is a directed edge from method m to class B to denote their RET relationship.
- **Con**tain (**CON**): If one method m or variable a is the member of class A, then there is a directed edge from class A to method m or variable a to denote their CON relationship.
- **Clo**ne (**CLO**): If one member method m clones another member method n, then there is a bidirectional edge between method m and n to denote their CLO relationship.

We are now ready to construct the ***software relationship graph (SRG)*** dataset.

Definition 1. SRG is a directed graph: $SRG = (V, E)$ where the vertices (V) are code entities and the edges (E) are relationships between code entities. Multiple edges are allowed if there are more than one relationship between two code entities. Each vertex has a label indicating the type of the code entity (i.e., Class, Method and Variable), while each edge has a label indicating the type of the relationship (11 types, such as EXT, IMPL, CALL, etc.). A SRG dataset is a set of SRGs.

The relationships are extracted from each commit. For structural dependencies and contain relationship, we enhance an open-source static analysis tool, *Depends*[5], to support dependency types such as *EXT*, *ACC* and *MVAR* and to better incorporate the dependency extraction process in our analysis. For code clone relationships, we employ a code clone detection tool named *SAGA* (Li et al., 2020) to extract method-level clone relationships. The similarity threshold is set to 0.7, a typical value adopted in literatures (Li et al., 2020; Sajnani et al., 2016) with balanced precision and recall. Based on the code entities and relationships extracted, an SRG is constructed for each commit. To ensure the quality of the relationships, three authors with rich Java development experiences investigated 500 randomly-selected cases, including 400 cases about structural dependencies and 100 cases about clone relationships. They confirmed over 99% correctness of the relationships.

4.2. Step 2: Spatial-temporal window processing

The goal of this step is to process every commit that satisfies certain criteria (i.e., the *core* commit) one by one, and construct the Spatial–Temporal Change Relationship Graphs (ST-CRGs) by collecting the changed code entities and corresponding relationships within limited time and space (i.e., ST-Window).

Fig. 4 illustrates this construct process. We first determine a time window which covers a number of commits before and after the core commit (Section 4.2.1). For convenience, we call the changed code entities in the core commit ***core entities***. Then, we establish mappings between the code entities in the SRGs of the adjacent commits in the time window so that the core entities are traced in all the commits in the time window (Section 4.2.2). Finally, starting from the core entities, we construct the spatial–temporal change relationship graph (Section 4.2.3).

We recognize a commit as a ***core commit*** which satisfies the following criteria: (1) Not all changed files are test files; (2) At least one code entity is changed; and (3) The number of changed files is no more than a preset threshold N. The first two constraints ensure that the changes in the core commit are the source of change impacts that are of interest. We currently do not consider the change impacts on and from the test code, which is left for future work. The third constraint ensures that the commit under consideration does not touch too many files. A commit that has too many changes are more likely to be a house-keeping commit (Zimmermann et al., 2005; Moonen et al., 2016) and may impose a negative effect on the change propagation analysis. Moreover, the qualified core commits that are committed by the same developer within *one* hour[6] are regarded as a single commit operation because it is very likely that the changes are closely related already.

4.2.1. Step 2.1: Determining the time window

The *time window* is a period of time that covers a number of commits before and after the core commit. The definition of time window is different from the previous works (Herzig and Zeller, 2011; Jaafar et al., 2014, 2011; Feng et al., 2019). This is because we notice that the changes in the core commit could be impacted by the previous changes, while also impacting the later changes. Therefore, we consider the changes before and after the core commit so that we could capture the long-term changes related to the core entities.

The size of the time window (in days or hours) is decided based on both the length of time and the number of commits. Particularly, we set the default size of the time window as the average length of time per

[5] https://github.com/multilang-depends/depends.

[6] Section 5.1 will discuss the reason why we choose such a time of period.

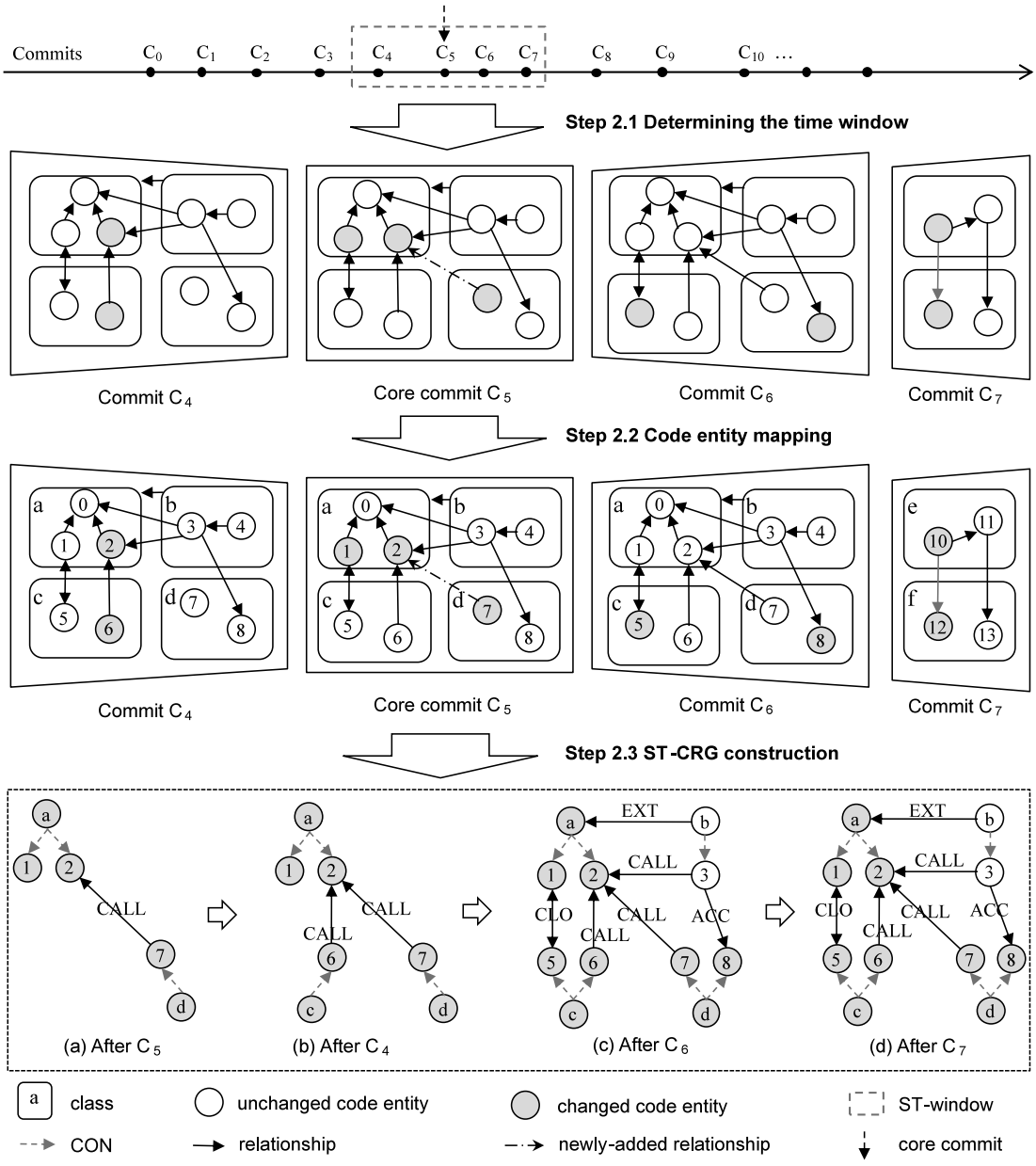

Fig. 4. An example of ST-CRG construction.

15 commits in the whole development history. Once the default size of D days is set, two time periods of $D/2$ days are set before and after the core commit to get the full time window. Meanwhile, we also limit the number of commits up to 15. If there are more than 15 commits covered by the time window, we then drop out the commits from the start and end points of the time window proportionally to the numbers of commits before and after the core commits. For example, if there are 20 commits covered by a time window, among which 11 are before the core commit, 8 are after the core commit, then we drop 3 commits before and 2 commits after the core commit, respectively, such that the time window contains no more than 15 commits. Such settings prevent a huge number of changed code entities in too many commits that may cause infeasible change propagation analysis.

Example. The Step 2.1 in Fig. 4 demonstrates the determination of the time window for core commit C_5. The time window is shown as the dashed rectangle, covering 4 commits (including the core commit). Then we obtain the SRGs of these commits from SRG dataset. In this step, all changed code entities are recognized, as shown in gray circles in Fig. 4. Unchanged code entities are shown in blue. The circular nodes represent code entities. To facilitate reading, we use rounded boxes to represent classes in Step 2.1 and Step 2.2. The arrows represent relationships between code entities. The newly-added relationships are in red. We omit the types of the relationships in the figure only for simplicity.

4.2.2. Step 2.2: Code entity mapping

In this step, we establish the mappings between any two adjacent commits to align the code entities. The mappings build up the evolution trace of the code entities along the commits in the time window. In order to do this, we first use the built-in mechanism of Git to establish the file-level tracing. File moving and renaming are also tracked by Git by default. Then we use CLDiff, an AST-based code differencing tool, to establish the mappings between code entities in two adjacent commits by AST-based matching technology. The classes can be mapped according to the mapped files and classes, and the variables can be mapped by the name along with the class name. The mapping of methods is established by tracking method line number changes in

successive commits. Using the AST-based matching technology could effectively avoid the situation of changing the method signature, such as method renaming, parameters adding or deleting, etc., which greatly improves the accuracy of method tracking. Two authors with rich Java development experiences investigated 200 randomly-selected cases, and confirmed over 95% correctness of the mapping.

Based on the mappings, we assign a globally unique index for each mapped entity to distinguish the entities across different commits in the time window. All code entities in each commit, either changed or unchanged, are all traced by the mappings. Particularly, if a code entity is deleted, the entity in the previous commit is mapped to a null entity; if a code entity is added, the new entity is assigned a new unique index.

Example. The Step 2.2 in Fig. 4 exemplifies the code entities that are traced between adjacent commits, labeled with the same identity number. We omit the classes other than *a-d* in commits C_4, C_5 and C_6 and the classes other than *e-f* in commit C_7 only for the purpose of simplicity. All the code entities in these files are traced in this example.

4.2.3. Step 2.3: ST-CRG construction

Spatial–Temporal Change Relationship Graph (ST-CRG) is a graph representation for the range of potential change propagation from the *core entities* within the time and space window. The range of potential change propagation is defined as the code entities that (1) are changed in the commits in the *time window* and (2) are within k hops along the path of relationships from the core entities in the SRG (*space window*). Specifically, given a core commit and the corresponding time window, the ST-CRG contains the changed code entities in the core commit and the changed code entities, in the time window, within k hops to the core entities. The ST-CRG also contains the code entities that are not changed in the time window if they are on the k-hop path between the core entity. Formally we define ST-CRG as follows.

Definition 2. Given a core commit c and the corresponding time window D, the ST-CRG is a directed graph G_c^w=(V, E), where V represents the code entities and E represents relationships between the entities. Each vertex has a label indicating the type of the code entity (i.e., Class, Method and Variable), and each edge has a label indicating the type of the relationship (11 types, such as EXT, IMPL, CALL, etc.). The vertices include three subsets of code entities: (1) the core entities that are changed in the core commit; (2) the changed non-core entities in the SRG of each non-core commit in the time window w, within k hops to the core entities (mapped from the core commit); and (3) the unchanged entities that are on the shortest paths between vertices in Subset (1) and Subset (2). Multiple edges are allowed if there are more than one relationship between two code entities.

The procedure for ST-CRG construction for a time window w of a core commit c is described in Algorithm 1. We first obtain the core entities in the commit c as the initial vertices of the ST-CRG (line 2). The initial edges are the relationships between the core entities (line 3). Then, for each non-core commit in D, we find the vertices that represent the changed code entities (line 6) and check whether these vertices are within k hops from a core entity in the SRG of the commit (line 9–10). If so, we add the vertices to the ST-CRG for each code entity in the shortest path from the changed entity (inclusive) to the core entity (line 11–12). Otherwise, the changed code entities are discarded.

In this procedure, all changed code entities in w that are within the range of k-hop to the core entities in each SRG and within the time window w are included in the ST-CRG. Therefore, an ST-CRG essentially embody the changes related to the core entities in a given spatial–temporal window.

Example. The lower part of Fig. 4 illustrates the construction procedure of the ST-CRG, starting from the core commit C_5 (Step 2.3(a)) followed by commits C_4 (Step 2.3(b)), C_6 (Step 2.3(c)), and C_7 (Step 2.3(d)). In this example, the range of impact threshold k is set to 2.

Algorithm 1: Generate the ST-CRG from a time window of a core commit

Input: *corecommit*, a core commit
commitlist, the commits list in the time window
SRGs, the SRGs for the commits list
Output: A ST-CRG, *st-crg* = (V, E)

```
1  Initialize a directed graph st-crg = (V, E)
2  Add core entities to V
3  Add relationships between core entities to E
4  foreach commit c in commitlist do
5      Let MappedCoreEntity_c be the set of code entities in c that
         are mapped to the core entities
6      Let ces_c be the set of changed entities in commit c
7      foreach e in ces_c do
8          foreach e_core in MappedCoreEntity_c do
9              if e and e_core are not in the same file then
10                 if exists a k-hop shortest path P between e and
                     e_core in SRG_c then
11                     V = V ∪ all vertices in P
12                     E = E ∪ all edges in P
13                 end
14             end
15         end
16     end
17 end
18 return st-crg
```

First, the changed code entities in Commit C_5 (Step 2.3(a)), or core entities, 1, 2, and 7, are added to the ST-CRG as vertices. Since Class *a* contains methods 1 and 2 and Class *b* contains methods 7, the CON relationships are added as the edges correspondingly. Then, other commits in the time window are processed sequentially.

In Commit C_4 (Step 2.3(b)), the changed code entities are 2 and 6. Since code entity 6 depends on core entity 2, and is located in the two-hop range with core entity 2, code entity 6 is a potential impact entities and added into ST-CRG. Meantime, the CALL and CON relationships are added as edges into graph.

In Commit C_6 (Step 2.3(c)), the changed code entities are 5 and 8. The code entity 5 has a CLO relationship with the core entity 1, and is one-hop away (within the two-hop range). Thus code entity 5 is added to the graph as vertex 5. Code entity 8 is two-hop away from the core entity 2, also in the valid range of impact. Thus, both the code entity 8 is also added as a vertex in the graph. Also note that, since code entity 8 is related to code entities 2 via code entity 3 as the intermediate, the code entity 3 is also added even if it is not changed in Commit C_6. The relationships EXT, CALL, ACC and CON are added correspondingly.

In Commit C_7 (Step 2.3(d)), the changed code entities are 10 and 12. However, both entities are out of the two-hop range from the core entities. Thus, they are not added to the ST-CRG, so the graph is not changed.

After all commits in the time window are processed, the ST-CRG is constructed as in the lower right part of Fig. 4. For each core commit, an ST-CRG is constructed.

4.3. Step 3: Graph transformation and mining

The goal of this step is to find the recurring CPCs, which can be modeled as a frequent subgraph mining problem. In each ST-CRG, a potential CPC starts from a changed code entity and ends with another changed code entity. All relationship types and the intermediate entities in the path are recognized as part of the potential CPC. Specially, we do not consider the unique IDs of each vertex but only keep the *changed* or *unchanged* status of each vertex and the edges that represent the relationship types.

Table 1
The edge direction labels (L_1) of CPC-G.

ID	L_1	Description
1	A	Afferent: $e_1 \xrightarrow{SD} e_0 \xleftarrow{SD} e_2$
2	E	Efferent: $e_1 \xleftarrow{SD} e_0 \xrightarrow{SD} e_2$
3	AE	Afferent and Efferent: $e_1 \xrightarrow{SD} e_0 \xrightarrow{SD} e_2$ or $e_1 \xleftarrow{SD} e_0 \xleftarrow{SD} e_2$
4	AC	Afferent and Clone: $e_1 \xrightarrow{SD} e_0 \xleftrightarrow{CLO} e_2$ or $e_1 \xleftrightarrow{CLO} e_0 \xleftarrow{SD} e_2$
5	EC	Efferent and Clone: $e_1 \xleftarrow{SD} e_0 \xleftrightarrow{CLO} e_2$ or $e_1 \xleftrightarrow{CLO} e_0 \xrightarrow{SD} e_2$
6	CC	Clone relationships: $e_1 \xleftrightarrow{CLO} e_0 \xleftrightarrow{CLO} e_2$

Note: SD = structural dependence CLO = clone relationship.

However, directly mining the frequent subgraphs of the ST-CRG is challenging. First, the ST-CRG is a directed graph, where each vertex and edge have different labels. Due to the presence of bidirectional edges representing clone relationships and the allowance of multiple edges between code entities, it is a challenging task to perform subgraph mining on such directed graphs. Second, we assume the possibility that changes in the same class or file are transitive to change impacts. For example, a method *m* in class *A* possibly has an internal relationship or impact on other code entities in class *A* so that the latter code entities may have an external impact on other code entities in another file. In this case, directly mining the ST-CRG requires more preprocess on the Contain relationships and the code entities that are in the same file.

Based on this observation, we opt not to mine frequent subgraphs in the ST-CRGs directly but to develop a new graph representation, called ***CPC Graph (CPC-G)***, which is an undirected graph transformed from the ST-CRG (Section 4.3.1). Based on the transformed CPC-Gs, we then find the recurring CPCs by frequent subgraph mining (Section 4.3.2). We detail these two steps in the following subsections.

4.3.1. Step 3.1 transforming ST-CRGs into a CPC-G dataset

In this step, we transform each ST-CRG into a CPC-G. Recalling the definition that the CPC is a path of two code entities linked by two relationships with one or two intermediate code entities, in which the contained code entities cross *three* files with *two types* of relationships. In other words, a case of the CPC is a minimal ST-CRG which consists of only three or four code entities and two relationships. Here, assuming that e_0 represents the intermediate code entity,[7] e_1 and e_2 represent the other two ends code entities, respectively. The relationship between code entities e_0 and e_1 is represented as $\langle e_0, e_1 \rangle$, while the relationship between code entities e_0 and e_2 is represented as $\langle e_0, e_2 \rangle$.

Such a minimal ST-CRG is transformed to the minimal CPC-G. Each minimal CPC-G contains only two vertices and one edge, in which the two vertices are transformed from the two relationships $\langle e_0, e_1 \rangle$ and $\langle e_0, e_2 \rangle$ in the ST-CRG, and the label of each vertex represents the relationship type. The edge connecting the two vertices reflects the *change status* of the entity e_0 and the *directions* of the two edges $\langle e_0, e_1 \rangle$ and $\langle e_0, e_2 \rangle$. In order to capture this information, we use a *direction label* (L_1) to represent the directions of the relationships $\langle e_0, e_1 \rangle$ and $\langle e_0, e_2 \rangle$ with respect to the entity e_0. The direction labels are shown in Table 1, where *SD* represents any type of structural dependencies and *CLO* represents the clone relationship in the ST-CRG. We also use a *change state label* (L_2) to represent whether the intermediate code entity e_0 is changed (CE) or not (UE). When the entity e_0 is represented as CE, it indicates that at least one of the intermediate code entity has been changed. Therefore, each edge in the CPC-G contains two labels, namely the direction and the change state. Formally, we define a CPC-G as follows.

Definition 3. A CPC-G, transformed from an ST-CRG, is an undirected graph (V, E), where each vertex v in V is a relationship type with a label of a relationship type (e.g., CALL, ACC) in the ST-CRG, and each edge e in $E \subseteq V \times V$ is undirected and labeled as $L_1@L_2$ where L_1 is one of the direction labels (i.e., A, E, AE, AC, EC, and CC), and L_2 is one of the entity-change-status labels (i.e., CE and UE).

Fig. 5 illustrates six possible cases of minimal CPC-Gs where the intermediate code entity e_0 is changed (CE). There are other six cases where the intermediate code entity e_0 are unchanged, the status of other two code entities is the same as the illustrated six cases, thus we omit them for saving spaces. In each subfigure, the left part is the ST-CRG and the right part is the corresponding CPC-G. In the ST-CRGs, the gray circles represent the changed entities, and the arrows represent relationships between the code entities.

Algorithm 2: Generate the CPC-G from a ST-CRG

```
   Input: a ST-CRG represented by (V^{ST-CRG}, E^{ST-CRG})
   Output: a CPC-G, cpc-g = (V^{CPC-G}, E^{CPC-G})
1  Let edgeindexlist store the edge indexes of E^{ST-CRG}, besides
     CON edges
2  Initial an undirected graph cpc-g = (V^{CPC-G}, E^{CPC-G})
3  foreach unordered pair of edges edge_i and edge_j
     (edge_i, edge_j ∈ E^{ST-CRG}) do
4      Let index_i represent the edge index of edge_i
5      Let index_j represent the edge index of edge_j
6      if edge_i and edge_j share only one common file then
7          Let f_0 represent the common file
8          Let e_01 represent one code entity of edeg_i in f_0
9          Let e_1 represent the other code entity of edeg_i
10         Let e_02 represent one code entity of edeg_j in f_0
11         Let e_2 represent the other code entity of edeg_j
12         if (e_1 and e_2 are all changed)
13         and (not exist direct relation between e_1 and e_2)
14         and (exist core entity in e_01, e_02, e_1 and e_2) then
15             node_i = findVertexByIndex(index_i)
16             node_j = findVertexByIndex(index_j)
17             if node_i is not found then
18                 node_i = a new vertex, indexed by index_i, labeled
                     by the relation type of edge_i
19                 Add node_i to V^{CPC-G}
20             end
21             if node_j is not found then
22                 node_j = a new vertex, indexed by index_j, labeled
                     by the relation type of edge_j
23                 Add node_j to V^{CPC-G}
24             end
25             edl= decideEdgeLabels(edeg_i, edeg_j)
26             Add an undirected edge to E^{CPC-G}, with label edl
                 to connect node_i and node_j
27         end
28     end
29 end
30 return cpc-g
```

The procedure of generating CPC-Gs is formulated as Algorithm 2. In the ST-CRG, we consider any two edges (i.e., relationships) to determine whether they could be transformed into a CPC-G with the following four rules: (1) the two edges share only one common file,[8] that is, there are three files in all related to these two edges (line 6); (2) the two code entities, e_1, e_2, in the two non-common files must be changed (line 12); (3) there are no direct relationships between the

[7] There might be two intermediate code entities involved in change propagation, in which case e_0 represents two code entities.

[8] Though there is no file node in ST-CRG, every code entity belongs to one certain file.

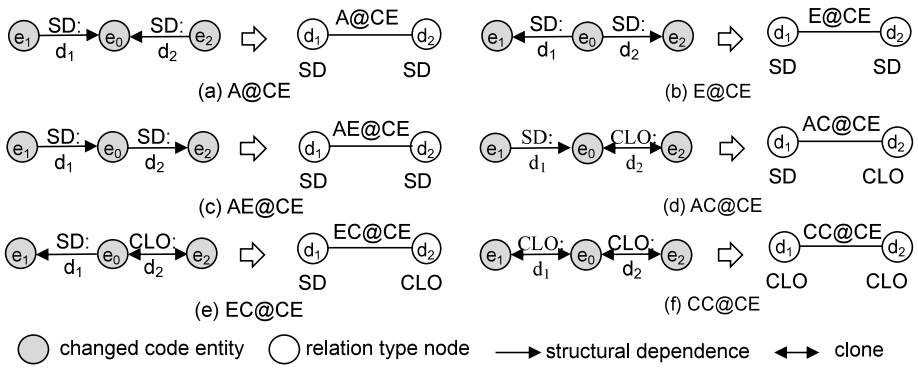

Fig. 5. The examples of transforming ST-CRGs into CPC-Gs.

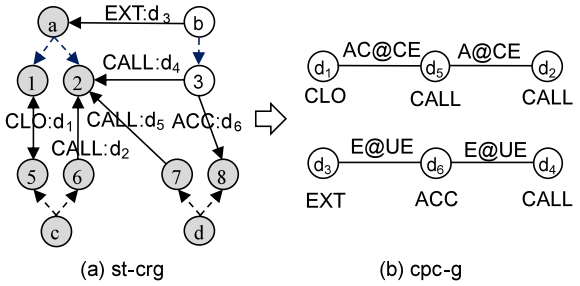

Fig. 6. An example of CPC-G construction.

code entities, e_1, e_2, in the two non-common files (line 13); (4) the two edges cover at least one core entity (line 14); If all rules are satisfied, the two edges are then transformed into two vertices in the CPC-G, and an edge linking the two vertices is created.

Example. Fig. 6 illustrates the transformation. Fig. 6(a) is the ST-CRG; Fig. 6(b) is the transformed CPC-G. In ST-CRG, each edge is labeled by an ID followed by the type of the relationship. In CPC-G, each node corresponds to an edge in ST-CRG. For example, the node d_1 and d_5 in CPC-G comes from the pair of edges $\langle d_1, d_5 \rangle$ in ST-CRG. The transformation is done because the code entities 1 and 2 are in the same file and entities 5 and 7 are not in the same file, and the code entities 5 and 7 are marked as changed in ST-CRG. The edge between d_1 and d_5 in CPC-G is labeled by (AC, CE) because d_1 is a bidirectional relationship (CLO) (the C in AC) and d_5 is a directed relationship that points *inward* (the A in AC) to the entity 2 from the perspective of the file that contains code entity 1 and 2 that are changed (the CE). Then, we traverse all these edges in pairs and finally get the CPC-G as depicted.

4.3.2. Step 3.2 frequent subgraph mining

Based on the CPC-G dataset, we are able to mine frequent subgraphs that represents ***frequent change propagation channels (FCPC)***. In this work, we intend to find the minimal CPCs, meaning that any connected subgraph containing two nodes in the CPC-G is a target subgraph. Therefore, we set the node threshold to 2 in the subgraph isomorphism algorithm VF2 (Cordella et al., 2004) to mine frequent subgraphs.

After mining the common CPC-G subgraphs, we can obtain the corresponding FCPCs. For example, assuming that $d_5 \xrightarrow{\text{A@CE}} d_2$ in Fig. 6(b) is a frequent subgraph, we then find a FCPC: $\texttt{CE} \xrightarrow{\text{CALL}} \texttt{CE} \xleftarrow{\text{CALL}} \texttt{CE}$. This means that a change on the intermediate code entity in this FCPC can have an impact on all code entities that have inward (afferent) CALL relationships. We can also find a FCPC, $\texttt{CE} \xleftarrow{\text{CALL}} \texttt{UE} \xrightarrow{\text{ACC}} \texttt{CE}$, which means the change in one code entity can impact another code entity through an unchanged code entity, where the two changed code entities have relationships with the unchanged code entity through method calls and variable access. These two cases demonstrate that code changes can propagate through both direct relationships (when the intermediate code entity is changed) and indirect relationships (when the intermediate code entity remains unchanged).

5. Empirical study

In order to answer the research questions, we carry out an empirical study on five large-scale open-source projects. In this section, we first introduce the setup of our empirical study (Section 5.1), and then discuss our findings for each research question (Section 5.2).

5.1. Study setup

Subject Project Selection. We choose five well-maintained Java open source projects sponsored by Apache Software Foundation as the subject projects. They are: (1)Tomcat[9], an implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies; (2)Jmeter[10], a Java application designed to measure performance and load test applications; (3)Kafka[11], an open-source distributed event streaming platform; (4)Pinot[12], a real-time distributed OLAP datastore; and (5)Iceberg[13], a core component for high-performance data lake solutions. The basic information of these projects is listed in Table 2.

Each of these projects has a long evolution history and is actively maintained at the time of our study. They have well-maintained issue databases, which provide rich information about the bugs or defects that could be linked to the commits for code quality evaluation.

In our evaluation, we select a period of time (*Study Timespan* in Table 2) when the projects are comparatively stable in the architecture, i.e., no major refactoring happened. This is for the purpose of improving the mapping between code entities in adjacent commits so that the change impact could be precisely traced if the changes repeatedly happen. In this period of time of each project, there were almost the same number of commits (around 1,000) that touched source code files (none-test-files), so that we are able to do cross-project comparison.

Although we select almost the same number of commits, the corresponding development history time varies due to the different density of commits in the projects. The lowest commit density occurs in project *Iceberg*, containing 1,006 source-code-touching commits in 839 days. The highest commit density occurs in project *Tomcat*, containing 1,002 source-code-touching commits in 477 days. The differences in commit density lead to different sizes of the time window for each project.

[9] https://github.com/apache/tomcat.
[10] https://github.com/apache/jmeter.
[11] https://github.com/apache/kafka.
[12] https://github.com/apache/pinot.
[13] https://github.com/apache/iceberg.

Table 2
Subject systems.

System	Full history	# of commits	# of files	KLOC	Study timespan	Study releasespan	Days	# of commits in the timespan	# of commits touching Java files (%)	# of files in the timespan
Tomcat	03/2006-05/2022	72,470	2,560	356	01/2017-05/2018	9.0.0-9.0.8	477	1,449	1,002 (69%)	1,685-1,765
Jmeter	09/1998-05/2022	20,611	1,394	148	03/2017-04/2019	3.1.0-5.2.1	760	2,098	1,012 (48%)	1,029-1,093
Kafka	08/2011-05/2022	15,205	3,437	499	06/2018-02/2020	2.0.0-2.5.0	594	1,957	1,005 (51%)	1,281-1,604
Pinot	01/2002-05/2022	8,945	2,934	336	01/2019-07/2020	0.0.9-0.4.0	534	1,513	1,007 (67%)	2,175-2,359
Iceberg	12/2017-05/2022	2,847	1,807	357	06/2019-05/2022	0.6.0-0.13.2	839	2,148	1,006 (49%)	290-826

Threshold Selection. First, in our study time-span, about 85% of the commits have five or fewer files changed. As we assume that larger commits may consist of unrelated code changes, we limit the number of files touched in a core commit up to N (i.e., 5). This is conservative estimation that the change impacts from this commit are not overwhelmingly scattered. This estimation does not necessarily ensure the atomic of the commit (Shen et al., 2021) but could reduce the negative effect on the change propagation analysis by the housekeeping or tangled commits (Herzig and Zeller, 2013; Zimmermann et al., 2005; Moonen et al., 2016).

Second, we regard the qualified core commits which are committed by the same developer within *one* hour as a single commit operation. We do so for three reasons: (1) A developer might commit an incomplete change or update the remaining files in a separate commit, with the result that the code entities which are highly related via the same change might become spread across several commits. (2) As the developer may commit similar changes in these commits, that merging these commits could avoid the repeated creation of ST-CDGs. (3) Several previous works have proposed strategies to merging commits for co-change analysis, such as grouping commits by the same developer that happen more than once in some period of time. However, there is currently no unified standard regarding the duration of time, with some studies using a few minutes (Mockus et al., 2002; Zimmermann et al., 2005; Silva et al., 2019), while others using several hours or even days (Jaafar et al., 2014; Silva et al., 2015b). Therefore, we select *one* hour as a compromise time threshold. For example, in order to address the issue of an infinite loop[14] in *Tomcat*, a developer committed two distinct code changes[15] within a time interval of less than one hour. These two commits, when merged, serve as a core commit for the creation of ST-CDG. We randomly selected 250 ST-CDGs (i.e. 50 per project in average) involving core commits merging to check manually the commit messages whether the core commits have highly related changes. Two authors confirmed that over 95% of cases refer to same or highly related maintenance task.

Third, in this study, we set the size of the time window based on the commit density of each project and limit the numbers of commits should be no more than 15 for computation feasibility purposes. As different projects have different commit frequencies, we do not choose a fixed time periods or numbers of commits as the time window. Moreover, we set two-hops relationships as the range of change impact (i.e., the space window). This is conservative estimation that the changes in the ST-Window are more likely to be interrelated with the core entities.

Data Preparation. We apply the three steps of our approach in each subject project. In Step 1, we choose a graph database Neo4J for SRG storage. The graph database is designed for efficient graph traversal and operations, which is suitable for SRG storage and processing.

In Step 2, the ST-window settings are shown in the *TW* and *SW* columns in Table 3. The numbers of ST-CRGs constructed for all projects are also shown in the same table.

In Step 3, we transform each ST-CRG into a CPC-G according to our transformation rules, and then mine frequently-occurring common subgraphs in the set of CPC-Gs with the frequent subgraph mining algorithm (i.e., VF2) to obtain FCPCs. Because a change propagation channel requires at least three code entities with two relationships, some ST-CRGs could not be transformed into CPC-G. As listed in Table 3, about 67%–92% of ST-SRGs are transformed for each project, averagely 83% in total.

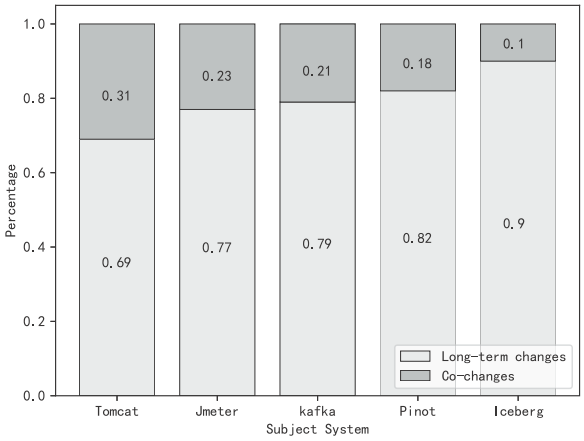

Fig. 7. The distribution of co-changes and long-term changes in ST-CRGs.

Reproduction data. We provide a reproduction package including all SRGs used in our study, the graph representations as well as all mined FCPCs.[16]

5.2. Findings

5.2.1. RQ1: Do the long-term changes widely exist in software maintenance history in contrast to the co-changes?

As the actual proportion of long-term changes in software history is difficult to directly measure, we calculate the percentage of ST-CRGs that can be constructed by considering only co-changes as an indirect assessment of the frequency of long-term changes. In all projects, we constructed 263 to 561 ST-CRGs, of which approximately 10% to 31% can be constructed by considering only co-changes, as shown in Fig. 7. This implies that approximately 69% to 90% of ST-CRGs involve long-term changes.

To verify whether ST-CRGs do involve long-term changes, we organized five developers who had an average of about two years of Java development experience, and randomly selected 100 ST-CRGs involving long-term changes from all projects (i.e., about 20 ST-CRGs for each one). Our investigation was centered on analyzing the correlation between code entity changes within the time window of the ST-CRG, such as whether they complete the same task or have similar or parent–child issue related changes. As an instance, a ST-CRG was constructed, encompassing three commits[17], all of which were directly linked to

[14] https://bz.apache.org/bugzilla/show_bug.cgi?id=60970.

[15] Their commit hashes are '45c2b0f2281bf3fe24d58943d982e8b9dc38518a' and '1516a4fbb13d87130a41cba2e738939cf74c2130'.

[16] https://github.com/FudanSELab/cpcminer.

[17] Their commit hashes are '11aeac2d624e88272e6a932a3b1ebea103ab1a77', '737307506b8f14b2eccde5a44c05a4e0b2f4b1cf', and '1955fee1e3ba6fdec43d3717222226cbe49f19e1'.

Table 3
Experimental settings and resulting graphs.

System	# of commits per day	# of days per 15 commits	TW (days)	SW (k-hop)	# of ST-CRGs	# of files (%)	# of CPC-Gs (%)	# of FCPCs ≥2 CPC-Gs	≥3 CPC-Gs	≥4 CPC-Gs	≥5 CPC-Gs
Tomcat	2.10	7.13	8	2	299	503 (28%)	201 (67%)	133	89	71	59
Jmeter	1.33	11.26	12	2	263	468 (43%)	210 (80%)	79	57	47	43
Kafka	1.69	8.87	8	2	516	904 (56%)	440 (85%)	153	127	103	89
Pinot	1.88	7.99	8	2	515	1,113 (47%)	465 (90%)	156	123	102	91
Iceberg	1.20	12.51	12	2	561	666 (81%)	515 (92%)	172	147	122	104
							Total (∪)	**243**	**203**	**174**	**144**

Note: TW = time window SW = space window.

Table 4
The percentages of CPC-Gs covered by Top-n FCPCs that occur in at least two subject projects.

System	# of CPC-Gs	All FCPCs				Type-0 FCPCs			Type-1 FCPCs		
		Top-10	Top-20	Top-30	Top-40	Top-10	Top-20	Top-30	Top-10	Top-20	Top-30
Tomcat	201 (100%)	78%	88%	94%	95%	78%	82%	85%	36%	44%	45%
Jmeter	210 (100%)	90%	93%	96%	99%	90%	93%	95%	28%	30%	30%
Kafka	440 (100%)	93%	97%	98%	98%	93%	95%	96%	28%	30%	32%
Pinot	465 (100%)	93%	97%	98%	98%	93%	95%	96%	35%	38%	39%
Iceberg	515 (100%)	94%	97%	98%	99%	94%	97%	97%	30%	34%	37%
	Average	**90%**	**94%**	**97%**	**98%**	**90%**	**92%**	**94%**	**31%**	**35%**	**37%**

the task concerning ALPN support. Each ST-CRG was verified by at least two developers independently. If there was disagreement between them, a third developer was brought in for independent verification, and determined the final result with consideration of all results. The results showed that the code entities in most of ST-CRGs (about 90%) were semantically related, and their changes spanned multiple different commits. We notice that the project *Iceberg* has the highest proportion of ST-CRGs involving long-term changes. This is because that this project is in its early stages, which is quite active and unstable, resulting in a higher proportion of long-term changes.

This result indicates the common existence of the long-term changes in the process of software development. Only considering co-changes may miss quite a few related source code changes that have remote impact on other code entities after a period of time. This also demonstrates the necessity to consider long-term changes in change propagation analysis.

> **Answer to RQ1**: *69%-90% of ST-CRGs involve long-term changes, which means when code change propagation occurs, the affected code entities are changed in multiple commits. High proportions of long-term changes imply the necessity to consider long-term changes in change propagation analysis.*

5.2.2. RQ2: Do change propagation channels exist? If so, which types of relationships between code entities frequently play the role of channels that propagate changes?

We obtained 201-515 CPC-Gs transformed from the ST-CRGs constructed from the histories of the subject project. To find frequent change propagation channels (FCPCs), we mined the frequent subgraphs from the CPC-Gs with the *support* thresholds of 2, 3, 4, and 5, respectively. The numbers of FCPCs mined from the CPC-Gs are listed in the last four columns of Table 3. We found 243 distinct FCPCs in all projects when the *support* threshold was set to 2. We also found that, although the numbers of the mined FCPCs decrease as the *support* threshold increases, there were still 144 distinct FCPCs remaining when 5 CPC-Gs should support the FCPC. This means that the FCPCs commonly exist in the projects.

Among the 243 distinct FCPCs that have the minimum *support* of 2 in all projects, we identify 183 FCPCs that occur in *at least two* subject projects. We believe that analyzing these FCPCs contributes to the cross-project generality of our findings.

Table 4 shows the percentages of the coverage on CPC-Gs of Top-n of the most frequently-occurring CPCs (i.e., with the highest supports). We found that Top-10 of the most frequently-occurring CPCs covers over 90% of all CPC-Gs. If we consider Top-40 of the most frequently-occurring CPCs, the coverage rises to over 98%. This implies that most of the changes on a certain code entity impose an impact to other code entities through a few kinds of (inter-entity) relationships.

With this observation, we considered only Top-40 FCPCs in all projects, as listed in Table 5. For each FCPC, we listed the *support value* in all projects as well as in each individual projects. The most common FCPC is $CE \xrightarrow{\text{LVAR}} UE \xleftarrow{\text{PAR}} CE$ with a support value of 1,061 in all five subject projects. This FCPC describes the relationship between two changed code entities in the given time window, which are very likely to be related by a local variable declaration and a parameter in a call to an intermediate code entity which does not change. Take the method `createSSLContext` (e_1) in class `AprEndpoint`, the method `storeChildren` (e_2) in class `SSLHostConfigSF` and the class `SSLHostConfig` (e_0) from *Tomcat* for example. In order to add support for the configuration of OpenSSL, the method e_1 and e_2 changed in three commits[18], in which the changes were related by the PAR and LVAR relationships in a call to the intermediate class e_0.

We observed in this list that, among the top 20 FCPCs, 19 has the `UE` (i.e., unchanged entity) as the middle code entity. This means that the source code changes in one entity impose an *indirect* impact to the other code entity on the other end of the FCPC through an unchanged intermediate. We call such a FCPC a ***Type-0 FCPC***. If the intermediate code entity is the `CE` (i.e., changed entity), we call the FCPC a ***Type-1 FCPC***. We find that *Type-0* FCPCs occur more frequently than *Type-1* FCPCs, as shown in Table 4. Top 10-30 *Type-0* FCPCs cover 78%–97% of CPC-Gs, whereas top 10-30 *Type-1* FCPCs only cover 28%–45% of CPC-Gs. The column *Type* in Table 5 shows this type information.

We investigated the *Type-0* FCPC cases and found that the unchanged code entities typically acted as a bridge to propagate the changes. For example, when two methods being changed in multiple commits both have a CALL relationship to a shared unchanged method, the unchanged method is usually a getter/setter, a utility method, a logging method, or a constructor. In most cases, we find the *Type-0* FCPCs represent *loose coupling* between the changed code entities through the intermediate unchanged code entity.

[18] Their commit hashes are '5b1f5e389ec3238b66d9bd7a724ead63b843cce1', '5777168f58476da9d120b3b186126703e93df224', and '5777168f58476da9d-120b3b186126703e93df224'.

Table 5
The top 40 FCPCs that occur in at least two subject projects.

No.	Type	FCPC	# sum of	Tomcat		Jmeter		Kafka		Pinot		Iceberg	
			support	rank	support	rank	support	rank	support	rank	support	rank	support
0	0	$CE \xrightarrow{\text{LVAR}} UE \xleftarrow{\text{PAR}} CE$	1,061	1	57	0	111	1	236	0	319	0	338
1	0	$CE \xrightarrow{\text{LVAR}} UE \xleftarrow{\text{LVAR}} CE$	948	3	49	1	98	2	230	1	301	2	270
2	0	$CE \xrightarrow{\text{PAR}} UE \xleftarrow{\text{PAR}} CE$	816	4	48	2	85	5	157	3	219	1	307
3	0	$CE \xrightarrow{\text{CALL}} UE \xleftarrow{\text{CALL}} CE$	749	0	79	5	59	7	136	2	246	5	229
4	0	$CE \xrightarrow{\text{CRE}} UE \xleftarrow{\text{CRE}} CE$	621	10	22	8	50	0	247	6	158	9	144
5	0	$CE \xrightarrow{\text{LVAR}} UE \xleftarrow{\text{RET}} CE$	616	17	15	4	61	8	124	5	178	4	238
6	0	$CE \xrightarrow{\text{CRE}} UE \xleftarrow{\text{LVAR}} CE$	613	6	29	3	63	3	221	4	194	14	106
7	0	$CE \xrightarrow{\text{CRE}} UE \xleftarrow{\text{PAR}} CE$	520	12	21	6	59	4	165	8	143	10	132
8	0	$CE \xrightarrow{\text{PAR}} UE \xleftarrow{\text{RET}} CE$	516	23	12	7	56	13	71	9	121	3	256
9	0	$CE \xrightarrow{\text{LVAR}} UE \xleftarrow{\text{MVAR}} CE$	468	18	15	22	13	6	153	12	95	7	192
10	0	$CE \xrightarrow{\text{CALL}} UE \xleftarrow{\text{LVAR}} CE$	412	7	29	9	38	12	82	7	143	11	120
11	0	$CE \xrightarrow{\text{MVAR}} UE \xleftarrow{\text{PAR}} CE$	403	15	17	27	10	9	102	17	70	6	204
12	0	$CE \xrightarrow{\text{CALL}} UE \xleftarrow{\text{PAR}} CE$	361	5	37	12	32	14	69	10	111	13	112
13	1	$CE \xrightarrow{\text{CALL}} CE \xleftarrow{\text{CALL}} CE$	321	2	49	10	34	15	57	13	92	15	89
14	0	$CE \xrightarrow{\text{RET}} UE \xleftarrow{\text{RET}} CE$	309	21	13	15	26	16	56	19	69	8	145
15	0	$CE \xrightarrow{\text{ACC}} UE \xleftarrow{\text{ACC}} CE$	300	8	28	11	33	11	83	14	92	19	64
16	0	$CE \xrightarrow{\text{CRE}} UE \xleftarrow{\text{RET}} CE$	232	50	5	13	31	18	48	15	89	20	59
17	0	$CE \xrightarrow{\text{CALL}} UE \xleftarrow{\text{CRE}} CE$	214	25	12	14	27	17	51	11	100	36	24
18	0	$CE \xrightarrow{\text{CALL}} UE \xleftarrow{\text{RET}} CE$	211	48	6	16	22	33	19	16	77	16	87
19	0	$CE \xrightarrow{\text{MVAR}} UE \xleftarrow{\text{RET}} CE$	198	27	11	32	7	21	33	24	31	12	116
20	0	$CE \xrightarrow{\text{CRE}} UE \xleftarrow{\text{MVAR}} CE$	184	43	6	20	15	10	84	27	24	21	55
21	1	$CE \xleftarrow{\text{CALL}} CE \xrightarrow{\text{CALL}} CE$	177	16	16	19	16	22	31	18	69	22	45
22	1	$CE \xrightarrow{\text{CALL}} CE \xrightarrow{\text{CALL}} CE$	168	13	21	21	15	20	40	20	50	23	42
23	0	$CE \xrightarrow{\text{MVAR}} UE \xleftarrow{\text{MVAR}} CE$	158	19	15	24	11	19	44	36	18	18	70
24	0	$CE \xrightarrow{\text{CALL}} UE \xleftarrow{\text{MVAR}} CE$	142	31	10	37	6	29	24	25	29	17	73
25	1	$CE \xrightarrow{\text{CALL}} CE \xleftarrow{\text{LVAR}} CE$	140	22	13	17	17	26	25	21	46	24	39
26	1	$CE \xrightarrow{\text{CALL}} CE \xleftarrow{\text{CRE}} CE$	109	36	8	28	10	30	24	22	38	31	29
27	1	$CE \xrightarrow{\text{CALL}} CE \xleftarrow{\text{PAR}} CE$	108	30	10	25	11	35	17	23	34	28	36
28	1	$CE \xrightarrow{\text{EXT}} CE \xleftarrow{\text{EXT}} CE$	107	14	17	26	10	24	29	39	17	29	34
29	1	$CE \xrightarrow{\text{CALL}} CE \xleftarrow{\text{EXT}} CE$	102	9	27	23	11	31	22	33	20	38	22
30	1	$CE \xrightarrow{\text{ACC}} CE \xleftarrow{\text{ACC}} CE$	95	29	11	18	17	32	20	29	22	35	25
31	1	$CE \xrightarrow{\text{CALL}} CE \xleftarrow{\text{RET}} CE$	81	52	5	40	5	42	13	26	25	30	33
32	0	$CE \xrightarrow{\text{IMPL}} UE \xleftarrow{\text{LVAR}} CE$	77	66	4	35	6	27	25	97	4	25	38
33	1	$CE \xrightarrow{\text{ACC}} CE \xleftarrow{\text{CALL}} CE$	71	20	15	38	6	36	16	38	17	46	17
34	0	$CE \xrightarrow{\text{IMPL}} UE \xleftarrow{\text{RET}} CE$	67	42	6	47	3	34	17	95	4	26	37
35	1	$CE \xleftarrow{\text{ACC}} CE \xrightarrow{\text{CALL}} CE$	66	49	5	29	10	46	12	30	22	48	17
36	0	$CE \xrightarrow{\text{IMPL}} UE \xleftarrow{\text{PAR}} CE$	66	59	4	59	2	28	24	–	–	27	36
37	0	$CE \xrightarrow{\text{ACC}} UE \xleftarrow{\text{PAR}} CE$	65	44	6	78	2	23	30	42	15	65	12
38	0	$CE \xrightarrow{\text{ACC}} UE \xleftarrow{\text{LVAR}} CE$	65	83	3	45	4	25	29	32	21	80	8
39	1	$CE \xrightarrow{\text{CALL}} CE \xleftarrow{\text{IMPL}} CE$	65	11	21	42	5	58	8	45	12	42	19

Answer to RQ2: *The channels of change propagation commonly exist in our studied open source projects. There are 183 distinct FCPCs mined from the five subject projects, among which the top 40 FCPCs are the most commonly occurring ones, covering over 98% cases of change propagation. FCPCs that propagate changes through an unchanged intermediate code entity, or Type-0 FCPCs, are more common than Type-1 FCPCs with a changed intermediate code entity.*

5.2.3. RQ3: Are there significant differences in the change time intervals for changed code entities involved in the different types of frequent change propagation channels?

The specific code entities instantiated from the UE/CE placeholders in a FCPC, along with the relationships in between, are recognized as an *instance* of the FCPC. We considered the changes on the code entities in the instances of the FCPCs. The changes are either *co-changes* or *long-term changes* in the time window. We used the *change time interval* to represent the time interval between the changes in the code entities in the FCPC. Obviously, the larger change time intervals imply more time the developer will spend to correctly propagate changes.

As each FCPC may have multiple instances in a time window, we first calculated the average change time interval for all instances of the FCPC in the time window, and then obtained the median value on all time windows as the median change time interval of the FCPC. For each instance of the FCPC in a time window, we calculated the average change time interval of all changed code entities in pairs with the following procedure.

Considering two code entities E_1 and E_2 in an instance of the FCPC that was changed in the time window, we collected the time intervals between *the nearest commit pairs* where one commit was on E_1 and the other was on E_2. Fig. 8 illustrates the procedure, which mainly contains four steps: (1) Commits collection. For each code entity in an instance of FCPC, we get its related changed commits. As shown in the figure, E_1 was changed in commit C_1, C_2, C_4, C_5, C_6, and C_8, and E_2 was changed in commit C_3, C_5, and C_7, and both E_1 and E_2 have been changed in the

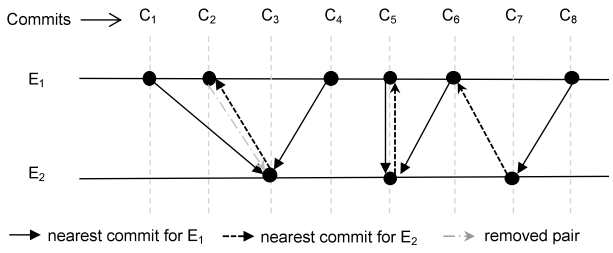

Fig. 8. An example of calculating the change time intervals.

commit of C_5. (2) Nearest commit selection. For each commit of a code entity, we find the nearest commit of other code entity. That is, if two code entities were changed in a same commit, the co-change commit is the nearest commit for each other code entity; if there is no co-change commit, we search forward to get the latest commit of another code entity; if the commit does not exist, then we search backward to get the nearest one. For E_1, the nearest commit pairs are $C_1 \rightarrow C_3$, $C_2 \rightarrow C_3$, $C_4 \rightarrow C_3$, $C_5 \rightarrow C_5$, $C_6 \rightarrow C_5$ and $C_8 \rightarrow C_7$. For E_2, the nearest commit pairs are $C_3 \rightarrow C_2$, $C_5 \rightarrow C_5$ and $C_7 \rightarrow C_6$. (3) Duplicate commit pairs removal. Duplicate commit pairs will be removed besides co-change commit pairs, then $C_2 \rightarrow C_3$ is removed; (4) Time interval calculation. We count the absolute value of the time difference and calculate the mean value, which is the average change time intervals between E_1 and E_2.

The *M*edian change time intervals in *H*ours (**MH**) of the FCPCs as per project are shown in the MH columns in Table 6. For each FCPC, we also calculated a *W*eighted average of *MH* value (**WMH**) for all projects by considering the *support* values of each FCPC listed in Table 5. Each WMH value indicates the overall change time interval for an FCPC. For example, the WMH for the FCPC *No. 30* is 0.8 h ((2.7*11 + 0.3*17 + 1.9*20 + 0.1*22 + 0*25)/(11 + 17 + 20 + 22 + 25), where the numbers 11, 17, 20, 22, and 25 are the *support* values in each project), meaning that the related source code changes covered by FCPC *No. 30* are medially 0.8 h apart in our observed maintenance history.

We ranked the 40 FCPCs by the WMH values in ascending order in Table 6 and noticed that almost all *Type-1* FCPCs rank higher than the *Type-0* FCPCs.

To confirm this observation, we then conducted Mann–Whitney U tests (Mann and Whitney, 1947) on the change time intervals for *Type-0* and *Type-1* FCPCs to see whether they exhibit a significant difference. The Mann–Whitney U test is a non-parametric test, i.e, it does not make the assumption of a normal distribution of each group. Since the data of the MH to be tested do not necessarily follow a specific distribution, it is reasonable to use the Mann–Whitney U test in our study. The magnitude of differences was measured by the Cliff's delta effect size (Cliff, 1993). The effect size (v) was categorized as follows (Romano et al., 2006): if $|v| < 0.147$, the effect size is *negligible*; if $0.147 \leq |v| < 0.330$, the effect size is *small*; and if $0.330 \leq |v| < 0.474$, the effect size is *medium*; and otherwise the effect size is *large*.

Table 7 shows the results. We found that the values of MH for *Type-0* FCPCs were significantly larger than those for *Type-1* FCPCs in all projects ($P < 0.05$). The majority of projects exhibited *large* effect sizes, with the exception of the project *Tomcat*, which demonstrated a *medium* effect size. This means that two related changes in the code entities in *Type-0* FCPCs are separated in a longer period of time than those in *Type-1* FCPCs. In other words, *Type-0* FCPCs show *slower* change impact propagation than *Type-1* FCPCs.

Answer to RQ3: *There are significant differences in the change time intervals for changed code entities involved in different FCPCs. The FCPCs with an unchanged intermediate code entity (Type-0) typically indicate more time spent on the involved changed code entities, in terms of significantly longer time intervals between the code changes in the channels, with comparison to those with a changed intermediate code entity (Type-1).*

5.2.4. RQ4: Are there significant differences in the change-proneness and bug-proneness of code entities involved in the different types of frequent change propagation channels?

To evaluate the change-proneness of each FCPC, we counted the number of commits involved in each instance of this FCPC, and then calculated the *M*edian *C*ommit count (**MC**) of all instances of this FCPC.

To evaluate the bug-proneness of each FCPC, we first found ***b**ug-**f**ixing-related **i**nstances (**BFI**s)* for each FCPC. A BFI of FCPC is an FCPC instance whose time window covers at least one ***b**ug-**f**ixing **c**ommit (**BFC**)*. To detect BFCs, we used the Stanford CoreNLP[19] to normalize the commit messages and examined the processed messages using the heuristic proposed by Mockus and Votta (2000) to identify those commits that occurred for the purpose of fixing bugs. The way we detect the BFCs was also following previous works (Kim et al., 2008; Barbour et al., 2011; Mondal et al., 2019). We also manually checked 100 randomly selecting BFIs and confirmed that all instances were bug-fixing related. Then, we calculated the *B*FIs *P*ercentage (**BP**) for each FCPC to represent the bug-proneness for the FCPC.

Similar to the *WMH*, we also calculated the *W*eighted average of *MC*s and *BP*s for all projects to get the **WMC** and **WBP**. The results are shown in Table 6. In order to find out whether there are significant differences between *Type-0* and *Type-1* FCPCs in change-proneness and bug-proneness, we also apply Mann–Whitney U tests to see if the values of MC/BP significantly differ. The results are shown in Table 8 and Table 9, respectively.

We found no significant difference in the median numbers of commits (MC) between *Type-0* and *Type-1* FCPCs, showing that the change-proneness of different FCPCs was approximately the same. Furthermore, it seems significant differences in the BFIs percentage (BP) between *Type-0* and *Type-1* FCPCs in almost all projects, except for *Kafka*. However, *Type-1* FCPCs are more likely to contain bug-fixing source code changes than *Type-0* FCPCs only in project *Tomcat* and *Jmeter*, while such a conclusion is opposite for the project *Pinot* and *Iceberg*. Therefore, it is necessary to conduct larger scale experiments to analyze the bug-proneness of different FCPCs. Further investigation is still to be carried out in our future study.

Answer to RQ4: *The change-proneness of code entities involved in the FCPCs has no significance difference. But the bug-proneness of different FCPCs needs more experiments to confirm.*

5.2.5. RQ5: Which code entities are frequently involved in the change propagation channels? are the involved files for these code entities more bug-prone?

Recalling that the FCPCs are mined from the CPC-Gs, i.e, each instance of FCPC comes from a CPC-G. We focus on the instances of FCPC that covers the same code entities, which repeatedly propagate code changes through a specific type of relationship. Therefore, we counted the frequency of FCPC instances appearing in different time windows, i.e., the number of CPC-Gs covered. The more frequent the instances of FCPC, the more frequently their corresponding code entities propagate code changes.

Table 10 shows the number of FCPC instances that cover 3, 4, and 5 CPC-Gs (support thresholds). When the support threshold was

[19] https://stanfordnlp.github.io/CoreNLP.

Table 6
The Median change time interval in Hours (MH), Median Commit count (MC), and BFIs Percentage (BP, %) of the FCPCs (RQ3 and RQ4).

No.	Type	FCPC	Tomcat			Jmeter			Kafka			Pinot			Iceberg			WMH	WMC	WBP
			MH	MC	BP	MH	MC	BP	MH	MC	BP	MH	MC	BP	MH	MC	BP			
30	1	$CE \xrightarrow{ACC} CE \xleftarrow{ACC} CE$	2.7	2.0	64	0.3	2.0	100	1.9	1.3	13	0.1	1.3	32	0	1.0	14	**0.8**	1.4	39
13	1	$CE \xrightarrow{CALL} CE \xleftarrow{CALL} CE$	1.4	2.0	46	0.4	2.0	94	1.3	1.0	39	3.9	2.0	45	9.8	2.0	23	4.3	1.8	43
33	1	$CE \xrightarrow{ACC} CE \xleftarrow{CALL} CE$	0.9	2.0	57	0.3	2.5	100	4.5	2.0	38	0	1.0	32	14.7	2.0	9	4.7	1.8	39
35	1	$CE \xleftarrow{ACC} CE \xrightarrow{CALL} CE$	1.7	2.0	50	0.5	2.0	100	12.9	2.0	29	2.3	2.0	36	9.9	1.5	18	5.9	1.9	41
22	1	$CE \xrightarrow{CALL} CE \xrightarrow{CALL} CE$	1.3	3.0	48	7.7	3.0	93	0.4	1.0	50	6.0	2.0	44	17.6	2.0	17	7.1	2.0	44
29	1	$CE \xrightarrow{CALL} CE \xleftarrow{EXT} CE$	1.5	3.0	56	0.4	2.0	91	8.2	2.0	52	3.2	2.0	55	19.8	2.0	18	7.1	2.3	51
39	1	$CE \xrightarrow{CALL} CE \xleftarrow{IMPL} CE$	2.3	3.0	48	3.4	2.0	100	2.6	1.5	13	0	1.0	50	20.1	2.0	29	7.2	2.1	43
26	1	$CE \xrightarrow{CALL} CE \xleftarrow{CRE} CE$	0.1	2.0	19	0.3	2.0	100	5.6	2.0	35	1.7	2.0	36	21.0	2.0	24	7.4	2.0	37
28	1	$CE \xrightarrow{EXT} CE \xleftarrow{EXT} CE$	1.7	3.0	53	0.5	2.8	90	12.2	2.0	29	0	1.0	41	15.1	2.0	18	8.4	2.1	37
15	0	$CE \xrightarrow{ACC} UE \xleftarrow{ACC} CE$	3.4	2.0	34	3.5	2.0	94	3.5	1.5	30	5.5	2.0	39	30.2	2.0	23	9.8	1.9	39
21	1	$CE \xleftarrow{CALL} CE \xrightarrow{CALL} CE$	0.7	2.0	25	4.1	2.0	88	6.7	2.0	47	10.5	2.0	43	25.3	2.0	26	12.1	2.0	42
25	1	$CE \xrightarrow{CALL} CE \xleftarrow{LVAR} CE$	5.1	2.0	31	0.3	2.0	94	11.2	2.0	42	10.5	2.0	33	22.9	2.0	26	12.3	2.0	40
27	1	$CE \xrightarrow{CALL} CE \xleftarrow{PAR} CE$	6.4	3.0	40	0.5	1.5	100	14.0	2.0	12	9.5	2.0	31	22.7	2.0	35	13.4	2.0	37
17	0	$CE \xrightarrow{CALL} UE \xleftarrow{CRE} CE$	0.2	1.0	38	9.6	2.0	76	10.6	2.0	40	17.2	2.0	57	15.6	2.0	19	13.5	1.9	50
31	1	$CE \xrightarrow{CALL} CE \xleftarrow{RET} CE$	3.4	3.0	20	1.1	3.0	100	12.1	2.0	31	9.3	2.0	28	21.8	2.0	35	14.0	2.1	35
37	0	$CE \xrightarrow{ACC} UE \xleftarrow{PAR} CE$	3.3	2.0	33	13	2.3	100	11.5	2.0	40	15.3	1.0	27	32.6	2.0	46	15.5	1.8	39
3	0	$CE \xrightarrow{CALL} UE \xleftarrow{CALL} CE$	4.5	2.0	42	5.8	2.0	90	13.2	2.0	44	17.5	2.0	49	24.7	2.0	30	16.6	2.0	45
14	0	$CE \xrightarrow{RET} UE \xleftarrow{RET} CE$	0.4	2.0	23	8.5	2.0	73	20.5	2.0	38	3.0	2.0	50	25.5	2.0	23	17.1	2.0	36
16	0	$CE \xrightarrow{CRE} UE \xleftarrow{RET} CE$	1.2	2.0	20	7.4	2.0	74	28.6	2.0	42	11.5	2.0	47	27.6	2.0	30	18.4	2.0	45
23	0	$CE \xrightarrow{MVAR} UE \xleftarrow{MVAR} CE$	6.9	2.0	13	0	1.0	82	21.5	2.0	35	2.9	1.5	39	25.9	2.0	19	18.5	1.9	30
10	0	$CE \xrightarrow{CALL} UE \xleftarrow{LVAR} CE$	11.9	2.0	29	11.2	2.0	86	10.8	2.0	48	18.4	2.0	49	33.8	2.0	33	20.2	2.0	46
38	0	$CE \xrightarrow{ACC} UE \xleftarrow{LVAR} CE$	5.9	2.5	0	5.4	2.3	100	31.3	2.0	41	7.9	2.0	40	27.9	2.0	50	20.5	2.0	44
34	0	$CE \xrightarrow{IMPL} UE \xleftarrow{RET} CE$	4.3	2.0	33	4.0	2.0	0	21.8	2.0	18	3.3	1.5	50	27.8	2.0	26	21.6	2.0	25
18	0	$CE \xrightarrow{CALL} UE \xleftarrow{RET} CE$	1.1	2.0	50	9.3	2.0	75	6.9	2.0	47	21.0	2.0	49	31.5	2.0	25	22.3	2.0	42
12	0	$CE \xrightarrow{CALL} UE \xleftarrow{PAR} CE$	6.1	2.0	28	7.3	2.0	86	18.6	2.0	48	18.6	2.0	55	39.9	2.0	33	22.9	2.0	47
36	0	$CE \xrightarrow{IMPL} UE \xleftarrow{PAR} CE$	2.6	2.5	25	17.0	2.0	100	22.8	2.0	35	–	–	–	27.5	2.0	31	24.0	2.0	34
4	0	$CE \xrightarrow{CRE} UE \xleftarrow{CRE} CE$	12.1	2.0	48	12.6	2.0	85	25.3	2.0	48	12.1	2.0	53	41.9	2.0	33	24.3	2.0	49
32	0	$CE \xrightarrow{IMPL} UE \xleftarrow{LVAR} CE$	3.1	2.0	50	43.1	2.5	67	20.9	2.0	54	41.6	2.3	25	29.6	2.0	30	27.1	2.1	41
2	0	$CE \xrightarrow{PAR} UE \xleftarrow{PAR} CE$	3.8	2.0	44	15.4	2.0	91	23.4	2.0	42	18.6	2.0	51	42.1	2.0	38	27.1	2.0	48
6	0	$CE \xrightarrow{CRE} UE \xleftarrow{LVAR} CE$	7.2	2.0	36	16.0	2.0	87	29.7	2.0	47	23.7	2.0	52	42.7	2.0	35	27.6	2.0	50
1	0	$CE \xrightarrow{LVAR} UE \xleftarrow{LVAR} CE$	9.6	2.0	43	14.3	2.0	91	26.0	2.0	54	23.8	2.0	55	41.4	2.0	39	27.6	2.0	53
5	0	$CE \xrightarrow{LVAR} UE \xleftarrow{RET} CE$	0.9	2.0	53	12.9	2.0	84	28.2	2.0	45	24.0	2.0	47	38.5	2.0	35	28.8	2.0	46
7	0	$CE \xrightarrow{CRE} UE \xleftarrow{PAR} CE$	5.6	2.0	21	26.9	2.0	83	26.8	2.0	45	22.2	2.0	53	43.8	2.0	31	29.0	2.0	47
20	0	$CE \xrightarrow{CRE} UE \xleftarrow{MVAR} CE$	7.0	2.0	33	36.5	2.0	67	32.4	2.0	37	27.0	2.0	25	27.5	2.0	42	29.7	2.0	39
24	0	$CE \xrightarrow{CALL} UE \xleftarrow{MVAR} CE$	1.8	2.0	15	19.9	2.0	83	9.4	2.0	35	26.1	2.0	45	43.1	2.0	27	30.0	2.0	34
0	0	$CE \xrightarrow{LVAR} UE \xleftarrow{PAR} CE$	19.9	2.0	40	27.9	2.0	87	24.1	2.0	48	25.8	2.0	54	45.5	2.0	44	31.6	2.0	52
9	0	$CE \xrightarrow{LVAR} UE \xleftarrow{MVAR} CE$	7.8	2.0	27	36.2	2.0	77	29.5	2.0	44	25.4	2.0	46	42.4	2.0	31	33.4	2.0	39
11	0	$CE \xrightarrow{MVAR} UE \xleftarrow{PAR} CE$	0.4	2.0	12	3.8	2.0	90	22.7	2.0	40	29.0	2.0	46	45.6	2.0	36	33.9	2.0	39
19	0	$CE \xrightarrow{MVAR} UE \xleftarrow{RET} CE$	6.4	2.0	9	26.5	2.0	43	44.8	2.0	26	25.3	2.0	31	37.1	2.0	24	34.5	2.0	25
8	0	$CE \xrightarrow{PAR} UE \xleftarrow{RET} CE$	1.5	2.0	13	25.7	2.0	84	22.8	2.0	37	21.6	2.0	48	47.7	2.0	35	**34.7**	2.0	43

Table 7
Mann–Whitney U test (P < 0.05) of MH between *Type 0* and *Type 1*.

	Tomcat	Jmeter	Kafka	Pinot	Iceberg
Mean (*Type0/Type1*)	5.1/2.2	15.5/1.5	21.8/7.2	18.8/4.4	35.0/18.4
P-value	3.1E−02	1.1E−05	5.2E−05	3.2E−05	1.6E−06
Cliff's delta	0.43 (medium)	0.87 (large)	0.8 (large)	0.83 (large)	0.95 (large)

Table 8
Mann–Whitney U test (P < 0.05) of MC between *Type 0* and *Type 1*.

	Tomcat	Jmeter	Kafka	Pinot	Iceberg
Mean (*Type0/Type1*)	2.0/2.5	2.0/2.2	2.0/1.8	1.9/1.7	2.8/1.9
P-value	2.8E−03	0.2	1.5E−02	0.9	0.1
Cliff's delta	0.44 (medium)	0.19 (small)	0.28 (small)	0.24 (small)	0.19 (small)

Table 9
Mann–Whitney U test (P < 0.05) of BP between *Type 0* and *Type 1*.

	Tomcat	Jmeter	Kafka	Pinot	Iceberg
Mean (*Type0/Type1*)	30%/43%	80%/96%	41%/33%	45%/33%	32%/22%
P-value	2.9E−02	1.0E−04	0.1	3.1E−02	1.5E−03
Cliff's delta	0.46 (medium)	0.76 (large)	0.36 (medium)	0.43 (medium)	0.63 (large)

Table 10
The resulting instances of FCPCs (on the same code entities).

System	# instances of FCPCs			Relevance verification	
	≥3 CPC-Gs	≥4 CPC-Gs	≥5 CPC-Gs	Commits	Accuracy
Tomcat	374	158	3	47	0.9
Jmeter	120	22	12	49	0.8
Kafka	555	134	41	70	1.0
Pinot	1,247	464	191	54	0.8
Iceberg	3,356	647	63	86	1.0
Total	**5,652**	**1,425**	**310**	**306**	**0.9***

Note: * = average value.

set to 3 CPC-Gs, we found a total of 5,652 FCPC instances in all projects. Although the number of FCPC instances decreases as the support threshold increases, there are still 310 FCPC instances when the support threshold is set to 5 CPC-Gs. This implies that in these projects, there are many code entities are frequently involved in the change propagation channels (i.e., FCPC). These code entities tend to be change-prone and are likely to propagate changes to others.

To evaluate the accuracy of the mined instances of FCPC on the same code entities, we randomly investigated 50 instances (i.e., about 10 from each project), including instances of *Type-0* and *Type-1*, and manually analyzed the changes in the code entities in multiple commits. We found that the code changes in the FCPCs were most likely related in terms of functionality or semantics, such as completing the same task or having similar or parent–child issue related changes. Two of the authors individually determined the relevance of the changes in the FCPCs and tagged the instance *positive* if the changes were related (i.e., completing the same task or having similar or parent–child issue related changes) or *false* otherwise. If there was disagreement between them, a third author was brought in for independent verification, and determined the final result with consideration of all results. We found that, among the 50 instances of FCPCs covering 306 commits, 45 (90%) were *positives*. Details are shown in the last two columns of Table 10.

Reasons behind the semantic relevance. We further summarized the semantic relevance of changes in multiple commits for *Type-0* and *Type-1* categories. For each FCPC instance, we use e_0 to represent the intermediate code entity, and use e_1 and e_2 to represent the other two changed entities. Our purpose is to confirm whether the changes between e_1 and e_2 are semantically related, and the intermediate entity e_0 is related to the changes through the mined relationships. We discussed the scenarios in the following.

- ***Type-0***: There are 42 instances in this category. The two code entities, e_1 and e_2, have multiple relationships with the intermediate e_0, such as LVAR, PAR, CALL, RET, and ACC. We found that e_1 and e_2 were changed together in four typical scenarios. (1) In about 54% of studied instances, the changes in e_1 and e_2 are indirectly related to e_0 via the mined relationships, such as the changes in e_1 or e_2 have control or data dependency with the entity e_0. (2) In about 19% of the instances, e_1 and e_2 commonly share the same or similar code changes, typically in a code clone snippet, such as adding similar statements or updating the way to invoke the same method. (3) In about 17% of the instances, the changes in e_1 and e_2 are directly related to e_0 via the mined relationships. For example, both e_1 and e_2 experienced consistent changes to use (i.e., CALL, ACC, PAR) the entity e_0. (4) In the other 10% of the instances, the changes between e_1 and e_2 are irrelevant. This scenario mostly occurs when e_1 and e_2 are constructor or long methods, which lacks of coupling inside the method.
- ***Type-1***: There are 8 instances in this category. We found three typical scenarios in source code changes of *Type-1* FCPCs. (1) In about 70% of interested instances, both e_1 and e_2 depend on e_0, and the changes in e_0 result in changes to e_1 and e_2. For example, a caller method should be changed if the parameters of the callee method are changed. (2) In about 15% of the instances, e_1 depends on e_0 and e_0 depends on e_2, so the changes in e_2 result in cascade changes on e_0 and e_1. (3) In the other 15% of the instances, e_0 depends on e_1 and e_2 so when e_1 and e_2 are changed, e_0 also need to be changed.

The bug-proneness of the involved files. We first calculated the percentages of involved files for the top-*N* instances of FCPC on the same code entities, and then calculated the percentages of *bug-fixing commits (BFCs)* touching these involved files. The results are shown in Table 11. We find that although the top-*N* instances involved very few files, these files consume a significant percentage of the bug-proneness of the project. For example, top-200 instances of FCPC, which only contain 3%–11% of files, cover 38%–52% of BFCs. This probably indicates that the developers only need to focus on the top a few FCPC instances instead of reviewing all the identified in software maintenance work.

> **Answer to RQ5**: We find that a small proportion of code entities do appear frequently in the FCPCs. The involved files for these code entities contribute to a project's bug-proneness.

6. Discussion

In this section, we present in-depth reflections based on the results reported in the previous section.

6.1. The necessity of considering long-term changes

Developers change source code for various reasons, such as functionality enhancement, new features, or bug fixes. However, one of the difficulties in software maintenance is that source code entities have various relationships, including but not limited to static dependencies, to each other. Due to the complexity of various kinds of relationships, it is likely that developers miss out some changes when responding to a change request. Long-term changes typically come from the fact that multiple changes serving for the same purpose may scatter in both time and space. Herzig (2010), Herzig and Zeller (2011) tried to capture long-term changes, and had proven that the long-term impact of changes were important for software quality, maintainability, and development effort. They grouped changes into change genealogies to predict the long-term cause effect chains at method level. However, due to the complexity of long-term changes, and the difficulty of capturing,

Table 11
The percentages of involved files and BFCs covered by top-n instances of FCPCs that occur in the same code entities.

System	# of all files	# of all BFCs	Top-10		Top-100		Top-200	
			#F	#BFC	#F	#BFC	#F	#BFC
Tomcat	1,765(100%)	310(100%)	1%	11%	1%	11%	2%	22%
Jmeter	1,093(100%)	509(100%)	1%	13%	5%	32%	6%	36%
Kafka	1,604(100%)	307(100%)	1%	16%	3%	31%	5%	45%
Pinot	2,359(100%)	281(100%)	1%	11%	2%	14%	3%	41%
Iceberg	826(100%)	241(100%)	1%	12%	1%	27%	8%	39%

Note: #F = the percent of involved files, #BFC = the percent of bug-fixing commits.

few works conduct change propagation analysis considering long-term changes.

In this work, we used a spatial–temporal window (ST-Window) to capture changes and also found them commonly existing in the propagation of impacts. As far as we know, this is the first work that conduct change propagation analysis considering long-term changes in code entity level. We believe that it is necessary to use long-term changes for change propagation analysis, and our proposed method can be a great way for such analysis.

6.2. Change patterns vs change propagation channels

Recurring code changes are usually observed as *change patterns*, which could be mined by various approaches (Silva et al., 2019; Jiang et al., 2021; Janke and Mäder, 2022; Nguyen et al., 2019; Feng et al., 2019). In this paper, we focus on *change propagation channels* instead of *change patterns* because we want to uncover the relationships between changed code entities and to explain the interrelated changes. We argue that these relationships contribute to the propagation of changes.

The concept of change propagation channel is a generalized presentation of the concept of change patterns. In our work, some change propagation channels are similar to some change patterns mined in other research work. For example, as shown in Table 4, the FCPCs *No. 13, 21* and *22* are typical change (propagation) patterns extracted by Feng et al. (2019), who named the three patterns *Dissemination*, *Concentration*, and *Domino*. However, they did not further distinguish the dependencies that enable the change propagation.

Another example comes with the FCPCs *No. 30, 33* and *13*. For *No. 30*, the source code changes mainly impose impacts on other code entities through the Member Access (ACC) relationship, representing that, if a member variable is changed, the other code entities accessing the member variable are likely to be modified correspondingly. These three FCPCs (*No. 30, 33 and 13*) are also similar to the change patterns found by Wang et al. (2018b), Jiang et al. (2020, 2021). However, they did not further investigate the time intervals between the changes, whereas our empirical study reveals that these relationships propagate changes in relatively short time intervals, indicating stronger coupling between the related code entities.

Therefore, analysis with change propagation channels provides more insightful results than traditional change pattern analysis.

6.3. Structural dependencies and clone relationships in change propagation channels

Structural dependencies and clone relationships have been proven to cause change propagation (Oliva and Gerosa, 2015; Cui et al., 2019; Mondal et al., 2019, 2020a). As discussed in RQ2, we find common combinations of structural dependencies in the top 40 FCPCs.

However, there is no clone relationship involved in the top 40 FCPCs. We find that the top-2 rankings of FCPCs involving clone relationships are *No. 99* and *107*. The *No. 99* and *107* FCPCs are $CE \xleftarrow{CALL} CE \xleftrightarrow{CLO} CE$ and $CE \xrightarrow{CALL} CE \xleftrightarrow{CLO} CE$, respectively. The reason that they ranked low is partially because we only consider clones at method-level and thus the number of clones are not very high. However, in our case study in RQ5, we find with our manual investigation that segment-level clones are common in FCPCs. In our future work, we will further employ segment-level clone detection in our approach.

6.4. Longer change propagation channels

In our approach, we defined the *minimal case* of a change propagation channel (Section 4.3.1), which contains code entities only in *three* files with *two types* of relationships. However, longer change propagation channels may exist in practice. We argue that the minimal cases of the change propagation channels could be linked by their shared code entities so as to form up longer change propagation channels.

Recall the example in Section 2. We use the index e_i to represent the code entity for short, i.e. the method `createSSLEngine` is e_1. The mined cases of $e_2 \xrightarrow{CALL} e_1 \xleftarrow{CALL} e_3$, $e_1 \xrightarrow{CALL} e_6 \xleftarrow{CALL} e_7$ and $e_1 \xrightarrow{CALL} e_d \xleftarrow{EXT} e_e$ could be linked by the common code entity e_1. The method `createSSLEngine` e_1 is a core method in this maintenance task, in which a new parameter `clientRequestedApplicationProtocols` is added in the first commit, and the changes in the second and third commits are made to handle the newly-added parameter. The linked channels form up a longer change propagation channel which could help developer better understand the path of change propagation and complete the next maintenance task.

There are various ways to link the minimal cases of the change propagation channels. For example, two distinct cases of change propagation channels that share the intermediate code entity, or share the code entity at both ends of the channels. We count the most common linked types with these two ways and find the top-three most frequently recurring linked channels are: $CE \xrightarrow{\text{LVAR}} UE \xleftarrow{\text{PAR}} CE$, $CE \xrightarrow{\text{LVAR}} UE \xleftarrow{\text{LVAR}} CE$, and $CE \xrightarrow{\text{PAR}} UE \xleftarrow{\text{PAR}} CE$. Further research on more linked types of the change propagation channel and the longer change propagation channels are scheduled as our future work. We believe that the minimal change propagation channels provide a starting point for developers to understand the longer cases of change propagation channels.

6.5. Tangled changes

During interactions with version control systems, developers may commit unrelated or loosely related code changes in a single commit, resulting in a tangled change (Herzig and Zeller, 2013). Consequently, when examining the code change history, such tangled changes can falsely establish connections among all modules, thereby potentially compromising resulting analyses through noise and bias (Herzig and Zeller, 2013). To address this challenge, we have endeavored to mitigate the impact of tangled changes through the following two ways.

First, we limit the number of files touched in a core commit up to N (i.e., 5), which is discussed in Section 5.1. A commit that has too many changes are more likely to be a house-keeping commit and may impose a negative effect on the change propagation analysis.

Second, we create a time window for the core commit and construct an ST-CRG representing changes related to core entities within the given spatial–temporal window (Section 4.2.3). Only changed code entities within the k-hop range are considered, removing irrelevant

changes. The relevance defined in this paper is based on dependency relationships. Although this is an instinctive decision, we find it a feasible solution to capture the related code changes for the core entities.

We are also trying to explicitly isolate the tangled changes with techniques like history slicing, which is ongoing work and not included in this paper.

6.6. Limitations

First, the programming language supported by our approach is currently limited to Java. Although CLDIFF and Depends support many other programming languages, we have not yet integrated other language support due to engineering limit. We are planning to support more languages in our further work.

Second, it is worth noting that our current AST-based matching technology may not provide adequate support for frequent branch merging and refactoring. Nevertheless, alternative code entity mapping technologies may be employed in our approach. Additionally, we intend to improve the AST-based matching technology in future work.

Third, our approach is mostly applicable to uncover long-term changes where the link between issues and commits is obscure. Since changes belonging to different issues may also impact each other, such as in similar or parent–child issues, our work offers a flexible methodology to capture the long-term changes for these issues. Moreover, due to the complexity dependency analysis, we currently limit the scope of long-term changes with a range of space and time. This is a way to filter out some loosely-related changes serving for different development activities in a composite commit (Herzig and Zeller, 2013). Of course, there may exist more ways to capture the long-term changes, such as transaction dependency graphs (TDGs) (Herzig, 2010). Utilizing other ways to capture such impacts of changes is scheduled in our further work.

Fourth, only structural dependencies and code clone relationships are considered for change propagation analysis in this work. However, many more relationships among code entities may also contribute to propagation of changes, such as semantic couplings (Ajienka et al., 2018), developer interactions (Ashraf et al., 2019), or change genealogies (Herzig and Zeller, 2011). Considering more relationship types is part of our future work.

Last, we only extract clone relationship at method level. However, we also find that segment-level clones are also suitable for representing similar code changes among code entities, which is also observed by Mondal et al. (2019, 2020a). We plan to consider finer-grained clones for change propagation analysis in the future.

7. Potential applications

Based on the tool CPCMiner, we collected and publicly released software relationship graphs (SRGs) and code entity change information for a total of five high-quality open-source projects, consisting of 5,032 code snapshot versions. Furthermore, we proposed a novel graph representation (i.e., ST-CDG) to capture long-term changes based on spatial–temporal windows, and mined 40 most frequent change propagation channels (FCPCs) using graph mining techniques. In the following, we will introduce the potential applications of our work from four different perspectives.

Researchers. First, considering the recurring nature of code co-changes, researchers have been trying various approaches to utilize co-changes in order to complete specific tasks (Oliva and Gerosa, 2015; Wang et al., 2018b; Feng et al., 2019; Silva et al., 2019; Jiang et al., 2020, 2021), such as bug fixes (Wang et al., 2018b), code change recommendation (Jiang et al., 2020, 2021), software evolution analysis (Feng et al., 2019), etc. They have demonstrated the effectiveness of recurring co-change patterns in these specific tasks. In this study, we explored the time intervals between changes and confirmed that these co-change patterns are characterized by short time intervals. We find that co-changed code entities usually share certain commonality, e.g., depending on common code entities with different relationships or common relationships. Therefore, our findings validate the observations made in prior researches.

Second, our tool CPCMiner and SRGs dataset can also enhance existing works. If one utilizes our SRGs dataset, which contain various types of software relationships, this could increase the work's effectiveness. For example, Wang et al. (2018b) identified six recurring change patterns from the change dependency graphs of different bug fixes. However, they only considered three types of relationships between the code entities, i.e, Containment, Overriding and Access. By leveraging our dataset encompassing a diverse array of software relationship types, they could identify a broader spectrum of recurring change patterns. This facilitates a more comprehensive understanding of multi-entity changes in real bug fixes, such as interface implementation-related errors, inconsistent fixes in clone code, and other similar instances. Some other works (Hassan and Holt, 2004; Oliva and Gerosa, 2015; Feng et al., 2019; Jiang et al., 2021) could also been enhanced either.

Last, some specific tasks, such as change impact analysis or change recommendation (Zimmermann et al., 2005; Rolfsnes et al., 2016; Jiang et al., 2020; Mondal et al., 2020a; Jiang et al., 2021), could benefit from our approach or dataset. We have demonstrated that long-term changes are widely existing in the process of software development, and our proposed spatial–temporal change relationship graph is effective for capturing the long-term changes. Based on our approach and dataset, researchers could further improve the performance of impact prediction or recommendation tools, e.g., increasing its recall. For example, Mondal et al. (2020a) successfully associated code clones with association rules to better identify the impact sets. Our preliminary experiments shows that the integration of evolutionary coupling and spatial–temporal change relationships among code entities effectively enhances the recall of impact prediction, which could serve as our future work.

Software Designers or Architects can use our tool to optimize the design of the software. For example, the *No. 30* FCPC, $CE \xrightarrow{ACC} CE \xleftarrow{ACC} CE$, represents that a change on the intermediate code entity (i.e., Variable) can have an impact on all code entities that have inward (afferent) ACC relationships. Note that our mined FCPCs cross three files, which means that the code entities outside the class the member variable belongs to could directly access the variable. This clearly violates the encapsulation feature of member variables in object-oriented languages. Another example comes from the clone-related FCPCs, i.e., if the changed code entities (clone related) always have a change time interval, these entities may suffer inconsistent changes that should draw the attention of software designers.

Tool Builders can use our tool or dataset to build useful tools, such as dependency analysis tools, code entity mapping tools, technical debt detection tools, etc. For example, DV8 (Cai and Kazman, 2019), an automated architecture analysis tool suite, is inspired from architecture anti-pattern. Our mined FCPCs, e.g., *No. 30*, could provide more inspiration for tool builders to create debt detection tools.

Developers. In Sections Section 5.2.5, we observe evidence that some code entities do appear frequently in the FCPCs and the involved files contribute to a project's bug-proneness. The mined FCPCs can be a bridge to link the code entities which always change together in a period of time. The FCPCs could be used as a signal or hint for developers to search for the related bug-prone files or code entities even if they have no direct relationships.

8. Threats to validity

Our results and findings in the empirical study suffer from several internal and external threats.

Threats to internal validity: Our approach largely depends on the precision and recall of the detection of the static dependencies and code

clones. Wrongly detected dependencies and code clones are threats to the validity of our results. To mitigate this threat, we have carefully chosen the static analysis tool and the clone detection tool and spent large effort in optimizing the tools. Manual verification on random samples confirms that the precision and recall of the detection of dependencies and code clones are acceptable in our study.

Another threat comes from our choice of the *two-hop* relationships among code entities. It is possible that longer hops also contribute to the propagation of impacts. Considering the possibility that changes in the same class or file may propagate change impacts, the minimal case of a CPC covers four code entities in at most three files. This means that the two code entities in the common class or files are treated together as an intermediate entity, which extends the length of relationship hops and mitigates the threat. Considering larger length of relationship hops will be our future work.

Moreover, we limit the number of files touched in a core commit up to 5, and the number of commits in ST-window up to 15. We find that such settings are balanced between generality and performance, which mitigates the threat. We will also test the sensibility of these thresholds in the future.

Threats to external validity: We only consider five open source projects in this work, all of which are developed in Java and are from the Apache ecosystem. Our findings may not apply to other software systems, such as systems developed in other languages or enterprise proprietary software systems.

9. Related work

Our work is related to change pattern mining, change recommendation, and software evolution analysis.

9.1. Change pattern mining

The closest related work is change pattern mining, which is used to understand code changes and support software evolution (Silva et al., 2014; Jaafar et al., 2014; Silva et al., 2015a; Wang et al., 2018b; Silva et al., 2019; Feng et al., 2019; Jiang et al., 2020, 2021; Huang et al., 2022). For instance, Wang et al. (2018b) identified six recurring change patterns from the change dependency graphs of different bug fixes. They focused on three kinds of software entities: classes, field, and method, and created change dependency graphs (CDGs) to connect the related changed entities. Similarly, Jiang et al. (2020, 2021) investigated the recurring co-change patterns commonly exist in JavaScript project, and they found five most popular recurring change patterns. Silva et al. (2019) conducted a large-scale study with GitHub projects to evaluate system modularity using the co-change clustering technique (Silva et al., 2014, 2015a), and revealed six co-change patterns by projecting frequently co-changed files over the directory structure. They extracted co-change graphs from history information and mine co-change clusters in file granularity. The six co-change pattern were mainly differed on the directory of co-change files. Feng et al. (2019) proposed an active hotspot model to detect and monitor the emergence and evolution of software degradation. They discovered four recurring propagation patterns at the file level.

Several researchers mined code change patterns from a fine-grained code changes (Negara et al., 2014; Kreutzer et al., 2016; Molderez et al., 2017; Nguyen et al., 2019; Janke and Mäder, 2022). For instance, Negara et al. (2014) identified high-level, in-the-wild frequent code change patterns from a fine-grained sequence of code changes. They identified ten popular high-level program transformations describing frequent code changes. Nguyen et al. (2019) proposed a graph-based mining approach, CPATMINER, to identify repetitive changes in the wild, by mining fine-grained semantic code change patterns from a large number of repositories. Janke and Mäder (2022) mined code change patterns by capturing the relational context of individual edits.

However, our research is different from theirs in three ways. First, we constructed ST-CRGs for the core commit by a sliding spatial–temporal window, which means not only the co-changes but also the long-term changes were all considered. As we know, ripple effects could happen in a series of commits, or caused by direct and indirect dependencies (Yau et al., 1978; Hassan and Holt, 2004; Feng et al., 2019). Our approach could capture the propagation of impacts in both temporal and spatial space, considering the co-changes and long-term changes together. Second, we extracted eleven dependency types for software source code to construct ST-CRGs. It allowed us to capture the related changed entities as many as possible and filter out lots of irrelevant change information. Third, we translated the ST-CRGs into CPC-Gs by aggregating relationships between the changed code entities and mined frequent change propagation channels automatically. Compared with the previous manual summary of change patterns, our method greatly reduces the workload of pattern mining.

9.2. Change recommendation

Various tools were built to mine version histories for code change recommendation (Ren et al., 2004; Ying et al., 2004; Zimmermann et al., 2005; Hassan and Holt, 2004; Rolfsnes et al., 2016, 2017; Islam et al., 2018; Rolfsnes et al., 2018; Agrawal and Singh, 2020; Jiang et al., 2020; Mondal et al., 2020b; Moonen et al., 2020; Jiang et al., 2021). For instance, Zimmermann et al. (2005) developed a prototype tool, named ROSE, to suggest further code changes by the association rules. Rolfsnes et al. (2016) proposed an algorithm called TARMAQ for mining evolutionary coupling and predicted co-change candidates in file level. Some researches focused on code change recommendation in method level. Islam et al. (2018) used transitive association rules to realize evolutionary coupling among program entities that did not co-change in the past. By combined with regular and transitive association rules, their co-change prediction mechanism outperformed TARMAQ. These works conduct code change recommendation mainly depending on the association rules.

Some other researches predicted code changes based on change pattern mining. For instance, Ying et al. (2004) mined frequent changed together files from the code change history and helped developers identify relevant source code for changing. Wang et al. (2018a), Jiang et al. (2020) introduced an automatic approach, named CMSuggester, to suggest complementary changes for multi-entity edits in Java programs. Jiang et al. (2021) conducted an empirical study to find the recurring co-change patterns commonly exist in JavaScript project and built a machine learning (ML)-based approach to recommend code changes. Similar to these works, our research could also be used to suggest code changes, which may construct our future work.

9.3. Software evolution analysis

A variety of studies are concerned about the evolution of software systems across evolutionary coupling analysis (Yu, 2007; Wong and Cai, 2011; Jiang et al., 2017; Zhou et al., 2019). Yu (2007) studied the evolution of 12 Linux kernel modules and found a linear correlation between evolutionary coupling and structural coupling, which could help developers understand software components co-evolution. Jiang et al. (2017) presented an approach to group and aggregate relevant code changes into six types of trajectory patterns which were useful for software evolution management and quality assurance. Zhou et al. (2019) conducted an empirical study on six open source systems to understand evolutionary coupling by fine-grained co-change relationship analysis. They revealed six co-change relation types and visualized them in pixelmaps to understand the co-evolution file pairs.

There are works which focus on architectural evolution and decay (Behnamghader et al., 2017; Le et al., 2018; Feng et al., 2019; Garcia et al., 2022; Cui, 2021). Behnamghader et al. (2017) utilized ARCADE to compute the decay and change metrics to observe and

track architecture evolution. Le et al. (2018) investigated the nature and impact of architectural smells and found its strongly correlation to implementation issues. Garcia et al. (2022) utilized multiple architectural views and architectural metrics as features to predict the quality of an architectural element.

Our research focus on mining change propagation channels to understand code changes during software evolution. These change propagation channels could be used to reveal the relationship types between code entities that frequently propagate changes and help software evolution analysis.

10. Conclusion

To reveal the path of change propagation, we introduced the concept of *change propagation channel* to represent the relationship types between code entities that frequently propagate changes during a period of time and within a limited space. Our approach mine frequent change propagation channels with considerations of both *co-changes* and *long-term changes* in the software evolution history. Our empirical study has shown that the long-term changes widely exist in the cases of code change propagation and should not be ignored in change propagation analysis. The most frequent change propagation channels revealed by our study are sufficiently representative with regard to the target systems because over 98% cases of code change propagation are covered by the channels, showing the possibility that developers may only consider the small part of the change propagation channels. Manual validation also shows that the changes covered by the top frequent change propagation channels are mostly relevant, to some extent, and could be helpful for developers to get alerted on the directions to which the changes on the code entities are likely to be propagated.

We notice great opportunities in the research on developing new code change recommendation approaches or impact prediction approaches by finding the missing changes. We also schedule extensive studies on more projects from real world to investigate longer change propagation channels and to improve the explainability of the channels in our future work.

CRediT authorship contribution statement

Daihong Zhou: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Yijian Wu:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing. **Xin Peng:** Conceptualization, Supervision. **Jiyue Zhang:** Software, Validation. **Ziliang Li:** Software, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data used in this work are publicly available at https://github.com/FudanSELab/cpcminer.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (62172099).

References

Agrawal, R., Imielinski, T., Swami, A.N., 1993. Mining association rules between sets of items in large databases. In: Buneman, P., Jajodia, S. (Eds.), Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. Washington, DC, USA, May 26–28, 1993, ACM Press, pp. 207–216. http://dx.doi.org/10.1145/170035.170072.

Agrawal, A., Singh, R.K., 2020. Predicting co-change probability in software applications using historical metadata. IET Softw. 14 (7), 739–747.

Ajienka, N., Capiluppi, A., Counsell, S., 2018. An empirical study on the interplay between semantic coupling and co-change of software classes. Empir. Softw. Eng. 23 (3), 1791–1825.

Asaduzzaman, M., Roy, C.K., Schneider, K.A., Penta, M.D., 2013. LHDiff: A language-independent hybrid approach for tracking source code lines. In: 2013 IEEE International Conference on Software Maintenance. Eindhoven, the Netherlands, September 22–28, 2013, IEEE Computer Society, pp. 230–239. http://dx.doi.org/10.1109/ICSM.2013.34.

Ashraf, U., Mayr-Dorn, C., Egyed, A., 2019. Mining cross-task artifact dependencies from developer interactions. In: Wang, X., Lo, D., Shihab, E. (Eds.), 26th IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2019, Hangzhou, China, February 24-27, 2019, IEEE, pp. 186–196. http://dx.doi.org/10.1109/SANER.2019.8667990.

Barbour, L., Khomh, F., Zou, Y., 2011. Late propagation in software clones. In: IEEE 27th International Conference on Software Maintenance. ICSM 2011, Williamsburg, VA, USA, September 25–30, 2011, IEEE Computer Society, pp. 273–282. http://dx.doi.org/10.1109/ICSM.2011.6080794.

Behnamghader, P., Le, D.M., Garcia, J., Link, D., Shahbazian, A., Medvidovic, N., 2017. A large-scale study of architectural evolution in open-source software systems. Empir. Softw. Eng. 22 (3), 1146–1193.

Brudaru, I.I., Zeller, A., 2008. What is the long-term impact of changes? In: Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering. RSSE 2008, Atlanta, GA, USA, November 9, 2008, ACM, pp. 30–32. http://dx.doi.org/10.1145/1454247.1454257.

Cai, Y., Kazman, R., 2019. DV8: automated architecture analysis tool suites. In: Avgeriou, P., Schmid, K. (Eds.), Proceedings of the Second International Conference on Technical Debt. TechDebt@ICSE 2019, Montreal, QC, Canada, May 26-27, 2019, IEEE / ACM, pp. 53–54. http://dx.doi.org/10.1109/TechDebt.2019.00015, URL: https://dl.acm.org/citation.cfm?id=3355333.

Cliff, N., 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol. Bull. 114 (3), 494.

Cordella, L.P., Foggia, P., Sansone, C., Vento, M., 2004. A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. 26 (10), 1367–1372.

Cui, D., 2021. Early detection of flawed structural dependencies during software evolution. IEEE Access 9, 28856–28871.

Cui, D., Liu, T., Cai, Y., Zheng, Q., Feng, Q., Jin, W., Guo, J., Qu, Y., 2019. Investigating the impact of multiple dependency structures on software defects. In: Atlee, J.M., Bultan, T., Whittle, J. (Eds.), Proceedings of the 41st International Conference on Software Engineering. ICSE 2019, Montreal, QC, Canada, May 25–31, 2019, IEEE / ACM, pp. 584–595. http://dx.doi.org/10.1109/ICSE.2019.00069.

Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M., 2014. Fine-grained and accurate source code differencing. In: Crnkovic, I., Chechik, M., Grünbacher, P. (Eds.), ACM/IEEE International Conference on Automated Software Engineering. ASE '14, Vasteras, Sweden - September 15–19, 2014, ACM, pp. 313–324. http://dx.doi.org/10.1145/2642937.2642982.

Feng, Q., Cai, Y., Kazman, R., Cui, D., Liu, T., Fang, H., 2019. Active hotspot: An issue-oriented model to monitor software evolution and degradation. In: 34th IEEE/ACM International Conference on Automated Software Engineering. ASE 2019, San Diego, CA, USA, November 11–15, 2019, IEEE, pp. 986–997. http://dx.doi.org/10.1109/ASE.2019.00095.

Fluri, B., Würsch, M., Pinzger, M., Gall, H.C., 2007. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. 33 (11), 725–743.

Garcia, J., Kouroshfar, E., Ghorbani, N., Malek, S., 2022. Forecasting architectural decay from evolutionary history. IEEE Trans. Softw. Eng. 48 (7), 2439–2454.

Han, J., 1997. Supporting impact analysis and change propagation in software engineering environments. In: Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering. pp. 172–182. http://dx.doi.org/10.1109/STEP.1997.615479.

Hassan, A.E., Holt, R.C., 2004. Predicting change propagation in software systems. In: 20th International Conference on Software Maintenance. ICSM 2004, 11–17 September 2004, Chicago, IL, USA, IEEE Computer Society, pp. 284–293. http://dx.doi.org/10.1109/ICSM.2004.1357812.

Herzig, K.S., 2010. Capturing the long-term impact of changes. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (Eds.), Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2. ICSE 2010, Cape Town, South Africa, 1–8 May 2010, ACM, pp. 393–396. http://dx.doi.org/10.1145/1810295.1810401.

Herzig, K., Zeller, A., 2011. Mining cause-effect-chains from version histories. In: Dohi, T., Cukic, B. (Eds.), IEEE 22nd International Symposium on Software Reliability Engineering. ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011, IEEE Computer Society, pp. 60–69. http://dx.doi.org/10.1109/ISSRE.2011.16.

Herzig, K., Zeller, A., 2013. The impact of tangled code changes. In: Zimmermann, T., Penta, M.D., Kim, S. (Eds.), Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13, San Francisco, CA, USA, May 18–19, 2013, IEEE Computer Society, pp. 121–130. http://dx.doi.org/10.1109/MSR.2013.6624018.

Huang, K., Chen, B., Peng, X., Zhou, D., Wang, Y., Liu, Y., Zhao, W., 2018. Cldiff: generating concise linked code differences. In: Huchard, M., Kästner, C., Fraser, G. (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ASE 2018, Montpellier, France, September 3–7, 2018, ACM, pp. 679–690. http://dx.doi.org/10.1145/3238147.3238219.

Huang, Y., Jiang, J., Luo, X., Chen, X., Zheng, Z., Jia, N., Huang, G., 2022. Change-patterns mapping: A boosting way for change impact analysis. IEEE Trans. Softw. Eng. 48 (7), 2376–2398.

Islam, M.A., Islam, M.M., Mondal, M., Roy, B., Roy, C.K., Schneider, K.A., 2018. [Research paper] detecting evolutionary coupling using transitive association rules. In: 18th IEEE International Working Conference on Source Code Analysis and Manipulation. SCAM 2018, Madrid, Spain, September 23–24, 2018, IEEE Computer Society, pp. 113–122. http://dx.doi.org/10.1109/SCAM.2018.00020.

Jaafar, F., Guéhéneuc, Y., Hamel, S., Antoniol, G., 2011. An exploratory study of macro co-changes. In: Proceedings of the 18th Working Conference on Reverse Engineering. WCRE 2011, Limerick, Ireland, October 17–20, 2011, IEEE Computer Society, pp. 325–334.

Jaafar, F., Guéhéneuc, Y., Hamel, S., Antoniol, G., 2014. Detecting asynchrony and dephase change patterns by mining software repositories. J. Softw.: Evol. Process 26 (1), 77–106.

Janke, M., Mäder, P., 2022. Graph based mining of code change patterns from version control commits. IEEE Trans. Softw. Eng. 48 (3), 848–863.

Jiang, Q., Peng, X., Wang, H., Xing, Z., Zhao, W., 2017. Understanding systematic and collaborative code changes by mining evolutionary trajectory patterns. J. Softw.: Evol. Process 29 (3).

Jiang, Z., Wang, Y., Zhong, H., Meng, N., 2020. Automatic method change suggestion to complement multi-entity edits. J. Syst. Softw. 159.

Jiang, Z., Zhong, H., Meng, N., 2021. Investigating and recommending co-changed entities for JavaScript programs. J. Syst. Softw. 180, 111027.

Kim, S., Whitehead, Jr., E.J., Zhang, Y., 2008. Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng. 34 (2), 181–196.

Kreutzer, P., Dotzler, G., Ring, M., Eskofier, B.M., Philippsen, M., 2016. Automatic clustering of code changes. In: Kim, M., Robbes, R., Bird, C. (Eds.), Proceedings of the 13th International Conference on Mining Software Repositories. MSR 2016, Austin, TX, USA, May 14–22, 2016, ACM, pp. 61–72. http://dx.doi.org/10.1145/2901739.2901749.

Le, D.M., Link, D., Shahbazian, A., Medvidovic, N., 2018. An empirical study of architectural decay in open-source software. In: IEEE International Conference on Software Architecture. ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018, IEEE Computer Society, pp. 176–185. http://dx.doi.org/10.1109/ICSA.2018.00027.

Li, G., Wu, Y., Roy, C.K., Sun, J., Peng, X., Zhan, N., Hu, B., Ma, J., 2020. SAGA: efficient and large-scale detection of near-miss clones with GPU acceleration. In: Kontogiannis, K., Khomh, F., Chatzigeorgiou, A., Fokaefs, M., Zhou, M. (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2020, London, on, Canada, February 18-21, 2020, IEEE, pp. 272–283. http://dx.doi.org/10.1109/SANER48275.2020.9054832.

Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. 50–60.

Mockus, A., Fielding, R.T., Herbsleb, J.D., 2002. Two case studies of open source software development: Apache and mozilla. ACM Trans. Softw. Eng. Methodol. 11 (3), 309–346.

Mockus, A., Votta, L.G., 2000. Identifying reasons for software changes using historic databases. In: 2000 International Conference on Software Maintenance. ICSM 2000, San Jose, California, USA, October 11–14, 2000, IEEE Computer Society, pp. 120–130. http://dx.doi.org/10.1109/ICSM.2000.883028.

Molderez, T., Stevens, R., Roover, C.D., 2017. Mining change histories for unknown systematic edits. In: González-Barahona, J.M., Hindle, A., Tan, L. (Eds.), Proceedings of the 14th International Conference on Mining Software Repositories. MSR 2017, Buenos Aires, Argentina, May 20-28, 2017, IEEE Computer Society, pp. 248–256. http://dx.doi.org/10.1109/MSR.2017.12.

Mondal, M., Roy, B., Roy, C.K., Schneider, K.A., 2019. An empirical study on bug propagation through code cloning. J. Syst. Softw. 158.

Mondal, M., Roy, B., Roy, C.K., Schneider, K.A., 2020a. Associating code clones with association rules for change impact analysis. In: Kontogiannis, K., Khomh, F., Chatzigeorgiou, A., Fokaefs, M., Zhou, M. (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2020, London, on, Canada, February 18–21, 2020, IEEE, pp. 93–103. http://dx.doi.org/10.1109/SANER48275.2020.9054846.

Mondal, M., Roy, B., Roy, C.K., Schneider, K.A., 2020b. HistoRank: History-based ranking of co-change candidates. In: Kontogiannis, K., Khomh, F., Chatzigeorgiou, A., Fokaefs, M., Zhou, M. (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2020, London, on, Canada, February 18–21, 2020, IEEE, pp. 240–250. http://dx.doi.org/10.1109/SANER48275.2020.9054869.

Moonen, L., Binkley, D.W., Pugh, S., 2020. On adaptive change recommendation. J. Syst. Softw. 164, 110550.

Moonen, L., Di Alesio, S., Binkley, D.W., Rolfsnes, T., 2016. Practical guidelines for change recommendation using association rule mining. In: Lo, D., Apel, S., Khurshid, S. (Eds.), Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE 2016, Singapore, September 3–7, 2016, ACM, pp. 732–743. http://dx.doi.org/10.1145/2970276.2970327.

Negara, S., Codoban, M., Dig, D., Johnson, R.E., 2014. Mining fine-grained code changes to detect unknown change patterns. In: Jalote, P., Briand, L.C., van der Hoek, A. (Eds.), 36th International Conference on Software Engineering. ICSE '14, Hyderabad, India - May 31 - June 07, 2014, ACM, pp. 803–813. http://dx.doi.org/10.1145/2568225.2568317.

Nguyen, H.A., Nguyen, T.N., Dig, D., Nguyen, S., Tran, H., Hilton, M., 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In: Atlee, J.M., Bultan, T., Whittle, J. (Eds.), Proceedings of the 41st International Conference on Software Engineering. ICSE 2019, Montreal, QC, Canada, May 25–31, 2019, IEEE / ACM, pp. 819–830. http://dx.doi.org/10.1109/ICSE.2019.00089.

Oliva, G.A., Gerosa, M.A., 2015. Experience report: How do structural dependencies influence change propagation? An empirical study. In: 26th IEEE International Symposium on Software Reliability Engineering. ISSRE 2015, Gaithersbury, MD, USA, November 2–5, 2015, IEEE Computer Society, pp. 250–260. http://dx.doi.org/10.1109/ISSRE.2015.7381818.

Pan, W., Hua, M., Chang, C.K., Yang, Z., Kim, D., 2021. ElementRank: Ranking java software classes and packages using a multilayer complex network-based approach. IEEE Trans. Softw. Eng. 47 (10), 2272–2295.

Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.C., 2004. Chianti: a tool for change impact analysis of java programs. In: Vlissides, J.M., Schmidt, D.C. (Eds.), Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada, ACM, pp. 432–448. http://dx.doi.org/10.1145/1028976.1029012.

Rolfsnes, T., Di Alesio, S., Behjati, R., Moonen, L., Binkley, D.W., 2016. Generalizing the analysis of evolutionary coupling for software change impact analysis. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14–18, 2016 - Volume 1. IEEE Computer Society, pp. 201–212. http://dx.doi.org/10.1109/SANER.2016.101.

Rolfsnes, T., Moonen, L., Binkley, D.W., 2017. Predicting relevance of change recommendations. In: Rosu, G., Penta, M.D., Nguyen, T.N. (Eds.), Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, IEEE Computer Society, pp. 694–705. http://dx.doi.org/10.1109/ASE.2017.8115680.

Rolfsnes, T., Moonen, L., Di Alesio, S., Behjati, R., Binkley, D.W., 2018. Aggregating association rules to improve change recommendation. Empir. Softw. Eng. 23 (2), 987–1035.

Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J., 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In: Annual Meeting of the Florida Association of Institutional Research, Vol. 177. p. 34.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V., 2016. Sourcerercc: scaling code clone detection to big-code. In: Dillon, L.K., Visser, W., Williams, L.A. (Eds.), Proceedings of the 38th International Conference on Software Engineering. ICSE 2016, Austin, TX, USA, May 14–22, 2016, ACM, pp. 1157–1168. http://dx.doi.org/10.1145/2884781.2884877.

Shen, B., Zhang, W., Kästner, C., Zhao, H., Wei, Z., Liang, G., Jin, Z., 2021. SmartCommit: a graph-based interactive assistant for activity-oriented commits. In: Spinellis, D., Gousios, G., Chechik, M., Penta, M.D. (Eds.), ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Athens, Greece, August 23–28, 2021, ACM, pp. 379–390. http://dx.doi.org/10.1145/3468264.3468551.

Silva, L.L., Valente, M.T., de Almeida Maia, M., 2014. Assessing modularity using co-change clusters. In: Binder, W., Ernst, E., Peternier, A., Hirschfeld, R. (Eds.), 13th International Conference on Modularity. MODULARITY '14, Lugano, Switzerland, April 22–26, 2014, ACM, pp. 49–60. http://dx.doi.org/10.1145/2577080.2577086.

Silva, L.L., Valente, M.T., de Almeida Maia, M., 2015a. Co-change clusters: Extraction and application on assessing software modularity. LNCS Trans. Aspect Oriented Softw. Dev. 12, 96–131.

Silva, L.L., Valente, M.T., de Almeida Maia, M., 2019. Co-change patterns: A large scale empirical study. J. Syst. Softw. 152, 196–214.

Silva, L.L., Valente, M.T., de Almeida Maia, M., Anquetil, N., 2015b. Developers' perception of co-change patterns: An empirical study. In: Koschke, R., Krinke, J., Robillard, M.P. (Eds.), 2015 IEEE International Conference on Software Maintenance and Evolution. ICSME 2015, Bremen, Germany, September 29 - October 1, 2015, IEEE Computer Society, pp. 21–30. http://dx.doi.org/10.1109/ICSM.2015.7332448.

Wang, Y., Meng, N., Zhong, H., 2018a. CMSuggester: Method change suggestion to complement multi-entity edits. In: Bu, L., Xiong, Y. (Eds.), Software Analysis, Testing, and Evolution - 8th International Conference, SATE 2018, Shenzhen, Guangdong, China, November 23–24, 2018, Proceedings. In: Lecture Notes in Computer Science, 11293, Springer, pp. 137–153. http://dx.doi.org/10.1007/978-3-030-04272-1_9.

Wang, Y., Meng, N., Zhong, H., 2018b. An empirical study of multi-entity changes in real bug fixes. In: 2018 IEEE International Conference on Software Maintenance and Evolution. ICSME 2018, Madrid, Spain, September 23–29, 2018, IEEE Computer Society, pp. 287–298. http://dx.doi.org/10.1109/ICSME.2018.00038.

Wong, S., Cai, Y., 2011. Generalizing evolutionary coupling with stochastic dependencies. In: Alexander, P., Pasareanu, C.S., Hosking, J.G. (Eds.), 26th IEEE/ACM International Conference on Automated Software Engineering. ASE 2011, Lawrence, KS, USA, November 6–10, 2011, IEEE Computer Society, pp. 293–302. http://dx.doi.org/10.1109/ASE.2011.6100065.

Xing, Z., Stroulia, E., 2005. Umldiff: an algorithm for object-oriented design differencing. In: Redmiles, D.F., Ellman, T., Zisman, A. (Eds.), 20th IEEE/ACM International Conference on Automated Software Engineering. ASE 2005, November 7–11, 2005, Long Beach, CA, USA, ACM, pp. 54–65. http://dx.doi.org/10.1145/1101908.1101919.

Yau, S.S., Collofello, J.S., MacGregor, T., 1978. Ripple effect analysis of software maintenance. In: The IEEE Computer Society's Second International Computer Software and Applications Conference. COMPSAC 1978, 13–16 November, 1978, Chicago, Illinois, USA, IEEE, pp. 60–65. http://dx.doi.org/10.1109/CMPSAC.1978.810308.

Ying, A.T.T., Murphy, G.C., Ng, R.T., Chu-Carroll, M., 2004. Predicting source code changes by mining change history. IEEE Trans. Softw. Eng. 30 (9), 574–586.

Yu, L., 2007. Understanding component co-evolution with a study on Linux. Empir. Softw. Eng. 12 (2), 123–141.

Zhou, D., Wu, Y., Xiao, L., Cai, Y., Peng, X., Fan, J., Huang, L., Chen, H., 2019. Understanding evolutionary coupling by fine-grained co-change relationship analysis. In: Guéhéneuc, Y., Khomh, F., Sarro, F. (Eds.), ICPC 2019, Montreal, QC, Canada, May 25–31, 2019, IEEE / ACM, pp. 271–282. http://dx.doi.org/10.1109/ICPC.2019.00046.

Zimmermann, T., Weißgerber, P., Diehl, S., Zeller, A., 2005. Mining version histories to guide software changes. IEEE Trans. Softw. Eng. 31 (6), 429–445.

Daihong Zhou is a Ph.D. student in School of Computer Science of Fudan University, Shanghai, China. He received his master degree in China University of Mining and Technology (Beijing) in Beijing, China. His research interest include software evolution analysis, mining software repository, technical debt, and graph data analysis.

Yijian Wu received his Ph.D. degree from Fudan University in 2006. He is currently an associate professor in Fudan University. His research interests mainly focus on software architecture, software reuse and product lines, and software evolution in industrial environments.

Xin Peng received his Ph.D. degree from Fudan University in 2006. He is currently a full professor and Deputy Dean of School of Computer Science in Fudan University. His research interests mainly focus on intelligent software engineering techniques, including intelligent software development and operation, microservices and cloud native architecture, and software development data analysis.

Jiyue Zhang is a master student in School of Computer Science of Fudan University. He received his Bachelor degree in the School of Computer Science and Technology of Donghua University in Shanghai, China in 2020. His research interests include software evolution analysis, mining software repository, and technical debt.

Ziliang Li is a master student in School of Computer Science of Fudan University. He received his bachelor degree in School of Computer Science and Technology of Tianjin University in Tianjin, China in 2019. His research interest is software evolution analysis, mining software repository, and technical debt.