SmartCommit: A Graph-Based Interactive Assistant for Activity-Oriented Commits

Bo Shen
Key Lab of High Confidence Software Technology (Peking University), MoE
Beijing, China

Wei Zhang
Key Lab of High Confidence Software Technology (Peking University), MoE
Beijing, China

Christian Kästner
Carnegie Mellon University
Pittsburgh, PA, USA

Haiyan Zhao
Key Lab of High Confidence Software Technology (Peking University), MoE
Beijing, China

Zhao Wei
Huawei Technologies Co., Ltd.
Beijing, China

Guangtai Liang
Huawei Technologies Co., Ltd.
Beijing, China

Zhi Jin
Key Lab of High Confidence Software Technology (Peking University), MoE
Beijing, China

ABSTRACT

In collaborative software development, it is considered to be a best practice to submit code changes as a sequence of cohesive commits, each of which records the work result of a specific development *activity*, such as adding a new feature, bug fixing, and refactoring. However, rather than following this best practice, developers often submit a set of loosely-related changes serving for different development activities as a *composite commit*, due to the tedious manual work and lack of effective tool support to decompose such a *tangled changeset*. Composite commits often obfuscate the change history of software artifacts and bring challenges to efficient collaboration among developers. To encourage *activity-oriented* commits, we propose *SmartCommit*, a graph-partitioning-based interactive approach to tangled changeset decomposition that leverages not only the efficiency of algorithms but also the knowledge of developers. To evaluate the effectiveness of our approach, we (1) deployed *SmartCommit* in an international IT company, and analyzed usage data collected from a field study with 83 engineers over 9 months; and (2) conducted a controlled experiment on 3,000 synthetic composite commits from 10 diverse open-source projects. Results show that *SmartCommit* achieves a median accuracy between 71–84% when decomposing composite commits without developer involvement, and significantly helps developers follow the best practice of submitting *activity-oriented* commits with acceptable interaction effort and time cost in real collaborative software development.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468551

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; **Collaboration in software development**.

KEYWORDS

revision control system, code commit, changes decomposition

ACM Reference Format:
Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. 2021. SmartCommit: A Graph-Based Interactive Assistant for Activity-Oriented Commits. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468551

1 INTRODUCTION

In collaborative software development, software artifacts are usually under continuous evolution. Developers modify software artifacts for various development tasks/activities, such as implementing new features, fixing bugs or vulnerabilities, and refactoring. To efficiently manage the life-cycle of software artifacts, the *version control system* (VCS) like Git is employed to track the change history of software artifacts as a sequence of *commits*: each commit consists of a set of differences (*diffs*) representing the changes of source code since the last version, as well as a textual description (*commit message*) summarizing the content and intent of these code changes. Consequently, commits act as the basic unit to organize changes on software artifacts, upon which many other VCS functions are built.

It is considered to be a best practice to submit code changes as a sequence of cohesive commits, each of which records the work result of a specific development *activity* [4, 28]; such kind of commits is also called *atomic*, *single-activity* or *activity-oriented* commits. When following this practice, a clear change history of software artifacts can be maintained, which benefits various activities related to software development [22]. For an individual developer, activity-oriented commits make it easier to revert a buggy commit or reuse

specific changes. For a team of developers, activity-oriented commits make it easier for a developer to understand and review code changes made by other developers [1, 3, 28]. For a researcher in the field of software development, activity-oriented commits bring less noise when carrying out studies that depend on historical data (e.g. mining software repositories) [8, 12]. Because of these benefits, developers are encouraged to follow this best practice in many open-source communities (e.g. Git official[1] and Angular[2]) and companies (e.g. Google[3]), through explicit regulations and guidelines.

However, rather than following this best practice, developers often submit a set of loosely-related changes serving for different development activities as a *composite commit* [8], due to the tedious manual work and lack of effective tool support to decompose such a *tangled changeset*. In daily development, developers are often observed to perform several independent activities simultaneously [14], such as refactoring the code structure while working on a new feature (*floss refactoring* [20, 21]) or fixing a bug along the way while optimizing a function, leading to a tangled changeset in the workspace. Prior empirical studies repeatedly and consistently confirmed that composite commits are common in software projects, both for open-source and industrial ones. Herzig and Zeller [8] found that composite commits occur in up to 15% of all bug-fix commits in 5 open-source projects. Tao and Kim [28] studied 453 code changes from 4 open-source projects and concluded that on average 17% and up to 29% of all commits are composite and address two or more issues. Barnett et al. [1] found that changesets can be decomposed in over 40% of code reviews at Microsoft. In researches on defect prediction and localization, Nguyen et al. [23] reported that 11% to 39% of bug-fix commits include other changes. Worse still, the chances of depending on developers to eliminate composite commits are slim, because of unawareness or tedious manual effort [22]. On one hand, some developers simply do not know the best practice. On another hand, although some VCSs (like Git interactive staging[4]) or tools (like *GitKraken* and *Fork*) do provide functions to decompose changes in a developer's workspace, these functions totally depend on manual operations.

Researchers have proposed various approaches to address the problem of composite commits by decomposing them with algorithms, typically based on heuristic rules [8, 34], program slicing techniques [22], code dependency [1, 32] or pattern matching [4]. But there are still two weaknesses in these approaches. First, most of these approaches identify and decompose composite commits temporarily at the code review stage, where composite commits have been recorded in the version history or shared with collaborators. We argue that it is more meaningful to prevent the occurrence of composite commits, by proactively decomposing the corresponding tangled changeset before the submission, as also suggested in several works [12, 13, 26]. Second, these approaches often generate too fine-grained decomposition results [22], which can hardly be used as activity-oriented commits. The cause is that these approaches usually rely on a limited set of information retrieved from source code; many valuable relations between diff hunks are out of consideration, leading to the result that two diff hunks that should be submitted together are separated. We argue that a promising approach should identify and utilize a diverse set of information, even including knowledge from developers.

For the purpose of reminding, leading, and assisting developers to follow the best practice of activity-oriented commits, in this work we propose an interactive approach, named *SmartCommit*, which leverages both the efficiency of the algorithm and the knowledge of the developer. In particular, this approach has two features. First, it assumes that it is very difficult to design a fully automated algorithm that can decompose tangled changesets accurately and without developers' intervention, since what constitutes an activity-oriented commit is closely related to a project's characteristics and a developer's subjective criteria; for this reason, it employs an intuitive GUI-based interaction mechanism for a developer to participate in the decomposing process efficiently. Second, it adopts an extensible graph representation, named *Diff Hunk Graph*, to capture comprehensive semantic and heuristic relations between changes, and transforms the problem of changeset decomposition into a graph partition problem correspondingly.

We evaluated *SmartCommit* with regard to effectiveness and scalability under industrial as well as open-source settings. On one hand, we deployed *SmartCommit* in an international IT company and recruited 83 engineers to use it for 9 months. On the other hand, we generated 3,000 synthetic tangled changesets from 10 popular open-source projects (with different scales and from different domains), and compared the decomposition suggested by *SmartCommit* with the original commits (i.e. *the ground truth*). The industrial field study shows that *SmartCommit* could help developers submit activity-oriented commits with acceptable manual effort (3.41 operations on average). The open-source experiment shows that *SmartCommit* achieves a median accuracy of 71.00–83.50%, with stable performance under different input sizes (less than 5 seconds in 90% cases).

Overall, our work makes the following contributions:

- An extensible graph representation to capture 4 categories of relations between code changes, including the *Hard Links*, *Soft Links*, *Refactoring Links* and *Cosmetic Links*.
- A graph-partitioning-based algorithm to decompose tangled changesets, which outperforms the state-of-the-art approach in terms of accuracy and interpretability.
- An interaction mechanism between the decomposing algorithm and the developer, which leverages developers' knowledge through two kinds of simple adjustment operation.
- A well-engineered open-source tool for *SmartCommit*, which stands the test of a 9-month practical use as well as an extensive evaluation on 10 diverse open-source projects.

2 MOTIVATING EXAMPLE AND EXPLORATORY STUDY

In Figure 1, we show a real composite commit from *jruby* project.[5] In this commit, two Java source files were changed with 71 diff regions, but the diff content and the commit message both indicate that this commit contains code changes for multiple activities, including refactoring, fixing two issues, and cleaning up some dead code. Although this commit comes with a clear bullet-list description

[1] https://git-scm.com/docs/gitworkflows#_separate_changes
[2] https://github.com/angular/angular/blob/master/CONTRIBUTING.md
[3] https://google.github.io/eng-practices/
[4] https://git-scm.com/book/en/v2/Git-Tools-Interactive-Staging
[5] https://github.com/jruby/jruby/commit/f79d6c

(whereas many composite commits do not [8]), it is still difficult to review, integrate, or reuse, since changes corresponding to different activities are mixing up and intersecting with each other.

To better understand the state-of-the-practice about composite commits, we conducted semi-structured interviews with 6 software practitioners for their viewpoints about composite commits. They were recruited through our professional network: four (P1–P4) are industrial software engineers, two are open-source contributors (P5 and P6), and all of them have more than 2-year experience of Java programming. After showing them the above motivating example, we asked each participant to find 3 composite commits from the historical commits or pull requests in their own projects, so as to ground the later discussion. Inspired by *contextual inquiry techniques* [9], we discussed with participants about the composite commits they have submitted recently to collect insights about the cause and effect of composite commits. Each interview was conducted with an *observation* phase (we observed how each participant decomposed composite commits with Git interactive staging) and a *discussion* phase (we asked them 5 questions about the cause and effect of composite commits as well as the expected assistance). Details about questions, process and quoted user responses are available in our *supplementary data repository* [2].

We categorized the collected insights into 4 topics:

(1) *Benefits*: All participants agreed that the best practice of activity-oriented commits would generally be beneficial for development, maintenance, and collaboration among developers, e.g., making it easier to describe, understand, revert, and integrate code changes. For example, one developer (P3) mentioned that "When using git-bisect to locate bugs, it will be good if the buggy commit is small to safely revert."

(2) *Causes*: By recalling the submission process of historical composite commits, participants pointed out three major causes of composite commits: interleaved development tasks, the time stress, and the absence of regulations/guidelines/tool support in practice, e.g., "Our team have regulations about code style but nothing about commit style." (P2)

(3) *Clues*: When manually decomposing changesets, participants tended to find clues like code structure and dependency, change similarity, and refactoring. Other clues included systematic edits, change couplings, and frameworks. For example, one developer (P6) told us his experience: "To fix a buggy condition in if/for/while, I am used to also check other similar ones."

(4) *Expectations*: When asked to imagine an automatic tool for changeset decomposition, they all thought it is difficult and also unsafe to fully automate this process. But they did expect such a tool to save their effort in reviewing and organizing changes with an intuitive GUI, and if it would not change their code, some inaccuracy would be find "A few errors are tolerable as long as I can correct them quickly." (P5)

The *Benefits* and *Causes* insights confirmed the findings in the previous works, and the *Clues* insights motivated us to design an approach to decomposing changesets by simulating the process of manual decomposition. The *Expectations* from developers motivated us to make it *interactive* and co-work with the developer to improve its feasibility in practice.

Figure 1: An example composite commit from the jruby project, in which multiple activities were done and two issues were addressed in single commit.

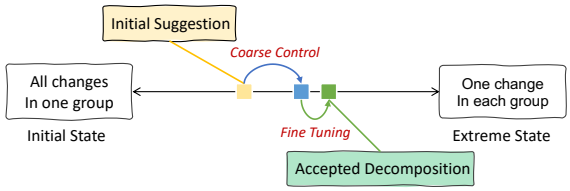


Figure 2: Getting close to the accepted decomposition through coarse and fine interactions.

3 APPROACH

In this section, we first describe the target problem as well as the major challenge. After that, we present an approach named *SmartCommit*, to address the problem and challenge.

3.1 Problem Statement

The target problem is to decompose a tangled changeset into a sequence of activity-oriented subsets, each of which corresponds to the work result of a specific development activity. The *major challenge* of this problem is how to group low-level operational code changes according to implicitly-specified high-level development activities, which can hardly be fully automated for two reasons. First, what constitutes an activity-oriented commit is often specific to project and subjective to developer. Existing approaches try to decompose changesets typically based on predefined heuristics about source code and code changes, and have been proved to be too rigid to meet the developer's expectation [22]. Second, it is difficult to encode information beyond source code (e.g. project/change-specific knowledge, or developer-specific criteria) into an algorithm to improve the decomposition process.

To cope with the above challenge, we argue that a promising approach should be a *human-in-the-loop* solution, which integrates the developer knowledge with a semi-automatic algorithm in the decomposition process. As shown in Figure 2, we consider the target problem in general as finding an accepted decomposition between the *initial* state (where all changes are submitted as a single commit) and the *extreme* state (where each change is submitted as a separate commit). In this way, the decomposition is a process where an algorithm co-works with the author of code changes to find an activity-oriented decomposition.

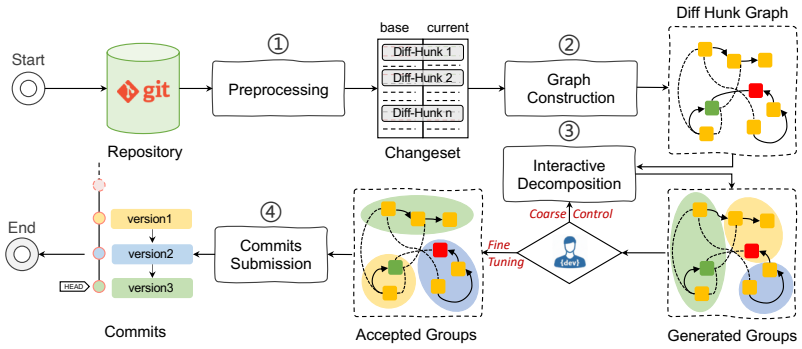


Figure 3: An overview of the proposed approach.

3.2 Approach Overview

Based on the collected insights in the exploratory study (Section 2), we propose an interactive approach named *SmartCommit* to addressing the target problem. Initially, without the developer's involvement, *SmartCommit* generates a decomposition as the *initial suggestion*, which will be presented to the developer for review. The developer then can perform *coarse control* and *fine tuning* (see Figure 2) to quickly adjust it towards the *accepted decomposition*. As a result, the decomposition generated by the algorithm offers a good start to save manual effort, whereas adjustment operations (i.e., coarse control and fine tuning) improves flexibility of our approach as well as trustworthiness of the final result.

Specifically, *SmartCommit* takes the changed files in a developer's workspace as input, and forms the accepted decomposition through 4 consecutive stages, as shown in Figure 3. In a *preprocessing* stage, changes contained in the changed files are retrieved and structured as a set of *diff hunks*. In the subsequent *graph-construction* stage, a *diff hunk graph* (cf. Definition 3.1) is constructed, which models a series of explicit and implicit *links* between diff hunks. In the next *interactive-decomposition* stage, first, an edge-shrinking algorithm is used to partition the diff hunk graph into a set of sub-graphs, each of which represents a group of closely-related diff hunks, as the initial decomposition. The developer can review the decomposition to determine whether to adjust it or not. In particular, via *coarse control*, the developer can adjust the parameter of the partition algorithm, which will lead to an updated decomposition at a different granularity, until it is close enough to the expectation; and via *fine tuning*, the developer can directly move diff hunks between change groups to further refine it accurately. In the final *commit-submission* stage, the developer selects all or a sub-set of change groups, writes commit messages, and submits them as a sequence of separate commits. Next, we will explain each of these stages in detail.

3.3 Preprocessing

The preprocessing stage targets at collecting the internal and context information for code changes, and representing them in the form of *diff hunks*. In our approach, we treat *diff hunk* as the *basic unit* to organize code changes.

Intuitively, a *diff hunk* is a combination of the textual diff with involved syntactic code elements (abstract syntax subtree(s)). A diff hunk is represented as a 5-tuple $h \doteq$ (*file_index, index, change_type, base_hunk, current_hunk*), a pair of *hunks* (namely *base_hunk* and *current_hunk*) corresponds to the version before and after the change respectively. The combination of *file_index* and *index* indicates its location in the changeset and also serves as an unique identifier of a *diff hunk*.

3.4 Graph Construction

We design a graph representation named *Diff Hunk Graph* for later decomposition, which is defined as follows:

Definition 3.1 (Diff Hunk Graph). A diff hunk graph is a *labeled*, *weighted*, and *directed* graph $G \doteq (V, E, w, T)$, where:

- V is the vertex set, each element of which represents a diff hunk in the changeset, with a unique id <file_index:index>.
- $E \subseteq \{(source, target, type) \mid (source, target) \in V \times V, source \neq target, type \in T\}$ is the edge set. Each element in E stands for a specific link between two diff hunks.
- $w : E \rightarrow (0, 1]$ is the function that maps each element in E to a weight value, which denotes the strength of the link.
- $T \doteq \{depend, refactor, similar, proximate, reformat, cleanup, moving, non\text{-}code\}$ is the set of link types currently supported, which is amenable to be extended.

Edges in this graph represent relational links between diff hunks identified from code changes that suggest two changes may result from the same development activity. Currently, we consider 4 categories of link:

(1) *Hard Links* represent syntactic and semantic constraints between AST nodes covered by diff hunks, where code in a diff hunk explicitly or implicitly depends on code in others. To detect such links, we first build a graph that combines the AST, data dependency, and call graph from the *base* and *current* versions of all changed files respectively, then locate all nodes covered by diff hunks. If two diff hunks contain nodes with direct hierarchical relation or dependency, then a *direct dependency* link will be added; if they contain nodes with multi-hop dependency, then an *indirect dependency* link will be added.

(2) *Soft Links* represent heuristic factors like similarity and proximity of diff hunks, to associate repetitive change patterns and spatially related changes. Our approach compares each

pair of diff hunks to measure the similarity textually and syntactically, and the proximity according to the syntactic hierarchy of their covered nodes.

(3) *Refactoring Links* represent multiple edits in different locations that likely originate from the same refactoring, e.g., renaming a method and all corresponding calls to it. We first detect all refactored nodes in the whole changeset, then connect the involved diff hunks.

(4) *Cosmetic Links* relate diff hunks that are cosmetic changes in nature (i.e. they do not affect the program semantics), such as reformatting, fixing typos in comments, or cleaning up comments. We detect such links by comparing the snippet and covered_nodes of the base_hunk and current_hunk.

Each link has a *weight* to indicate the relative confidence that two linked diff hunks should be committed together. For the direct dependency in *Hard Links*, the weight is assigned a maximum value of 1.0 to indicate that if such links exist, the source and target vertex should almost certainly be committed together (but the developer can still override the decision with adjustments). For other links (indirect dependency, similarity, proximity, etc.), we assign lower weights based on a measure that is specific to the kind of link. For example, the weight of indirect dependency is the reciprocal of the hop number of the shortest path between all nodes covered by two diff hunks, the similarity is with the average of textual similarity of code snippets and tree similarity of covered syntax tree nodes, and the proximity is the reciprocal of the maximal hop number from the covered nodes to their common ancestor node (see *supplementary data repository* [2] for detailed calculation rules).

Different from previous works, our approach does not depend on particular heuristic rules or features, although it does cover common heuristics in previous works. There are often multiple links of different types between two diff hunks, and we combine the information carried by different links in a flexible and extensible way via an extensible *graph* representation. It is easy to add new kinds of links through extra analyses, e.g., the evolutionary coupling [31] and the timing information of recorded edits [14].

3.5 Interactive Decomposition

3.5.1 Generating the Initial Suggestion. Based on the concept of diff hunk graph, we transform the problem of tangled-changeset decomposition into a graph partitioning problem. Inspired by the multilevel graph partitioning [11] and Kruskal's algorithm [16], we design an *edge-shrinking* algorithm (see Algorithm 1) to partition the diff hunk graph based on edges, which encode relations for decomposition. The main steps of this algorithm are described as follows. Initially, the algorithm creates an empty priority queue Q and a map representing a disjoint-set S, which initializes single-vertex partition (i.e. each element in S is a group consisting of exactly one vertex connected with itself). Then, each edge $(u, v, type)$ in the graph is added into the priority queue as a triple (wt, u, v), where the priority of an edge is determined by the weight of the edge $wt = w(u, v, type)$ first (descending), the source vertex id u next (ascending), and the target vertex id v last (ascending). After that, the algorithm enters a loop where the edge with the highest priority in Q is popped out in each iteration. If the edge's two endpoints are already in the same group, the loop will continue to the next iteration. Otherwise, if the edge's weight is above a *threshold* (0.6 by default, tuned through experiments on real composite commits in the exploratory study), the two groups that respectively contains the edge's two endpoints will be merged via the union-find algorithm. If the edge's weight is under the threshold or there are no edges left in Q, the loop will break, then we traverse S to generate groups from connected subsets. In the last, if there exist groups still with only one original vertex, these groups will be merged into one, gathering the isolated vertices that stand for *trivial* changes. A set of groups will be generated from the connected subsets of S, as the initial *suggested decomposition* for the tangled changeset.

Algorithm 1: Edge-shrinking Graph Partitioning

```
Input  : G ≐ (V, E, w, T), the diff hunk graph of a changeset;
         𝕎, the edge filtering threshold
Output : A suggested decomposition D
Function partition(G, 𝕎)
    Q ← PriorityQueue(), S ← {u : (u, 1) | u ∈ V}
    foreach e = (u, v, type) ∈ E do
        Q.add((w(e), u, v))
    end
    while Q not empty do
        wt, u, v ← Q.pop() // the edge with the highest priority
        if wt ≤ 𝕎 then break
        // already in the same group then skip
        if find(u) == find(v) then continue
        if find(u) == u&&find(v) == v then
            // both u and v are in single-vertex groups
            S[u] ← (v, wt) // group them with confidence wt
        else
            union(u, v, S) // union the groups that contains u and v
        end
    end
    D ← generateGroups(S) // get connected subsets as groups
    return D
end
Function generateGroups(S)
    D ← Set(), trivial ← ∅
    foreach u, (v, wt) ∈ S do
        if u == v then
            trivial.add(u) // collect still isolated vertices
        else
            // merge the groups that contains u and v
            merge(getGroup(D, u), getGroup(D, v))
        end
    end
    D.add(trivial) // append the group consisting of all trivial changes
    return D
end
```

Figure 4 shows an example of diff hunk graph to illustrate how the algorithm works. Each vertex in the graph represents a diff hunk, with its *id* as the label. Each edge represents the link with the maximum *weight* value between a pair of diff hunks (other links are hidden for simplification). At the beginning of the algorithm, all edges are added into a priority queue in the form of (*weight, source_index, target_index*). In each iteration of the loop, the edge with the largest weight value and the smallest id is popped out from the queue, and the source and target vertices will be put into one group, if the weight is above the threshold and the two vertices do not belong to the same group yet. The algorithm terminates when there are no unvisited edges with a weight value above the threshold. Finally, it partitions the 14 vertices into 4 groups, each of which is expected to correspond to an activity (e.g. feature addition, refactoring, adding test, and reformatting).

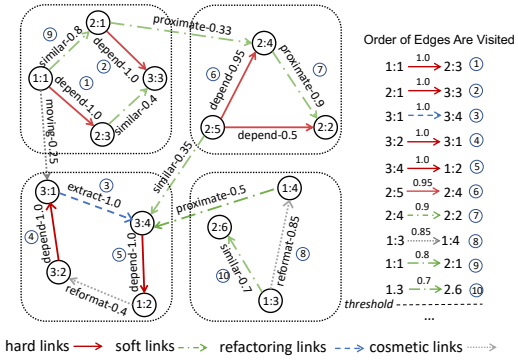


Figure 4: A simplified example of *Diff Hunk Graph*. The 14 vertices are decomposed into 4 groups. Numbers on vertices denote the *file_index:diff_hunk_index*, while numbers besides edges indicate the order they are visited.

3.5.2 Interacting with the Developer. The initial suggestion will be presented through a graphical user interface, with which the developer can review and adjust the suggested decomposition. Corresponding to Figure 2, we design two kinds of adjustment operation: *coarse control* and *fine tuning*:

- *Coarse control* is an operation that regenerates a suggested decomposition at different granularity by assigning a new value to the threshold $\mathbb{W}$ that controls the shrinking criteria of edges and the stop criteria of the decomposition algorithm.
- *Fine tuning* is an operation through which the developer can manually move diff hunks across groups or move a group to change its order.

We consider the above interacting mechanism as important as the decomposition algorithm for two reasons. First, the algorithm makes use of the feedback from the developer to quickly reach a better state that requires no or minimal extra manual effort, and can greatly save manual effort to follow the best practice of activity-oriented commits. Second, considering that the code submission is essentially a human-computer interaction process (choosing changes to stage and writing descriptions), the interaction can be seamlessly embedded in developers' normal workflow, with the additional benefits of greatly improving the credibility and reliability of our approach.

3.6 Commits Submission

The last stage is to submit the groups as a list of consecutive commits appended in the version history. When the developer is ready to commit, they should select part or all of the generated groups, complete the corresponding messages, and submit them with one single click. In this stage, *SmartCommit* will perform two steps:

- *Correctly reorder the groups*: by default, diff hunks with directed dependency are kept in the same group, to avoid commits with compilation errors. However, this property can be broken by developers unconsciously or on purpose (e.g. for *stacked pull requests*[6]). Therefore, before submission the selected groups are reordered by the topological order of their vertices in the diff hunk graph, to make sure the diff hunk that is depended on by others is committed before or with its dependents. If this criterion cannot be satisfied, an error will be reported for further fine-tuning.
- *Incrementally apply the changes*: since all diff hunks are computed upon the original *HEAD* commit (the last commit before all submission), but it moves forward as groups are committed, so the changes cannot be applied directly. We solve this problem with *incremental patch*, which includes changes in the to-be-submitted group and all changes in the previously-submitted groups, so as to be applied on the original *HEAD* version to get a correct workspace for submission.

[6]https://unhashable.com/stacked-pull-requests-keeping-github-diffs-small/

4 EVALUATION

We implement *SmartCommit* as an open-source tool (see Figure 5), which works either as a standalone software, a plugin in *IntelliJ IDEA*, or as a customized Git sub-command *git sc*. The source code (with about 13,000 lines of Java code) is available on GitHub.[7] In the implementation, we extract the hierarchical and dependency relations with *Eclipse JDTParser*[8] and symbol resolver, and detect refactoring with the state-of-the-art tool *RefactoringMiner* [29]. We adopt parallelization and cache diff hunk graph in memory to quickly respond to coarse control. *SmartCommit* takes the working directory of a Git repository as input, and presents the suggested decomposition as a list of groups, which can be adjusted in two ways: (1) change the threshold with the sliding bar on the left side to coarsely change the granularity, and (2) drag&drop specific diff hunks for fine-tuning. When ready to commit, the user can select one or several groups (with the check marks in green background), complete the commit message, and then click the *Commit* button at the top right corner to trigger the submission.

4.1 Research Questions

The goal of our approach is to quickly reach a good initial decomposition which the developer then can refine without too much effort. Hence, we evaluate accuracy of the initial decomposition, interactivity, performance and usefulness with 4 research questions:

(1) **RQ1 (Accuracy)**: What is the accuracy of initial decomposition suggested by *SmartCommit*?
(2) **RQ2 (Interactivity)**: Are the manual effort to adjust the suggested decomposition acceptable for developers?
(3) **RQ3 (Performance)**: What about the performance and how does it change with different input sizes?
(4) **RQ4 (Usefulness)**: Is *SmartCommit* useful for developers to follow the best practice?

4.2 Research Design

We conducted the evaluation with mixed methods research [5, 30], and *triangulate* results from two complementary studies conducted in parallel: (1) a field study in which we observed 83 software engineers in an international IT company use *SmartCommit* for 9 months and interviewed a subset of them, and (2) an experiment (without human subjects) in a controlled setting of 3,000 synthetic composite commits from 10 well-known open-source projects from

[7]https://github.com/Symbolk/SmartCommitCore
[8]https://www.eclipse.org/jdt/

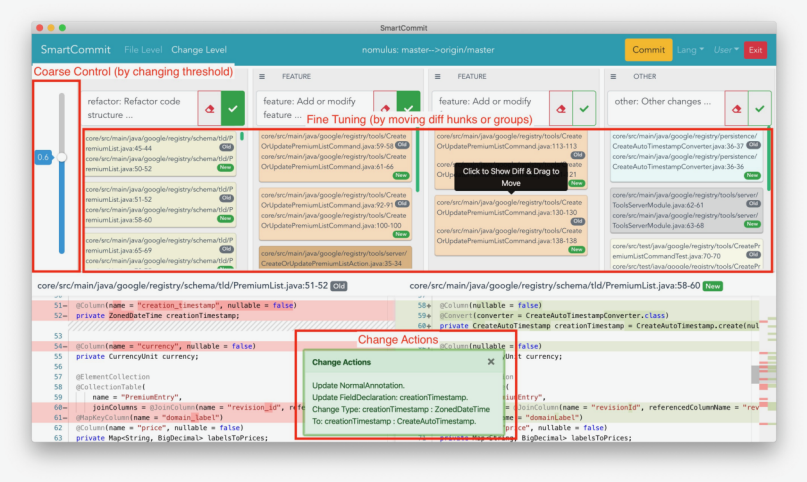


Figure 5: A screenshot of the GUI of *SmartCommit*.

various domains. By mixing both qualitative and quantitative evaluations in the field and in controlled settings, we triangulate between evaluations in different settings and environments to gain more confidence in the conclusion [30].

In particular, the two studies exhibit the traditional *tradeoff* between internal and external validity [25]. The field study was performed within the realism of an industrial setting when developers perform their daily tasks – but this realism (supporting external validity) comes at the cost of limited control (threatening internal validity). In contrast, the open-source experiment was highly controlled and allows us to evaluate the approach under various experimental conditions on an extensive dataset (supporting internal validity) – but these tangled changesets were generated *synthetically* to provide a reliable ground truth (threatening external validity). Taken together, triangulating results from the two studies increases confidence that our conclusions are not simply accidental due to a weakness in one specific study design.

4.2.1 Industrial Field Study. We conducted our field study in a large international IT company from 2020-05-18 to 2021-01-23. A total of 83 experienced software engineers from two product teams were involved: the first team included 32 developers working on a music application for Android (*Project a*), and the second team comprised 51 developers working on a general cloud computing platform (*Project b*). All participants have more than 2 years of programming experience with Java, and write code in *IntelliJ IDEA*. Correspondingly, we deployed *SmartCommit* as a plugin. We conducted a quick-start training course through an online conference about the best practice of activity-oriented commits and the usage instructions of *SmartCommit*.

Although we initially recommended these engineers to use *SmartCommit* for every commit, we did not intervene in their daily work; they were free to choose any other existing tools to submit commits (like Git commands or the built-in Git support in *IDEA*). If they chose to use *SmartCommit*, this tool would analyze changes in the local workspace, and open a web-based GUI (similar to Figure 5) when the analysis was done. From startup to exit, *SmartCommit* collected the following telemetry data for further analysis: (1) the meta information, e.g. timestamp, user id, repository name; (2) the change information, e.g. number of changed files and lines of code; (3) the decomposition information, i.e. the initial suggestion and the final commits; and (4) interactive operations.

After the end of the field study, we conducted a focus group session [17] to collect comments and feedback from users. Specifically, we recruited the top 5 active users by usage frequency from each team (active users), and 5 users who just tried a few (less than 5) times (inactive users, 3 from *Project a* and 2 from *Project b*). The online meeting with all 15 participants was started by one of the researchers with warm-up questions about the pros and cons of *SmartCommit*, which laid the foundation for further discussions. During the discussion, another researcher was responsible for recording the responses and discussions. After the meeting, researchers worked together to analyze the collected information using inductive thematic analysis [6].

4.2.2 Controlled Open-Source Experiment. The main challenge of a controlled study that evaluates the accuracy of *SmartCommit* is to establish the *ground truth* decomposition for a large set of diverse composite commits. Unfortunately, we are not aware of any existing validated dataset, and it will also open doors for many forms of bias if asking experienced developers to decompose commits or contacting commit authors for the ground truth.[9] Therefore, we adopted an approach that synthesizes composite commits from existing individual commits — in this way these individual commits provide the *ground truth* decomposition.

The key idea, inspired by past research [8, 36], is to artificially simulate a composite commit by merging multiple consecutive non-composite commits of the same author from the version history. That is, we merged code where a developer actually split work

[9]We did try the latter approach, but discarded it for two reasons: (1) given a commit, determining whether it is composite is challenging, especially for project outsiders [8]; (2) due to the complexity of studied projects, manually decomposing suspected composite commits could bias the ground truth, especially when the changeset is large. We also built an online platform and contacted authors of suspected composite commits to validate the decomposition, but the response rate was too low for rigorous analysis.

Table 1: Studied Projects

Project	Field	#Stars	#Total	#Atomic
rocketmq	Messaging and streaming	14.1k	1,144	459
nomulus	Domain name registry	1.4k	3,034	572
glide	Image loading library	29k	2,383	650
antlr4	Parser generator	7.7k	5,888	1,117
storm	Realtime computation	6.1k	10,287	1,444
realm-java	Mobile database	10.9k	5,465	1,614
netty	Web application	23.5k	9,688	2,786
cassandra	Partitioned row store	6.1k	25,210	3,459
deeplearning4j	Deep learning	11.8K	24,119	4,699
elasticsearch	Search engine	48.7k	47,564	6,728

The last three columns represent the number of stargazers (#Stars), the number of all commits (#Total) and atomic commits (#Atomic), collected on the master, main, or default branch on GitHub.

into multiple commits to simulate a setting where they did not. Specifically, we generated 3,000 *synthetic composite commits* from 10 open-source projects, by merging 2, 3, or 5 consecutive non-composite commits. We picked these numbers to mirror the normal size and number of activities observed in real composite commits, and these numbers are also consistent with finally-submitted groups in our later statistics for the field study results. Then we decomposed the tangled changeset in each synthetic composite commit with *SmartCommit*, and compared suggested decomposition with original commits to measure the accuracy as well as the *distance* in terms of the minimal number of *fine-grained* operations to get from the suggested decomposition to the ground truth.

Select studied projects. To select representative and diverse sample projects, we first searched for projects with Java as the main language on GitHub, and ranked them by the number of stargazers. Then we chose projects that are popular (with at least 1,000 stargazers), mature (with at least 1,000 commits) and well-known in their domains. The selected projects are listed in Table 1.

Collect atomic commits. To generate synthetic composite commits, we first needed to identify atomic/non-composite commits as inputs. According to common characteristics of composite commits from previous works [8, 23, 28] and our manual validation on 100 sample commits from studied projects, we considered a commit as non-composite if its commit message did not satisfy any of the following criteria:

- references multiple issue ids (e.g. *Fix #X and #Y...*),
- contains a bullet list of performed actions (e.g. *- Fixed ... - Changed ...*),
- contains multiple phrases/sentences with verbs (e.g. *Rename ... Removed ...*), or
- contains parallel sentences starting with conjunction keywords like *and, also, plus* (e.g. *Refactor ... Also ... And ...*).

Synthesize composite commits. For each project, we randomly selected sequences of *consecutive* non-composite commits from the same author. For each of the 10 projects, we selected 100 sequences of length 2, 100 sequences of length 3, and 100 sequences of length 5, getting a total of 3,000 sequences. For each sequence, we generated one composite commit by merging all changes, and recorded the original commits and their order as the ground truth. When there were overlapping changes between merged commits, we only kept the last change on the same lines.

4.2.3 Threats to Validity. As discussed, the field study and controlled experiment have different limitations regarding internal and external validity, but triangulation helps us to interpret findings despite these limitations. The field study was conducted in a realistic setting, but with less control and limited scope of projects and teams. For the controlled study, threats to validity lie in the process of simulating composite commits to establish ground truth.

Nonetheless, it is worth acknowledging that, despite best efforts in our study design (e.g., long-period observation in the field study), common experimental biases (like the novelty effect of new tools and the response bias in interviews) may still affect our results in the field study. Also in our controlled experiment, despite best efforts of simulating realistic merged commits (e.g., mirroring real composite commits in size and using a large dataset to reduce the impact of random noise), the synthetic commits are only a proxy for realistic commits. Readers should interpret results, especially results that are not consistent across both studies, carefully in the context of these limitations.

4.3 Evaluation Results

We summarize and triangulate results from the two studies in Table 2. Details about the evaluation metrics, process and results for each research question are presented as follows. More detailed quantitative data and qualitative results are publicly available in our *supplementary data repository* [2].

4.3.1 Evaluation of Accuracy (RQ1). We evaluate the accuracy of the initial suggested decomposition of *SmartCommit*, by comparing it with the ground truth, as well as three baselines.

Metric. In both studies, we adopted *Rand Index* [15], a commonly-used measure of the similarity between two data clusterings, to measure the *accuracy* of the suggested decomposition.

Definition 4.1 (Accuracy). Given a set of *n diff hunks* H, the *ground truth* decomposition D_g (a *partition* to H), and an decomposition suggested by an approach D_s (also a *partition* to H), the accuracy of D_s relative to D_g is defined as follows:

$$acc(D_s|D_g) \doteq \frac{|Pair(D_s) \cap Pair(D_g)|}{|Pair(D_g)|} = \frac{|Pair(D_s) \cap Pair(D_g)|}{n(n-1)/2}$$

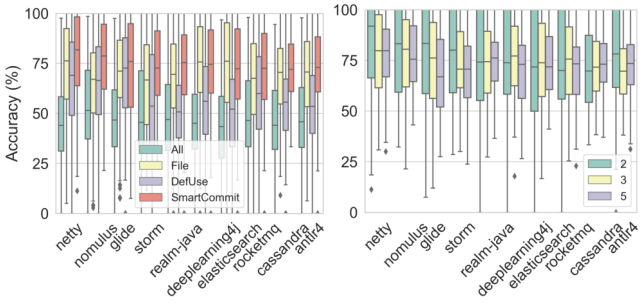
where, $Pair(D)$ is the set of element pairs in partition D, which is defined as $\doteq \{(h_i, h_j, 1) | h_i, h_j \in H, i < j, \exists S \in D \cdot \{h_i, h_j\} \subseteq S\} \cup \{(h_i, h_j, 0) | h_i, h_j \in H, i < j, \nexists S \in D \cdot \{h_i, h_j\} \subseteq S\}$.

In the field study, we regard the decomposition submitted by developers as the ground truth. In the controlled experiment, we derived the ground truth from the original commits. Furthermore, in the controlled experiment, we also compare the accuracy of our approach with three baselines: (1) putting all changes into one group (denoted as *All*); (2) putting changes in each file into one group (denoted as *File*); and (3) considering only *def-use*, *use-use* and *same-enclosing-method* relations (denoted as *DefUse*, corresponding to the strategy used in ClusterChanges [1]).

Results. In the field study, *SmartCommit* achieves the median accuracy of 74.70% for *Project a* and 70.45% for *Project b*. In over 60%

Table 2: Triangulation of 2 Studies to Answer 4 Research Questions

Research Question	Results from Field Study	Results from Controlled Experiment
RQ1: Accuracy	*SmartCommit* achieves median accuracy of 74.70% and 70.45% for 2 industrial projects, and over 60% of generate groups are accepted without adjustment. All interviewed long-term users agreed that it could suggest reasonable decomposition.	*SmartCommit* achieves median accuracy between 71.00–83.50% for the 3,000 synthetic composite commits from 10 projects, and outperforms the state-of-the-art baseline.
RQ2: Interactivity	Users frequently need adjustments, but usually fewer than 5. Coarse control is important, as it greatly reduces the manual effort for fine tuning.	Without human intervention and coarse control, typically 1-15 reassigns are needed, 1-3 reorders are needed.
RQ3: Performance	Most interviewees consider the performance as acceptable, and seldom complained or abandoned *SmartCommit* for run time.	Computation time is under 5 seconds for 90% composite commits, and does not change drastically as input size increases.
RQ4: Usefulness	Interviewees reported that *SmartCommit* was particularly useful when they realized there were interleaved works done. All interviewed active and long-term users agreed that it effectively improved their *commit style*.	(The controlled experiment can not provide *direct* insights to usefulness, since it does not involve human subjects.)



(a) Accuracy with Baselines **(b) Accuracy of *SmartCommit***

Figure 6: (a) Accuracy comparison of different approaches, each bar aggregates 300 composite commits, (b) accuracy distribution of *SmartCommit* with the number of merged commits, each bar aggregates 100 composite commits.

invocations, the initial suggested decomposition was committed with no adjustments (see Figure 7a). In over 95% invocations, the number of manually-committed groups is equal or less than 3, but we find that the accuracy for changesets from 4 or more activities is similar. Interviewees also reported that in most cases, *SmartCommit* could generate decompositions close to their expectations.

In the controlled experiment, *SmartCommit* achieves a median accuracy of 71.00–83.50% and outperforms the three baselines in most projects, see Figure 6a for details. Note that although *File* (group changes by file) achieves higher accuracy in some cases, it generates far more groups than *SmartCommit* and the ground truth, which would require more adjustment effort in practice. As shown in Figure 6b, *SmarCommit* performs best for composites commits synthesized with 2 commits, and the accuracy reduces for 3 and 5 as expected but not much (around 75%).

Answer to RQ1: *While not perfect, results from both field study and controlled experiment indicate that SmartCommit can provide a good initial suggestion for decomposing tangled changesets.*

4.3.2 Evaluation of Interactivity (RQ2). We observe and evaluate how many interactions are commonly needed to adjust the suggested decomposition.

Metric. In the field study, we logged the number of *coarse control* and *fine tuning* operations performed by developers in each submission. In the controlled experiment, since no humans were involved, we computed the *minimum* number of *fine tuning* adjustments required to change the initial suggested decomposition D_s to the ground truth D_g. To this end, we measured the edit distance between D_s and D_g, by considering two kinds of edit operation: (1) *reassign* to move a *diff hunk* to another group, and (2) *reorder* to move a group to change the order. The shortest *edits distance* between D_s and D_g is computed in two steps: (1) find a mapping between groups in D_s and D_g with *maximum weight bipartite matching*, in which D_s and D_g are treated as the two vertex-subsets of a bipartite graph respectively, and the weight between two groups is the number of their common elements; (2) compute the number of wrongly-assigned diff hunks as the number of *reassign* operations, and the moving distance between the two lists of groups as the number of *reorder* operations.

Results. In the field study, users only needed less than 3 coarse control operations in over 95% cases, and less than 5 fine tuning operations in over 90% cases (see Figure 7a). We observed from the data that in some cases with large changesets, users often tried the coarse control several times to find the most appropriate decomposition. In the focus group session, interviewees also agreed that the interaction mechanism is necessary and acceptable for them, especially the coarse control. Two interviewees used the coarse control in a novel way: they first submitted those suggested groups that met their expectation, and then decomposed the remaining changeset with *coarse control* to get a new decomposition, upon which they handled overwhelming changes and groups iteratively.

In the controlled experiment, without developer involvement, we see a little larger number of *reassign* operations (see Figure 7b). The reason is that the decomposition process purely based on source code tends to generate more groups than actually desired, mirroring findings from previous work [22]. However, even for changesets

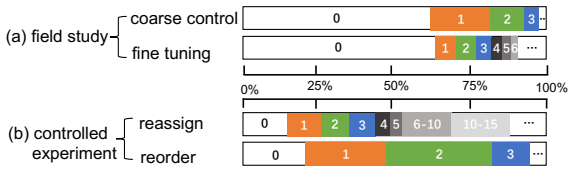


Figure 7: Distribution of adjustments in the field study (a) and the controlled experiment (b) (ignoring those under 2%).

with dozens or hundreds of diff hunks, the algorithm still keeps the number of reassign operations relatively small, and thus provides an acceptable start for later manual adjustment. In over 95% cases, the number of *reorder* adjustment is less than 3.

Answer to RQ2: *Adjustments are commonly needed for activity-oriented decomposition, but the manual effort is kept acceptable through the interaction mechanism of SmartCommit.*

4.3.3 Evaluation of Performance (RQ3). We evaluate the run time of *SmartCommit* with inputs of different sizes, since performance matters for user acceptance and experience.

Metric. We evaluated the performance qualitatively in the field study, and quantitatively in the controlled experiment. For the qualitative part, we report results from the focus group session and informal user feedback during the field study. For the quantitative part, for the 3,000 changesets of different sizes, we recorded the *elapsed time* in miliseconds from the start of *SmartCommit* to the output of the suggested decomposition. We analyze the performance with regard to two factors of the input size: the number of diff hunks (#diff_hunks), and the number of changed lines (#changed_lines, including both removed and added ones).

Results. In the field study and the focus group session, most users thought that the run time is usually acceptable, only two users complained about too long run time of *SmartCommit* (over 10 seconds in their cases, one with many entire-file changes, and one with many resolved conflicts after a huge merge), which led them to terminate it. These observations suggest that the typical performance of *SmartCommit* is acceptable for daily practical use, but could benefit from optimizations for corner cases.

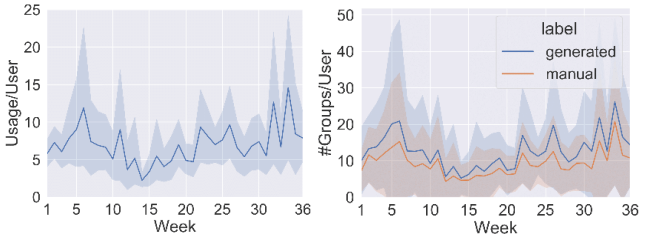
In the controlled experiment, over 90% changesets can be processed within 5 seconds, and the analysis time slightly increases with the growth of *#diff_hunks* and *#changed_lines* (the detailed distribution is available in the *supplementary data repository* [2]). The worst-cased overhead occurred with several extra-large changesets that consist of more than 500 *diff hunks* and 8,000 changed lines, where the run time exceeds 10 seconds. However, we found in the field study that most real changesets in the workspace are much smaller in size than these, so we view the extra-large samples as a stress test for our implementation.

Answer to RQ3: *SmartCommit shows acceptable performance in practical use, and keeps run time stably under 5 seconds for a wide range of input sizes in the controlled experiment.*

4.3.4 Evaluation of Usefulness (RQ4). We focus on the field study to analyze the usefulness of *SmartCommit* in practice.

Table 3: Usage Data Statistics (2020-05-18 - 2021-01-23)

Scope		Effect	
#invocations	2,632	#generated_groups	4,838
#diff_files	8,763	#committed_groups	3,552
#diff_hunks	31,086	#unchanged_groups	2,262
#diff_lines	422,128	#avg_adjustments	3.41



(a) Usage Frequency **(b) Diff Hunk Groups**

Figure 8: (a) The trend of usage, (b) the distribution of generated and manually-committed groups, both aggregated by week. Shaded area represents the standard deviation.

Quantitative analysis. We examine the *scope* and *effect* of *SmartCommit*. *Scope* is characterized by the number of usages and the size of processed changesets, while *Effect* is characterized with the generated groups and manually-committed groups.

During the study period from 2020-05-18 to 2021-01-23 (see Table 3), 83 developers invoked *SmartCommit* for 2,632 times (#invocations), with more than 422K lines of changed code processed. In total, the tool generated 4,838 (#generated_groups) groups as the initial suggestion, and the developer submitted 3,552 commits (#committed_groups), in which 2,262 (63.68%) (#unchanged_groups) were exactly the same as the initial suggestion. In over 60% invocations, there was no adjustments performed (see Figure 7); when adjustments were required, the average number of adjustments is 3.41 (including both coarse control and fine tuning).

Over time, the number of users per week stabilized within range [10, 15] after the 5th week. About a half of developers (40/83) used *SmartCommit* in at least 2 weeks, and 13 of them were particularly active and used it in at least 10 weeks. According to Figure 8a, the overall usage frequency increased in the first 5 weeks, but dropped between the 5th and 13th week; later on, the usage frequency of active users increased gradually with time, suggesting that for these users *SmartCommit* was considered to be useful and helpful. As a result, the distribution shows a long-tail phenomenon: 40 users invoked it 5–551 times (the most active one used the tool for 551 times in 32 weeks), but the others used it less than 5 times, mainly because their work has less need for large-scale changes (see qualitative analysis).

SmartCommit tended to generate more groups than the user expectation *initially* (see Figure 8b), and the extra groups were mostly small (with a few diff hunks), which were either moved to the other groups or left uncommitted. This is reasonable, because not all valuable links between diff hunks can be retrieved from source code, so the automatic decomposition tends to be more fine-grained than activities [22]; this also explains why the interaction is necessary in this problem. Besides, interviewees responded that

when the tool suggested more than 4 groups (although in less than 5% cases), coarse control is especially useful for them to rapidly find the appropriate granularity.

Qualitative analysis. After the focus group with 10 active users and 5 inactive users, we summarize the collected insights for future improvements as follows:

- All interviewed active users agreed that *SmartCommit* could suggest reasonable decomposition, and effectively improved their *commit style* by making them concentrate on and double-check code changes before submission. For example, users commented that "Basically the given grouping makes sense.", "I am getting used to checking my change size consciously before committing."
- Eight of interviewed active users reported that *SmartCommit* was particularly useful when they realized there were interleaved works done. But when the change was small and trivial (e.g. small bug fix towards an issue, or non-code changes), especially for interviewed inactive users, developers tended to prefer plain Git commands instead of any other clients or tools, e.g., "Comparing with Git, I did not find [it] necessary since my project is stable and issue-driven."
- Three of interviewed active users mentioned that *SmartCommit* additionally helped them to avoid accidentally submitting unwanted or dangerous changes, such as local/personal configurations, resources, personal and sensitive information/data/secrets. For example, one user appreciated that "The tool can often group my local test config in one group, which will pollute the server config if committed."
- Most interviewees (12/15) expected to use *SmartCommit* as the main committing tool if it could provide a more complete workflow by supporting other Git functions (like pushing and creating merge request after committing). Besides, supporting other languages and suggesting commit messages were also highly wanted, e.g., "I find that it becomes easier for me to describe my changes for more compact commits, so will it also be easier to apply some automatic commit message generators?"

Answer to RQ4: *SmartCommit is considered and proved to be useful in assisting developers to submit activity-oriented commits.*

5 RELATED WORK

Several approaches have been proposed to address the change decomposition problem in the literature. Herzig and Zeller [8] used a heuristic-based algorithm to untangle changes, in which they considered their relations as *confidence voters*. Similarly, Wang et al. [32] adopted a heuristic method based on code dependency analysis and similar code detection to decompose composite commits for code review. Barnett et al. [1] proposed *CLUSTERCHANGES*, which relies on the *def-use* and *use-use* links to cluster changes. Guo and Song [7] presented *CHGCUTTER*, which generates related change subsets with the program dependence. Muylaert and De Roover [22] used program slicing technique on program dependence graphs to decompose tangled changesets. Dias et al. [4] developed *EpiceaUntangler* to untangle changes by predicting the probability of two changes belonging to one cluster, based on code structure and recorded IDE events when developers write code in SmallTalk. A few works suggest the necessity of developer involvement in the decomposition process. Taeumel et al. [27] proposed *Thresher*, an interactive tool to group changes in Squeak/Smalltalk, based on user-implemented scripts. Recently, Yamashita et al. [34] introduced *ChangeBeadsThreader*, an environment that untangles changes based on four metrics and allows the user to split/merge clusters totally by hand.

Our approach differs from these works in three aspects. (1) *Target scenario*: while most previous works decompose composite commits temporarily for code review, our approach focuses on preventing the occurrence of composite commits before submission. Our target scenario is more challenging because the generated change groups must align with high-level development activities. (2) *Technique*: comparing with previous works, we propose an extensible graph representation that uniformly covers common heuristic links among code changes, and design a graph-partition-based algorithm for changeset decomposition correspondingly. (3) *Interaction*: comparing to other interactive approaches, we provide a more intuitive and efficient interaction mechanism that greatly saves manual effort for adjustments. In this way, we integrate human knowledge into the algorithm without bringing extra burden to developers. Furthermore, we have conducted a much more solid evaluation of our approach than any other previous works, and have the tool undergone a rather long-time practical use.

6 CONCLUSION AND FUTURE WORK

In this paper, we present *SmartCommit*, an interactive approach to help developers follow the best practice of activity-oriented commits. Given a changeset as input, *SmartCommit* generates suggested decomposition and co-work with the developer to refine it through interactions. We evaluate the effectiveness and scalability of *SmartCommit* under both industrial and open-source settings. The results show that: (1) when decomposing synthetic composite commits from open-source projects, *SmartCommit* achieves a median accuracy of 71.00–83.50% with run time mostly under 5 seconds; (2) when used in real industrial developments, *SmartCommit* manifested its capability to facilitate the adoption of the best practice of activity-oriented commits. Following the current work, we plan to extend more types of links for the diff hunk graph to further improve the accuracy, and extend *SmartCommit* by integrating state-of-the-art *commit message generation and suggestion* works [10, 18, 19, 33].

In the long term, we aim to construct an artificial collective intelligence (ACI) system [35] for collaborative software development, a system that enables a continuously executing loop about *information exploration*, *integration*, and *feedback*. The approach *SmartCommit* presented in this paper is about organizing the result of a developer's information exploration in a cohesive way, and our previous work *IntelliMerge* [24] focuses on integrating information from different developers into a consistent code version.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China under Grant Nos. 61690200 and 61620106007.

REFERENCES

[1] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 134–144. https://doi.org/10.1109/ICSE.2015.35

[2] BoShen. 2020. Supplementary data repository. https://github.com/Symbolk/SmartCommitEvaluation-Viz.

[3] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. 2019. The effects of change decomposition on code review—a controlled experiment. *PeerJ Computer Science* 5 (may 2019), e193. https://doi.org/10.7717/peerj-cs.193

[4] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 341–350. https://doi.org/10.1109/saner.2015.7081844

[5] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*. Springer London, 285–311. https://doi.org/10.1007/978-1-84800-044-5_11

[6] Douglas Ezzy. 2013. *Qualitative Analysis*. Routledge. https://doi.org/10.4324/9781315015484

[7] Bo Guo and Myoungkyu Song. 2017. Interactively decomposing composite changes to support code review and regression testing. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 118–127. https://doi.org/10.1109/compsac.2017.153

[8] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 121–130. https://doi.org/10.1109/msr.2013.6624018

[9] Karen Holtzblatt and Hugh Beyer. 1998. Contextual design. In *CHI 98 Conference Summary on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/286498.286629

[10] Shuyao Jiang. 2019. Boosting Neural Commit Message Generation with Code Semantic Analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1280–1282. https://doi.org/10.1109/ase.2019.00162

[11] George Karypis and Vipin Kumar. 1998. Multilevel algorithms for multi-constraint graph partitioning. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 28–28. https://doi.org/10.1109/sc.1998.10018

[12] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! are you committing tangled changes?. In *Proceedings of the 22nd International Conference on Program Comprehension*. 262–265. https://doi.org/10.1145/2597008.2597798

[13] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2016. Splitting Commits via Past Code Changes. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 129–136. https://doi.org/10.1109/apsec.2016.028

[14] Martin Konopka and Pavol Návrat. 2015. Untangling development tasks with software developer's activity. *CSD@ICSE* (2015), 13–14. https://doi.org/10.1109/csd.2015.10

[15] Abba M. Krieger. 1999. A Generalized Rand-Index Method for Consensus Clustering of Separate Partitions of the Same Data Base. *Journal of Classification* 16, 1 (jan 1999), 63–89. https://doi.org/10.1007/s003579900043

[16] Haiming Li, Qiyang Xia, and Yong Wang. 2017. Research and Improvement of Kruskal Algorithm. *Journal of Computer and Communications* 05, 12 (2017), 63–69. https://doi.org/10.4236/jcc.2017.512007

[17] Pranee Liamputtong. 2011. *Focus group methodology: Principle and practice*. Sage Publications.

[18] Qin Liu, Zihe Liu, Hongming Zhu, Hongfei Fan, Bowen Du, and Yu Qian. 2019. Generating commit messages from diffs using pointer-generator network. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 299–309. https://doi.org/10.1109/msr.2019.00056

[19] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic Generation of Pull Request Descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 176–188. https://doi.org/10.1109/ase.2019.00026

[20] Emerson Murphy-Hill and Andrew P. Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software* 25, 5 (sep 2008), 38–44. https://doi.org/10.1109/ms.2008.123

[21] R. Emerson Murphy-Hill, Chris Parnin, and P. Andrew Black. 2012. How We Refactor, and How We Know It. *Software Engineering, IEEE Transactions* (2012), 5–18. https://doi.org/10.1109/icse.2009.5070529

[22] Ward Muylaert and Coen De Roover. 2018. Untangling Composite Commits Using Program Slicing. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 193–202. https://doi.org/10.1109/scam.2018.00030

[23] Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2013. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 138–147. https://doi.org/10.1109/issre.2013.6698913

[24] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: a refactoring-aware software merging technique. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 170:1–170:28. https://doi.org/10.1145/3360596

[25] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE. https://doi.org/10.1109/icse.2015.24

[26] Sarocha Sothornprapakorn, Shinpei Hayashi, and Motoshi Saeki. 2018. Visualizing a tangled change for supporting its decomposition and commit construction. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 74–79. https://doi.org/10.1109/compsac.2018.00018

[27] Marcel Taeumel, Stephanie Platz, Bastian Steinert, Robert Hirschfeld, and Hidehiko Masuhara. 2017. Unravel Programming Sessions with THRESHER: Identifying Coherent and Complete Sets of Fine-granular Source Code Changes. *Computer Software* (2017). https://doi.org/10.11185/imt.12.24

[28] Yida Tao and Sunghun Kim. 2015. Partitioning composite code changes to facilitate code review. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 180–190. https://doi.org/10.1109/msr.2015.24

[29] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 483–494. https://doi.org/10.1145/3180155.3180206

[30] Alison Twycross. 2004. Research design: qualitative, quantitative and mixed methods approachesResearch design: qualitative, quantitative and mixed methods approaches Creswell John W Sage 320 £29 0761924426 0761924426. *Nurse Researcher* 12, 1 (sep 2004), 82–83. https://doi.org/10.7748/nr.12.1.82.s2

[31] Laszlo Vidacs and Martin Pinzger. 2018. Co-evolution analysis of production and test code by learning association rules of changes. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 31–36. https://doi.org/10.1109/maltesque.2018.8368456

[32] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2019. CoRA: Decomposing and Describing Tangled Code Changes for Reviewer. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1050–1061. https://doi.org/10.1109/ase.2019.00101

[33] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. International Joint Conferences on Artificial Intelligence Organization*, Vol. 7. 3975–3981. https://doi.org/10.24963/ijcai.2019/552

[34] Satoshi Yamashita, Shinpei Hayashi, and Motoshi Saeki. 2020. ChangeBeadsThreader: An Interactive Environment for Tailoring Automatically Untangled Changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 657–661. https://doi.org/10.1109/saner48275.2020.9054861

[35] Wei Zhang and Hong Mei. 2020. A constructive model for collective intelligence. *National Science Review* 7, 8 (05 2020), 1273–1277. https://doi.org/10.1093/nsr/nwaa092

[36] Shurui Zhou, Stefan Stanciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wasowski, and Christian Kästner. 2018. Identifying features in forks. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 105–116. https://doi.org/10.1145/3180155.3180205