Untangling Composite Commits Using Program Slicing

Ward Muylaert, Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
{ward.muylaert;coen.de.roover}@vub.be

Abstract—**Composite commits are a common mistake in the use of version control software. A composite commit groups many unrelated tasks, rendering the commit difficult for developers to understand, revert, or integrate and for empirical researchers to analyse. We propose an algorithmic foundation for tool support to identify such composite commits. Our algorithm computes both a program dependence graph and the changes to the abstract syntax tree for the files that have been changed in a commit. Our algorithm then groups these fine-grained changes according to the slices through the dependence graph they belong to. To evaluate our technique, we analyse and refine an established dataset of Java commits, the results of which we also make available. We find that our algorithm can determine whether or not a commit is composite. For the majority of commits, this analysis takes but a few seconds. The parts of a commit that our algorithm identifies do not map directly to the commit's tasks. The parts tend to be smaller, but stay within their respective tasks.**

I. Introduction

Version control systems (VCS) are widely used to manage the history of code bases. Prominent examples include Git, SVN, and Mercurial. A developer may "save" their changes into units called commits. Best practice suggests each commit should only contain changes related to one task. Such commits are called single-task or atomic commits [1, 2]. In this manner, the VCS can be used to keep track of how the program under development evolves. VCS also support, for example, reverting individual changes or porting changes to other versions of the code base. On the research side, VCS provide a trove of software evolution information open to analysis.

However, developers do not necessarily follow the best practice of creating only single-task commits [3]. For example, a small bug may end up being fixed while work is underway on another feature. The bug fix and the new feature then end up in the same commit. Floss refactoring is another problem: refactoring in order to prepare an implementation for a new feature, after which all changes are committed together [4]. These situations result in composite commits: larger commits that combine many unrelated changes.

Composite commits occur on a regular basis. A study by Herzig et al. found that up to 15% of Java bug fixes contain multiple unrelated changes [5]. Tao and Kim found that between 17% and 29% of investigated revisions were composite [2]. Nguyen et al. found that 11% to 39% of all the fixing commits used for mining archives were composite [6].

Composite commits may cause several problems. We provide four examples. Individual changes are more difficult to revert if they are a part of a larger commit. Changes are also more difficult to integrate if they are part of a composite commit with other unrelated changes. A code reviewer will have a harder time understanding larger commits of unrelated changes [7]. This in turn will lead to lower quality feedback [8]. A researcher analysing commit data, finally, may need to decide on the "one" motivation for a commit even though many comprise various unrelated changes.

The chances of developers abandoning composite commits by themselves are slim. Tool support is required to identify commits as composite and to decompose them into single-task commits. The first type of tool suffices to warn developers that are about to commit unrelated changes. The second type of tool is also of use to researchers analysing the individual tasks commits are composed of.

We propose program slicing as a foundation for such tool support. Program slicing is a program analysis that answers questions about the influence of program statements on other program statements [9, 10]. We extend this idea: our foundation for tool support applies program slicing to source code changes. We hypothesise that *related changes affect source code from the same program slice* and thus that *a commit may be decomposed into related changes using the created program slices*. Intuitively, this states that changes that belong together also have control or data dependencies on one another.

Tao and Kim [2] propose a related approach that also incorporates program slicing. Their approach slices line-based changes. Our approach slices changes to the abstract syntax tree (AST) and is therefore more fine-grained. Our AST-level changes are computed through a change distiller, a program that accepts two file versions and distils what changes were made to go from one version to the other. The approach taken by Tao and Kim also incorporates various other techniques unrelated to slicing. We focus solely on slicing in this work, to assess its value in isolation.

To analyse our approach, we make use of a dataset of commits stemming from five Java projects, gathered by Herzig and Zeller [11]. We first analyse this dataset and refine it further to fit the context of this work. We make the results of this refinement available via https://soft.vub.ac.be/~wmuylaer/publications.

Our results indicate that slicing on changes to the abstract syntax tree largely meets the stated goals. Our technique is able to categorize commits as single-task or composite. In identifying the individual tasks our technique at times produces finer-grained results. That is to say, it may identify several different parts that should actually belong together as one task. This is still better than the alternative in which the technique considers different tasks as one big part. As such, the results of our technique can still prove useful for code reviewing, reverting, or integrating.

Specifically, our paper has the following contributions:

1) A technique to slice around changes to an abstract syntax tree.
2) The application of this technique to decide whether a commit handles a single task.
3) The application of this technique to identify different parts of a commit.
4) An evaluation of our approach.

This paper is structured as follows. We detail the components of our technique in Section II, and the dataset used in its evaluation in Section III. The evaluation method and its results are presented in Section IV. Finally, Section V discusses related work.

II. Overview of the Approach

We want our technique to take as input a commit that needs to be analysed. We want our technique to output the clusters of related changes that it considers the commit to comprise. To achieve this, our technique consists of four major parts, as depicted in Figure 1. First, the commit is distilled into fine-grained changes to the program's abstract syntax tree (AST). Second, the system dependence graph (SDG) is created for every file in the commit. Third, for every fine-grained change, our technique slices on it in the system dependence graph. Finally, changes are grouped by means of the slices they belong to. We have implemented our technique for commits to Java programs. The rest of this section provides further detail into each of the four steps. We will use the example shown in Figure 2 as a running example. The example in question is part of composite commit 5e2cdc06[1] of the ArgoUML project, trimmed for the sake of the example.

A. Fine-Grained Change Distilling

In step one, we make use of a change distiller. A change distiller can be used after the fact, when handed "blobs" of changes (in our case: commits). A change distiller will consider these blobs and split them into fine-grained changes following some algorithm. Another option to obtain fine-grained changes would be by means of a change logger. However, this would require the logger to be installed on developers' machines beforehand. While possible within a company, this is not a feasible approach for many researchers.

To distil the changes from a commit, we make use of ChangeNodes [12]. ChangeNodes is an implementation of the ChangeDistiller algorithm [13] which in turn is based on work by Chawathe et al. [14]. The ChangeNodes implementation operates on the abstract syntax tree (AST) of a Java program. The AST in question is created using the Eclipse Java Development Tools (JDT). Given two versions of a program, ChangeNodes performs tree differencing on their ASTs and returns a list of Insert, Update, Move, and Delete operations. One could compare this list of operations to the changes as produced by the `diff` tool. Applying `diff`'s changes on the first version of the program results in the second version of the program. Similarly, applying the list of operations on the AST of the first version of the program, results in the AST of the second version of the program. ChangeNodes thus provides fine-grained changes describing the `diff` style changes in the commit. In our scenario the two versions used as input for ChangeNodes are (1) the version of the program with the changes of the commit under analysis not yet applied, and (2) the version of the program after the commit under analysis is applied.

Figure 3 depicts the output of ChangeNodes for our running example. ChangeNodes computed nine AST-level change operations that have the same effect as the original commit. In our example, they are all of the Update type: the names of the variables are updated. We numbered the distilled changes for future reference.

B. System Dependence Graph

In the second step, our technique uses program dependence graphs (PDGs). Program dependence graphs contain both control and data flow dependencies as dependence edges. For the sake of the explanation of the following paragraph, we make the following explicit distinction. A procedure dependence graph is the program dependence graph for one method or procedure. Method calls are not resolved. A system dependence graph is the program dependence graph for a combination of procedure dependence graphs. The method calls are used to link different procedure dependence graphs. Our SDGs are for the entire file in which changes occur.

For the implementation of this second step, we opted to extend the open-source TinyPDG tool [15, 16, 17]. TinyPDG creates a procedure dependence graph of a Java method from an AST provided by the Eclipse Java Development Tools (JDT). This was a convincing point in its selection. It enables our implementation to link results from TinyPDG back to ChangeNodes, as both operate on the same AST. TinyPDG does, however, only work on an intra-procedural level. Our implementation therefore renders TinyPDG inter-procedural using the algorithm introduced by Horwitz et al. [18]. Using this algorithm, our extended version of TinyPDG combines the procedure dependence graphs of the different methods into one SDG. Note that our implementation does this on a per file basis.

[1]Full identifier is 5e2cdc061a0572d4007f4bc84382fff80f29e726. Note that this identifier does not match up with what may be found online. At the time of the creation of the dataset (see Section III), not all projects were managed by Git. Herzig and Zeller converted these other repositories to Git themselves, so this identifier only makes sense within their dataset.

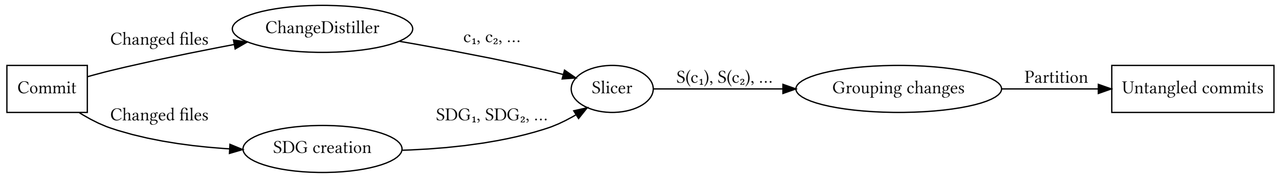

Fig. 1. Overview of our approach. The input is a commit, the output the untangled single-task commits that make up the commit. The four main parts of our approach are each described in detail in Section II.

```
  public FigActionState() {
-   _bigPort = new FigRRect(10 + 1, 10 + 1, 90 - 2,
    ↪ 25 - 2, Color.cyan, Color.cyan);
+   bigPort = new FigRRect(10 + 1, 10 + 1, 90 - 2,
    ↪ 25 - 2, Color.cyan, Color.cyan);
-   _bigPort.setCornerRadius(_bigPort.getHalfHeight
    ↪ ());
+   bigPort.setCornerRadius(bigPort.getHalfHeight())
    ↪ ;
-   _cover = new FigRRect(10, 10, 90, 25, Color.
    ↪ black, Color.white);
+   cover = new FigRRect(10, 10, 90, 25, Color.black
    ↪ , Color.white);
-   _cover.setCornerRadius(_cover.getHalfHeight());
+   cover.setCornerRadius(getHalfHeight());
-   _bigPort.setLineWidth(0);
+   bigPort.setLineWidth(0);
-   addFig(_bigPort);
+   addFig(bigPort);
-   addFig(_cover);
+   addFig(cover);
  }
```

Fig. 2. Difference view for commit 5e2cdc061a0572d4007f4bc84382fff80f29e726 of ArgoUML. Used as running example for Section II.

```
1 Update: _bigPort SimpleProperty[org.eclipse.jdt.core
      ↪ .dom.SimpleName,identifier]
2 Update: _bigPort SimpleProperty[org.eclipse.jdt.core
      ↪ .dom.SimpleName,identifier]
3 Update: _bigPort SimpleProperty[org.eclipse.jdt.core
      ↪ .dom.SimpleName,identifier]
4 Update: _bigPort SimpleProperty[org.eclipse.jdt.core
      ↪ .dom.SimpleName,identifier]
5 Update: _bigPort SimpleProperty[org.eclipse.jdt.core
      ↪ .dom.SimpleName,identifier]
6 Update: _cover SimpleProperty[org.eclipse.jdt.core.
      ↪ dom.SimpleName,identifier]
7 Update: _cover SimpleProperty[org.eclipse.jdt.core.
      ↪ dom.SimpleName,identifier]
8 Update: _cover SimpleProperty[org.eclipse.jdt.core.
      ↪ dom.SimpleName,identifier]
9 Update: _cover SimpleProperty[org.eclipse.jdt.core.
      ↪ dom.SimpleName,identifier]
```

Fig. 3. The distilled changes for commit 5e2cdc061a0572d4007f4bc84382fff80f29e726 of ArgoUML, the running example of Section II. Note that the actual distilled changes also indicate where in the abstract syntax tree they are supposed to be inserted, updated, moved, or deleted.

The entire system dependence graph created for the running example is too large to include here. Instead we show an extract of the relevant parts in Figure 4.

C. Program Slicing

In step three, our technique performs program slicing [9]. The idea behind program slicing is as follows. Given a variable of interest v in a statement s, backwards program slicing on v retrieves the statements that may affect that v in that location. Executing a program reduced to those statements that affect variable v should, in theory, compute the same run-time values for v as if the entire program were executed.

A common static aproach to backwards slicing relies on a program dependence graph. Since the program dependence graph establishes the dependencies for all parts of a program, the information to slice is already present. Specifically, backwards slicing on a node n in a program dependence graph amounts to determining what other nodes n can be reached from.

Our extension to TINYPDG implements the backwards slicing algorithm for system dependence graphs introduced previously by Horwitz et al. [18]. To determine what node to slice on, our technique once more considers the fine-grained change operations provided by CHANGENODES. For a fine-grained change operation c, we identify the original location o_c of the abstract syntax tree node affected by the change. This location is used to find the node n_c in the system dependence graph such that o_c is present in n_c. Slicing on node n_c produces the slice $S(c)$. For ease of notation we will just write $c_i \in S(c_j)$ to indicate the situation in which node n_{c_i} belongs to the slice around node n_{c_j}. Such a slice is computed for every fine-grained change operation.

For the running example, our technique links the change operations described in Figure 3 to the nodes of the system dependence graph in Figure 4. We refer to the nodes in the graph by the number between < and > present in the node. Change operations c_1 through c_5 are linked to nodes $74 \ldots 75$, 76, 76, 80, and 87, respectively. This leads to slices $\{73 \ldots 94, 74 \ldots 75\}$, $\{73 \ldots 94, 74 \ldots 75, 76\}$, $\{73 \ldots 94, 74 \ldots 75, 76\}$, $\{73 \ldots 94, 74 \ldots 75, 80\}$, and $\{73 \ldots 94, 74 \ldots 75, 87\}$. The calculations for change operations c_6 through c_9 happen analogously.

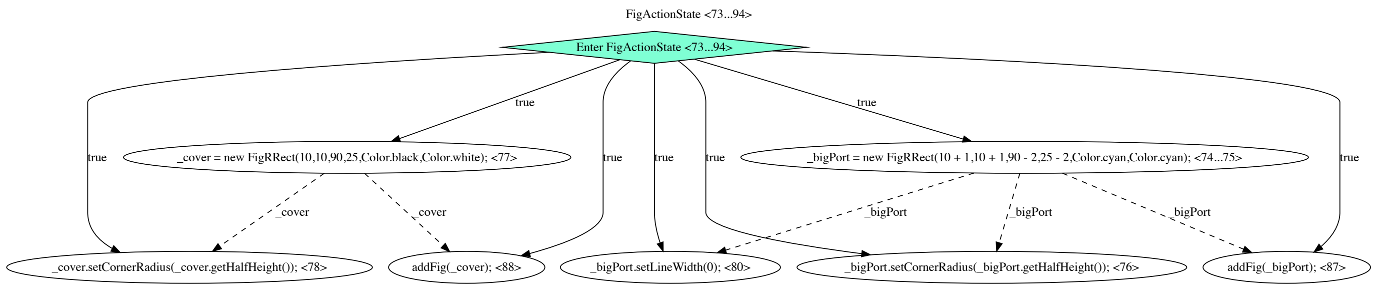
FigActionState <73...94>
Enter FigActionState <73...94>
true
_cover = new FigRRect(10,10,90,25,Color.black,Color.white); <77>
_bigPort = new FigRRect(10 + 1,10 + 1,90 - 2,25 - 2,Color.cyan,Color.cyan); <74...75>
_cover
_bigPort
_cover.setCornerRadius(_cover.getHalfHeight()); <78>
addFig(_cover); <88>
_bigPort.setLineWidth(0); <80>
_bigPort.setCornerRadius(_bigPort.getHalfHeight()); <76>
addFig(_bigPort); <87>

Fig. 4. Relevant parts of the system dependence graph for the running example of Section II. Solid lines with the label `true` are control dependencies. Dashed lines are data dependencies. The label indicates what makes it a data dependency.

D. Change Grouping

Finally, our technique needs to decide which change operations belong together. To do so, we define the equivalence relation $\equiv_S$ between changes. Using the equivalence relation then enables partitioning the set of change operations into disjunct sets.

To define $\equiv_S$, we first define the helper relation $\equiv'_S$.

$$c_i \equiv'_S c_j \iff c_i \in S(c_j) \lor c_j \in S(c_i)$$

In other words, change operations c_i and c_j are related by $\equiv'_S$ if and only if c_i is in the backwards slice on c_j or vice-versa. Note that the relation $\equiv'_S$ is *not* an equivalence relation. By definition of how slicing works, this relation is reflexive. The relation is also clearly symmetric due to its symmetric definition. The relation is however not necessarily transitive. We cannot state that if $c_i \equiv'_S c_j$ and $c_j \equiv'_S c_k$, then $c_i \equiv'_S c_k$. Consider for this the following simplified situation. c_j is part of the root node of a program dependence graph with two children. c_i is part of one of the child nodes. c_k is part of the other child node. Slicing in this situation results in $S(c_i) = \{c_i, c_j\}$, $S(c_j) = \{c_j\}$, and $S(c_k) = \{c_j, c_k\}$. Then $c_i \equiv'_S c_j$ and $c_j \equiv'_S c_k$, but $\neg(c_i \equiv'_S c_k)$. The relation $\equiv'_S$ is thus not an equivalence relation.

We use $\equiv'_S$ to define $\equiv_S$. Specifically, $\equiv_S$ is the transitive closure of $\equiv'_S$:

$$c_i \equiv_S c_j \iff \exists c_{k_1}, \ldots, c_{k_n} : c_i \equiv'_S c_{k_1}, \ldots, c_{k_n} \equiv'_S c_j$$

In other words, there is a relation $\equiv_S$ between two change operations c_i and c_j if there is a chain of change operations, each connected with the next one by relation $\equiv'_S$, that links c_i to c_j.

Algorithmically, our technique uses $\equiv'_S$ to build the partition for $\equiv_S$. The following steps are followed, given the change operations, their slices, and an empty set to hold the partition.

1) If a change operation is not in relation $\equiv'_S$ with any change operation in any of the existing subsets in the partition, create a new subset with that change operation in it.
2) If a change operation is in a relation with an element (or more elements) of exactly one existing subset in the partition, place the change operation in that subset.
3) If a change operation is in a relation with two (or more) elements of different subsets, join the subsets together and add the change operation to it.

In terms of the partition, i.e., these subsets of change operations, we rephrase our hypothesis as: A commit is a single-task commit if and only if our technique does not split up the commit into different subsets of change operations.

Applying this relation to the running example, we see that

$$c_1 \in S(c_2) \land c_1 \in S(c_3) \land c_1 \in S(c_4) \land c_1 \in S(c_5)$$

and thus

$$c_1 \equiv'_S c_2 \land c_1 \equiv'_S c_3 \land c_1 \equiv'_S c_4 \land c_1 \equiv'_S c_5$$

giving finally

$$c_1 \equiv_S c_2 \equiv_S c_3 \equiv_S c_4 \equiv_S c_5.$$

Similarly, we can see that

$$c_6 \equiv_S c_7 \equiv_S c_8 \equiv_S c_9.$$

There is no further connection between changes, so there are two equivalence classes. Our algorithm considers the running example to consist of two distinct parts.

III. Dataset

We now introduce the well-established dataset of commits that we will use to evaluate our hypothesis. Subsequent sections will evaluate our approach on a refinement of this dataset, to which we apply data cleansing through automated filtering and manual commit verification first.

The dataset of commits stems from five Java programs as used by Herzig and Zeller in [5, 11]. We are not aware of version numbers assigned to this dataset. For reproducibility purposes, we provide the exact one we used via https://soft.vub.ac.be/~wmuylaer/publications. The programs in question are: ArgoUML, GWT, Jaxen, JRuby, and XStream. These projects were chosen for meeting the following quality criteria: to be under active development (at the time of their analysis), to have at least 48 months of active history, to have more than ten active developers, and to feature a reasonable number of identifiable bug fixes. For each of the projects, Herzig and Zeller manually identified single-task and composite commits using commit and issue information. Using the single-task commits, Herzig and Zeller also created artificial composite commits for each of the five projects. In order to see how our approach deals with actual situations, we do not consider this set of artificial composite commits for our evaluation. Instead, we limit ourselves to the real-world commits in the dataset. Table I depicts the number of commits present for each type of commit in each of the projects. Table I also provides the median number of Java files found per commit.

The prototype implementation of our technique is limited in the types of files it supports. We used this information to perform an automated filtering of the commits in the dataset which were known to be affected by these limitations. In terms of file types, our prototype does not support non-Java files that might appear in a commit. Our prototype also failed to construct the system dependence graph for some of the Java files, as evidenced by an exception being thrown during its analysis. We marked the corresponding commits in the dataset as causing failures. In the case of two files, moreover, no exception was thrown but graph construction timed out (i.e., took longer than an hour).

In terms of changes to the files affected by a commit, there are some limitations too. Program or system dependence graphs do not take comments into account. It follows that our approach cannot either. Moreover, in our prototype implementation, the graph construction algorithm only works on code contained within methods. As such, changes to, for example, class or field declarations are ignored. Finally, for some change operations, our implementation failed to identify a matching node in the system dependence graph. This can be the case when something is inserted without a tie to the original code. Meaning, CHANGENODES categorises the change as an insert and not a move or an update of code that was already in the program. As our system dependence graph considers the original version of the code, this insert may not be linked to any meaningful node. Taking these limitations into account, our automated filtering step removes commits for which no files with valid change operations remain. Only the ones remaining are considered for any further evaluation. Table I depicts the number of each type of commit per project after the automated filtering step. The table also depicts the median number of Java files per commit for these remaining commits. Finally, Table I depicts the median number of valid change operations per Java file.

Following the automated filtering described above, we performed a detailed manual verification of the 504 commits that remain from the original dataset. We inspected the code changed by each commit as well their accompanying commit message. We looked for the presence of the following aspects.

- One of the tasks in a composite commit is the fixing of comments, formatting, or other style issues. The presence of such a task in a composite commit means our technique would not be able to correctly distinguish tasks.
- The commit consists of many changes to statements that TINYPDG cannot handle, or is centred around such changes. An example is the `try-catch` statement.[2]
- The commit consists of many repeated changes in otherwise unrelated locations. Consider, for example, commit ba2f8bd2[3] of the JRuby project. In this commit, a `null` check is added to several different methods. While conceptually related, this is not a relation our technique can possibly discern.
- Our approach considers the project before the changes. If the commit primarily consists of new files, then our approach cannot do anything.
- Finally, we also consider whether the dataset categorised a commit correctly. By this we mean the commit was marked as comprising many tasks while it was actually just a single task, or vice-versa.

In case of uncertainty in our analysis, we left the data of the dataset as is. In this manner, we avoid personally influencing the data.

Our manual analysis filtered out another 116 of the 504 commits. Among the 116, 35 had formatting (e.g., whitespace changes) as one of the tasks. In 13 occasions, TINYPDG would not be able to handle the types of changes. Repeated changes occurred 53 times. Finally, 20 of the commits had a majority of new files being added. We point out that these numbers do not add up to 116; some commits were placed in multiple categories.

We are left with 388 commits for the evaluation of our approach. Of these, 359 commits passed all our scrutiny. We found the other 29 commits to be categorised incorrectly. The main culprit for incorrect categorisations, may be a different interpretation of what a composite commit is. Take, for example, commit 26d69d46[4] of the GWT project. This commit refers to two issues in its commit message: "Fix for issues #966 and #867; escapes HTML end tags from string literals in compiler output.". Despite there being two issues mentioned, there is but one fix that happens to fix the both of them. These types of commits were marked as being composite in the dataset, but we consider them to only perform a single task. As such, to perform the evaluation, we switch the categorisation for these commits.

Table I describes what is left of the dataset. With these remaining commits, we will evaluate our approach. As mentioned before, Table I depicts the number of commits, the median number of Java files per commit, and the median number of valid change operations per Java file.

After filtering out commits that cannot be used to evaluate our approach, the dataset contains 388 commits which will be used for the evaluation.

IV. EVALUATION

To analyse our hypothesis, we consider two research questions.

[2] While there is code for this present within TinyPDG, we saw no notice of it in the generated PDGs. We decided not to dig into debugging this particular potential bug.

[3]The full identifier is ba2f8bd229c62aa68acf176fa5c7578a4e7670e1.

[4]The full identifier is 26d69d46ad7fbb01ac5a2cd9a03084f73e9cff51.

TABLE I
DESCRIPTIVE STATISTICS FOR THE ORIGINAL DATASET BY HERZIG AND ZELLER [5, 11], THE DATASET AFTER AUTOMATED FILTERING, AND THE DATASET AFTER OUR MANUAL COMMIT VERIFICATION. IN CASE OF AGGREGATED NUMBERS (I.E., THOSE PER COMMIT OR PER FILE), THE MEDIAN IS GIVEN.

		Original		After automatic filtering			After manual filtering and verification		
		Commits	Java files per commit	Commits	Java files per commit	Change operations per file	Commits	Java files per commit	Change operations per file
ArgoUML	single-task	125	1	75	1	5	76	1	5
	composite	168	2	112	2	4	63	2	6
GWT	single-task	44	1	19	1	7.5	21	1	6.5
	composite	68	3.5	46	2	7.5	30	2	7.5
Jaxen	single-task	32	1	22	1	5	16	1	5.5
	composite	12	1	6	1	5	3	1	9
JRuby	single-task	200	1	60	1	4	60	1	4
	composite	271	1	112	1	5	81	1	6
XStream	single-task	37	2	26	1	4	21	1	6
	composite	37	3	26	2	7.5	17	2	9
Total	single-task	438		202			194		
	composite	556		302			194		

RQ1 Does our technique correctly identify composite commits?

RQ2 Does our technique correctly identify the single tasks within a commit?

We will use the dataset described in Section III to answer these research questions. The remainder of this section consists of the following three parts. We describe the research method for each research question, provide the results for each answer, and discuss any threats to validity.

A. Research Method

Our hypothesis concerns entire commits. The implementation of our technique, however, works on a per file basis. Its output is the number of sets in the partition of the file, i.e., the number of clustered changes for that file. Commits may contain many changed files and as such we need to reconcile the two. We define the following metric to do this.

$$\text{PARTITION}_{\text{commit}} = \max_{\text{file} \in \text{commit}} \text{PARTITION}_{\text{commit}}$$

Using that, we define classification by our technique as follows.

$$\text{COMPOSITE}_{\text{commit}} = (\text{PARTITION}_{\text{commit}} > 1)$$

In other words, our technique considers a commit composite if at least one file had more than one set of change operations in the file's partition.

1) Composite Commit Identification: We apply our technique to all commits in the dataset. Commits are classified as either composite or single-task in the dataset. Our technique classifies a commit as either composite or single-task. Thus there are four possible results to consider in the evaluation of a commit.

1) Composite commit correctly identified as composite. A true positive.
2) Single-task commit correctly identified as single-task. A true negative.
3) Single-task commit incorrectly identified as composite. A false positive.
4) Composite commit incorrectly identified as single-task. A false negative.

We will provide the number of times each of these possibilities occurred. We will also provide the precision and recall based on those numbers. Finally, the F-score, which combines precision and recall into a single number, is provided. These three metrics provide a number between 0 and 1 where 1 is the situation in which everything is correct. Results will be reported on a per project basis.

2) Single-Task Identification: Here too our technique is applied to all commits in the dataset. The results from the previous research question remain relevant here. However, for this research question we will look at the results with a focus on the single-task commits, rather than on the composite commits. If our technique is good at identifying single-task commits, then that is a strong indication it is good at identifying single tasks. However, it is not sufficient: what if the technique just overapproximates single tasks?

To avoid this possibility, we will also look into the number of sets reported by our technique for composite commits (the $\text{PARTITION}_{\text{commit}}$ metric mentioned before). The dataset states for some composite commits how many tasks they comprise. We will compare the numbers reported by our technique to the numbers present in the dataset. If the numbers match up, this is an indication that the sets in the partition identify single tasks correctly within a composite commit. We will make this comparison in two ways.

1) Looking at how often the number of sets of the partition matches up exactly.
2) By performing a Wilcoxon signed-rank test to see if there is a significant difference between the two. As usual with this test, the null hypothesis H_0 states that the numbers are drawn from the same population. In other words, rejection by this test indicates the number

of sets of the partition reported by our technique is significantly different from the number of sets expected by the dataset.

Note that this would just be an indication of correct partitioning, not a certainty. To be a certainty, the partition calculated by our technique should match up with one in the dataset. We cannot make this comparison here as we lack the data to compare to, i.e., the dataset does not specify which groups of fine-grained changes a composite commit is comprised of.

To further mitigate this issue, we have three computer scientists evaluate the output produced by our tool. For a random selection of 31 commits, the computer scientists are tasked to rate the output of our tool on a five level Likert scale. We will provide the mean of their replies per person. They are also asked whether the clustered changes should be further combined, further split up, or neither of the two. For this, we will consider for how many commits they thought it should go one way or the other. Even if only one of the reviewers wants to see the clustering done differently, we will still count that commit as needing improvement. This final question is important in knowing whether our clusters cross task boundaries or not.

B. Results

We present the results and a conclusion for research questions one and two.

1) Composite Commit Identification: An overview of the results in identifying composite commits is given in Table II. As mentioned, this table depicts the total number of commits, the number of true positives, the number of true negatives, the number of false positives, and the number of false negatives. It also gives the calculated precision, recall, and F-score based on those results. All the numbers are provided per project.

In interpreting these results, it is important to keep the number of commits per commit type in mind for each project. This data was provided in Section III, specifically in Table I. All else being equal, a higher or lower number of composite commits versus single-task commits would result in higher or lower precision, respectively.

In the results, the numbers for the Jaxen project are noticeably lower than for the other results. There were few commits for this project *and* the majority of those commits were single-task. Only three composite commits were analysed for the Jaxen project. We are thus inclined to attribute this result, at least partially, to these factors. For the other projects, the numbers are more positive. Both precision and recall seem to be around 70%. The F-score too is around that ratio. Here we feel obliged to point out that for the GWT and JRuby projects, more composite than single task-commits are present. This affects the precision in a positive manner.

The results are positive, but not overwhelmingly so. Our technique correctly identifies a large number of commits, but also fails more often than desired.

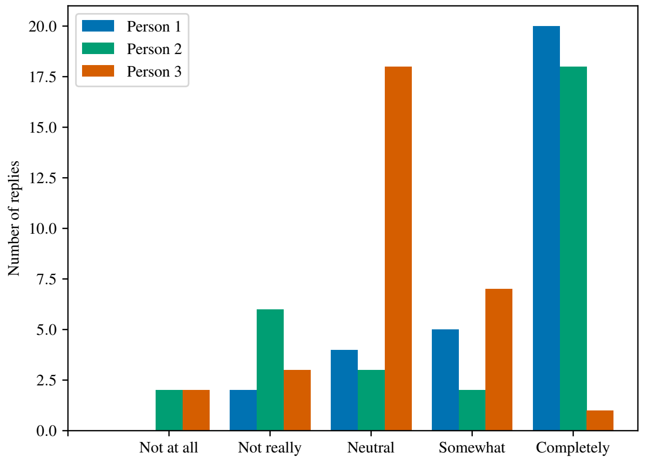

Fig. 5. Results of a small survey into whether the results by our change clustering makes sense.

Our technique correctly identifies composite commits. The F-score of the identification is 70%.

2) Single-Task Identification: We consider again Table II, but now from the point of view of the single-task commits. To do this, we calculate the relevant precision, recall, and F-scores. The results are depicted in Table III. Besides the GWT project, all resulting F-scores are over 65%. For Jaxen and XStream the results are even more positive. As in the previous section, the result is positive, but not overwhelmingly so. We consider the other test results.

The next step was the comparison of the number of tasks our technique identifies for a commit against the number of tasks in that commit according to the dataset. While we performed a validation of our dataset in Section III, we note that this validation did not include an analysis of the number of partitions of a commit as described by the dataset. Table IV summarises the results of comparing our technique to the numbers present in the dataset. The dataset did not provide any information regarding the number of tasks in a commit for any of the commits in the Jaxen project. As such, no results are present for the Jaxen project.

The results we encounter here are negative. Identifying the exact number of results proves difficult for our technique. The dataset's number of tasks making up the commit was found in only a quarter of the cases for the four projects for which data was available. Similarly, the p-value of the Wilcoxon signed-rank test for all four cases does not reject the null hypothesis. Recall that H_0 states that the numbers (from our technique and from the actual task count) are drawn from the same population, i.e., that there is no significant difference between the two. Note that not rejecting H_0 does not imply that there *is* a connection.

The results of our small survey are more positive. The raw results are visualised in Figure 5. The mean result for each of the reviewers was 4.39, 3.90, and 3.06. As mentioned, this is a five level Likert scale: a 3 is the middle, anything higher is

TABLE II
IDENTIFICATION OF COMPOSITE COMMITS ON A PER PROJECT BASIS. TIME IS THE MEDIAN TIME TAKEN TO ANALYSE A COMMIT.

	Commits	True positive	True negative	False positive	False negative	Precision	Recall	F-score	Time (s)
ArgoUML	139	45 32%	51 37%	25 18%	18 13%	0.64	0.71	0.68	2
GWT	51	18 35%	12 24%	9 18%	12 24%	0.66	0.6	0.63	2
Jaxen	19	2 11%	13 68%	3 16%	1 5%	0.4	0.67	0.5	0
JRuby	141	48 34%	46 33%	14 10%	33 23%	0.77	0.59	0.67	5
XStream	38	12 32%	17 45%	4 11%	5 13%	0.75	0.71	0.73	0

TABLE III
THE PRECISION, RECALL, AND F-SCORE FOR THE IDENTIFICATION OF SINGLE-TASK COMMITS. THESE ARE CALCULATED USING THE ABSOLUTE NUMBERS PRESENT IN TABLE II.

	Precision	Recall	F-score
ArgoUML	0.74	0.67	0.7
GWT	0.5	0.57	0.53
Jaxen	0.93	0.81	0.87
JRuby	0.58	0.77	0.66
XStream	0.77	0.81	0.79

TABLE IV
IDENTIFYING THE NUMBER OF SINGLE TASKS WITHIN A COMPOSITE COMMIT. ONLY COMMITS FOR WHICH THE NUMBER OF PARTITIONS IS PRESENT IN THE DATASET ARE CONSIDERED.

	Composite commits	Correct number of sets	Wilcoxon p-value
ArgoUML	33	7 (21%)	0.57
GWT	28	7 (25%)	0.97
Jaxen	0	—	—
JRuby	65	15 (23%)	0.61
XStream	16	4 (25%)	0.13

a positive response. Two of the reviewers were rather positive about the output, one remained more neutral. In terms of ways to improve the clustered changes, 3 of the 31 cases were considered to need further splitting up of the created clusters of changes. For 10 of them the opposite was true: more changes should be combined into one cluster. No changes should be made in the other 18 cases.

This result mitigates, to some extent, the issue of not having the same number of clusters as the dataset. Our clusters of changes are more fine-grained: a need to combine changes implies a created cluster of changes does not span across multiple tasks in a commit. If the clusters would span across tasks, they would no longer help with the analysis. Instead, the individual clusters stay within their tasks and each manage to identify a part of their task. This ensures the clusters are not rendered useless when analysing or reviewing a commit. A reviewer may still use the different clusters of changes knowing each cluster contains changes that belong together. The reviewer can then still further combine the clusters as they deem necessary.

Our approach is able to identify when a commit performs a single task with an F-score around 70%. Our approach creates more fine-grained parts than those that are counted in the dataset. A manual review by some computer scientists finds that the clusters of changes are contained within their respective tasks in a commit. As such, the clusters can still be used for code review, integration, and reversion.

Finally, we have a brief look into the scalability of our approach. To do so, we consider the time it takes to analyse a commit. The median time for analysis per commit on a per project basis is depicted in Table II. Note that time was recorded with a precision of seconds, thus entries stating 0 imply a time under 1 second. For the majority of commits, the analysis takes but a few seconds. There were outliers, however, with 42 commits taking a minute or more and six of those taking over five minutes.

C. Threats to validity

The evaluation of our approach is dependent on the correctness of the dataset we use as a ground truth. This is true in its most basic form: stating whether a commit is composite or not. This is also true in the number of single tasks it discerns in a composite commit. As described in Section III, we tried to mitigate this to some extent by performing a manual filtering phase. We were conservative in this manual filter phase, so errors may still be present. Mistakes in either could partially invalidate our evaluation.

TINYPDG is not able to handle some Java statements, like `try-catch`. Our manual filter phase only removed commits where these statements were the *main* part of what was being changed. This results in a possibly subpar analysis.

Our implementation may have bugs. This in turn affects the results of analysing commits and the evaluation of those results. Also in our implementation, we enable binding resolution, as provided by the Eclipse JDT library, to create the abstract syntax tree. We enable binding resolution in order to resolve method calls and the like to their definition. In this, we are bound by the precision of this static binding resolution. This can further influence the results.

Horwitz et al.'s algorithm [18] is not entirely state of the art. Improvements have been made over the years to enable a better handling of, for example, classes and objects. This may negatively affect our results.

It is possible unrelated tasks touch overlapping parts of the code in the same way. When committed together, our technique may not be able to distinguish the two. We do not see a way around this situation with just our approach.

V. Related Work

Tao and Kim [2] perform a similar approach to ours. They employ the ZeroOneCFA points-to analysis built into the T.J. Watson Libraries for Analysis (WALA). This is done at the level of changed lines, not at the level of fine-grained changes like we use. Moreover, the work uses more than only program slices to determine which changes are related. Formatting changes, for instance, are also considered to separate changes into change groups. In addition, string comparisons are used to relate, for example, the addition of a `.clone()` method call in several locations without any static dependencies. While this string comparison adds some extra relations between changes, it does not always prove beneficial. The authors mention an example in which their technique related changes that did not belong together. Despite the similarity in program slicing, we feel our work is sufficiently different. We focus specifically on the program slicing in isolation to analyse how well it behaves on its own. Tao and Kim only look at the results of their combined analysis.

Barnett et al. [19] also attempt to decompose changes. To do so, they consider "diff-regions". These regions are the result of performing a textual difference and splitting them up to stay within one method or within one type. Barnett et al. work with limited information: only the before and after of changed files are provided in one changeset (i.e., one commit). They do not have access to the entire project. Instead they make use of any method definitions that are present to link the diff-regions if they belong to the same method, if one diff-region uses a method whose definition appears in another diff-region, or if two diff-regions use a method defined in a file present in the changeset. Our work differs in that we make use of more than just the relation between definitions and uses. As we make use of a program dependence graph, all control and data flow dependencies are mapped, providing a richer picture.

Kreutzer et al. [20] use both line-based changes, by making use of the `diff` tool, as well as fine-grained changes by making use of ChangeDistiller [13]. To match changes together, Kreutzer et al. look at a string representation of these changes. They then look for the longest common subsequence to determine similarity in changes. This information is then used to cluster changes together. We too make use of change distilling, but our methods of defining similarity strongly differ. We rely on the dependencies that show up in the program dependence graph, not on patterns.

Just like Kreutzer et al., Kirinuki et al. [21] also use the longest common subsequence to look for patterns. They analyse the changed program statements between many revisions. If patterns are repeated across many revisions, Kirinuki et al. consider them composite and add them to their database. When new commits are made to that project, the new commit can be compared to the patterns in the database. If there are more similarities than a certain threshold, the commit is deemed composite. This differs from our work in the same way as the previous paragraph, we do not look for patterns in the changes.

Herzig and Zeller [11] combine many different metrics by means of confidence voters. They looked at changes on the level of addition and removal of method calls and method definitions, which is more coarse-grained than our approach. To decide similarity, they combined various metrics such as, for example, the distance within a file, the similarity in package names, or the distance in a call graph. We do not take any of these metrics into account and instead rely on slicing in the program dependence graph.

A completely different approach is performed by Dias et al. [1]. They make use of a change logger, a program installed by developers that tracks fine-grained changes as they are made. They then link the fine-grained changes together based on attributes such as date, were the changes performed close together, or are they part of the same class. They employ a form of machine learning to use these metrics to decide whether commits are composite. We purposefully did not opt to use change loggers. Requiring a change logger renders the throves of repositories already out there useless. Even with yet-to-be-developed software, it may not be feasible to require its developers to install appropriate change loggers. Privacy reasons could be cited or the developer may just find it too cumbersome to bother to set things up.

Arima et al. [22], finally, try to achieve the opposite. They consider the issue of one task being spread across multiple commits. To detect the issue, they construct a weighted directed graph of methods in which two commits are represented. Depending on the distance between two methods in the graph, their approach decides whether the different commits should have been one commit.

VI. Conclusion

We want to help developers, code reviewers, and researchers with tool support for decomposing composite commits according to the tasks they perform. For the foundation of this tool support, we start from the hypothesis that related changes belong to one and the same program slice in a program dependence graph. The corresponding algorithm performs program slicing on the change operations computed for a commit by a change distiller, and clusters the resulting fine-grained change operations according to the slices they belong to. We evaluated our technique on a dataset of commits stemming from five Java projects [11]. We first analysed this dataset and further refined it to fit our context. We found that our technique is able to identify single-task and composite commits. We also found that our technique creates more fine-grained clusters than those counted by the dataset we used. A manual review indicated that in the situations where there is no one-to-one mapping from cluster to task, each cluster of changes still stays within one single task. We conclude that our approach is capable of alerting developers about commits that are composite, prompting action on their part. It also enables identifying the individual parts of said commit. This way, the commit can be corrected before being pushed to other members of the team or the public at large.

VII. Future Work

In future work, we may consider different clustering criteria. For example, employing a relation similar to the *useUsesInDiff* relation used by Barnett et al. [19] could create extra connections in our graph and thus fewer sets in the partitions.

In this paper we sliced on the program dependence graph of the version of the code before the commit was applied. For larger additions, it may be more interesting to slice in the program dependence graph for the new version of the code.

Slicing can be done in two directions. We performed backwards slicing, considering the statements that affect a certain statement. One could also slice forwards, considering the statements affected by a certain statement. Forwards slicing might contain forks that match larger areas of the program dependence graph, this might influence precision. Alternatively, a best-of-both-worlds approach might combine the two directions to perform some sort of "optimal" slicing.

For an industrial setting, our technique needs to be able to analyse a commit reasonably fast. Our analysis only took a few seconds for the majority of analysed commits. However, several commits took over a minute to analyse. Optimisations to our research prototype are likely to be in order.

Notes

We thank Herzig and Zeller for sharing the dataset used to evaluate our commit slicing technique. The results of the analysis (both manual and automated) and the scripts used to produce the evaluation can be found via https://soft.vub.ac.be/~wmuylaer/publications. Ward Muylaert is an SB PhD fellow at FWO.

References

[1] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[2] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," in *International Conference on Mining Software Repositories (MSR)*, 2015.

[3] M. Konopka and P. Navrat, "Untangling development tasks with software developer's activity," in *International Workshop on Context for Software Development*, 2015.

[4] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, 2008.

[5] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Software Engineering*, 2015.

[6] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[7] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? - an exploratory study in industry," in *International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[8] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *International Conference on Software Engineering (ICSE)*, 2013.

[9] M. Weiser, "Program slicing," in *International Conference on Software Engineering (ICSE)*, 1981.

[10] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Computing Surveys*, 2012.

[11] K. Herzig and A. Zeller, "The impact of tangled code changes," in *International Conference on Mining Software Repositories (MSR)*, 2013.

[12] R. Stevens and C. De Roover, "Extracting executable transformations from distilled code changes," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[13] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, 2007.

[14] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *International Conference on Management of Data (SIGMOD)*, 1996.

[15] Y. Higo and S. Kusumoto, "Enhancing quality of code clone detection with program dependency graph," in *Working Conference on Reverse Engineering*, 2009.

[16] ——, "Code clone detection on specialized PDGs with heuristics," in *European Conference on Software Maintenance and Reegineering*, 2011.

[17] Y. Higo. TinyPDG: A library for building intraprocedural PDGs for Java programs. [Online]. Available: https://github.com/YoshikiHigo/TinyPDG

[18] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, 1990.

[19] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," *IEEE International Conference on Software Engineering*, 2015.

[20] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen, "Automatic clustering of code changes," in *International Conference on Mining Software Repositories (MSR)*, 2016.

[21] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *International Conference on Program Comprehension (ICPC)*, 2014.

[22] R. Arima, Y. Higo, and S. Kusumoto, "A study on inappropriately partitioned commits — how much and what kinds of IP commits in Java projects?" in *International Conference on Mining Software Repositories (MSR)*, 2018.