2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets

Mike Barnett
Microsoft Research
Redmond, WA, USA
mbarnett@microsoft.com

Christian Bird
Microsoft Research
Redmond, WA, USA
cbird@microsoft.com

João Brunet
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
joao.arthur@computacao.ufcg.edu.br

Shuvendu K. Lahiri
Microsoft Research
Redmond, WA, USA
shuvendu@microsoft.com

***Abstract*—Code Reviews, an important and popular mechanism for quality assurance, are often performed on a changeset, a set of modified files that are meant to be committed to a source repository as an atomic action. Understanding a code review is more difficult when the changeset consists of multiple, independent, code differences. We introduce CLUSTERCHANGES, an automatic technique for decomposing changesets and evaluate its effectiveness through both a quantitative analysis and a qualitative user study.**

I. INTRODUCTION

Code review is an important mechanism for quality assurance in software development: it provides feedback and avoids introducing bugs. However, code review is effective only to the degree that reviewers are able to understand the changes being made. Prior work has shown that the easier it is for a reviewer to understand a change, the more likely they are to provide feedback that improves quality [1]. We and others have observed that when changes are small and/or cohesive, reviewers are most able to understand them [2], [3].

Unfortunately, developers often make changes that incorporate multiple bug fixes, feature additions, refactorings, etc. [4], [5], [6]. These result in changes that are both large and only loosely related, if at all, leading to difficulty in understanding. Developers have indicated that they are able to understand these changes better if authors annotate them with comments in the review tool, but this is a cumbersome task and occurs less than 5% of the time in practice. Developers have also provided feedback that decomposing such composite changes would help them to understand changes during review, but to date no such tools exist or are being used [2].

To address this, we developed CLUSTERCHANGES, a lightweight static analysis technique for decomposing changesets. The insight underlying CLUSTERCHANGES is that we can relate separate regions of change within a changeset by using static analysis to uncover relationships such as definitions and their uses present in these regions. For example, if a method definition is changed in one region and its callsites are changed in two other regions, these three regions are likely to be related and should be reviewed together. Connected subgraphs of related code entities form partitions that can be explored and understood independently; a formal description of our algorithm is presented in Section III.

In an effort to validate the results of CLUSTERCHANGES, understand differences between different types of partitions, and gauge its potential usefulness, we built a prototype graphical tool and used it to investigate changesets submitted for review in Bing and Office at Microsoft. Our quantitative evaluation shows that over 40% of changes submitted for review at Microsoft can be potentially decomposed into multiple partitions, indicating a high potential for use. We also characterized and quantified the nature of suggested partitions across 1000 changesets and have performed a careful manual investigation of over 100 reviews with an eye towards (i) spurious relationships for changesets with fewer than 2 *non-trivial* partitions (Section III-B3), and (ii) missing relationships for changesets with 10 or more *trivial* partitions. We then conducted a qualitative user study with twenty developers, where we found that most developers agree with our automatic partitioning and believe the decomposition is useful for reviewers to understand their changes better (some even asked for the prototype to use on their own reviews going forward).

Finally, we discuss limitations of our algorithm and our implementation and describe how CLUSTERCHANGES can be integrated into the code review process.

II. THE PROBLEM

Developers submit code to be reviewed on a daily basis. Although good development practices prescribe that developers perform small and cohesive commits, they often contain large and independent changes. Code review is dependent on understanding the change being made. While large commits make review difficult (there is more to read and try to understand), the presence of multiple independent changes make matters worse. Developers are forced to read and make sense of more code than they would need to if the independent changes were examined separately. In this section we triangulate different sources of information to show the existence of this problem and to motivate our work.

a) Concrete example: Here is an arbitrarily chosen commit made to the Roslyn project [7] on 18 August 2014 [8]. It contains edits to eight files and the addition of one file. The commit message says:

> Fix #244 by adding inmethod binders in between properties and indexers. Also refactors expression-

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.35

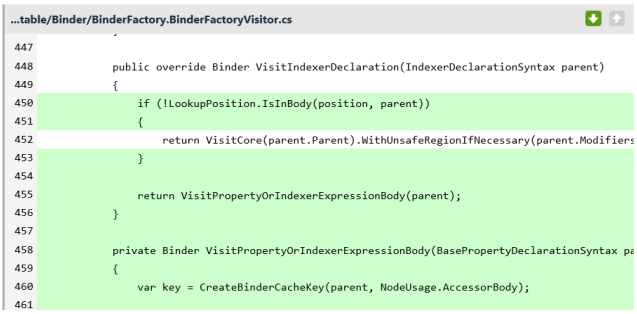

Fig. 1. Example change from a real changeset. The white lines are unchanged lines. The green lines are added lines. There are no deleted lines; those would be shown highlighted in red.

bodied member semantic model tests into their own file and adds some extra tests.

("#244" refers to a bug report that had been made a few weeks prior.) Part of the relevant change is shown in Figure 1. Of particular interest for our work is that line 450 adds a call to a method which is introduced in another file included in the changeset, whereas the method call added on line 455 is to the method introduced on line 458 and so is easily found and understood. CLUSTERCHANGES does indeed decompose the commit into two independent partitions, one of which contains the two relevant files for the bug fix.

b) Literature: Prior literature has found that large changes are problematic to examine [9]. For instance, Rigby et al. [3] found that in order to facilitate review activity, changes should be small and independent. For example, changes that combine refactorings with bug fixes can be difficult to review together, especially if the reviewer doesn't know that there are multiple things going on in the change. Rigby et al. raised the need to enable a divide-and-conquer review style that keeps each change logically and functionally independent.

In a later study, Rigby et al. [10] analyzed change size (number of added and deleted lines, and number of changed files) and its relationship to several complexity metrics for changes submitted for peer review in 6 open source projects. They found that the more files changed and differences produced in a submitted change, the more likely the change actually consisted of multiple relatively independent changes that might affect diverse sections of a system. As a consequence, developers experience more difficulty when trying to understand large code changes.

Herzig and Zeller [6] found that a non-trivial proportion of changes in five open source projects are *tangled*, meaning that they contain more than one bug fix, feature, refactoring, etc. While their study was on the impact of such changes on research in mining software repositories, they provide evidence that changes do exist which can in some way be decomposed. They point out (and we agree) that there may be a completely valid reason that a developer commits multiple changes that appear unrelated as one change. One of our study participants indicated that the overhead to run regression tests after each commit often leads to a developer combining multiple changes in a commit.

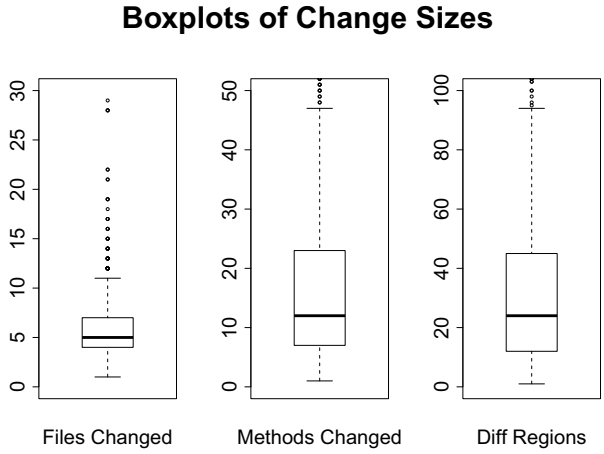

Fig. 2. Boxplots showing distributions of the size of changes in terms of the number of files and methods changed as well as the number of *diff-regions*.

Our goal is not to force developers to commit independent changes separately, but rather to identify independent parts of changes to facilitate understanding.

Tao et al. conducted a large scale study of how developers examine and understand code changes [2] with one of their main goals being to determine "How to improve the effectiveness and efficiency of the practice in understanding code changes." They observed that "engineers sometimes mix multiple bug-fixing changes or changes with other purposes in a single checkin. Understanding [these] composite changes requires non-trivial efforts." One of their main conclusions is that "Participants call for a tool that can automatically decompose a composite change into separate sub-changes." and point out that "however, currently such decomposition lacks tool support." As one of the developers in their study indicated: "It would be useful to be able to analyze groups of functional changes instead of having to go file by file and context switch between changes. I want to be able to see all the changes that related [sic] to one bug, or possibly one variable within the changelist".

c) Discussions with developers: We were also motivated by data gathered from our previous empirical study on code review [1] and follow on observations and discussions with developers. One common theme that developers mentioned when discussing code review is how large a review is. As one developer indicated *"If I get a code bomb to review with like thirty files, I can either look at it now and skim it in five to ten minutes or I can save it for later and I may or may not get to it. Both aren't good."* A Windows development manager indicated that one of the bottlenecks in their process was reviewing changes made to resolve conflicts after merges from branch to another. It is not uncommon for such changes to include upwards of 50 files and contain many unrelated changes.

d) Code change size at Microsoft: To quantitatively investigate the change size at Microsoft, we conducted a preliminary analysis on 1000 code changes submitted to be reviewed within Microsoft Office during the Office 2013 development cycle. Figure 2 shows quantitative data on the change sizes. "Files changed" shows a boxplot of the number of

files that are added, modified, or deleted in changes submitted for code review. "Methods changed" is a count of the number of individual methods that are added, modified, or deleted. We define a *diff-region* as a contiguous sequence of added, deleted, or modified lines. For example, Figure 1 shows two diff-regions. The median number of changed methods and diff-regions are 12 and 24, while over 25% of reviews modify 23 methods and comprise 45 diff regions. These numbers suggest that many changes submitted for review may be large and/or difficult to understand. Some developers that we talked to during this and earlier studies ([1], [11]) indicated that it is difficult to review a change that has many regions of change scattered across the files even if there are a small number of modified files.

III. ClusterChanges

A. Starting Point

The predominant code review tool used at Microsoft is CodeFlow [12]. CodeFlow provides access only to individual *changesets*: these are standalone packages containing the set of pairs of changed files. Each pair has a *before-file* and an *after-file* (for added or deleted files, one element of the pair may be empty). It is important to note that analyzing changesets is quite different than analyzing the history of a source repository. In particular, changesets do not provide full information: we do not have the full set of project files, nor the compiler options used to actually compile the code. We use a standard text differencing tool on each pair of files to get a set of pairs of *diff-regions*. Because our analysis applies only to C# files, we split diff-regions that cross type and method boundaries so that no diff-region crosses more than one type or method. This corresponds to the predominant organizing units of object-oriented code.

We restricted ourselves to C# [13] because it is one of the primary programming languages used within Microsoft. It is a modern object-oriented language, with types (primarily classes, but also structs) composed of members such as fields, events, properties, and nested types. Another reason for picking C# is our ability to use Roslyn [7], the new Microsoft compiler, with its open API. We use Roslyn to create a synthetic project comprising the after-files from a changeset and the basic .NET assembly references that provide definitions of basic types like string, etc. In return, we access the resulting symbol table and abstract-syntax tree that Roslyn provides after making a best-effort (relative to the available definitions) parse of the project.

B. Definitions and uses.

We use the *def-use* relationship as the primary organizing principle for clustering diff-regions. Programmers often introduce interesting functional changes to code by introducing or modifying definitions along with their uses.

1) Computing def-use information: Given the project corresponding to the changeset, we collect the set of *definitions*, D, for types, fields, methods (including constructors), and local variables. D is the set of definitions present only in the changeset (that is, the definitions that occur anywhere in the files that were modified in the changeset). Then we scan the project for the set of all uses (i.e., references to a definition), U, as provided by the Roslyn API. For instance, any occurrence of a type, field, or method either inside a method or a field initialization is considered to be a use. We focused on this one relationship in order to evaluate the effectiveness of just this one source of information.

We define a function $Def : U \rightarrow D \cup \{\bot\}$ that maps any use to the corresponding definition. For any use $u \in U$, $Def(u) \in D$ whenever the definition is present in the changeset; otherwise, $Def(u) = \bot$ indicating that the definition lies outside of the changeset.

Each definition and use has an associated *span*, which is the sequence of (contiguous) characters in the source that represents the definition or use.

2) Projecting def-use on diffs: Since we are concerned with organizing the diff-regions, we consider only uses whose span intersects that of some diff-region. We define the following sets (in all formulas, the variable f represents a diff-region):

$$defs(f) \quad = \quad \{d \mid d \in D \wedge span(f) \cap span(d) \neq \{\}\}$$

Note that a definition (of a method or type, e.g.) that is changed in two non-contiguous regions results in the definition appearing in the $defs$ set for two different diff-regions. Similarly, we define the references found in a diff-region:

$$uses(f) \quad = \quad \{u \mid u \in U \wedge span(f) \cap span(u) \neq \{\}\}$$

In contrast with definitions, each use is distinct, so that given two distinct diff-regions, f_1 and f_2, $uses(f_1) \cap uses(f_2) = \{\}$. Next we project the definition-use relationship over the set of diff-regions. A pair $(d, u) \in D \times U$ is in the relation $defUsesInDiffs$ if and only if:

$$(\exists f_1, f_2 : f_1 \neq f_2 : d \in defs(f_1) \wedge u \in uses(f_2) \wedge Def(u) = d)$$

Finally, we also project uses of the same definition where the definition itself is present in the changeset, but does not appear within any diff-region. This captures changes made to all uses but where the definition has not been changed. A pair (u_1, u_2) is in the relation $useUsesInDiffs$ if and only if:

$$\begin{array}{l}(Def(u_1) = Def(u_2) \neq \bot) \wedge \\ (\forall f :: Def(u_1) \notin defs(f)) \wedge \\ (\exists f_1, f_2 : f_1 \neq f_2 : u_1 \in uses(f_1) \wedge u_2 \in uses(f_2))\end{array}$$

The first two conjuncts denote that (i) the two uses share a definition present in the changeset, and (ii) the definition has not been changed. The third conjunct denotes that the two uses are present in distinct diff-regions.

Our decision to restrict $useUsesInDiffs$ to definitions in D is motivated by two factors. First, definitions outside D are not necessarily resolved correctly, given the partial information available in the changeset. Second, changes to method calls defined in the framework (e.g. addition of calls to `Console.WriteLine` in two different methods) leads to spurious relationships among diff regions. We discuss this further in Section VI-A.

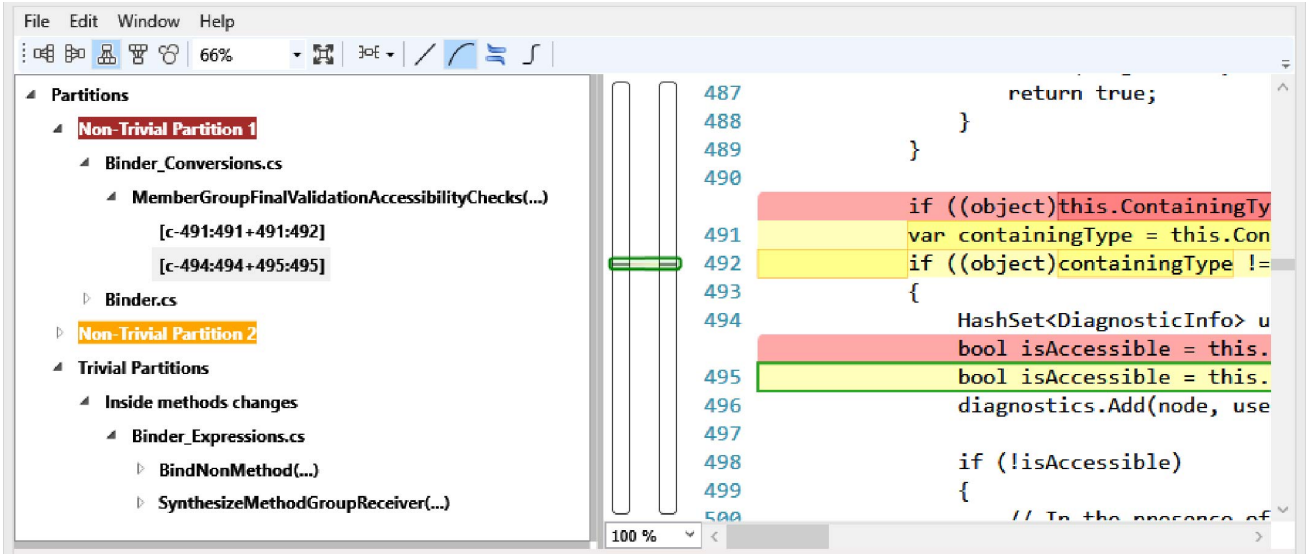

Fig. 3. Cluster Changes - Tree view displaying a change from the Roslyn project (located at https://roslyn.codeplex.com/SourceControl/changeset/4c74a47ca896)

3) Partitioning the set of diff-regions: Now we can define precisely what we mean by two diff-regions being related, using the relation $RelatedDiffs$. Distinct diff-regions f_1 and f_2 are in the relation $RelatedDiffs$ if and only if:

$$SameEnclosingMethod(f_1, f_2) \vee$$
$$(defs(f_1) \times uses(f_2)) \cap defUsesInDiffs \neq \{\} \vee$$
$$(uses(f_1) \times uses(f_2)) \cap useUsesInDiffs \neq \{\}$$

The relation $SameEnclosingMethod$ is true for pairs of distinct diff-regions whose span intersects the span of the same method (or property) definition. We group diff-regions in the same method together because a) in practice, we observe that changes to the same method are often related, and b) in prior research [1], we observed that reviewers usually review methods atomically (i.e., they rarely review different diff-regions in a method separately). Given these relations we create a partitioning over the set of diff-regions by computing the reflexive, symmetric and transitive closure of $RelatedDiffs$.

We then distinguish the set of *trivial partitions* as those partitions where all of the diff-regions within the partition are within the same method or where there is only one diff-region in the partition and it is outside of a method definition. We refer to the former category as *trivial in-method* partitions. That is, the trivial partitions are those diff-regions which we did not group with any other diff-regions (except those that are related only because they occur within the same method). All other partitions are *non-trivial partitions*: they contain diff-regions from multiple methods or changes in one method along with at least one change outside of any method definition.

C. Tool description

Summarizing, we have built a tool CLUSTERCHANGES that takes as input a CodeFlow changeset and produces a partitioning of the diff-regions from the after-files. Reviewers visualize the partitions in a *tree* view, as shown in the left pane of Figure 3. Currently, we number and assign colors to identify partitions. The tree view is similar to the current code review tool's user interface. In the tree view, each partition's leaves are the individual diff-regions which are then organized by method, type, and file. However, we do not explicitly show the relationships used to create the partitions. Prior to the study, some developers had indicated they were confusing and more importantly, there was a threat that showing them might bias the study.

The tree view is linked to a standard textual difference view of the before-file and the after-file: selecting a diff-region in the tree view produces a view of the source file with the changes highlighted, as shown in the right pane of Figure 3.

Our tool is not meant to replace the current code review tool, but rather is a prototype for validating our techniques.

IV. QUANTITATIVE EVALUATION

We initially took a random sample of 100 changesets submitted for review from Microsoft's search engine, Bing, in an effort to identify missing relationships and bugs in our implementation. We also used the tool on our own changesets as we worked on it. We selected changesets from reviews rather than commits because a review doesn't always correspond directly to a commit, as the former is used to solicit feedback and drive change while the latter is not intended to be rolled back.

We then applied CLUSTERCHANGES to a randomly chosen set of 1000 changesets submitted for review in Microsoft Office during Office 2013 development to determine the distribution of suggested trivial and non-trivial partitions. Figure 4 shows a histogram of the number of non-trivial partitions in these changesets.

While the most common case are changesets containing just one non-trivial partition, this still makes up only 45%. Nearly 42% of all changes contain more than one non-trivial partition.

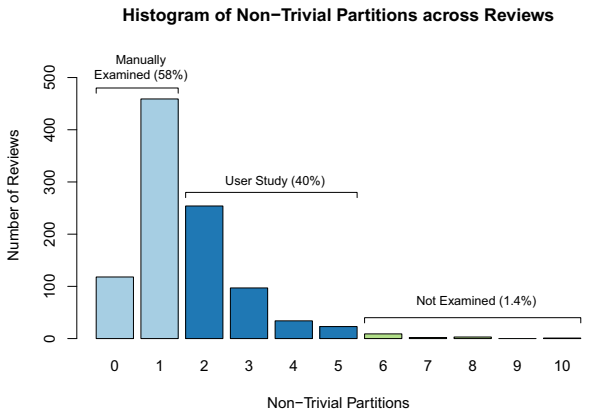

Fig. 4. Distribution of non-trivial partitions from 1000 changesets submitted as reviews

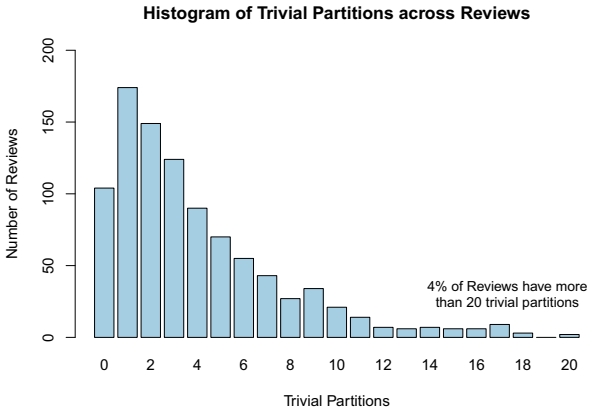

Fig. 5. Distribution of trivial in-method partitions from 1000 changesets submitted as reviews

In addition, the proportion of changed methods that end up in non-trivial partitions is 66% on average per review. To the degree that CLUSTERCHANGES correctly identifies non-trivial partitions, this indicates that i) a large proportion of changesets can be decomposed into multiple independent changes, and ii) our decomposition covers a large fraction of changed methods in a review. At the same time, the other 34% of changed methods that form trivial in-method partitions indicate that there is room for improvement for our technique. Figure 5 shows the distribution of trivial in-method partitions across the same changesets. There is a long tail in this distribution (one change had 326 trivial partitions) and just under 4% of the changesets actually have more than 20 trivial in-method partitions.

Ideally, we would evaluate our technique by validating the suggested partitions with the ground truth about the relative independence of these partitions. The best source of ground truth is the developer who created the changeset, which is feasible for only a very small fraction of the developers that made the 1000 changes. We therefore developed the following strategy to loosely group the set of reviews based on the number of partitions:

- ≤ 1 *non-trivial partition:* We manually examined changesets that contained fewer than two non-trivial partitions (a changeset can have no non-trivial partitions if it is composed only of trivial partitions) in a manner similar to Herzig and Zeller [6] by examining commit messages and looking at the modifications to the source code to determine if the partitioning appeared correct (Section IV-A).
- $2-5$ *non-trivial partitions:* We selected changes that comprised two to five non-trivial partitions for a user study to obtain the ground truth (Section V). These changes represent a significant fraction (around 40%) of all changes in Figure 4.
- ≥ 6 *non-trivial partitions:* We excluded reviews with six or more partitions from our study since (a) they constitute only 1.4% of reviews in Figure 4, and (b) consulting developers for these changes would take longer than our goal of a short interview.
- ≥ 10 *trivial partitions:* Under the hypothesis that a single developer's change cannot consist of so many independent changes, it is evident that there are relationships not captured by our technique. We performed a manual investigation of several of these reviews (Section IV-B).

A. Reviews with ≤ 1 non-trivial partition

We manually investigated the changesets that had one or fewer non-trivial partitions, indicated by the leftmost two bars in Figure 4 in an effort to determine if CLUSTERCHANGES ever puts sets of diff-regions into one non-trivial partition when they should actually be split. We examined the commit message to determine if more than one task (bug fix, feature implementation, refactoring, etc.) was accomplished. We also investigated the changeset to determine if the identified non-trivial partition in each changeset should actually be split into multiple partitions (trivial and/or non-trivial) or if any of the identified relationships were spurious.

We randomly sampled 50 of the changesets that had one or fewer non-trivial partitions. Of these, six had changeset descriptions indicating more than one task. In four of the six, CLUSTERCHANGES separated the tasks into partitions, but only one of the tasks comprised a non-trivial partition; the others were contained in trivial partitions. In the other two, the two tasks were actually semantically related. For example, one changeset addressed a bug in which a particular type of data file and its backup were lost or corrupted when it was moved. The changeset contained changes to the file manipulation code and updated tests that made calls to the same code. Since the test code did have a def-use relationship with the code in the bug fix, the diff-regions were placed into the same non-trivial partition.

We observed three cases where one or more trivial partitions in a changeset would have fit into existing non-trivial partitions,

but were kept separate due to relationships that CLUSTERCHANGES does not capture. For instance, the addition of a method call to a method in a base class was not put in the same partition as a change to an overriding method in a subclass (see Section VI-A for a discussion of these sorts of missing relationships).

We observed no cases of spurious relationships in non-trivial partitions. That is, whenever CLUSTERCHANGES put a group of diff-regions into one partition, we never found a case where the partition should have been split into multiple partitions.

B. Reviews with ≥ 10 trivial partitions

We randomly sampled 15 out of 119 changes with 10 or more trivial partitions, and performed manual investigation. It seemed very unlikely that there could be so many unrelated changes. Indeed, manual inspection revealed almost all of them were due to missing relationships. The three most prevalent missing relationships that we encountered are: (a) annotating several methods with common C# attributes such as `Serializable` or `Obsolete`, (b) a common refactoring (e.g. addition of a log message or variable renaming) across a large number of methods, and (c) relationships between overridden methods and their implementations. See Section VI-A for a detailed discussion of these and other missing relationships.

Next, we describe our user study where we approached review creators with the partitionings (2–5 non-trivial partitions).

V. USER STUDY

In this section we describe our user study methodology and its results. The following three research questions guided our user study:

- **RQ1:** *Do developers agree with our decomposition of their changes?* Our technique groups related code changes into partitions that should be independent. We were interested in whether participants agree with our decomposition.
- **RQ2:** *What role do trivial partitions play?* Our partitioning results in both non-trivial partitions and trivial partitions. We investigated how participants perceive the trivial partitions regarding their relevance and understanding with respect to the non-trivial partitions.
- **RQ3:** *Can organizing a changeset using our decomposition help reviewers?* We were interested in participants' opinion about whether our decomposition would make it easier for reviewers to understand their code changes.

To answer these research questions, we conducted semi-structured interviews with developers. We present the study and discuss some limitations of our approach identified during the study.

A. Methodology

We used a *firehouse research* study design [14], [15] to conduct 20 interviews with participants from 13 projects; we refer to them as P1 through P20. This research method gets its name from the fact that the research requires events to occur that cannot be induced by the researchers themselves. Researchers wishing to study those affected by fires must literally sit in a firehouse waiting for a fire to be reported. In our study we would "rush to the scene" soon after a change was submitted as a code review so that the author still had a mental model of his or her change fresh in mind. Through this method, we were able to interview developers within at most three days from the day they submitted a code change to be reviewed. Hence, we could get fresh feedback on their reasoning about their own changes, since developers were still familiar with them.

B. Participant Selection

We selected interviewees based on two criteria: i) their office should be located inside the Microsoft Redmond Campus, and ii) they should have submitted a code change that contained two to five non-trivial partitions. could easily travel to the building where the code change author worked and so that our decomposition had enough structure but was still be simple enough to fit in a twenty minute interview. In summary, the reviews we chose have: between 2 and 5 non-trivial partitions (median 2), between 2 and 13 changed files (median 5), between 7 and 52 diff-regions (median 25), and between 1 and 8 trivial partitions (median 2).

Each morning, we identified interview candidates by querying code review data for reviews that were created on the previous day. Then we sent out an email to review authors briefly describing our study and inviting the code change owner to participate by letting us visit them for approximately 20 minutes. In total, we sent out 43 invitations. Our response rate was 46%.

We interviewed 20 developers from 13 different projects. Participants have between 6 months and 13 years of experience at Microsoft (median 6 years). Among the 20 developers, 5 of them have the term "tester" in their job description.

C. Protocol

Once we had scheduled an interview, we went to the author's office. We started each interview by briefly explaining why we were there and the general purpose of our study. Before showing CLUSTERCHANGES, we made clear to developers that that **they were not under evaluation. Rather, our approach was**. After that, we showed our decomposition of their code change and explained how they could use it to inspect code changes. Then, we let them use CLUSTERCHANGES to look through their own change and drove the interview according to each participant's behavior. Most of the participants were quite communicative and, right after spending some time learning how to use the tool, they started explaining their change, the meaning of the partitions, if they made sense or not, if we missed any information, etc. For these participants, we tried to not interrupt them and let them drive the conversation during the interview. For the few participants that were less communicative, we asked questions to stimulate them to think aloud, such as, "Would you explain what this partition means?" or "Can you tell us the difference between these partitions?"

TABLE I
LIST OF QUESTIONS

RQ1	• For this review do you consider the decomposition we've created to be intuitive (correspond to your intention in having made the changes)? • For this review is the decomposition "correct"?
RQ2	• For this review, are the non-trivial partitions more important than trivial partitions? • For this review, are the trivial changes easier for reviewers to understand than the partitions? I.e., would they need less context to understand those changes?
RQ3	• For this review, do you think this decomposition would help your reviewers to understand your changes? Why? • Do you think it would help to structure the changes in a code review? • Would you like to use this tool for your next code review?

At the end of each session, we asked 7 detailed questions whose answers provide data to support conclusions for our three main research questions. Table I lists such questions and maps them to our main research questions. Surprisingly, while using CLUSTERCHANGES, all the 20 participants provided answers to some of these questions even before we asked them. We discuss these answers with more details in the next section.

The interviews lasted approximately 15-35 minutes. One participant did not allow us to record the interview; we had technical recording problems in three other interviews. We transcribed each recorded interview and had detailed notes from the others. At the end of each interview, although we had not offered it as an incentive, we compensated participants with a $5 gift card for lunch.

D. Coding Interviews

All four authors coded four transcribed interviews in a meeting together to define and calibrate the codes. Codes were created for each question. Besides that, sentences that could not be related to any of the questions but provided valuable feedback we coded as "other". For example, we coded the sentence "It would be nice to rename the partitions" as *other*; we found that many of these statements gave us valuable information about a missing feature of our tool. Then all four authors coded the same seven interviews individually. Since we used more than two raters for the transcriptions we used Fleiss' Kappa [16] on the individually coded responses to determine inter-rater reliability. The inter-rater reliability was 72%, which Landis and Koch consider to be substantial agreement [17]. The remaining interviews were each coded by an individual author.

In total, we coded 572 sentences extracted from our notes and audio transcripts: 366 (64%) were coded as *none*. These were sentences that do not provide any information to answer our questions. Of the remaining 206 (36%) sentences, 115 were coded to RQ1, 37 to RQ2, 46 to RQ3, and 33 to the *other* category. Note that 18 sentences were coded to more than one category.

```
public class MarkupHandler {
 private Regex listRe = new Regex("\s*\d+\.\s");
 private Regex listRe = new Regex("\s*#\.\s");

  public static Markup MarkupList(string text)
  {
    Match result = listRe.Match(text)
    ...
  }
}
public class MarkupTest {
  public bool TestList()
  {
    string testStr = "1. this is a test"
    string testStr = "#. this is a test"
    Markup testOutput = MarkupHandler.MarkupList(testStr);
    ...
  }
}
```

Fig. 6. Missed relationship between two diff-regions in a changeset.

E. Results and Analysis

1) RQ1: Of the 20 participants, 16 said that our non-trivial partitions were both correct and complete, i.e., the non-trivial partitions were indeed independent, the diff-regions within each partition were related and there were no missing conceptual groups. Of the remaining 4, 2 said that there should not have been more than one partition, while the other 2 said that while the non-trivial partitions were correct, some of the trivial partitions should have been grouped together in a new non-trivial partition.

Figure 6 shows an example of two trivial partitions that one participant thought should be in the same partition. In each pair of colored lines, the top red line had been removed and the bottom green line had been added. The developer changed both a string containing a regular expression (in `listRe`) and a string used to test the regular expression (`testStr`). The strings are related only through a method call to a method, `MarkupList`, which although defined in the changeset had not been modified. Such a relationship is not captured by CLUSTERCHANGES and the two diff-regions remain in separate partitions.

Interestingly, even though we did not explain the details of our analysis, one of the participants realized how difficult it would be to group such changes: " *In some sort of hypothetical perfect splitting that read my mind, there is one change in one line which was a variable changed (regular expression) that could be in a different partition. But I would not expect that, because it is difficult*" [P9].

The 2 participants that said their code changes shouldn't be partitioned were not clear about the reasons that led them to disagree with the partitions, but both mentioned that all the changes are somewhat related to a general concept, even if they are not related to each other. One of them was skeptical about our approach and stated at the beginning of the interview that "there is no reason [...] to commit unrelated changes" [P13].

We identified a pattern for the 16 cases in which we created the expected partitions. Even though all the diff-regions within non-trivial partitions were placed correctly and the partitions matched developers' reasoning about their own changes, 14 developers would have moved some (not more than 3) of the

trivial changes to one of the partitions. These were cases in which the analysis we employ could not identify the relationships. Most pertain to interface method usage and entities referenced by XML files or event-based framework calls. We discuss these cases with more details in Section VI.

Overall, developers were quite positive about our partitioning. All the aforementioned 16 developers confirmed that the partitions are indeed unrelated to each other and could be reviewed separately. In particular, three scenarios caught our attention.

In the first scenario, a developer mentioned that, before the interview, she realized that the commit could be split in two different changesets. She actually did split the changes and said that her commits match our two partitions: "*These were actually two different changes and I actually split them in two different things after this review*" [P7]. Six other participants mentioned that they should have split their changes according to our partitioning and committed them separately. For example, "*When I was writing this I was thinking to myself to separate this commit in two separate commits*" [P5].

In the second scenario, at the beginning of the interview, the participant was skeptical about our approach and mentioned that her changes were all related. However, when we presented the partitions to her, she seemed surprised: "*You are actually doing a good job. Actually you surprised me. It is quite intelligent*" [P6].

Third, according to a developer, we correctly split changes from two components into two different partitions. He mentioned that one of the components is responsible for the direct calls made to a service provider, while the other contained the callback methods which get called by the same service provider. In this case, the developer had changed the two components and added tests for each one. CLUSTERCHANGES created two partitions: one for each component and its tests. The developer mentioned that if he could, he would name partition 1 "Changes and tests for component 1" and partition 2 to "Changes and tests for component 2". In eight other interviews, developers mentioned we were able to separate different components into different partitions.

2) RQ2: As already mentioned, there were cases in which participants considered some of the trivial partitions incorrect: they should have been included in a non-trivial partition. Our questions about trivial partitions excluded those and focused on those that participants certified as being correct.

There were indeed several cases where the trivial method partitions identified by CLUSTERCHANGES were correct, i.e. they can be reviewed in isolation. The most common cases correspond to comments, log statements, blank lines, and internal logic changes (e.g. changes to if statements). For this reason, we had conjectured that trivial partitions were less important than non-trivial partitions. Surprisingly, 8 participants mentioned that trivial partitions are neither less or more important than non-trivial ones. Ten participants confirmed that they are less important than non-trivial ones, while the remaining 2 participants could not give an answer to our question. With respect to the 8 participants who said that trivial partitions are relatively equally important to the non-trivial ones, we found that they were somewhat not confident in assigning them less or more importance than other changes. As an example, a participant said "*I would not say that isolated changes are less important. It makes sense to separate them out. It is nice to see that they are not in that component, but they are still important*" [P11]. We also found a scenario in which a developer changed the internal logic of a method that doesn't directly impact its signature, but changed its semantics. The participant considered this change equally important as the ones within non-trivial partitions.

There was more consensus about whether trivial partitions were easier to understand than non-trivial partitions. Most participants (16) found them easier to understand, while 3 could not definitively answer the question, and 1 participant was unable to differentiate trivial partitions understanding from non-trivial ones. Overall, participants appreciated the fact that we separated trivial partitions. Among other reasons, participants mentioned the ability to prioritize non-trivial partitions, the fact that they need less context to understand trivial ones, and the fact that they are separated from the "meat" of the change.

3) RQ3: All participants were positive about the general concept of our approach in the sense that it can help reviewers to understand their changes. Some of them were quite optimistic, using terms such as "*perfect*" [P1] or "*amazing*" [P11], while others pointed that it could be helpful if we worked more on GUI details of our tool [P3,P7,P14]. We understand that participants tend to be positive during user studies and might be uncomfortable to point out negative aspects. While our intent was to gather observations from our qualitative study and not to perform a statistical analysis, we found some similarities among the answers. For example, 6 participants mentioned that, especially for large code changes, CLUSTERCHANGES could be helpful. During the interview, 1 of these 6 participants called the actual reviewer of the change to analyze the partitions. The participant started to explain the meaning of each partition and the reviewer agreed that it would help him to review the change as it was presented.

Also, 8 participants mentioned that our approach can be used to help assign reviewers to a specific partition based on expertise. Hence, reviewers could also prioritize partitions in which they could better contribute: "*[Decomposing changes] is useful because allow different reviewers with different purposes to focus on what they want*" [P12].

Finally, 5 developers mentioned that breaking changes into partitions would help reviewers to not spend too much time trying to understand the change among different contexts. Three mentioned the current view of CodeFlow and asked if we would combine both tools. CodeFlow currently organizes files in a review as a flat list: the diff-regions are organized only within the directories/files in which they occur.

> I think that could be really helpful. Usually when I am reviewing I open up CodeFlow and maybe there are hundreds of files and I start at the top and go in a directory order. Sometimes I find myself jumping around and saying there is this thing that was defined and I do not know where was defined and then I try to figure out what is going on and come back to

> the stack I was working on. So breaking things up into logical groups rather than just arbitrary directory order sounds like a big one for the reviewer. [P19]

In respect to CLUSTERCHANGES, two participants were unsure of its utility, one of them being the same participant (P13) who does not see any reason for a developer to commit unrelated changes. Nevertheless, she mentioned that the tool should allow developers to modify partitions, but "*the tool should be 95% correct or else I would not use it because it would be annoying*" [P13]. The other participant focused on criticizing GUI details of CLUSTERCHANGES. In particular, she mentioned she would like to see information on how we created partitions: "*The UI does not tell me how you created the partitions So, it is difficult for me to see its value*" [P17].

The remaining 18 developers (including two that disagreed with our partitioning) were positive about using CLUSTERCHANGES in their next changeset. Three of them indicated that they would like to use CLUSTERCHANGES even before committing the code: "*If I had a way to run this tool before I commit, I would have even considered splitting this partition 2 into a second commit*" [P4]. Three other developers asked for access to our prototype in order to use it for other changesets. One of them (P20) mentioned that we should contact another developer from her team because she has been committing several unrelated changes.

VI. DISCUSSION

We were pleasantly surprised that using the def-use and the use-use relationship did not have *any* false positives: i.e., there were no diff-regions that were incorrectly included in a partition. During our manual investigations and the user study, there was *never* a case where a non-trivial partition should actually have been split into multiple partitions. In retrospect, this makes sense: the def-use relationship is a fundamental relationship within a program and the use of a compiler makes it extremely unlikely to be mis-identified.

A. Missed Relations

As expected, there were many false negatives: diff-regions which should have been included in a partition, but were not. We categorize them by the difficulty of using a static analysis to find them.

1) Easy: There are many relationships that we know about and that we could automatically detect but which were not used. For instance, we did not create a def-use relationship for the definition of an interface method and its implementation or for an override (in a subclass) of a virtual method in a base class.

We found many changesets that consisted of annotating a large number of methods with a particular *custom attribute*. Custom attributes are user-defined tags that can appear on program elements, such as types, methods, and fields. For instance, many test methods may be added, each tagged with the `TestMethod` attribute, so that the test framework can automatically discover them. However, since we ignored use-use relationships if the definition was not present in the changeset, we did not group such methods, resulting in numerous trivial partitions. Although in general it is difficult to properly distinguish definitions that do not get fully resolved, this is common enough to warrant special treatment.

Also, we restricted ourselves to analyzing only C# files. It would be trivial to include Visual Basic files since Roslyn also is the compiler for that. We could also add other languages, given parsers for them.

2) Medium: We used only the after-files for creating the partitions, which means that code deletions are not represented.

Other use-use relationships besides those of custom attributes could be included as long as we do not conflate definitions that do not get fully resolved in the parsing.

Sometimes developers change names of entities to enforce a naming convention throughout the code; such changes do not share any semantic relationship. Clone detection [18] can be useful to identify relationships between such changes.

3) Hard: There are relationships due to the use of external tools and/or cross-language integration. For instance, there are XML files used as input to generate code: clearly the changed XML file should be grouped into the same partition as the code. XML files or other configuration files may indicate how to "connect" pieces of an application together at run time. While there is work on identifying cross-language dependencies [19], a multi-language analysis for the diverse set of languages and configuration formats that we encountered is beyond the scope of this work.

Code related by dynamic dependencies would be very difficult to analyze. For instance, many components use *callbacks* for two-way communication. That is, a component makes direct calls and also passes function pointers to allow the called component to call back into the calling component. This can be very difficult to precisely track, however some of these conventions can be heuristically recognized and added to our technique in the future.

Finally, there are always going to be changes that developers intend to be related, but which no static or dynamic analysis will ever find.

Adding a new relationship means that it is subject to the transitive closure of the partitioning algorithm. Alternatively, we found one particular strategy to address most of the developer's concerns with the trivial partitions. We can *post-process* the set of trivial partitions to merge them into any existing non-trivial partitions — creating one if needed — within the same enclosing class.

B. User involvement

It quickly became clear during the study that users would want to manually manipulate any proposed structuring as well as being able to *tag* or add a description to each partition. In fact, 6 developers mentioned this during the study. They also would like to have the partitions ordered so that the most important partition, the one with the "meat" of the change, would be first. Whenever any of these can not be done automatically, the developer should be provided with the ability to do it manually.

C. Threats to Validity

Clearly, our results are conditioned by several caveats. So far we have looked only at changesets from Microsoft. Even

within Microsoft, we have sampled only a very small subset of available changesets. Also, we have restricted ourselves to looking at diff-regions only within C# files. Finally, the small sample size and human factors involved mean that we are not able to achieve statistical significance, even within this scope.

VII. Related Work

Tao et al. present an empirical study to investigate the role of understanding code changes in software development [20]. Among other results, they observe that developers need to decompose changes in order to understand them — one of the primary goals of our approach.

To the best of our knowledge, there have been three prior approaches towards the problem of decomposing code changes. Kirinuki et al. [21] report a small experiment on identifying unrelated changes. They use the longest common subsequence algorithm to compare previous changes for the project to the one being committed. Herzig and Zeller [6] propose a heuristic-based algorithm to "untangle" changes based on information such as file distance and the call graph of a change. Similarly, Kawrykow and Robillard [4] apply a heuristic-based algorithm to analyze changesets; they focus on a statement-level analysis to find simple changes, such as, adding the keyword "this" when accessing fields.

Our work, in contrast, is concerned with developers' understanding of changes in the context of code reviews. Our exclusive use of *def-use* information, i.e., *semantic information* to partition changesets is novel and our validation with actual change owners enabled us to get the ground truth about independent changes being committed together.

Finally, there is a rich literature on techniques for understanding and summarizing changes. They range from change impact analysis [22], summarizing structural changes (automatically [23], [24] or interactively [25]) to the use of symbolic execution and program analysis [26], [27], [28]. These techniques are complementary to the problem of decomposing changes, and can in fact be used in conjunction to further summarize individual partitions suggested by our technique.

VIII. Conclusion

Changesets containing unrelated changes are not a rare event. This can negatively affect understanding: reviewers might need to switch contexts and manually separate unrelated changes to effectively review them. To tackle this problem, we designed an algorithm for partitioning the set of diff-regions present in a code review and implemented it in a tool which was used to perform both studies: quantitative and qualitative. We found that using a single relationship, that between the use of a type, method, or field and its definition, provided a useful decomposition with *no* false positives. We performed this initial validation with the author of the changes, rather than its consumers in order to see if the partitioning reflects the author's intent. Now that we have confidence in the accuracy of our partitioning, we can move on to do further studies with code reviewers.

Figure 7 illustrates one way that our work could fit into the development process. Initially, an author creates a changeset,

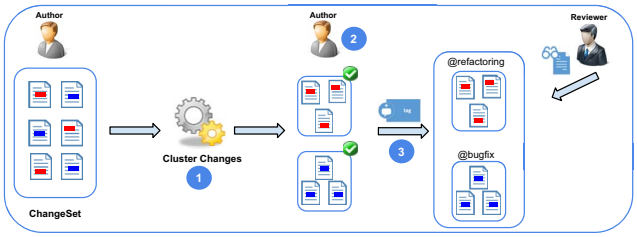

Fig. 7. Intended workflow for ClusterChanges.

which is then provided as input to ClusterChanges, which (step 1) decomposes it into independent partitions. Then, in step 2, the author can review the created partitions to ensure they are consistent with her understanding. In step 3, partitions can be ordered and tagged so that the reviewer sees the structured changeset. Steps 1 and 2 were already addressed by this work; future work includes adding more relationships without compromising precision (to step 1) and allowing developers to tag/order partitions (step 3). We also intend to conduct a broader quantitative study once the tool has been rewritten as an extension of the existing code review tool used at Microsoft.

Another possibility is in the context of other modern distributed revision control systems (e.g. GitHub) that have been promoting code review through lightweight mechanisms to annotate and discuss pull requests and commits. In particular, besides enabling general comments to summarize a commit, GitHub enables comments on code snippets, which can improve changes comprehension. In this context, ClusterChanges could be used as a mechanism to automatically identify where to include such comments/tags.

ClusterChanges is not coupled to a specific development environment, programming language, or application domain. Given the set of textual differences, which can be provided by any of the standard text differencing tools, we require only the ability to parse and semantically understand the source code. Many programming languages provide open APIs for retrieving such information.

It is our belief that there is a huge role for automated analysis and tooling for improving the code review process. Tools such as ClusterChanges or DiffCat [4] should be evaluated in a long-term study after they have been integrated into the "wild".

Acknowledgments

We thank Jack Tilford and Birendra Acharya from the CodeFlow team and Balaji Soundrarajan from the Roslyn team for all of their help. We would also like to thank Tom Ball, Yingnong Dang, Jacek Czerwonka and Andrew Begel for several interesting discussions about the problem and the tool. We would especially like to thank all of the study participants for their valuable time.

REFERENCES

[1] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[2] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: An exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 51:1–51:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393656

[3] P. C. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. M. German, "Contemporary peer review in action: Lessons from open source development," *Software, IEEE*, vol. 29, no. 6, pp. 56–61, 2012.

[4] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 351–360.

[5] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 5–18, 2012.

[6] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 121–130.

[7] [Online]. Available: https://roslyn.codeplex.com/

[8] [Online]. Available: https://roslyn.codeplex.com/SourceControl/changeset/8980c93e4d51eafbd088139afb6f262bbee40d33

[9] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 309–346, 2002.

[10] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, "Peer review on open source software projects: Parameters, statistical models, and theory," *ACM Transactions on Software Engineering and Methodology*, p. 34, 2014.

[11] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 202–212.

[12] [Online]. Available: http://www.microsoft.com/en-us/news/features/2012/jan12/01-05codeflow.aspx

[13] A. Hejlsberg, S. Wiltamuth, and P. Golde, *C# language specification*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[14] E. M. Rogers, *Diffusion of innovations*. The Free Press, 2003.

[15] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 332–341.

[16] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[17] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[18] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 187–196. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081737

[19] D. L. Moise and K. Wong, "Extracting and representing cross-language dependencies in diverse software systems," in *Reverse Engineering, 12th Working Conference on*. IEEE, 2005, pp. 10–pp.

[20] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 51.

[21] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *Proceedings of the 22Nd International Conference on Program Comprehension*, 2014, pp. 262–265.

[22] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," *SIGPLAN Not.*, vol. 39, no. 10, pp. 432–448, Oct. 2004. [Online]. Available: http://doi.acm.org/10.1145/1035292.1029012

[23] A. Loh and M. Kim, "Lsdiff: A program differencing tool to identify systematic structural differences," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 263–266. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810348

[24] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ser. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–13. [Online]. Available: http://dx.doi.org/10.1109/ASE.2004.5

[25] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive code review for systematic changes," in *Proceedings of 37th IEEE/ACM International Conference on Software Engineering*. IEEE, 2015.

[26] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859005

[27] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 226–237. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453131

[28] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare, "Differential static analysis: Opportunities, applications, and challenges," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 201–204. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882405