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

Interactive Code Review for Systematic Changes

Tianyi Zhang* Myoungkyu Song† Joseph Pinedo† Miryung Kim*
* University of California, Los Angeles
† University of Texas at Austin
tianyi.zhang@cs.ucla.edu, {mksong1117, joep24}@utexas.edu, miryung@cs.ucla.edu

***Abstract*—Developers often inspect a diff patch during peer code reviews. Diff patches show low-level program differences per file without summarizing *systematic changes*—similar, related changes to multiple contexts. We present CRITICS, an interactive approach for inspecting systematic changes. When a developer specifies code change within a diff patch, CRITICS allows developers to customize the change template by iteratively generalizing change content and context. By matching a generalized template against the codebase, it summarizes similar changes and detects potential mistakes. We evaluated CRITICS using two methods. First, we conducted a user study at Salesforce.com, where professional engineers used CRITICS to investigate diff patches authored by their own team. After using CRITICS, all six participants indicated that they would like CRITICS to be integrated into their current code review environment. This also attests to the fact that CRITICS scales to an industry-scale project and can be easily adopted by professional engineers. Second, we conducted a user study where twelve participants reviewed diff patches using CRITICS and Eclipse diff. The results show that human subjects using CRITICS answer questions about systematic changes 47.3% more correctly with 31.9% saving in time during code review tasks, in comparison to the baseline use of Eclipse diff. These results show that CRITICS should improve developer productivity in inspecting systematic changes during peer code reviews.**

I. INTRODUCTION

Code reviews are one of the most important quality assurance activities in software development [1]–[4]. According to a recent study, developers spend a significant amount of time and effort to comprehend code changes during peer code reviews [5].

When the information required to inspect code changes is distributed across multiple files, developers find it difficult to inspect a diff patch [6]. Suppose that an API gets modified in the latest release. All call sites using this API must be updated correctly [7]. Such edits tend to be systematic—involving similar but not identical edits to multiple locations. As another example, when programmers make changes to non-functional requirements such as security and persistence, these changes tend to be crosscutting edits to multiple locations [8].

Popular code review tools—PHABRICATOR,[1] GERRIT,[2] COLLABORATOR,[3] and CODEFLOW,[4]—all compute differences per file. This obliges the programmer to read changed lines file by file, even when those cross-file changes are done systematically with respect to the program's structure. Therefore, programmers are left to manually inspect individual edits to answer questions such as "what other code locations are changed similar to this change?" and "are there any other locations that are similar to this code but are not updated?"

Clone detection, code search, and matching approaches [9]–[12] can locate similar code fragments, but they do not directly work with diff patches. They do not summarize *similar edits* nor report *change anomalies* in a diff patch. These approaches also do not empower users to interactively investigate systematic changes, as they do not give users the control to iteratively generalize the search template. LSdiff [13] automatically summarizes coarse-grained structural differences, but naively enumerates all possible systematic change patterns as rules. This leads to the issue of poor scalability and a high rate of false positives. In Section VI, we discuss related work in detail.

This paper presents CRITICS, a new approach for interactively inspecting systematic changes during peer code reviews. Given a specified change, CRITICS creates a context-aware change template, extracting the surrounding control and data flow context. This approach models the template as Abstract Syntax Tree (AST) edits and allows reviewers to iteratively customize the template by parameterizing its content or by excluding certain statements. It then matches the customized template against the codebase to summarize systematic edits and locate potential inconsistent or missing edits. Our user study participants report that this interactive feature allows reviewers with little knowledge of a codebase to flexibly explore the diff patch with a desired pattern. They can incrementally refine the template and progressively search for systematic changes.

To demonstrate the benefits of our approach, we conducted two studies. First, professional software engineers at Salesforce.com used CRITICS to investigate diff patches authored by their own team. After they finished a hands-on trial of using CRITICS, we conducted semi-structured interviews with individual participants to understand the current challenges that they face during code reviews and whether and how CRITICS could help. The interviews helped us gather insights about the usability and benefit of CRITICS in an industry setting. All six participants said that they would like to have CRITICS integrated into their current code review environment, COLLABORATOR. CRITICS's feature to detect missing or inconsistent edits was valuable to their team and its interactive

[1] http://phabricator.org
[2] http://code.google.com/p/gerrit/
[3] http://smartbear.com/products/software-development/code-review/
[4] http://visualstudioextensions.vlasovstudio.com/2012/01/06/codeflow-code-review-tool-for-visual-studio/

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.33

usage was appropriate for novice developers to learn about the codebase.

In our second study, twelve participants reviewed diff patches using both CRITICS and Eclipse diff. This controlled experiment found that human subjects answered questions about systematic changes 47.3% more correctly and 31.9% faster on average with the assistance of CRITICS, in comparison to the baseline use of Eclipse diff. In addition to these two studies, we also compared the accuracy of CRITICS with our prior work LASE. The comparison found that in five out of six cases, interactively customizing a change template using CRITICS could achieve the same or even higher accuracy than LASE within a few iterations, showing the benefit of *interactive* template generation, as opposed to *fixed* template generation. In summary, our paper makes the following contributions.

- A new approach for interactively inspecting diff outputs. CRITICS provides a novel integration of program differencing and interactive code pattern search to locate and examine systematic changes. It is instantiated as an Eclipse plug-in and the tool is available online. Our replication package also includes tutorial materials, study tasks, and survey questions used for a lab study.[5]
- A user study with six professional software engineers at Salesforce.com. After using CRITICS to investigate the patches authored by their own team, the participants reported that CRITICS is helpful for inspecting system-wide changes and noticing oversight errors. All participants said they would like CRITICS to be integrated into their current code review environment. The study also shows that CRITICS is a mature tool that scales to an industry project and can be easily used by professional engineers.
- A lab study with twelve students at the University of Texas. Students answered questions about systematic changes more correctly and quickly using CRITICS than using Eclipse diff.

The rest of this paper is organized as follows. Section II illustrates our approach using a motivating example. Section III describes how CRITICS models code change as context-aware AST edits using static program analysis and how it matches a generalized change template using an adapted robust tree edit distance algorithm. Section IV-A describes a study with software engineers at Salesforce.com. Section IV-B describes our controlled experiment where participants complete code review tasks with and without CRITICS. Section V discusses the comparison between CRITICS and our prior work LASE and threats to validity. Section VI describes related work.

II. MOTIVATING SCENARIO

This section overviews CRITICS with a real world example drawn from Eclipse Standard Widget Toolkit (SWT) project. SWT is an open source widget toolkit with 400K lines of source code over 1000 files. This example is based on a diff patch at revision 13516, as shown in Figure 1. The patch is adapted and simplified for presentation purposes.

[5]https://sites.google.com/a/utexas.edu/critics/

Suppose Alice updates the program to use the new `sendEvent` API. Barry needs to review Alice's changes to ensure all locations using `sendEvent` are updated correctly and to check if there is any location that Alice forgot to change. The diff patch authored by Alice is over 450 lines of changes distributed across 42 different locations.

In order to find incorrect edits, Barry needs to inspect line level differences file by file. In particular, to identify missing updates, he must also inspect unchanged code as well, since the original diff patch does not show what did *not* change. The following shows how Barry may iteratively use CRITICS to inspect systematic changes and to detect potential missing or inconsistent updates.

Iteration 1. Suppose Barry first inspects changes in the `keyDownEvent` method in Figure 1(a). He wonders whether there are other methods that are changed similarly to `keyDownEvent`. So he selects the changed code in the diff patch. Given the selected change, CRITICS identifies the *change context*—unchanged, surrounding code relevant to these edits in terms of control and data dependences, which further serves as an anchor to locate missing updates during the searching process. So the default template generated by CRITICS consists of both the initial edit selection and the change context. Using the template, CRITICS locates code that matches the change context but is missing the update, shown in Figure 1(b).

Iteration 2. After examining the search result in the first iteration, Barry wants to explore further since he suspects other locations may use different identifier names. To match similar but not identical changes, CRITICS allows Barry to generalize the change template by parameterizing type, variable, and method names. So Barry generalizes the variable name `event` and searches again. This time, the location in Figure 1(c) is summarized although it uses a different variable name, `ev`.

Iteration 3. CRITICS includes the change context such as the `switch` and `case` statements from lines 3 to 5 in Figure 1(a) in the current template. Barry wonders if there are similar changes in different control-flow contexts such as a `for` loop or an `if-else` branch. He excludes the switch statement. Using the new refined template, CRITICS locates `buttonUpEvent` in Figure 1(d). This location uses an `if` statement instead of a `switch` statement. However, CRITICS flags this location as a potential inconsistent change, since Alice mistakenly swapped the two expressions, `EXPAND` and `COLLAPSE`. Such mistake is usually hard for the reviewer to detect during code inspection.

III. APPROACH

CRITICS provides a novel integration of program differencing and pattern-based interactive code search to help developers note inconsistent or missing changes during peer code reviews. It consists of the following three phases. Phase I takes a user specified change region and extracts the relevant context. Phase II allows developers to customize the change template by interactively generalizing its content. Phase III matches a template against the codebase to summarize similar changes and to detect potential anomalies. The reviewer can

```
1  int keyDownEvent (int wParam, int lParam) {
2  - ExpandItem item = items [focusIndex];
3    switch (wParam) {
4    case OS.VK_SPACE:
5    case OS.VK_RETURN:
6      Event event = new Event ();
7  -   event.item = item;
8  -   sendEvent(true, event);
9  +   event.item = focusItem;
10 +   sendEvent(focusItem.expanded ? COLLAPSE:EXPAND,
         event);
11 +   refreshItem(focusItem);
12     ...
13 }
```

(a) A changed region selected by Barry

```
1  int keyPressedEvent (int wParam, int lParam) {
2    ExpandItem item = items [focusIndex];
3    switch (wParam) {
4    case OS.VK_SPACE:
5    case OS.VK_RETURN:
6      Event event = new Event ();
7      event.item = item;
8      sendEvent(true, event);
9     ...
10 }
```

(b) Code location with exactly the same context but missing the update

```
1  int keyReleaseEvent (int wParam, int lParam) {
2  - ExpandItem item = items [focusIndex];
3    switch (wParam) {
4    case OS.GDK_RETURN:
5    case OS.GDK_SPACE:
6      Event ev = new Event ();
7  -   ev.item = item;
8  -   sendEvent(true, ev);
9  +   ev.item = focusItem;
10 +   sendEvent(focusItem.expanded ? COLLAPSE:EXPAND, ev);
11 +   refreshItem(focusItem);
12     ...
13 }
```

(c) A similar but not identical change using a different variable name, `ev`, instead of `event`

```
1  int buttonUpEvent (int wParam, int lParam) {
2  - ExpandItem item = items [focusIndex];
3    if (lParam == HOVER) {
4      Event bEvent = new Event ();
5  -   bEvent.item = item;
6  -   sendEvent(true, bEvent);
7  +   bEvent.item = focusItem;
8  +   sendEvent(focusItem.expanded ? EXPAND:COLLAPSE,
         bEvent);
9  +   refreshItem(focusItem);
10     ...
11 }
```

(d) Inconsistent change by mistakenly swapping two expressions, `EXPAND` and `COLLAPSE`

Fig. 1: Simplified examples of systematic changes, inconsistent changes, and missing updates. Code deletions are marked with '-' and additions are marked with '+'.

investigate the diff patch and achieve the desired result by iteratively refining the change template (Phase II) and searching change locations (Phase III).

A. Context Extraction

CRITICS parses the selected changed fragments into Abstract Syntax Tree (AST) edits and extracts the change context—surrounding unchanged code on which the selected edits are control and data dependent by performing static intra procedural slicing [14]. It selects all upstream dependent AST nodes based on a transitive relation within a method. The context could indicate where edits should be applied and serve as an anchor to locate systematic edits and to identify potential mistakes.

- Data dependence: AST node n_j is data dependent on node n_i, if node n_j uses a variable whose value is defined in node n_i. For example, by analyzing data dependencies between edits and surrounding unchanged code, CRITICS includes a variable declaration at line 6, whose variable `event` is referenced from the deleted line 7 in Figure 1(a).
- Control dependence: AST node n_j is control dependent on node n_i, if node n_j may or may not execute depending on a decision made by node n_i. For example, the `switch` and `case` statements from lines 3 to 5 in Figure 1(a) are included since the execution of the changed code depends on these control predicates.

B. Change Template Customization

CRITICS creates a default change template, including the initial selected fragments and the change context. A reviewer can customize the template by generalizing its content and to iteratively refine the template.

Parameterizing Identifiers. CRITICS allows a developer to parameterize type, variable, and method names, so that they can be regarded as equivalent to different identifiers during the matching process. Suppose that there is a statement `char[] data = foo()` in the change template. By parameterizing the variable name, `data`, CRITICS automatically propagates the parameterization to all statements referencing `data` and this statement can be matched to any other statements in the form of `char[] $V1 = foo()` where `$V1` represents any variable name.

Excluding Statements. CRITICS allows a user to exclude certain statements in the change template. Specially, by excluding contextual statements, CRITICS is able to find similar changes in multiple contexts. An excluded statement is mapped to a parameter `$EXCLUDED` in a generalized change template. For example, by converting `switch(x)` to `$EXCLUDED`, it can match `if(x == 1114)` in line 3 in Figure 1(d).

C. Matching and Anomaly Detection

Given a customized change template, CRITICS searches and summarizes systematic changes.

Tree Matching. To compute the similarity between different locations, CRITICS parses methods to abstract syntax trees and searches for similar subtrees by matching the template against other methods in the codebase. CRITICS extracts a *query tree* from an abstract diff template. This query tree is then matched

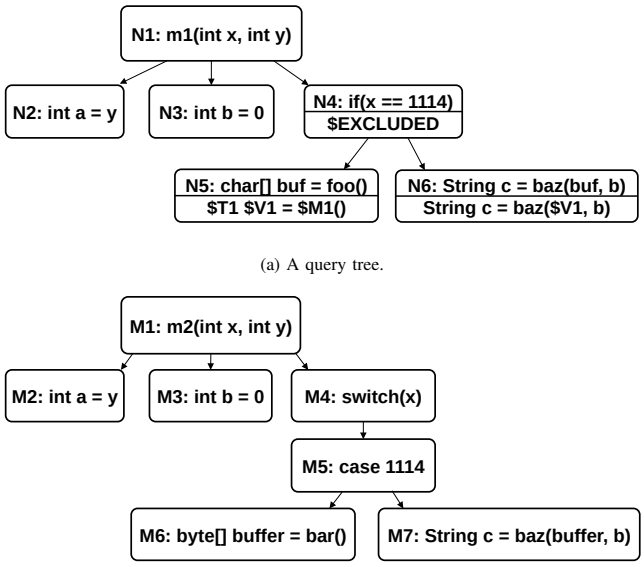
against a *target tree* of the rest of the codebase. CRITICS applies an efficient and worst-case optimal tree matching algorithm, Robust Tree Edit Distance (RTED) [15], which combines the strengths of Zhang's algorithm [16] and Demaine's algorithm [17]. Zhang's algorithm is efficient for trees with $O(log\ n)$ depth but has the worst-case time complexity $O(n^4)$. Demaine's algorithm has a better worst-case time complexity $O(n^3)$ but runs into the worst case frequently. RTED recursively decomposes the input trees into sub-forests, either removing the leftmost or the rightmost root node. It then computes the tree edit distance recursively by finding structural alignment. RTED then provides a list of matching node pairs with node edit operations that transform one tree into another.

The original RTED algorithm finds node-level alignment by calculating the minimum edit distance, producing many false positives. Therefore, CRITICS further computes token-level alignment between two matching AST nodes, as described in procedure `TokenMatch` in Algorithm 1. Given a list of token level matches, CRITICS checks whether a parameterized name is mapped to a concrete name. If the token labels are exactly the same, CRITICS considers them to be equivalent. While matching labels, we match the parameterized names such as `$V1` in the query tree with any concrete name in the target tree to support flexible matching. Suppose that RTED aligns two nodes: "`$T $V = $M(y)`" and "`int x = foo(y)`." CRITICS produces token level alignment: {("`$T`", '`int`"), ("`$V`", "`x`"), ("`$M(y)`", "`foo(y)`")}. Because these token level mappings are allowed via explicit parameterization in the previous step, CRITICS considers the aligned two nodes as identical and continues to check the next aligned pair.

As another adaptation to RTED, CRITICS checks whether there is an excluded node in the list of the aligned nodes computed by RTED. If RTED aligns an `$EXCLUDED` node with another node, CRITICS allows such matching, as described in line 11 in Algorithm 1. Suppose that CRITICS takes a node pair `switch(x)` in a query tree and `if(x == y)` in a target tree. If the node `switch(x)` is excluded by a user, CRITICS matches the two nodes. Figure 2 shows an example of node level and token level alignment.

CRITICS improves the performance of search by caching relevant data to reduce search load. CRITICS maps an identifier name to a set of source files using the identifier name and stores the mappings in a hash table. Before running RTED, CRITICS inspects each identifier name in a query tree and identifies a set of files using the same set of identifier names by looking up the hash table. Then, it only scans through the searched source files to avoid unnecessary matching.

Change Summarization and Anomaly Detection. Each template consists of a before state and an after state. The *before state* refers to code before edits. Conversely, the *after state* refers to the code after edits. Using the tree matching algorithm, CRITICS finds two sets of similar subtrees, matching the old and the new version respectively. If a method matches the before state, but not the after state, it implies that the programmer either made an incorrect edit or forgot to update the code. Similarly, if a method matches the after state but not the before state, CRITICS reports it as an anomaly as well, because similar edits are made to different contexts. We report two types of change anomalies: (1) *inconsistent changes*, where edits are applied but partially incorrect and (2) *missing updates*, where the required edits are completely missing. This feature of detecting change anomalies distinguishes CRITICS from other pattern mining and anomaly detection approaches that work with a single program version as opposed to a diff patch.



(a) A query tree.

(b) A target tree matched with the above query tree.

Fig. 2: RTED matches nodes, such as (N_1, M_1), (N_2, M_2), (N_3, M_3), (N_4, M_4), (N_5, M_6), and (N_6, M_7), and CCRITICS matches tokens in the labels of two matched nodes N_5 and M_6, such as ("`$T1`", "`byte[]`"), ("`$V1`", "`buffer`"), and ("`$M2`", "`bar`").

To summarize the matching systematic changes, CRITICS shows the individual matching locations and summarizes the similar edits using a change template derived from matching locations. Our implementation leverages ChangeDistiller [18] to compute AST edits, Crystal[6] for data and control flow program analysis, and RTED [15] for computing tree edit distance.

IV. EVALUATION

We evaluated CRITICS using two different methods. First, we conducted a user study with professional software engineers to understand how CRITICS can help them during code reviews. Engineers at Salesforce.com used CRITICS to investigate the real patches found in their version history, authored by their own team. This study emulates the realistic code review scenarios and solicits authentic feedback on the use of CRITICS in the real world. Second, we conducted a lab study at the University of Texas at Austin, where twelve participants investigated diff patches using both CRITICS and

[6]https://code.google.com/p/crystalsaf/

Algorithm 1: Searching similar subtrees.

```
Input  : Let AST be an Abstract Syntax Tree for a program.
Input  : Let QT be a query tree from a customized change template.
Output : Let MTs be a collection of the matched subtrees.

   Algorithm searchSimilarSubtrees(QT)
1  |  MTs := ∅;
2  |  foreach node_i from AST do
3  |  |  t := getSubtree(node_i);
   |  |  /* t is a target tree */
4  |  |  if RTED.match(QT, t) ≡ TRUE then
5  |  |  |  nodePairs := ∅;
6  |  |  |  nodePairs := nodePairs ∪ {(n_i, m_i) — n_i ∈ QT,
   |  |  |    m_i ∈ t, where (n_i, m_i) is a pair of nodes that
   |  |  |    RTED matches and aligns.};
7  |  |  |  if tokenMatch(nodePairs) ≡ TRUE then
8  |  |  |  |  MTs := MTs ∪ {t};
   |  |  |  end
   |  |  end
   |  end
9  |  return MTs;
   Procedure tokenMatch(nodePairs)
10 |  foreach pair_i ∈ nodePairs do
11 |  |  if pair_i.n is excluded then
12 |  |  |  continue;
   |  |  end
13 |  |  if match(pair_i.n.label, pair_i.m.label) ≡ FALSE then
   |  |  |  /* pair_i.n.label is different from pair_i.m.label. */
14 |  |  |  tokenPairs := ∅;
15 |  |  |  tokenPairs := tokenPairs ∪ {(t_j, u_j) —
   |  |  |    t_j ∈ pair_i.n.label, u_j ∈ pair_i.m.label, where
   |  |  |    (t_j, u_j) is a pair of a deleted token and an ins-
   |  |  |    erted token that CRITICS matches and aligns.};
16 |  |  |  if ∀t_j ∈ tokenPairs, t_j is parameterized then
17 |  |  |  |  continue;
   |  |  |  end
   |  |  |  else
18 |  |  |  |  return FALSE;
   |  |  |  end
   |  |  end
   |  end
19 |  return TRUE;
```

Eclipse diff and search. We selected Eclipse diff and search as a baseline, because they are default features in Eclipse. We cannot use existing clone-based search tools as a baseline, because they are not designed for inspecting diff patches and thus participants cannot use them without adapting the tools to inspect diff patches. These two evaluation methods (hands-on trials followed by semi-structured interviews and a controlled experiment using human subjects) complement each other by assessing the benefits of CRITICS both qualitatively and quantitatively.

A. User Study with Professional Developers at Salesforce

We recruited six participants from Salesforce.com. The participants included two software developers, three quality engineers, and a project manager from the same team. This team develops a platform for other teams to process and manage big data stored in the cloud. The participant names and the product name are anonymized.

All six participants had at least three years of Java development experience in industry. Five reported that they conduct code reviews at least weekly, using COLLABORATOR, a default code review tool at Salesforce.[7] Although one manager said he seldom reviews others' changes, we still interviewed him, because he could provide valuable feedback from a manager's perspective. Table I shows the demographic information about the six participants.

In terms of a study procedure, we first gave a presentation to introduce CRITICS's features to the participants. This presentation included a twenty-minute live demo of how to use CRITICS Eclipse plug-in. To get accurate and comprehensive feedback, participants were then asked to use CRITICS to investigate one of the four diff patches authored by their colleagues. This could simulate hands-on experience of using CRITICS in a real world setting, because the participants reviewed patches from their own system.

The four patches came directly from the version history of the Salesforce codebase that they currently work on. We selected the patches that include similar changes to multiple files, because the goal of CRITICS is to help developers examine systematic changes and find potential anomalies. Table II describes the associated commit log descriptions, the size of the patches in terms of changed lines of code, and the number of changed files from the actual version history. While we do not disclose the size of the Salesforce codebase for confidentiality, we report that CRITICS is a mature tool that scales to an industrial-scale project and the participants did not have any problems running CRITICS on their codebase and patches.

For individual participants, the hands-on use of CRITICS lasted about 20 to 30 minutes. Afterward, we conducted a semi-structured interview to solicit their feedback on the utility of CRITICS. The advantage of semi-structured interviews is that they are flexible enough to allow unforeseen types of information to be recorded [19]. The interviews were audio-recorded and transcribed later for further analysis. The interview questions are described below.

- What kind of challenges do you face when you conduct code reviews?
- In which situation, do you think CRITICS can help improve code reviews in your team?
- Would you like to have CRITICS be integrated with the code review tool you are currently using?
- How do you like or dislike CRITICS?

TABLE I: The demographic information of study participants

Subject	Role	Gender	Age	Java Experience	Code Review Frequency
1	Developers	Male	21-30	4	Weekly
2	Quality Engineer	Female	21-30	3	Weekly
3	Manager	Male	41-50	4	Seldom
4	Quality Engineer	Male	21-30	5	Weekly
5	Quality Engineer	Female	31-40	10	Weekly
6	Developers	Male	41-50	14	Daily

The interview results are organized by the questions raised during the interviews.

What kind of challenges do you face when you conduct peer code review? COLLABORATOR allows developers to

[7] http://smartbear.com/products/software-development/code-review/

TABLE II: Diff patches from Salesforce.com used for the study

No	Commit Description	Changed LOC	Num of Changed Files
1	Refactor test cases by moving bean maps to respective utils classes	743	22
2	Refactoring the API to get versioned field values by passing the version context as a parameter	943	34
3	Refactor tests by using try-with-resources statements to ensure the resource object is released after the program	484	10
4	Update common search tests by getting versioned test data	2224	12

upload, compare, and comment patches during code reviews. However, participants find it hard to review systematic changes, since COLLABORATOR only highlights differences on the uploaded patches, lacking the ability to identify underlying similar change patterns and pinpoint overlooked mistakes.

"Since REST APIs across different versions generally share similar code snippets, refactoring on versioned APIs often involves similar changes. Unfortunately, these changes are not always exactly the same, including subtle differences in different locations."

"It is hard for us to find missing updates, especially if the reviewer is not familiar with the codebase. So we totally depend on regression testing to check if there is any location we forgot to change, assuming it (the overlooked change) will break test cases. But, honestly, it does not work very well."

In which situations do you think CRITICS can help improve code reviews in your team? The participants mentioned that CRITICS can help them inspect system-wide changes, so that they do not need to manually walk through each changed location line by line. They also discussed that code reviews are usually assigned to senior developers and consequently piled up on them, since they are familiar with the codebase and are more likely to notice oversight errors. They believed that the interactive search process of CRITICS is an efficient method for novices to perform code reviews, unleashing the burdens of senior developers and spreading knowledge between team members.

"Because currently in our company, reviewers only ensure the logic correctness and coding style in uploaded patches. They barely check if there is any missing update, unless a reviewer is very familiar with the codebase and knows where the developer should update. That is also why we always assign code reviews to senior developers in the scrum team. The feature in your tool can free us from piling code review tasks on our senior developers, since it can do the inspection automatically without requiring deep knowledge of the codebase."

*"*CRITICS *would be helpful to check some API updates in our projects. For example, an API from one team is updated and the old API is deprecated. Since people only change the locations they know and reviewers usually do not intentionally check unchanged areas, we cannot guarantee all locations are updated as expected. So using* CRITICS *could help us find out all the locations that need to be updated in the early stage so that we do not need to wait for regression testing or even worse, the customer to tell us if there is any place that we updated incorrectly or forgot to update."*

Would you like to have CRITICS be integrated with your current code review tool? All six participants provided strong positive answers and believed that it would be useful to have CRITICS integrated to their code review tool, COLLABORATOR.

"Definitely. It makes sense to integrate it with COLLABORATOR*, since it will save a lot of time for code review."*

"Of course. Currently COLLABORATOR *only highlights the changed location in a very naive way. A feature like extracting and visualizing the change context can help us better understand the change itself as well as find some underlying change patterns between related changes."*

How do you like or dislike CRITICS? They thought CRITICS would be a good time saving tool for code reviews. Four participants replied that they like the search feature a lot because of its flexibility and interactivity compared with existing textual search. Two participants shared the UI is not very intuitive at a first glance and it took some time for them to grasp the UI.

"I like it since it is a great time saving tool for code review and I think its ability to find similar changes can be useful in our work."

"It will be more interesting if you can provide the change skeleton by default in the tree graph and enable users to expand a node to see details if they want to."

In summary, after using CRITICS to investigate their own team's patches, participants told us that CRITICS can improve developer productivity in code reviews and should be integrated to COLLABORATOR. Professional engineers encounter challenges when reviewing system-wide code changes. Currently, in their work environment, they barely have any reliable mechanism to guarantee all locations are correctly modified. Participants think CRITICS would help them detect unnoticed locations. Its interactive search feature also makes it easier for less experienced developers to use the tool. All participants strongly affirmed that they would like to have CRITICS's features as a part of their current code review environment.

B. Lab Study: Comparison with Eclipse Diff and Search

We conducted a user study with 12 participants to further evaluate the efficiency and usability of CRITICS.

- RQ1: How accurately does a reviewer locate similar changes with CRITICS in comparison to Eclipse diff and search?
- RQ2: How correctly can a reviewer detect change anomalies with CRITICS in comparison to Eclipse diff and search?
- RQ3: How much time can a reviewer save in a code review task when using CRITICS?

In this study, we used counterbalancing to control the order effect. Each participant carried out two different code review tasks, Patch 1 and Patch 2, once using CRITICS and once with Eclipse diff and search. In this section, we refer to the

TABLE III: The description of two patches and corresponding questions for code review tasks

	Versions	Change Description	Similar Change	Inconsistent Change	Missing Update	Size(LOC)
Patch1 (Simple)	JDT 9800 vs. JDT 9801	initiate a variable in a for loop instead of using a hashmap	getTrailingComments(ASTNode) getLeadingComments(ASTNode) getExtendedEnd(ASTNode)	getExtendedStartPosition(ASTNode)	getComments(ASTNode) getCommentsRange(ASTNode)	190
	Q1. Given the change in the method `getTrailingComments`, what other methods containing similar changes can you find? Count the number. A. 0 B. 1 C. 2 D. 3 or more Answer: C. `getLeadingComments` and `getExtendedEnd`. Q2. Which of the following methods contains inconsistent changes compared with the change in `getTrailingComments`? A. `storeTrailingComments` B. `getExtendedEnd` C. `getLeadingComments` D. `getExtendedStartPosition` Answer: It uses a wrong expression, `i<=this.leadingPtr` instead of `range==null && i<=this.trailingPtr`. Q3. How many methods share context similar to the change in `getTrailingComment` but missed the similar update? A. 0 B. 1 C. 2 D. 3 or more Answer: C. `getComments` and `getCommentsRange`.					
Patch2 (Complex)	JDT 10610 vs. JDT 10611	extract the logic of unicode traitement to a method	getNextChar() getNextCharAsDigit() getNextToken() ... 9 locations in total	getNextCharAsJavaIdentiferPart()	jumpOverMethodBody() getNextChar(char, char) getNextToken() ... 11 locations in total, all located in another file	680
	Q1. Given the change in the method `getNextChar`, what other methods containing similar changes can you find? A. 0 B. 1-5 C. 6-9 D. 10 or more Answer: C. `getNextCharAsDigit()` and `getNextToken()`, ... 8 methods in total Q2. Which of the following methods contains inconsistent change compared with the change in `getNextChar`? A. `getNextCharAsDigit` B. `getNextCharAsJavaIdentiferPart` C. `jumOverMethodBody` D. `jumpOverUnicodeWhiteSpace` Answer: It invokes a wrong method, `jumpOverMethodBody` instead of `getNextUnicodeChar`. Q3. How many methods share context similar to the change in `getNextChar` but missed the similar update? A. 0 B. 1-5 C. 6-9 D. 10 or more Answer: D.`jumpOverMethodBody`, `getNextToken` ... 11 methods in total.					

setting of Eclipse without CRITICS as *diff* in short. Patch 1 is a simple patch with 190 changed lines, and Patch 2 is a complex patch with 680 changed lines. Both the order of the assigned tools and the order of the assigned tasks were randomized to mitigate the learning effect. Table III describes the patches in terms of data source, patch size (LOC), change description as well as the number of methods that contain similar changes, inconsistent changes, and missing updates. It also describes the user study questions for each patch.

Four of the twelve participants were electrical and computer engineering undergraduate students and the other eight were all graduate students in software engineering. All participants had at least one year experience in using the Eclipse IDE. All but one participant had code review experience with diff tools such as Eclipse diff and Git/SVN diff. Participation was strictly voluntary with no compensation offered.

Prior to each study session, all participants were given a twenty-minute tutorial to learn how to use CRITICS. We gave them a live demo about inspecting a diff patch with CRITICS. The participants first answered two warm-up questions about the assigned diff patch. Then they were given a time to inspect a diff patch and answer three questions about systematic changes. All study tasks concern answering questions about similar changes, because the goal of CRITICS is to support inspection of similar changes, not all types of code changes. The questions required participants to identify methods that were changed similarly to a given location and to search for potential anomalous locations where similar edits were incorrect or completely missing. The three questions are described in Table III. Table IV shows the percentage of correct answers for each tool. To measure efficiency, we recorded the task completion time from when participants started to inspect source code to the time when they submitted the answers.

At the end of the user study, each participant was asked to complete a post study survey to evaluate their experience with CRITICS and Eclipse diff. First, they were asked to rate CRITICS and Eclipse diff separately on the aspects of *relevance*, *clarity*, and *usefulness* for locating similar changes and detecting anomalies. The survey also included open-ended questions to solicit qualitative feedback on how the users like or dislike CRITICS and the suggestions for improving CRITICS.

We recorded each user study session with screen capture software for further analysis. The participants were encouraged to speak out their thoughts, when conducting the code review tasks. Regarding multiple choice questions about the locations of systematic changes and anomalies, we asked the participants to explicitly identify individual method locations. The quotes from the study participants are extracted from these conversations. The data and transcript are anonymized.

Identifying Similar Changes. 10 out of 12 participants answered the question (Q1) about similarly changed locations correctly with CRITICS. 5 out of 12 did with Eclipse diff. We observe that Patch 2 usually required more template configuration and search iterations in CRITICS, and participants often stopped refining the template after two or three iterations. At that point, the customized template was still not general enough to find all similar changes. Using Eclipse *search* with a few keywords extracted from the patch often produced unstable search results depending on the choice of the keywords.

TABLE IV: Average correctness of participants' answers with and without CRITICS

	Q1 (Similar Changes)		Q2 (Inconsistent Changes)		Q3 (Missing Updates)		Time	
	CRITICS	diff	CRITICS	diff	CRITICS	diff	CRITICS	diff
Patch 1 (Simple)	100% (6/6)	50% (3/6)	100% (6/6)	33% (2/6)	100% (6/6)	83% (5/6)	0:18:32	0:20:24
Patch 2 (Complex)	67% (4/6)	50% (3/6)	100% (6/6)	83% (5/6)	83% (5/6)	33% (2/6)	0:20:20	0:30:53

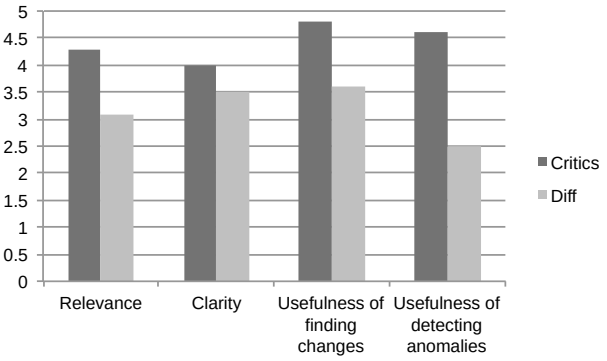


Fig. 3: Subjective Ratings for Critics and Eclipse Diff

Detecting Inconsistent Changes. All 12 participants found inconsistent change locations (Q2) correctly with CRITICS, as opposed to 7 out of 12 with Eclipse diff. We observed that CRITICS can help detecting inconsistent changes for both simple and complex patches.

Detecting Missing Updates. 11 out of 12 participants pinpointed all missing updates correctly with CRITICS, while only 4 out of 12 found missing updates with Eclipse diff. We observed that Eclipse diff was comparable to CRITICS, when inspecting a simple, small patch (Patch 1 with 160 line changes), while participants could locate a missed update more accurately when using CRITICS than Eclipse diff for the complex one (Patch 2 with 680 line changes).

Task Completion Time. Participants saved 6 minutes and 13 seconds with CRITICS on average, completing the tasks 31.9% faster than Eclipse diff. For the simple patch (Patch 1), CRITICS reduced task completion time by 9.2% on average, 48% at most. But for the complex patch (Patch 2), it reduced time by 34.2% on average, 60% at most. Consistent with the goal of CRITICS to support investigation of systematic changes, it was more useful when a patch consists of a large amount of scattered similar edits.

User Feedback. After completing the user study, participants completed a brief questionnaire to rate CRITICS and Eclipse diff based on their experience. Figure 3 shows the subjective ratings from the survey. CRITICS received higher ratings than Eclipse diff from all participants, including how *relevant* the found locations are, how *clear* the tool is and how *useful* the tool is in locating similar edits and detecting anomalies.

We also solicited qualitative feedback from the participants. They appreciated that CRITICS reduces the effort to investigate similar changes, especially in a large system. Using CRITICS, they only needed to inspect one location, as opposed to reading changed lines file by file without having the global context of what they are reviewing.

"I like the way it (Critics) automatically identifies possible similar edits that I could miss and detects anomalous changes. It really speeds up the code review process."

However, opinions were divided on the usability of CRITICS's UI. Three participants mentioned that its user interface is not intuitive and would benefit from extra visual options or instructions. They also suggested that CRITICS should provide configuration hints, e.g., which identifier should be generalized.

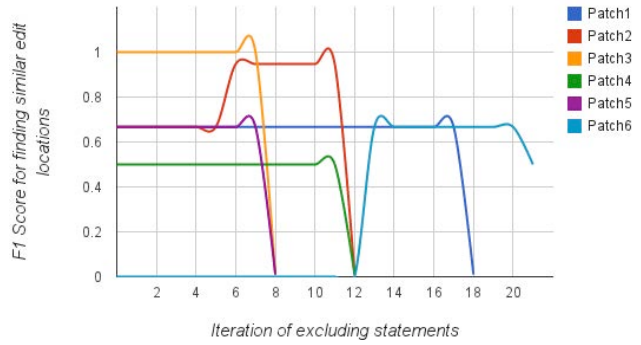
"It would be much more straightforward if CRITICS *gave some hints about which identifiers should be generalized. Currently it seems totally depends on developer's sense."*

In conclusion, participants were able to locate systematic changes and detect anomalies more correctly and quickly in code review tasks using CRITICS. They believed that CRITICS could complement the use of diff during inspection of systematic changes.

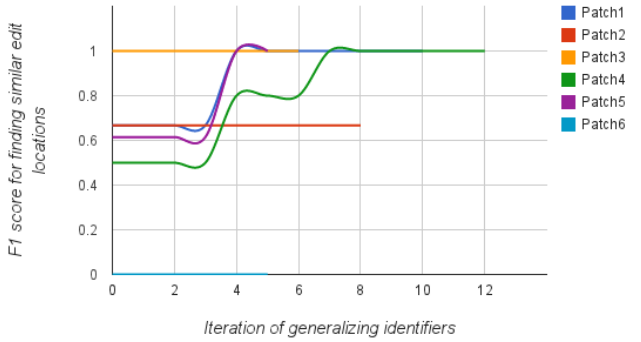
V. DISCUSSION

Comparison with LASE. Our prior work LASE [20] automates systematic editing by searching for locations and applying custom edits to individual locations. It requires multiple change examples as input to generate abstract transformation. It is challenging to directly compare CRITICS with LASE, because LASE's template generation requires multiple examples apriori and is fixed, while CRITICS is an interactive tool that a human can iteratively configure a template. Therefore, we simulate a human-driven template configuration process in CRITICS. From the lab study described in the previous section, we find that users follow common patterns while interactively generalizing the selected edit content and context. They usually generalize one identifier or statement at a time and re-run the search; if the search result degrades, they undo the generalization and try a different identifier or statement. In other words, their generalization strategy is similar to the typical *greedy search*. When generalizing identifiers, users first generalize a variable with a long name rather than a short one. When excluding statements, users prefer to exclude the context node on which a change is *control dependent*, such as `if` and `for`. We encode these patterns in a test script to simulate the interactive use of CRITICS. Then we compare LASE's accuracy with CRITICS's accuracy at each iteration.

The oracle test suite is drawn from the systematic edits identified by Park *et al.* [21] in Eclipse JDT and Eclipse SWT and consists six sets of systematic changes. In this test suite, the patch size ranges from 190 to 680 lines of edits. The number of locations with systematic changes ranges from three to ten locations. The first two changed locations are used as input examples to LASE, using the same approach



(a) F1 score for finding similar edit locations by excluding statements.



(b) F1 score for finding similar edit locations by parameterizing identifiers.

Fig. 4: F1 score on change template customization.

described in Meng et al. [20]. Figure 4 describes the accuracy variation in CRITICS's simulation. Figure 4a represents F_1 score (a harmonic mean of precision and recall) for finding similar changes while varying the number of excluded nodes. Figure 4b represents F_1 score for finding similar changes, while varying the number of generalized identifiers. Table V shows the comparison of search accuracy between CRITICS and LASE, including the iteration numbers till CRITICS achieves the best result and the average execution time for each iteration. In five out of six cases, CRITICS achieves the same or higher accuracy than LASE within a few iterations, showing the benefit of interactive template configuration as opposed to fixed template configuration.

TABLE V: Comparison between CRITICS and LASE

	CRITICS				LASE	
	Precision	Recall	Iteration	Time(sec)	Precision	Recall
Patch 1	1	1	4	1.66	1	1
Patch 2	1	0.9	6	8.95	0.92	0.75
Patch 3	1	1	0	13.52	1	1
Patch 4	1	1	7	71.98	1	0.33
Patch 5	1	1	4	6.86	1	1
Patch 6	1	0.33	3	1.47	1	1
Average	1	0.87	4	17.41	0.99	0.84

Threats to Validity. In terms of *construct validity*, in our lab study, we measured the correctness of the answers and the time taken to answer questions to measure developer productivity in inspecting systematic changes. Other measures such as the number of potential bugs detected could be used to measure developer productivity for peer code reviews. In our user study, we used both large and small patches and counterbalanced the order and task assignment to mitigate learning effect. Because the goal of CRITICS is to help inspect systematic changes, the questions mainly pertain to the questions about similar scattered changes, not general program comprehension questions.

In terms of *external validity*, in our lab study, twelve student developers were not familiar with Eclipse JDT, from where patches are drawn. The lab study may not generalize to professional developers who are familiar with their codebase. To overcome this limitation, in our user study at Salesforce.com, six engineers investigated the patches from their own system.

The study at Salesforce is a qualitative study based on six interviews. Because of the qualitative nature of the study, we do not make any quantitative statements about how much productivity gain CRITICS can provide in comparison to their current code review tool, COLLABORATOR. The study was conducted only in one company. We do not believe this is a significant limitation because the background of the participants and the code review practice at Salesforce.com are similar to other large software companies. In the comparison with LASE, our test suite of systematic changes includes only patches from Park et al.'s data set [21] and may not generalize to projects other than Eclipse JDT and SWT.

To mitigate *internal validity*, in our lab study, before the participants started the task, we asked them to inspect the change example first and answer two questions to calibrate their understanding. The first question required them to choose true or false about detailed statements about the change to ensure that they have carefully inspected the example. The second question required them to identify changes similar to the given example. The warm up questions helped them better understand change similarity.

VI. RELATED WORK

Modern Code Reviews and Code Change Comprehension. Rigby et al. conduct an investigation into code review practices in open source development and find that developers can understand small, logical, coherent units of code changes better rather than large, unrelated changes [22]. Rigby et al. also find general principles of code review practices and the benefit of code review for knowledge sharing among developers [23]. Bacchelli and Bird study modern code review practices and find that a key challenge is lacking tool support for code change comprehension [24]. Tao et al. also study the challenges that developers face when they comprehend code changes and find that modern code review tools must support the capability to divide a large chunk of code changes into sub logical groupings and to filter non-essential changes [5]. These findings motivate CRITICS. Barnett et al. also design a static analysis technique to help developers to understand code changes during code reviews [25]. They decompose composite changes and cluster relevant ones using dependence

analysis. While our work shares the same goal of assisting code change comprehension, our work focuses on inspecting similar changes and detecting anomalies.

Code Search and Anomaly Detection. Chang et al. use graph mining to detect implicit programming rules from system dependence graphs and use these rules to find violations [26]. Wang et al. propose a dependence-based code search technique [11]. Their query language is capable of capturing control and data dependences. Their work is later enhanced with semantic topic modeling [27].

Instant code search techniques take a code example as input and return other similar code examples on demand [28, 29]. In particular, CBCD detects recurring clone related bugs by finding similar ASTs with program dependence information. These techniques are based on code clone mining [10, 30]–[33].

Several approaches detect inconsistencies among code clones. CP-Miner [9], SecureSync [34] and Jiang et al.'s work [35] find cloning-related inconsistencies by searching for duplicated code. SPA categorizes four common types of porting inconsistencies and detects discrepancies between the surrounding context of systematic changes [36]. Lo et al. actively incorporate incremental user feedback to continually refine clone-based anomaly reports and selectively present clone-based bugs [37]. Lin et al. detect differences across multiple clone instances to help with clone comprehension [38].

CRITICS differs from these code search and clone detection techniques in two ways. First, CRITICS directly target investigation of diff patches as opposed to a single program version. Due to these differences, in our lab study, we could not directly compare with existing clone-based search tools, because these tools are not designed for inspecting diff patches. Second, CRITICS allows users to *interactively* generalize a search template to provide flexibility.

Systematic Change Inference. RefFinder finds the types and locations of refactoring edits using pre-defined refactoring rules [39]. LSdiff infers systematic change patterns at a coarse granularity and summarizes them as logic rules [13]. It also detects potential inconsistencies that violate the systematic change patterns. LSdiff supports only coarse-grained analysis at the level of method calls and field accesses. It also does not leverage any human input and therefore it often discovers a large amount of rules in an inefficient top-down manner, while discarding most of them in a post-processing step. In contrast, CRITICS allows a user to interactively refine an abstract diff template to be used. Chianti groups related code changes at a coarse granularity using predefined rules. To our knowledge, none of these were evaluated with a user study unlike ours [40].

Automating Systematic Edits. Andersen et al. propose generic patch inference, which takes a set of example program transformations and generate a generic patch to automate similar edits to multiple locations [41]. LibSync recommends similar API usage adaptation patterns but does not automate complex edits [42]. SYDIT takes a single code change as input and replicates similar change to a user provided target [43]. LASE uses multiple change examples to automate similar changes to multiple code fragments [20]. However, these approaches do not provide users interactivity and flexibility to tune change templates. In the comparison between CRITICS and our prior work LASE (Section V), we show that this interactive feature of CRITICS allows users to achieve high accuracy within a few iterations.

This paper makes a unique contribution of conducting two rigorous user studies to assess the effectiveness of CRITICS during peer code reviews. The above prior work on automated systematic editing was not evaluated using user studies. The demonstration paper on CRITICS describes the user interface features of CRITICS and does not include any technical algorithm description and user studies [44].

VII. CONCLUSION

We present CRITICS, a novel approach for searching systematic changes and detecting potential anomalies during peer code reviews. It takes as input a selected sub-region of diff patch and allows a reviewer to customize a change template by interactively generalizing the AST edits and surrounding context. A user study at Salesforce.com shows that after using CRITICS to investigate their own team's patches, all six participants said they would like CRITICS to be integrated into their current code review environment, COLLABORATOR. The participants saw the benefit of using CRITICS in detecting missing or inconsistent updates, which is difficult to find using their current code review tool. They also thought its *interactive* template configuration and search feature was appropriate for developers without deep knowledge of the codebase to gradually learn about the codebase. The lab study shows that human subjects could answer questions about systematic changes 47.3% more correctly and 31.9% faster on average with the assistance of CRITICS in comparison to the baseline use of Eclipse diff. These results indicate that CRITICS should help developers comprehend an underlying latent structure of a diff patch and locate missed or inconsistent updates during peer code reviews.

ACKNOWLEDGMENT

The authors would like to thank anonymous participants from Salesforce.com and University of Texas at Austin for their participation in the user study and their valuable insights and feedback. This work was supported in part by National Science Foundation under grants CCF-1117902, CCF-1149391, SHF-0910919, CNS-1239498, and a Google Faculty Award.

REFERENCES

[1] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: An effective verification process," *IEEE software*, vol. 6, no. 3, pp. 31–36, 1989.

[2] A. Dunsmore, M. Roper, and M. Wood, "Practical code inspection techniques for object-oriented systems: an experimental comparison," *IEEE software*, vol. 20, no. 4, pp. 21–29, 2003.

[3] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 38, no. 2-3, pp. 258–287, 1999, code inspection, checklist.

[4] K. E. Weigers, *Peer reviews in software: a practical guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] 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.

[6] A. Dunsmore, M. Roper, and M. Wood, "Object-oriented inspection in the face of delocalisation," in *ICSE '00: Proceedings of the 22nd International Conference on Software engineering*. New York, NY, USA: ACM, 2000, pp. 467–476, code inspection, code review, object-oriented, delocalized.

[7] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[8] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 107–119.

[9] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code." in *OSDI*, 2004, pp. 289–302.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[11] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 457–466. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859091

[12] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 365–383.

[13] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319.

[14] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/53990.53994

[15] M. Pawlik and N. Augsten, "RTED: a robust algorithm for the tree edit distance," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 334–345, 2011.

[16] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[17] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann, "An optimal decomposition algorithm for tree edit distance," in *Automata, languages and programming*. Springer, 2007, pp. 146–157.

[18] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling—tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, p. 18, November 2007.

[19] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Trans. Softw. Eng.*, vol. 25, no. 4, pp. 557–572, Jul. 1999. [Online]. Available: http://dx.doi.org/10.1109/32.799955

[20] N. Meng, M. Kim, and K. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *ICSE '13: Proceedings of 35th IEEE/ACM International Conference on Software Engineering (Accepted)*. IEEE Society, 2013, p. 10 pages.

[21] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *MSR '12: The 9th IEEE Working Conference on Mining Software Repositories*, 2012, pp. 40–49.

[22] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," in *ICSE '08: Proceedings of the 30th International Conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 541–550. [Online]. Available: http://flosshub.org/sites/flosshub.org/files/p541-rigby.pdf

[23] 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.

[24] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 712–721.

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

[26] R.-Y. Chang, A. Podgurski, and J. Yang, "Discovering neglected conditions in software by mining dependence graphs," *Software Engineering, IEEE Transactions on*, vol. 34, no. 5, pp. 579–596, Sept 2008.

[27] S. Wang, D. Lo, and L. Jiang, "Code search via topic-enriched dependence graph matching," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 119–123.

[28] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Instant code clone search," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 167–176.

[29] J. Li and M. D. Ernst, "CBCD: Cloned buggy code detector," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 310–320.

[30] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 48–62.

[31] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105.

[32] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 383–392.

[33] X. Yan and J. Han, "Closegraph: mining closed frequent graph patterns," in *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge discovery and data mining*. ACM, 2003, pp. 286–295.

[34] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 447–456. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859089

[35] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 55–64.

[36] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 367–377.

[37] D. Lo, L. Jiang, A. Budi *et al.*, "Active refinement of clone anomaly reports," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 397–407.

[38] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones." in *ICSE*, 2014, pp. 164–174.

[39] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1 –10.

[40] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of Java programs," in *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2004, pp. 432–448.

[41] J. Andersen, "Semantic patch inference," Ph.D. Disseratation, University of Copenhagen, Copenhagen, Nov. 2009, adviser-Julia L. Lawall.

[42] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *Proceedings of the ACM International Conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 302–321. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869486

[43] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 329–342. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993537

[44] T. Zhang, M. Song, and M. Kim, "Critics: An interactive code review tool for searching and inspecting systematic changes," in *International Symposium on Foundations of Software Engineering, Research Demonstration Track (to appear in FSE 2014)*, 2014, p. 4 pages.