Critics: An Interactive Code Review Tool for Searching and Inspecting Systematic Changes

Tianyi Zhang[*] Myoungkyu Song[†] Miryung Kim[*]
[*]University of California, Los Angeles, USA [†]University of Texas at Austin, USA
tianyi.zhang@cs.ucla.edu, mksong1117@utexas.edu, miryung@cs.ucla.edu

ABSTRACT

During peer code reviews, developers often examine program differences. When using existing program differencing tools, it is difficult for developers to inspect systematic changes—similar, related changes that are scattered across multiple files. Developers cannot easily answer questions such as "what other code locations changed similar to this change?" and "are there any other locations that are similar to this code but are not updated?" In this paper, we demonstrate CRITICS, an Eclipse plug-in that assists developers in inspecting systematic changes. It (1) allows developers to customize a context-aware change template, (2) searches for systematic changes using the template, and (3) detects missing or inconsistent edits. Developers can interactively refine the customized change template to see corresponding search results. CRITICS has potential to improve developer productivity in inspecting large, scattered edits during code reviews. The tool's demonstration video is available at `https://www.youtube.com/watch?v=F2D7t_Z5rhk`

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—Integrated Environments

General Terms

Design, Experimentation, and Measurement

Keywords

Software evolution, program differencing, code reviews

1. INTRODUCTION

Peer code reviews are widely used quality assurance activities in software development [1, 2]. During code reviews, developers spend a significant amount of time and effort to comprehend and inspect code changes [3]. When code changes involve *systematic edits*—similar but not identical changes to multiple contexts, developers may find it difficult to ensure that all locations are updated correctly and that there are no missing updates.

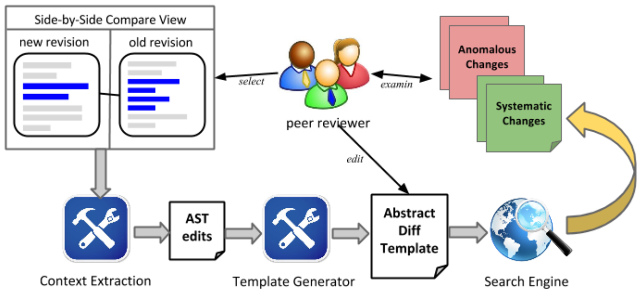


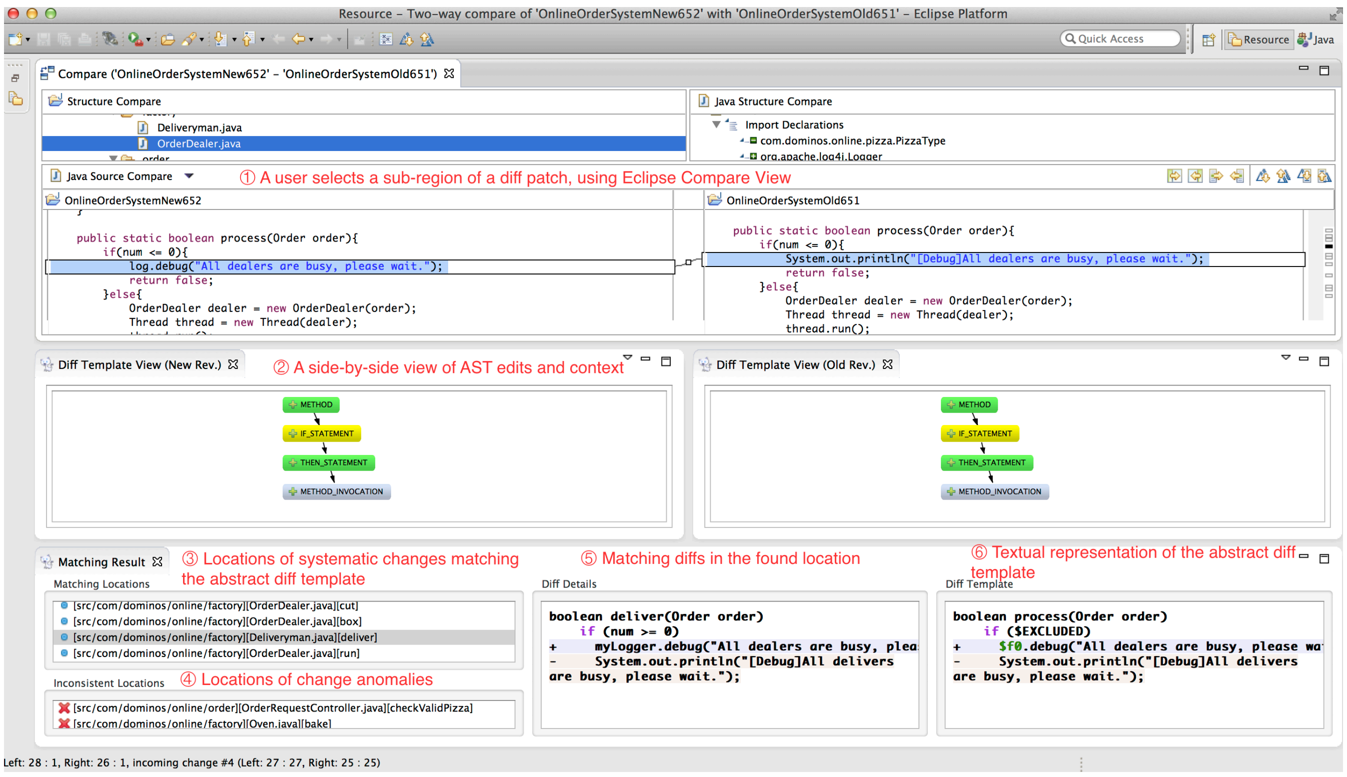
Figure 1: An overview of CRITICS Eclipse plug-in

The ubiquitous program differencing tool *diff* computes line-level differences per file, obliging the programmer to read changed lines file by file, even when those cross-file changes were done systematically with respect to the program's structure. Similarly, other differencing tools that work at different levels of abstraction (e.g., abstract syntax trees [4] and control flow graphs [5]) do not help developers grasp the underlying latent structure of systematic changes nor identify anomalies that violate systematic change patterns. Therefore, programmers are left to manually inspect individual code edits.

This paper introduces CRITICS, a novel interactive approach to investigate diff outputs. It combines program differencing with interactive code pattern search in order to locate and summarize systematic changes. Figure 1 gives an overview of CRITICS. It takes as input the old and new program versions. By selecting a certain region of a diff patch in Eclipse Compare (a viewer for investigating line-level differences), a user can specify the change he or she wants to search further. CRITICS extracts *change context*—surrounding statements that the selected change may depend on. It allows a reviewer to interactively generalize the change template to enable approximate matching. For example, a reviewer may parameterize type, variable, or method names in the AST edits or exclude certain statements in the surrounding contexts. Using the customized template, a user can then search the entire program and examine the matched diff regions. During this process, CRITICS also detects inconsistent or missing edits that violate the systematic change patterns.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2661675


Resource – Two-way compare of 'OnlineOrderSystemNew652' with 'OnlineOrderSystemOld651' – Eclipse Platform
Quick Access
Resource
Java
Compare ('OnlineOrderSystemNew652' – 'OnlineOrderSystemOld651')
Structure Compare
Java Structure Compare
Java Source Compare
① A user selects a sub-region of a diff patch, using Eclipse Compare View
② A side-by-side view of AST edits and context
Diff Template View (New Rev.)
Diff Template View (Old Rev.)
METHOD
IF_STATEMENT
THEN_STATEMENT
METHOD_INVOCATION
Matching Result
③ Locations of systematic changes matching the abstract diff template
④ Locations of change anomalies
⑤ Matching diffs in the found location
⑥ Textual representation of the abstract diff template
Matching Locations
Inconsistent Locations
Diff Details
Diff Template

```
Deliveryman.java
OrderDealer.java
```

```
Import Declarations
  com.dominos.online.pizza.PizzaType
  org.apache.log4j.Logger
```

OnlineOrderSystemNew652

```
    public static boolean process(Order order){
        if(num <= 0){
            log.debug("All dealers are busy, please wait.");
            return false;
        }else{
            OrderDealer dealer = new OrderDealer(order);
            Thread thread = new Thread(dealer);
```

OnlineOrderSystemOld651

```
    public static boolean process(Order order){
        if(num <= 0){
            System.out.println("[Debug]All dealers are busy, please wait.");
            return false;
        }else{
            OrderDealer dealer = new OrderDealer(order);
            Thread thread = new Thread(dealer);
            thread.run();
```

```
[src/com/dominos/online/factory][OrderDealer.java][cut]
[src/com/dominos/online/factory][OrderDealer.java][box]
[src/com/dominos/online/factory][Deliveryman.java][deliver]
[src/com/dominos/online/factory][OrderDealer.java][run]
```

```
[src/com/dominos/online/order][OrderRequestController.java][checkValidPizza]
[src/com/dominos/online/factory][Oven.java][bake]
```

```
boolean deliver(Order order)
    if (num >= 0)
+       myLogger.debug("All dealers are busy, plea
-       System.out.println("[Debug]All delivers
are busy, please wait.");
```

```
boolean process(Order order)
    if ($EXCLUDED)
+       $f0.debug("All dealers are busy, please wa
-       System.out.println("[Debug]All delivers
are busy, please wait.");
```

```
Left: 28 : 1, Right: 26 : 1, incoming change #4 (Left: 27 : 27, Right: 25 : 25)
```

Figure 2: A screen snapshot of CRITICS Eclipse Plug-in and its features.

This paper's main contribution is to describe the features of CRITICS from a user's perspective. The Eclipse plug-in is available for download at `https://sites.google.com/a/utexas.edu/critics/`. The detailed algorithm and evaluation is described in our separate technical report [6].

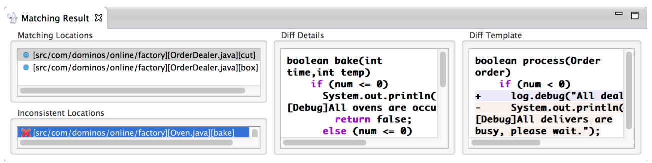
2. MOTIVATING EXAMPLES AND TOOL FEATURES

This section presents CRITICS features with a motivating example. Suppose Barry and Alice are developing an online sales system for a pizza store. Suppose Barry is conducting a code review of a diff patch authored by Alice. The check-in message says, "update to use log4j for log management." Barry wants to check that Alice refactored all locations that print log messages to the console, so that these locations can use Apache log4j APIs instead. Without CRITICS, Barry must inspect each changed location one after another because existing *diff* displays only line-level differences per file. Furthermore, he has to search the entire codebase to ensure that Alice did not miss anything, because missing updates do not appear in the diff patch. This manual reviewing process not only requires the deep knowledge of the codebase but also is tedious and error prone.

Eclipse Compare View. Barry first inspects a region of changed code in method `OrderDealer` using the **Eclipse Compare View** (see ① in Figure 2). In this method, Alice updated the original `System.out.println` statement with log4j API `debug`. Barry checked this change and now he wonders if Alice updated all other similar locations correctly. To automate this process, he selects the changed code in both old and new versions and then provides the selected code as an input to CRITICS for further search.

Diff Template View. CRITICS models and visualizes the selected change as an abstract *diff template*. Barry can review and customize the template in a side-by-side **Diff Template View** (see ② in Figure 2). The diff template serves as a change pattern for searching similar edits. In the template, CRITICS includes *change context*—unchanged, surrounding statements relevant to the selected change in the template. In this example, CRITICS includes an `if` statement because the changed code is executed only if the `if` condition is satisfied. Nodes with light blue color refer to statements that the user originally selected. Yellow nodes represent statements that the change is control dependent on. Green nodes represent parent nodes of the selected code. Orange nodes represent statements that the change is data dependent on. Barry can preview the textual template in the **Textual Diff Template View** (see ⑥ in Figure 2).

Matching Result View. Based on the diff template, CRITICS identifies similar changes and locates anomalies. It reports them in the **Matching Result View**. If the syntactic differences match the diff template in both the old and new versions, CRITICS summarizes this location as *systematic change* in **Matching Locations** (see ③ in Figure 2). If the target code matches the old version but does not match the new version, such unpairing is reported as *anomalies* in **Inconsistent Locations** (see ④ in Figure 2). In the first attempt, Barry does not edit the template and searches matching locations. CRITICS summarizes locations that are identical to the template and reports those violations against the template, as shown in Figure 3. The `bake` method is detected as a possible anomaly, because it shares the same context in the old revision but Alice did not update this



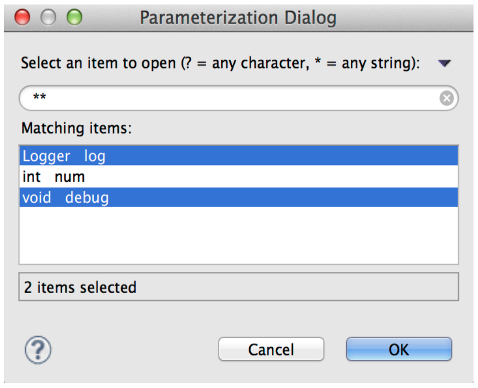
[src/com/dominos/online/factory][OrderDealer.java][cut]
[src/com/dominos/online/factory][OrderDealer.java][box]

[src/com/dominos/online/factory][Oven.java][bake]

```
boolean bake(int
time,int temp)
    if (num <= 0)
      System.out.println(
[Debug]All ovens are occu
        return false;
      else (num <= 0)
```

```
boolean process(Order
order)
    if (num < 0)
+      log.debug("All deal
-      System.out.println(
[Debug]All delivers are
busy, please wait.");
```

Figure 3: Searching results and diff details.



```
**
```

Logger log
int num
void debug

2 items selected

Figure 4: Dialog for parameterizing type, method, and identifier names in an abstract diff template.

method.

Diff Details View. When Barry clicks an individual change location in the **Matching Result View**, the corresponding differences are presented in the **Diff Details View** (see ⑤ in Figure 2). Changed code is highlighted in this view, and inserted code is marked with '+', while deletion is marked with '-'. By comparing contents in the **Diff Details View** and those in the **Diff Template View**, Barry can quickly figure out why each location is identified as a similar change or reported as an anomaly, without navigating different files back and forth. If he wants to drill down into the source code and double clicks a location, CRITICS redirects him to the change location in the **Eclipse Compare View**.

Template Refinement and Search. To match similar but not identical changes, Barry can generalize identifiers in the diff template, including type, variable, and method names, as shown in Figure 4. When he generalizes variable `log` and method `debug`, CRITICS locates method `deliver` which uses variable `myLogger` and invokes method `error` instead of `log` and `debug`. Barry further excludes a context node, an `if` statement, in the template by double clicking the node. This time CRITICS reports a change within a `while` loop in method `run` in the **Matching Result View** in Figure 2. Barry can progressively explore the diff patch and search for similar changes till he is confident that all locations are updated correctly.

3. IMPLEMENTATION

This section describes the implementation details of CRITICS. CRITICS is implemented as an Eclipse Plug-in and consists of four components: (1) selection of a diff region, (2) context extraction with dependency analysis, (3) template generation, and (4) a search engine for similar changes and potential mistakes, as shown in Figure 1.

Change Selection. Currently, CRITICS is integrated with Eclipse Compare, which displays line-level differences per file. CRITICS captures the selected code fragment in a side-by-side view and pipes it to the next phase of an abstract diff template extraction.

Context Extraction. Given a specified change, CRITICS computes change contexts from both the old and new versions using data, control, and containment dependence analysis. CRITICS first converts source code to abstract syntax tree (AST) with ASTParser.[1] ASTParser resolves binding information for each reference during the parsing process. A change is control dependent on a statement if the code in the change may or may not execute based on the decision made by the statement. CRITICS creates a control flow graph (CFG) using the Crystal static analysis framework.[2] By traversing a CFG, CRITICS identifies nodes that strictly dominate the changed nodes. Containment dependence captures the structure of the changed location, i.e., a parent and child relationship, which is computed by traversing abstract syntax trees.

Template Generation. The diff template is composed of the selected change and the extracted context. For the convenience of reviewing and editing template, CRITICS visualizes the template as a graph, using the Eclipse Zest visualization framework.[3] When an identifier (e.g., a variable, type, or method name) is generalized, it is replaced with a parameterized name, such as `$f`, `$t` and `$m`, in the template. An excluded context node is also marked as `$EXCLUDED`, as shown in ⑥ in Figure 2. These wildcard-like labels help CRITICS recognize the generalized content and support fuzzy matching in the following search process.

Template Matching and Search. Given a diff template, CRITICS extracts two AST query trees, a *before state* tree containing statements from the old version and an *after state* tree containing statements from the new version, and searches respectively against the old and new program. To do that, CRITICS parses methods and computes similarity between the query and target trees with an adapted Robust Tree Edit Distance (RTED) [7] algorithm. It first aligns tree nodes and pairs excluded nodes with any nodes in the target tree. For paired nodes, it tokenizes node labels to compute label similarity. A generalized identifier such as `$f1` is considered equivalent with any concrete element of the same type. If a changed location in a method matches both the before state tree and the after state tree, it is summarized as a systematic change. However, if it matches the before state but not the after state tree (or vice versa), it is reported as a potential anomaly textemdash a missing or inconsistent update.

4. RELATED WORK

Several approaches detect inconsistent updates to code clones. CP-Miner [8], SecureSync [9], and Jiang et al.'s work [10] find cloning-related inconsistencies by searching for duplicated code. SPA [11] categorizes four common types of porting inconsistencies and detects discrepancies between the surrounding context of systematic changes. Unlike CRITICS, none of these tools give the users the control to interactively tune the abstract template interactively. CRITICS also differs from these techniques by applying program differencing in

[1] ASTParser is provided by JDT toolkit, `http://projects.eclipse.org/projects/eclipse.jdt`.
[2] The Crystal framework – `http://code.google.com/p/crystalsaf/`
[3] The Zest framework – `http://wiki.eclipse.org/Zest`

tandem with interactive code pattern search.

Instant clone search techniques take a code example as input and returns other similar code examples on demand. For example, Wang et al. [12] propose a dependence-based code search technique. Critics differs from these code matching and pattern mining techniques by detecting anomalies in *changes* as opposed to a single program version. LASE [13] automatically generates an abstract edit script from multiple change examples to find and update similar code fragments. While LASE focuses on automated completion of similar changes, Critics's goal is to summarize similar changes and to detect change anomalies for peer code reviews. LASE requires users to provide multiple change examples, while Critics allows users to interactively generalize the diff template to be used for search. LSdiff [14] infers systematic change patterns at a coarse granularity and summarizes them as logic rules. It also detects potential inconsistencies that violate the systematic change patterns. However, because LSdiff does not leverage any human input, it often discovers a large amount of rules in an inefficient top-down manner.

5. SUMMARY

We present Critics, a novel code review approach that integrates interactive code search, program differencing, and anomaly detection. It takes as input a selected sub region of diff patch and allows a reviewer to customize the change template by interactively generalizing the AST edits and surrounding context.

As future work, to reduce the burden of refining an abstract diff template, we plan to provide generalization hints to users. We will further investigate heuristics that developers employ for parameterizing edit content and build a conceptual model for inferring best template configuration heuristics.

6. ACKNOWLEDGEMENT

This work was supported in part by the National Science Foundation under grants CCF-1149391, CCF-1117902, SHF-0910818, CNS-1239498, and a Google Faculty Award.

7. REFERENCES

[1] A Frank Ackerman, Lynne S Buchwald, and Frank H Lewski. Software inspections: An effective verification process. *IEEE software*, 6(3):31–36, 1989.

[2] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.

[3] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun 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*, page 51. ACM, 2012.

[4] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[5] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[6] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Critics: Interactive summarization of systematic changes and anomaly detection for peer code reviews. Technical report, University of Texas at Austin, TR-ECE-2014-4, April, 2014.

[7] Mateusz Pawlik and Nikolaus Augsten. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[8] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.

[9] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ASE '10, pages 447–456, New York, NY, USA, 2010. ACM.

[10] Lingxiao Jiang, Zhendong Su, and Edwin 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*, pages 55–64, New York, NY, USA, 2007. ACM.

[11] B. Ray, Miryung 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*, pages 367–377, Nov 2013.

[12] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering*, ASE '10, pages 457–466, New York, NY, USA, 2010. ACM.

[13] Na Meng, Miryung Kim, and Kathryn McKinley. Lase: Locating and applying systematic edits by learning from examples. In *ICSE '13: Proceedings of 35th IEEE/ACM International Conference on Software Engineering*, pages 502–511. IEEE Society, 2013.

[14] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *ICSE '09: Proceedings of the IEEE 31st International Conference on Software Engineering*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.