CRITICS2: A Hybrid Tool for Both Code Review and Program Transformation

by

Jun Wang

Department of Computer Science
College of Information Science and Technology
University of Nebraska at Omaha
Omaha, NE 68182-0500
402-310-0526
jwang7@unomaha.edu

Sunday, April 25, 2021

Abstract

This work will extend a previous interactive code review tool called CRITICS to implement a new hybrid tool for both code review and program transformation. CRITICS is an interactive code review tool for inspecting systematic changes between old and new revisions within a diff patch. When code reviewers selected a diff region, CRITICS will automatically build AST trees covering the selected codes and their data/control dependent context codes for both old and new revisions. A default change template will be generated and can be used to search for other similar consistent changes or inconsistent anomalies (either missing updates or incorrect changes) in the entire codebase. A unique feature of CRITICS is that it allows code reviewers to iteratively customize the default change template by parameterizing type, variable, and method names or excluding certain context statements so that the customized abstract change template can be flexible to match more similar systematic changes. As a pure code review tool, CRITICS does not allow code reviewers to make changes to the source codes being inspected. In this work we will develop an extended version of CRITICS called CRITICS2. CRITICS2 will not only allow code reviewers to summarize similar consistent changes and find any inconsistent anomalies, but also allow code developers to automatically mitigate those inconsistent anomalies (ie. applying missing updates or fixing incorrect updates). In this way CRITICS2 will be a new useful hybrid tool for both interactive code review and automatic program transformation.

Table of Contents

I. Introduction

I.1. Problem

Code review is one of the most widely employed activities in software development. A recent review work [Bacchelli and Bird 2013] concluded that the number of serious defects found by code review is much less than expected, though defect finding is still the top motivation of code review for many practitioners. When a code reviewer does not have a deep understanding of the codebase, all comments he/she might give are suggestions on improving formats/readability, consistency etc. Therefore, the key issue of code review is code understanding, not just the changed part, but also its context or even the entire code base. Finding defects only tells part of the story why we need code review (code review serves multiple purposes). Code review brings us other benefits such as code improvement, alternative solutions, knowledge transfer. Most of existing code review tools often neglect the request from code reviewers, especially novice programmers, to learn knowledge from code review. As pointed by [Tao et al. 2012] many developers would like to understand code changes done by other team members while performing code review because it is a good learning opportunity for them to learn from previous code changes and apply the acquired knowledge for their own future development needs. In this regard the interactive customization of the default change template provided by CRITICS [Zhang et al. 2015] offers a good opportunity for code reviewers to learn and understand code changes done by other team members. However, finding other similar consistent changes or inconsistent anomalies only benefits code reviewers. As code developers, they also need a tool that can automatically transform the programs to fix those inconsistent anomalies. Therefore, we will investigate the possibility of integrating automatic program transformation into a code review tool like CRITICS which could benefit both code reviewers and code developers.

As shown in Fig. 1 when code reviewers selected a diff region (highlighted in blue), CRITICS will automatically build AST trees covering the selected codes and their data/control dependent context codes for both old (rendered in “Diff Template View (Old Rev.)” view) and new (rendered in “Diff Template View (New Rev.)” view) revisions. A default change template will be generated (rendered in “Diff Template” view) and can be used to search for other similar consistent changes (rendered in “Matching Locations” view) or inconsistent anomalies (rendered in “Inconsistent Locations” view) in the entire codebase. A unique feature of CRITICS is that it

allows code reviewers to iteratively customize the default change template by parameterizing type, variable, and method names (rendered as $var in “Diff Details” and “Diff Template” views) or excluding certain context statements (rendered as $EXCLUDED in “Diff Template” view) so that the customized abstract change template can be flexible to match more similar systematic changes. However, CRITICS does not allow code reviewers to make changes to the source codes being inspected (There is only “Show Template” option in the popup context menu when a particular inconsistent anomaly is selected). In this work we will implement options like “Fix Incorrect Update” and “Apply Missing Update” in the popup context menu.

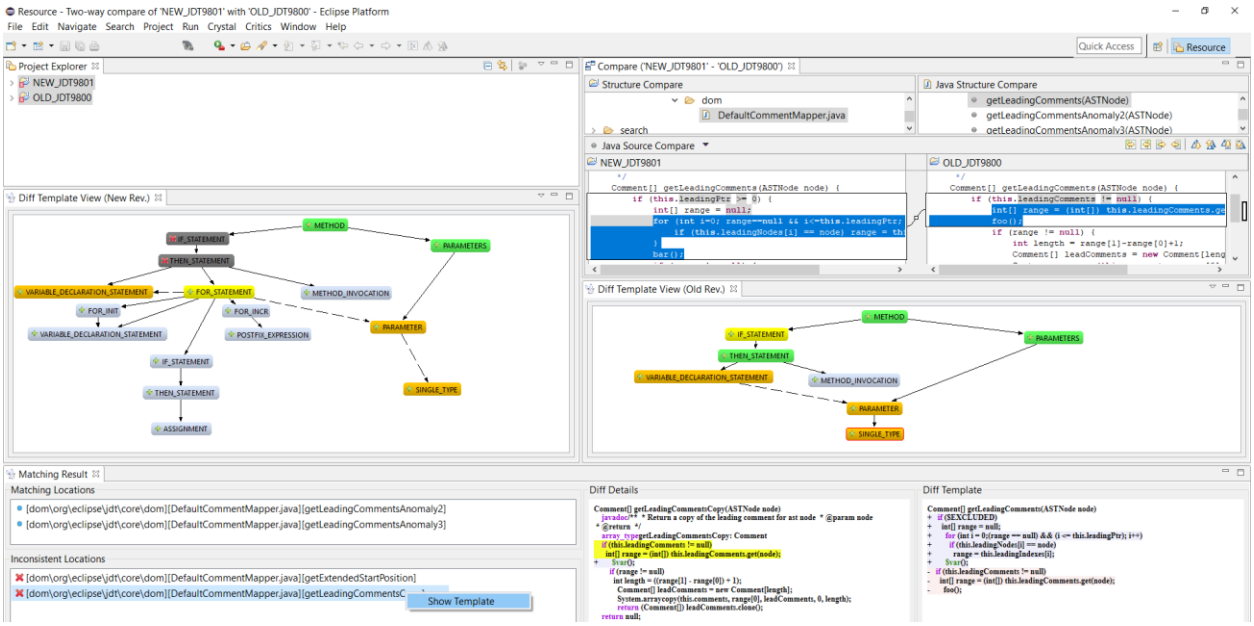


Fig. 1 CRITICS unable to fix detected inconsistent code changes

I.2. Motivations

When a software development process enters the evolution stage after initial specification, design & implementation and validation has completed, changes (adding features, fixing bugs, refactoring, and adapting to new APIs) are inevitable. Often these changes are systematic edits (similar, but not identical, changes to many locations). Making / reviewing these changes manually is tedious and error-prone. Locating and applying systematic edits has been proposed and implemented as a program transformation tool called LASE by [Meng et al. 2013]. However, LASE’s template generation requires multiple examples and is fixed. It does not allow code reviewers to iteratively customizing the change template like CRITICS. CRITICS allows reviewers with little knowledge of a codebase to flexibly explore the diff patch until a desired

pattern is satisfied. In this way code reviewers, especially novice programmers, could not only perform their code review duty, but also learn the codebase from code review. Unfortunately, CRITICS is unable to fix detected inconsistent code changes automatically. In this work we will combine advantages of LASE and CRITICS to build a new hybrid tool which will benefit both code reviewers and code developers.

I.3. Significance

As mentioned, the new hybrid tool CRITICS2 will not only allow code reviewers to summarize similar consistent changes and find any inconsistent anomalies, but also allow code developers to automatically mitigate those inconsistent anomalies (ie. applying missing updates or fixing incorrect updates). Therefore, the new tool will benefit both code reviewers and code developers.

I.4. Challenges

As mentioned, the proposed new hybrid tool will benefit both code reviewers and code developers. Therefore, it is worth investigating this problem. However, adding such complex new features to CRITICS which consists of six modules with a large codebase is not a trivial problem. Fortunately, it is not a problem unsolvable since LASE has already showed us how to automatically transform/update the code at locations with similar context. Besides that, Eclipse plugin development techniques learned from CSCI 8710 could help this project a lot.

Fully understanding the codebase of CRITICS is a major difficulty of this project. As shown in Table 1, the codebase of CRITICS consists of 6 modules with total 994636 lines of Java source codes. Even though the main CRITICS plugin module only has 138 files with 15619 LOCs, we still need to carefully investigate any other source files it references. Although majority of source codes in 3 org.eclipse.* modules are taken from original Eclipse plugins, they have been extended/modified to provide additional features required by CRITICS.

Another difficulty of this project is that we need to use relative old Eclipse Luna IDE which was used to develop CRITICS. Some bundles that CRITICS depends on may not be available in more recent Eclipse versions. For example, we cannot use new model fragment file fragment.e4xmi from Eclipse 4 RCP in more recent Eclipse IDE.

Table 1 CRITICS codebase analysis by tool cloc

module function	module name	files	blank	comments	code
main CRITICS plugin	edu.utexas.seal.plugins	138	2099	8390	15619
extended RTED & changedistiller	UT	1240	64866	190211	281292
data/control dependency analysis	edu.cmu.cs.crystal	245	3727	10885	16162
extended Eclipse Compare	org.eclipse.compare	164	4902	10222	25635
extended JDT Core	org.eclipse.jdt.core	1201	30824	124720	300588
extendded JDT UI	org.eclipse.jdt.ui	2096	73428	107335	355340
	CRITICS (Total)	5084	179846	451763	994636

I.5. Objectives

As shown in Fig. 1 there is only “Show Template” option in the popup context menu when a particular inconsistent anomaly is selected in CRITICS. In CRITICS2 we will implement options like “Fix Incorrect Update” and “Apply Missing Update” in the popup context menu so that it will automatically modify source codes of the new revision in a diff patch to apply the missing update or fix incorrect update when these options are chosen. Once those inconsistent anomalies are mitigated, those entries in “Inconsistent Locations” view will move up to “Matching Locations” view if we search the diff patch again with the same change template shown in “Diff Template” view of Fig. 1.

II. Overview

II.1. History of the problem

The history of modern code review and code change comprehension can be traced back to early work on change impact analysis (CIA). [Ren et al. 2004] proposed Chianti, a change impact analysis tool which is a promising technique for assisting developers with program understanding and debugging. Instead of running the entire regression test suite against the new program version, Chianti determines a subset that is potentially affected by the code changes. For an affected test Chianti also determines a subset of code changes that may have affected that particular test. [Xing and Stroulia 2005] proposed UMLdiff, a source code change differencing tool for automatically detecting structural changes in terms of (a) additions, removals, moves, renamings of packages, classes, interfaces, fields and methods, (b) changes to their attributes such as visibility and modifiers, and (c) changes of the dependencies among these entities. UMLdiff can assist software engineers in their tasks of understanding the rationale of code changes. [Fluri et al. 2007] proposed ChangeDistiller, a source code change differencing tool by comparing the difference of abstract syntax trees (AST) between the original and new versions. ChangeDistiller extracts changes by finding both a match between the nodes of the compared two ASTs and a minimum edit script that can transform one tree into the other given the computed matching. In contrast to GNU diff ChangeDistiller can detect changes more precisely and is able to assign a particular change to a concrete source code entity (such as the declaration or body part of a method), rather than just to a line number. [Kim and Notkin 2009] proposed LSdiff (Logical Structural Diff), a code review tool that infers systematic structural differences as logic rules. LSdiff iteratively infers logical rules with FB_Old (logical facts of old version), FB_New (logical facts of new version) and Delta_FB (fact-level difference) as inputs. At the end LSdiff outputs a list of all inferred logical rules and the remaining logical facts. Some inferred rules may include exceptions which are useful for detecting missing updates. [Barnett et al. 2015] proposed a new code review tool CLUSTERCHANGES to decompose composite code changes into independent partitions to facilitate understanding in code review. The def-use relationship was used as the primary organizing principle for clustering/partitioning diff-regions. CLUSTERCHANGES can help developers check their changesets before making actual commits and reviewers to understand their changes. [Zhang et al. 2015] proposed CRITICS, an interactive

code review tool for inspecting systematic changes. When code reviewers selected a diff region, CRITICS will automatically build AST trees covering the selected codes and their data/control dependent context codes for both old and new revisions. A default change template will be generated and can be used to search for other similar consistent changes or inconsistent anomalies (either missing updates or incorrect changes) in the entire codebase. A unique feature of CRITICS is that it allows code reviewers to iteratively customize the default change template by parameterizing type, variable, and method names or excluding certain context statements so that the customized abstract change template can be flexible to match more similar systematic changes. CRITICS not only help reviewers find all systematic changes, but also help them learn about the code base.

The history of automatic program transformation can be traced back to early work on automatic program repair. [Forrest et al. 2009] proposed GenProg, a Genetic Programming (GP) tool to automatically repair bugs in off-the-shelf legacy C programs. The basic idea of GP is to locate problematic statements along the execution path of the negative test cases with fault localization methods and then iteratively mutate problematic AST nodes with GP operators (insertion, deletion, swap, mutation, crossover) until a modified version passes all its positive and negative test cases. All genetic operators use existing code in other parts of the program with the assumption that programs contain the seeds of their own repairs. [Hartmann et al. 2010] proposed HelpMeOut, a social recommender system that aids the debugging of error messages by suggesting solutions that peers have applied in the past. HelpMeOut learns transformations to fix compilation and run-time errors from examples. HelpMeOut's automatic program transformation is limited to single-line changes since it relies on string-based textual difference, not AST-based difference. [Meng et al. 2011] proposed SYDIT, a program transformation tool that generates a context aware, abstract edit script learned from a given example edit and then applies the edit script to new program locations with similar context specified by developers. [Le Goues et al. 2012] further improved GenProg's algorithm with new patch-based representation (as a sequence of AST edit operations instead of a complete AST), fix localization assumption (choose seed of new repair from local scope instead of the entire program), that allows it to scale to large programs and finds successful repairs 68% more often. [Meng et al. 2013] further improved SYDIT as a new program transformation tool called LASE which generates an edit

script learned from two or more example edits and allows automatic search for target locations with similar context so that developers do not need to specify the target locations to apply systematic edits. [Kim et al. 2013] proposed PAR, a pattern-based automatic program repair tool, using fix patterns learned from existing human-written patches. The generated program repairs from PAR make more sense compared to those from GenProg. However, it is time consuming to create pattern-based fix templates.

II.2. State of the art

As mentioned by [Bacchelli and Bird 2013] a key challenge in modern code review practices is still lacking tool support for code change comprehension. Both [Bacchelli and Bird 2013] and [Tao et al. 2012] mentioned that many code changes in successive commits are composite changes which are difficult for code reviewers to comprehend. Therefore, modern code review tools must support the capability to divide a large chunk of code changes into sub logical groupings. This led to development of a couple of composite change decomposition tools [Herzig and Zeller 2013, Barnett et al. 2015, Tao and Kim 2015]. These findings motivated CRITICS [Zhang et al. 2015] as a state-of-the-art code review tool that assists code change comprehension with focuses on inspecting similar changes and detecting anomalies.

In the field of automatic program transformation, both SYDIT [Meng et al. 2011] and LASE [Meng et al. 2013] are typical example-based program transformations which share the same spirit of state-of-the-art programming-by-example (PBE) methodology. [Rolim et al. 2017] further proposed REFAZER, an automatic program transformation tool leveraging state-of-the-art PBE methodology and Inductive Programming (IP) framework PROSE. REFAZER learns domain specific language (DSL)-based abstract transformation rules even from one example or non-identical examples. LASE cannot learn from repetitive edits that appear in different statement types.

III. Techniques

III.1. Principles, Concepts, and Theoretical Foundations of the research problem

Most research on code review and automatic program transformation relies on the analysis of abstract syntax tree (AST) of the program which is the key concept in this field. Any program source code written in high-level language is nothing but a single string. Such unstructured string is usually converted into a structured full parse tree with a parser based on BNF grammar rules. Full parse tree derived from BNF is translated into a compact internal form called abstract syntax tree (AST) which removes nonessential non-terminal symbols. Complete parse expression provides a compact way to describe parse tree. For example, the entire program parse tree can be written as pgrm[[stmtList1]]. When a derivation is minimal and unique, we can use [[{ stmts1 }]] to denote a block of statements instead of block[[{ stmts1 }]]. A conditional if statement can be written as [[if (expr1) block1]] or [[if (expr1) [[{ stmts1 }]]]]. A variable declaration and assignment statement can be written as [[type1 id = expr1;]]. A for loop can be written as [[for (init1 ; expr1 ; modification1 ;) block1]]. A simple addition expression 3 + 5 in terms of AST can be written as [[[[3]] + [[5]]]].

Theoretical foundation of program transformation is equational reasoning. Given the equation "lhs = rhs", discover a substitution function σ such that σ(lhs) and target t are syntactically identical (matching lhs << t), then we can rewrite t as σ(rhs). For example, given equation x * (y + z) = x * y + x * z and we would like to transform 5 * (6 +7). To find the substitution function, we will try to match x * (y + z) with 5 * (6 +7) which will give us σ ≈ [x:=5][y:=6][z:=7]. Then we can rewrite/transform 5 * (6 +7) to σ(x * y + x * z) = 5 * 6 + 5 * 7. AST-based interactive code review tool like CRITICS for inspecting systematic changes follows the same basic idea. When code reviewers selected a diff region, CRITICS will automatically build AST trees covering the selected codes and their data/control dependent context codes for both old and new revisions. This basically set up the equation "lhs = rhs", where lhs represents the old revision and rhs represents the new revision. Then we can use lhs as the search query tree to search the entire program tree of old revision for any subtrees that matches lhs. The locations of the found matching subtrees indicate where systematic changes should apply in the new revision. By comparing the corresponding subtrees in the new revision, we can summarize similar consistent

changes and detect any inconsistent anomalies (missing updates or incorrect updates). AST-based program transformation tool like LASE also allow us to further rewrite/transform those found matching subtrees in the new revision.

III.2. Techniques that have been used by other researchers for the research problem

III.2.1. Change Distilling (tree differencing algorithm) [Fluri et al. 2007]

The tree differencing algorithm includes two tasks as shown in Fig. 2. Task 1 is to find a set of matching node pairs between T1 and T2. Task 2 is to find a minimum edit script that transforms T1 into T2. A general AST node n in ChangeDistiller has a label, n.mLabel, and a value, n.mValue. The label is the type of the statement. The value is the string representation of the statement. For example, a method invocation AST node "foo();" can be represented as (value: foo();)(label: METHOD_INVOCATION).

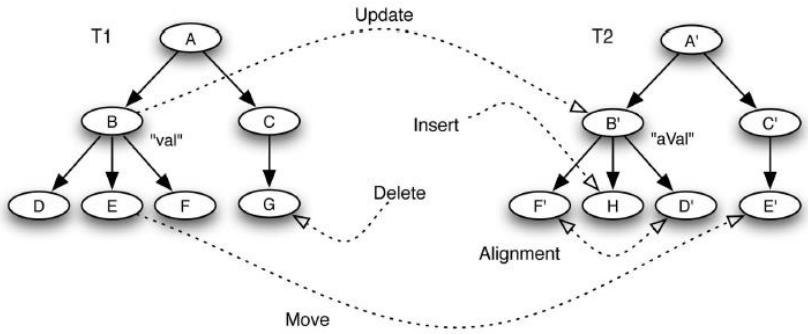


Fig. 2 matched node pairs (A~A', B~B', etc.) and minimum tree edit operations [Fluri et al. 2007]

Task 1 includes several steps. Given two labeled and valued trees T1 and T2 as input, the algorithm first calculates a complete matching of all leave nodes based on the matching criteria shown in Fig. 3(a). The leaf pairs are sorted according to their similarity and the best matches are added to the final matching set. At the end, the inner nodes are matched using the matching criteria shown in Fig. 3(b). Note that sim_{2g} in Fig. 3 represents bigram string similarity function. $|common(x,y)|$ in Fig. 3(b) means the number of matched leave nodes descended from the pair of inner nodes being compared between T1 and T2 and $max(|x|,|y|)$ means the maximum number of leave nodes descended from the inner node on either T1 or T2. In addition, there is an inner node similarity weighting exception: If the string similarity of inner node values is less than the threshold f, but $|common(x,y)| / max(|x|,|y|) \geq 0.8$, then $match_2(x, y)$ returns true. Besides that, the

threshold *t* is dynamically adjusted: if n ≤ 4, then *t* = 0.4, where *n* is the number of leaf descendants of the inner node.

$$match_1(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \wedge \\ & sim_{2g}(v(x), v(y)) \geq f \\ false & \text{otherwise,} \end{cases}$$

where $f = 0.6$.

(a) criteria to match leave nodes

$$match_2(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \wedge \\ & \frac{|common(x,y)|}{max(|x|,|y|)} \geq t \wedge \\ & sim_{2g}(v(x), v(y)) \geq f \\ false & \text{otherwise,} \end{cases}$$

where $f = 0.6$ and $t = 0.6$.

(b) criteria to match inner nodes

Fig. 3 matching criteria in Task 1 [Fluri et al. 2007]

The output of Task 1 (the matching set of node pairs between T1 and T2) is then passed to the edit script generation that runs through five phases to detect five basic tree edit operations (Insert, Delete, Alignment, Move, Update) as shown in Fig. 2 that transforms T1 into T2.

ChangeDistiller has been used in SYDIT, LASE and CRITICS to compute AST edits.

III.2.2. Robust Tree Edit Distance (RTED) tree matching algorithm [Pawlik and Augsten 2011]

The Change Distilling algorithm allows us to compare two small ASTs to get a minimum edit script that transforms T1 into T2 with a set of matching node pairs. However, it does not allow us to search a large AST for all subtrees that matches a small query AST since it cannot decompose the large AST. RTED tree matching algorithm allows us to recursively decompose the large AST to find all its possible subtrees and compare against a given query tree to get a list of matched subtrees.

In RTED a tree is decomposed to get a set of all possible sub-forests by recursively removing either the leftmost (r_L) or the rightmost root (r_R) nodes from a sub-forest as shown in Fig. 4. Note that a tree is a special case of forest. Initially we have a single tree forest. In this case both the leftmost and the rightmost root node refers to node A. After node A is removed, we get a forest that consists of a single node tree B and a four-nodes tree rooted at node C. Now we have a choice to make. If we remove r_L, we will get a single four-nodes tree sub-forest. If we remove r_R, we will get a sub-forest that consists of four trees (3 single node trees and 1 two-nodes tree). We continue the same process until we reach a single node forest.

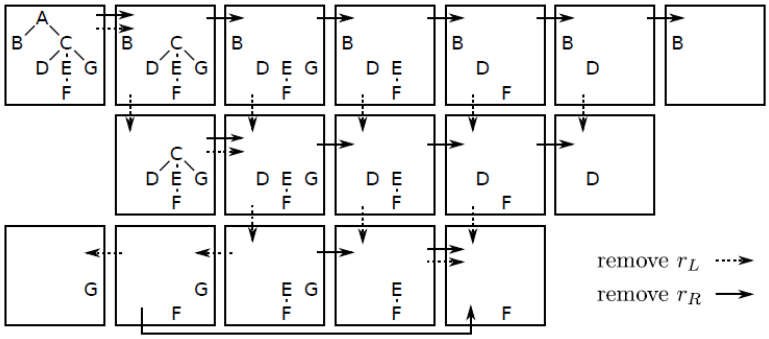


Fig. 4 recursive decomposition of a tree in RTED [Pawlik and Augsten 2011]

At each recursive step, as long as we have a forest that has more than one tree, we have a choice to make. Some algorithm always removes r_R. Others removes the largest subtree in a sub-forest last, after removing all nodes to the left and then all nodes to the right of that subtree. The key factor that leads to both efficient and worst-case optimal algorithm in RTED is that it dynamically chooses one of the above strategies to ensure an optimal choice.

The tree edit distance has the recursive solution shown in Figure 5. The nodes *v* and *w* are either both the leftmost or both the rightmost root nodes of the respective forests. RTED computes the tree edit distance recursively by finding structural alignment. RTED then provides a list of matching node pairs with a minimum tree edit distance. If the number of matching node pairs is equal to the expected value, then the target tree is treated as a match to the query tree. Of course, we could find multiple subtrees that match the query tree.

$$\delta(\emptyset,\emptyset) = 0,$$
$$\delta(F,\emptyset) = \delta(F-v,\emptyset) + c_d(v),$$
$$\delta(\emptyset,G) = \delta(\emptyset,G-w) + c_i(w),$$
$$\text{if } F \text{ is not a tree or } G \text{ is not a tree:}$$
$$\delta(F,G) = \min \begin{cases} \delta(F-v,G) + c_d(v) & (1) \\ \delta(F,G-w) + c_i(w) & (2) \\ \delta(F_v,G_w) + \delta(F-F_v,G-G_w) & (3),(4) \end{cases}$$
$$\text{if } F \text{ is a tree and } G \text{ is a tree:}$$
$$\delta(F,G) = \min \begin{cases} \delta(F-v,G) + c_d(v) & (1) \\ \delta(F,G-w) + c_i(w) & (2) \\ \delta(F-v,G-w) + c_r(v,w) & (5) \end{cases}$$

Fig. 5 recursive solution of tree edit distance [Pawlik and Augsten 2011]

RTED has been used in CRITICS to search for subtrees of old source code revision that match the query tree of selected old edit to find all locations where similar updates should apply. In a similar way RTED is also used to search for subtrees of new source code revision that match the query tree of selected new edit to summarize matching locations or inconsistent locations

(missing updates or incorrect updates). LASE does not use RTED to find edit locations, but directly compares the extracted maximum common context AST against each method AST to find all matching methods. Therefore, one important limitation of LASE is that it can find at most one edit location per method. Sometimes multiple similar code changes could occur in the same method. CRITICS is able to handle this issue with RTED.

III.3. Relevant technologies that would be useful to this research

Since the tool in this research will be developed as an Eclipse plugin, Eclipse plugin development techniques learned from CSCI 8710 are useful to this research, especially some relevant Eclipse plugin development frameworks such as Plug-in Development Environment (PDE) [Windatt 2011], Rich Client Platform (RCP) [Vogel 2016], Java Development Toolkit (JDT) [Herrmann 2020].

III.4. Algorithms, Process modules, and solution methods

Algorithm 1: Find Anomaly Nodes

Input: Let *diffQTree* be a ChangeDistiller difference Tree from a customized change template
Input: Let *diffTTree* be a ChangeDistiller difference Tree from a selected matching target tree
Output: Collections of incorrect inserted Nodes (IINs), incorrect deleted Nodes (IDNs), missed insertion Nodes (MINs), missed deletion Nodes (MDNs).

```
Algorithm findNodeOfAnomaly(diffQTree, diffTTree)
    IINs := IDNs := MINs := MDNs := ∅;
    insertNodeList_Q := diffQTree.getInsertNodeList(); insertNodeList_T := diffTTree.getInsertNodeList();
    deleteNodeList_Q := diffQTree.getDeleteNodeList(); deleteNodeList_T := diffTTree.getDeleteNodeList();
    foreach iNode from insertNodeList_T do
        is_iNode_in_insertNodeList_Q := FALSE;
        foreach jNode from insertNodeList_Q do
            if iNode.eq(jNode) ≡ TRUE then
                is_iNode_in_insertNodeList_Q := TRUE;
            end
        end
        if is_iNode_in_insertNodeList_Q ≡ FALSE then
            iNode.setAnomaly(TRUE);
            IINs := IINs ∪ {iNode};
        end
    end
    foreach iNode from insertNodeList_Q do
        is_iNode_in_insertNodeList_T := FALSE;
        foreach jNode from insertNodeList_T do
            if iNode.eq(jNode) ≡ TRUE then
                is_iNode_in_insertNodeList_T := TRUE;
            end
        end
        if is_iNode_in_insertNodeList_T ≡ FALSE then
            iNode.setAnomaly(TRUE);
            MINs := MINs ∪ {iNode};
        end
    end
    foreach iNode from deleteNodeList_T do
        is_iNode_in_deleteNodeList_Q := FALSE;
        foreach jNode from deleteNodeList_Q do
            if iNode.eq(jNode) ≡ TRUE then
                is_iNode_in_deleteNodeList_Q := TRUE;
            end
        end
        if is_iNode_in_deleteNodeList_Q ≡ FALSE then
            iNode.setAnomaly(TRUE);
            IDNs := IDNs ∪ {iNode};
        end
    end
    foreach iNode from deleteNodeList_Q do
        is_iNode_in_deleteNodeList_T := FALSE;
        foreach jNode from deleteNodeList_T do
            if iNode.eq(jNode) ≡ TRUE then
                is_iNode_in_deleteNodeList_T := TRUE;
            end
        end
        if is_iNode_in_deleteNodeList_T ≡ FALSE then
            iNode.setAnomaly(TRUE);
            MDNs := MDNs ∪ {iNode};
        end
    end
    return IINs, MINs, IDNs, MDNs;
```

Original findNodeOfAnomaly method from CRITICS is only able to mark those Nodes that should have been deleted as anomaly (highlighted with yellow background in "Diff Details" browser view). Though method markInsertNodeList is able to tell the difference between incorrect inserted Nodes and correct inserted Nodes (incorrect inserted nodes are rendered with default white background in "Diff Details" browser view), incorrect inserted nodes should be marked as anomaly and highlighted with yellow background. We implemented a new findNodeOfAnomaly method as shown in Algorithm 1 which is able to detect all four types of special Nodes, namely incorrect inserted Nodes, incorrect deleted Nodes, missed insertion Nodes and missed deletion Nodes. Except missed insertion Nodes, all three other special nodes are marked as anomaly and highlighted with yellow background in "Diff Details" browser view.

Algorithm 2: Fix Incorrect Update

Input: Let *selectedRightRev* and *selectedLeftRev* be ChangeDistiller general Tree Node for old and new target method, respectively.
Input: Collections of incorrect inserted Nodes (*IINs*), missing insertion Nodes (*MINs*) from findNodeOfAnomaly
Output: Updated Target Method Body Statements *updatedMethodBodyStms*.

```
Algorithm fixIncorrectUpdate(selectedRightRev, selectedLeftRev, IINs, MINs)
newMethodDecl := selectedLeftRev.getMethodDeclaration(); newMethodBody := newMethodDecl.getBody().toString();
newMethodBodyStms := newMethodBody.substring(1, newMethodBody.length() - 2);
updatedMethodBodyStms := newMethodBodyStms;
prjNameLeft := UTCriticsPairFileInfo.getLeftProjectName();
packageName := selectedRightRev.getPackageName();
className := selectedRightRev.getClassName();
foreach iIN, mIN from IINs, MINs do
    iIN_Str := getLabel(iIN.getLabel().name()) + iIN.getValue();
    mIN_Str := getLabel(mIN.getLabel().name()) + mIN.getValue();
    startIndex := updatedMethodBodyStms.indexOf(iIN_Str);
    endIndex := startIndex + iIN_Str.length();
    updatedMethodBodyStms := UTStr.replace(updatedMethodBodyStms, mIN_Str, startIndex, endIndex);
end
new ReplaceMethodBodyAnalyzer(prjNameLeft, packageName, UTStr.getClassNameFromJavaFile(className),
selectedLeftRev.getValue(), updatedMethodBodyStms);
```

Taking collections of incorrect inserted Nodes (IINs), missing insertion Nodes (MINs) from the new findNodeOfAnomaly method as input, we implemented a simple algorithm to fix those incorrect updates as shown in Algorithm 2. First, we can obtain a string representation of new target method body statements *newMethodBodyStms* from the target method Node *selectedLeftRev*. Then for each incorrect inserted Node's string representation we can locate its *startIndex* and *endIndex* in string *updatedMethodBodyStms* and replace the located substring with each corresponding missing insertion Node's string representation. Eventually, we can get our final *updatedMethodBodyStms* for use in method ReplaceMethodBodyAnalyzer which finds CompilationUnit of the new revision and calls ReplaceMethodBodyVisitor to rewrite the new

target method ASTNode. Both Algorithm 2 and Algorithm 3 are implemented in method addMenuToAnomalyTableViewer.

Algorithm 3: Apply Missing Update

Input: Let *selectedRightRev* and *selectedLeftRev* be ChangeDistiller general Tree Node for old and new target method, respectively.
Input: Collections of missing deletion Nodes (*MDNs*), missing insertion Nodes (*MINs*) from findNodeOfAnomaly
Output: Updated Target Method Body Statements *updatedMethodBodyStms*.

Algorithm applyMissingUpdate(*selectedRightRev*, *selectedLeftRev, MDNs, MINs*)
newMethodDecl := *selectedLeftRev*.getMethodDeclaration(); *lstNewStms* := *newMethodDecl*.getBody().statements();
fNewRev := UTCriticsPairFileInfo.getLeftFile(); *srcNewFile* := UTFile.getContents(*fNewRev*.getAbsolutePath());
newMethodBodyStms := *srcNewFile*.substring(*lstNewStms*.get(0).getStartPosition(), *lstNewStms*.get(*lstNewStms*.size()-1).getStartPosition() + *lstNewStms*.get(*lstNewStms*.size() - 1).getLength());
startIndexMissingDeletion := *MDNs*.get(0).getEntity().getStartPosition() - lstNewStms.get(0).getStartPosition();
endIndexMissingDeletion := *MDNs*.get(*missingDeletionNodesSize* - 1).getEntity().getEndPosition() - *lstNewStms*.get(0).getStartPosition();
missingDeletionNodesRange := *newMethodBodyStms*.substring(*startIndexMissingDeletion*, *endIndexMissingDeletion*);
startIndexMissingInsertion := *MINs*.get(0).getEntity().getStartPosition() - *lstNewQueryStms*.get(0).getStartPosition();
endIndexMissingInsertion := *MINs*.get(*missingInsertionNodesSize* - 1).getEntity().getEndPosition() - *lstNewQueryStms*.get(0).getStartPosition();
missingInsertionNodesRange := *newQueryMethodBodyStms*.substring(*startIndexMissingInsertion*, *endIndexMissingInsertion*);
updatedMethodBodyStms := UTStr.replace(*newMethodBodyStms*, *missingInsertionNodesRange*, *startIndexMissingDeletion*, *endIndexMissingDeletion*);
new ReplaceMethodBodyAnalyzer(*prjNameLeft*, *packageName*, UTStr.getClassNameFromJavaFile(*className*), *selectedLeftRev*.getValue(), *updatedMethodBodyStms*);

Taking collections of missing deletion Nodes (MDNs), missing insertion Nodes (MINs) from the new findNodeOfAnomaly method as input, we implemented a simple algorithm to apply missing updates as shown in Algorithm 3. First, we can obtain a string representation of new target method body statements *newMethodBodyStms* from the target method Node *selectedLeftRev*. Note that the strategy to obtain *newMethodBodyStms* in Algorithm 3 is different than that in Algorithm 2. Then we can determine the source range of all missing deletion Nodes as *missingDeletionNodesRange* and the source range of all missing insertion Nodes as *missingInsertionNodesRange* to be used to get our final *updatedMethodBodyStms* for use in method ReplaceMethodBodyAnalyzer which finds CompilationUnit of the new revision and calls ReplaceMethodBodyVisitor (Algorithm 4) to rewrite the new target method ASTNode.

Algorithm 4: Replace Method Body Visitor

Input: Let *targetMethodName* be the name of new target method whose method body will be replaced with *updatedMethodBodyStms*.
Output: Updated source code file of the new revision.

Algorithm visit(*methodDecl*, *targetMethodName, updatedMethodBodyStms*)
methodName := *methodDecl*.getName().getIdentifier();
if *methodName*.equals(*targetMethodName*) = *TRUE* **then**
    parser := ASTParser.newParser(AST.*JLS4*);
    parser.setSource(*updatedMethodBodyStms*.toCharArray());
    parser.setKind(ASTParser.*K_STATEMENTS*);
    newBlock := (Block) *parser*.createAST(null);
    oldBlock := *methodDecl*.getBody();
    rewrite.replace(*oldBlock*, *newBlock*, null);
end
return super.visit(*methodDecl*);

IV. Approach(s)

IV.1. Methodologies I am going to apply in this research

In order to automatically mitigate inconsistent anomalies found by CRITICS, we will modify the RTED tree matching procedure in CRITICS to record name mappings between abstract names from the query tree and concrete names from the found matching subtrees. Similar to LASE [Meng et al. 2013], CRITICS2 must customize the abstract AST edit for the found matching subtrees that miss updates. The customization algorithm replaces all abstract names in the AST edit with concrete names from the found matching subtrees recorded before. The result is a concrete AST edit, which fully specifies each edit operation as an AST modification with concrete labels and node positions. Then CRITICS2 will apply this concrete edit to rewrite corresponding AST nodes. For those subtrees that have incorrect updates, we just need to rewrite those AST nodes that have incorrect updates.

IV.2. Techniques I am going to use to solve the problem

We will use AST node rewrite techniques learned from CSCI 8710 to implement auto-mitigate inconsistent anomalies in the new source code revision.

IV.3. Processes I am going to engage in this research

CRITICS2 will be developed as an Eclipse plugin based on source codes of CRITICS with added source codes to implement features to mitigate detected inconsistent anomalies.
We will use original dataset from CRITICS to test if CRITCS2 can automatically mitigate the detected inconsistent anomalies. Once inconsistent anomalies have been fixed, those entries in "Inconsistent Locations" view will move up to "Matching Locations" view if we search the diff patch again with the same change template shown in "Diff Template" view of Fig. 1.

IV.4 Facilities and supplies needed for this research

- CRITICS source codes: https://github.com/tianyi-zhang/Critics
- CRITICS evaluation data set: https://www.dropbox.com/s/p2ikyu3iwm7lfww/DataSet.zip?dl=0
- Eclipse Luna IDE for Java EE Developers: https://www.eclipse.org/downloads/packages/release/luna/sr2/eclipse-ide-java-ee-developers
- CRITICS2 source codes: https://github.com/hawk2012en/CSCI8910Critics2

V. Work Conducted

V.1 Tasks performed in this research

Task 1: test existing CRITICS source codes (completed)

Download CRITICS source codes from https://github.com/tianyi-zhang/Critics.

Compile source codes with the recommended Eclipse Luna IDE for Java EE developers.

Download CRITICS evaluation dataset to test CRITICS.

Fix any runtime exceptions to make sure CRITICS works as expected as shown in its demo video https://www.youtube.com/watch?v=F2D7t_Z5rhk

This task has been completed in one week.

Task 2: recap Eclipse plugin development techniques learned from CSCI 8710 (completed)

Review lecture slides.

Review past programming assignments.

Fully understand how to modify target source codes via AST node rewrite.

ASTNode/ASTVisitor manipulation skills have been improved.

This task has been to be completed in one week.

Task 3: review codebase of CRITICS (Completed)

Fully comprehend codebase of CRITICS.

Understand how AST edits are generated with Change Distilling tree differencing algorithm.

Understand the adaptation of RTED tree matching algorithms in CRITICS.

Understand how inconsistent anomalies are distinguished from consistent updates.

Understand how inconsistent AST nodes are detected and shown in “Diff Details” view.

Understand how certain statements can be excluded from the change template.

This task has been completed in three weeks.

I have fully understood how CRITICS summarizes changes and detects anomalies and improved method findNodeOfAnomaly with new algorithm.

Task 4: implement feature to mitigate inconsistent anomalies (Completed)

Add codes to record name mappings between abstract identifiers and concrete token names.

Add codes to customize the change template for missing update locations.

Add codes to customize the change template for incorrect update locations.

Add codes to show options like “Fix Incorrect Update” and “Apply Missing Update” in the popup context menu in Fig. 1.

Add codes to handle the events while clicking the above new options so that corresponding inconsistent anomalies in the new source code revision can be fixed automatically.

Estimated to be completed in eight weeks.

V.2. Schedule, timeline, and milestones

Table 2. Tentative schedule of research project

Order	Dates	Task/activity	Prerequisites	Results obtained
1	1/24 - 1/30	Task 2	N/A	Improved ASTNode/ASTVisitor manipulation skills.
2	1/31 - 2/20	Task 3	N/A	Fully understood how CRITICS summarizes changes and detects anomalies. Improved method findNodeOfAnomaly with new algorithm.
3	2/21 - 4/17	Task 4	Task 2 / 3	Added two new pop-up menus “Fix Incorrect Update” and “Apply Missing Update”. Auto-repair feature implemented.

VI. Results and Analysis

VI.1. Results

VI.1.1. Fix runtime exception and bugs to make CRITICS works as expected

1. A FileNotFoundException was encountered after selecting "Summarize Changes and Detect Anomalies" popup menu in "Diff Template View". After adding and tracking [DBG] output, I successfully updated method getFullFilePath in file UTPlugin.java (also method getFile(ICompilationUnit icu) in file UTCriticsFileFilter.java) and fixed this issue:

```
120         //String fileName = unit.getPath().toFile().getAbsolutePath();
121         String fileName = unit.getPath().toFile().getPath();
```

Fig. 6 Code change to fix FileNotFoundException

2. Initially summarized similar changes are not shown up in the expected "Matching Locations" table viewer. After adding and tracking [DBG] output, I finally updated method getPackageName(File file) in file RTEDInfoSubTree.java to get this issue fixed:

```
197         //int index = pkgName.indexOf("/");
198         int index = pkgName.indexOf("\\");
```

Fig. 7 Code change to fix summarized similar changes not shown up issue

This is due to the difference of file path separator used between *nix OS and Windows.
In fact a better solution is to use the following getPackageName method:

```
202     private String getPackageName() {
203         CompilationUnit cUnit = getCunit();
204         return cUnit.getPackage().getName().getFullyQualifiedName();
205     }
```

Fig. 8 New method to get package name

3. A serious bug in CRITICS has been found while implementing the new auto-repairing feature. The "Select Diff Region" pop-up menu in "Jave Source Compare" View from Eclipse Compare (alternatively as "Critics" → "Select Context Nodes – Method Level") will lead to a Null pointer Runtime exception while applying to method getLeadingCommentsAnomaly2, getLeadingCommentsAnomaly3, etc. in Class DefaultCommentMapper from JDT 9800/9801diff patch, though it works OK for method getLeadingComments.

After a lengthy debugging process (about two weeks), eventually I figured out the root cause. The Null pointer exception was thrown from method printNode in file UTChange.java where the element *change* in List<Node> *aList* turned out to be null at runtime while applying to method like getLeadingCommentsAnomaly2 which caused null pointer exception due to referencing *change*.getEntity().getLabel(). Apparently this is due to *mInsertNodeList* from method getEditsFromQTree in file CriticsSelectContext.java returned a list of null elements. This can be further traced to *methodNode* from method getNodeListMethodLevel in file UTChange.java which returned the wrong method Node. Obviously this is caused by the fact that *methodDecl* from method getMethodNode in file UTChange.java did not return the expected target MethodDeclaration that matches our selected diff region. Finally I discovered that the string representation of the compilation unit *srcNewRev* obtained from source file is actually different from *srcNewViewer* obtained from org.eclipse.jface.text.source.IsourceViewer. It seems that the position value obtained from Node.getEntity().getStartPosition() is mostly based on *srcNewViewer* while that obtained from ASTNode.getStartPosition() is based on *srcNewRev*. Finally this bug can be fixed by making the following changes:

```
569         //mInsertNodeList = UTChange.getNodeListMethodLevel(mInsertList, unitNewRev, srcNewRev, fNewRev);
570         mInsertNodeList = UTChange.getNodeListMethodLevel(mInsertList, unitNewRev, srcNewViewer, fNewRev);
571         //mDeleteNodeList = UTChange.getNodeListMethodLevel(mDeleteList, unitOldRev, srcOldRev, fOldRev);
572         mDeleteNodeList = UTChange.getNodeListMethodLevel(mDeleteList, unitOldRev, srcOldViewer, fOldRev);
```

Fig. 9 Code change to fix Null Exception in file CriticsSelectContext.java

```
208         //MethodDeclaration methodDecl = finder.findCoveringMethodDeclaration(aUnit, new Point(startPosition, defaultLength));
209         MethodDeclaration methodDecl = finder.findMethod(aSource, change.getChangedEntity().getSourceRange(), false);
```

Fig. 10 Code change to fix Null Exception in file UTChange.java

4. Another Null pointer runtime exception I have fixed is for feature “Critics” → “Select Context Nodes – Class Level”. A null exception was thrown from method convertClassDeclaration in file UTASTNodeConverter.java while referencing type.declarationSourceStart. This bug can be fixed by making the following changes:

```
219             //else{
220                 if (String.valueOf(type.name).equals(typeName)) {
221                     return type;
222                 }
223             //}
```

Fig. 11 Code change to fix Null Exception in file UTCompilationUtils.java

A demo to show all above bugs and corresponding fixes can be viewed at https://use.vg/4E0nKn.

VI.1.2. Recap Eclipse plugin development techniques learned from CSCI 8710

I have fully understood how Visitor design pattern works, especially for ASTVisitor and ASTNode with a list of child nodes. I was able to retrieve path of current Project Name and Work Space from System.getProperty("org.osgi.framework.storage"). I learned how to get path of runtime WorkSpace from System.getProperty("osgi.logfile"). I reviewed how to add customized main menu and corresponding command / handler and knew two ways to add pop-up context menu. I am familiar with rendering data using TableViewer and TreeViewer. I mastered various ways to analyze / modify target Java source codes via manipulation of IJavaModel and ASTNode (ie. Two ways to obtain package name, class name from a MethodDeclaration ASTNode; How to add/delete a package, a class or a method; Two ways to rename a package, a class or a method; How to add additional method parameters and method body statements).

VI.1.3. Review codebase of CRITICS

Overall, I have fully understood how CRITICS summarizes changes and detects anomalies and improved method findNodeOfAnomaly with a new algorithm.

1. Minimum AST edits are obtained from method diffBlock in file UTChangeDistiller.java which returns a list of inserted Node and a list of deleted Node between an old ChangeDistiller Node and an new one.
2. The adaptation of RTED algorithm is implemented in method matchEditMapping(Node qTree, Node tTree) of file TreeMatch.java. After original RTED algorithm finds a node level match between a query tree qTree and a target tree tTree with method invocation treeEditMatch(qTree, tTree), CRITICS further checks if qTree also matches tTree at the token level. For each aligned child Node pair (qNode and tNode) between qTree and tTree obtained from original RTED, CRITICS checks if label of qNode is equal to label of tNode and value of qNode is equal to value of tNode. If the value of qNode has been parameterized, then CRITICS calls UTChange.checkEQByDiff(qNode, tNode) to check if tNode matches parameterized qNode. If all expected qNode matches tNode in terms of both label and possibly parameterized value, then qTree and tTree will be considered as a match at the token level and added to the list of found matching RTEDInfoSubTree mSubTrees with calculated tree edit distance ted.
3. Inconsistent anomalies are distinguished from consistent updates by methods detectAnomalies and detectAnomalyMethodLevel in file CriticsFindContext.java. The

idea is simple. Calling matchEditMapping(qTreeOld, tTreeOld) will give us a list of found subtrees lstSubTreeInfoOldRev that match old query tree (old revision of change template) in the old revision of source code. Calling matchEditMapping(qTreeNew, tTreeNew) will give us a list of found subtrees lstSubTreeInfoNewRev that match new query tree (new revision of change template) in the new revision of source code. If a subtree in lstSubTreeInfoOldRev is also found in lstSubTreeInfoNewRev, then it will be considered as consistent update with setAnomaly(false) and added to summaryGroup (rendered in "Matching Locations" TableViewer). Otherwise, it will be considered as inconsistent anomaly with setAnomaly(true) and added to anomalyGroup (rendered in "Inconsistent Locations" TableViewer). The course-grained sequence diagram of "Summarize Changes and Detect Anomalies" in CRITICS is shown in Fig. 12.

4. Inconsistent Nodes shown in "Diff Details" browser view are detected by methods findNodeOfAnomaly and markInsertNodeList in file CriticsOverlayViewEvent.java. Method findNodeOfAnomaly is designed to mark those Nodes that should have been deleted as anomaly (rendered with yellow background in "Diff Details" browser view). Method markInsertNodeList is designed to highlight those newly inserted Nodes with light blue background with the exception that the newly inserted Nodes with incorrect updates are highlighted with default white background. Those deleted Nodes with light brown background are marked by method putEditQTree in file CriticsOverlayNewBrowser.java. We have implemented a new method findNodeOfAnomaly which could replace the original findNodeOfAnomaly method and detect all four types of special Nodes, namely incorrect inserted Nodes, incorrect deleted Nodes, missed insertion Nodes and missed deletion Nodes as shown in Algorithm 1. Except missed insertion Nodes all three other special nodes are marked as anomaly and rendered with yellow background in "Diff Details" browser view. The course-grained sequence diagram of rendering data in "Diff Details" browser view is shown in Fig. 12.
5. CRITICS's unique feature to exclude certain statements from the change template is handled in methods treeEditMatch and matchEditMapping in file TreeMatch.java.

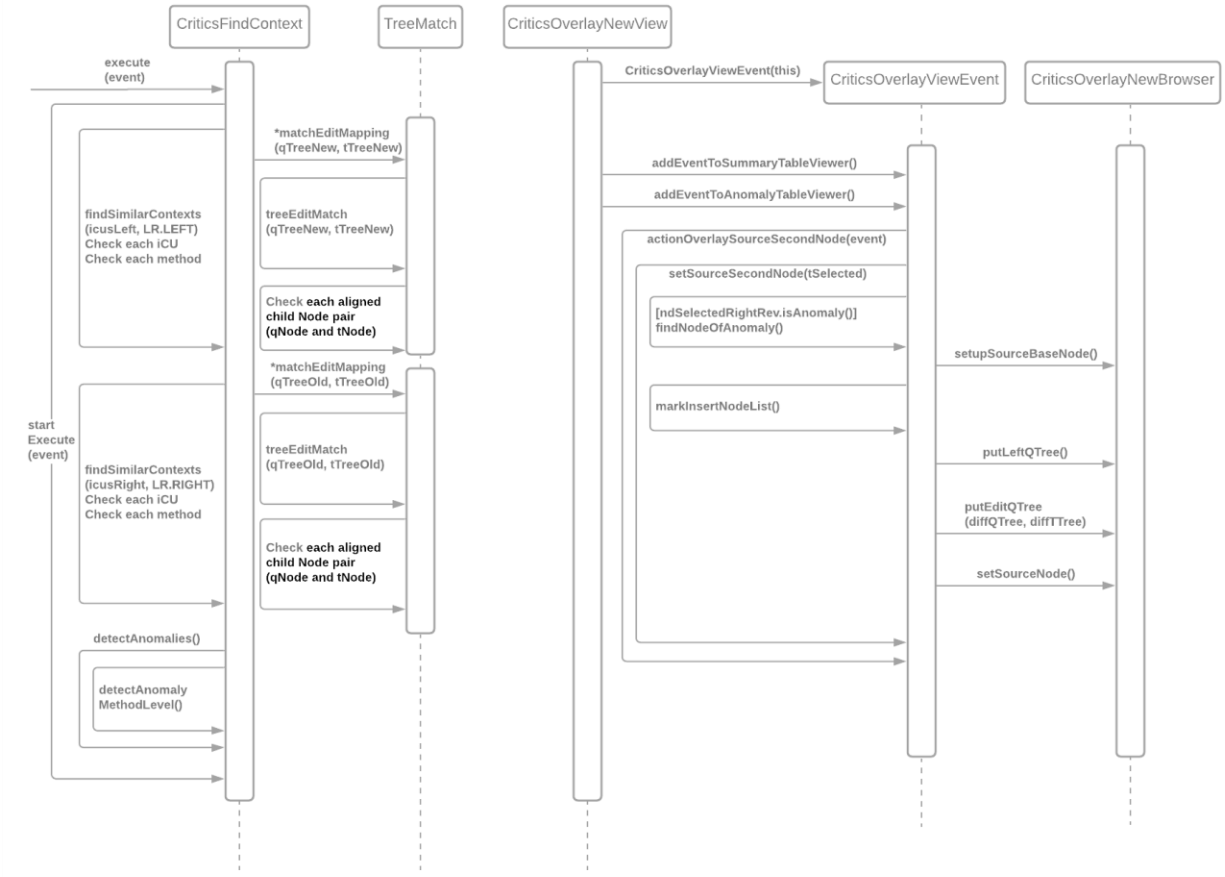


Fig. 12 "Summarize Changes and Detect Anomalies" and "Diff Details" sequences in CRITICS

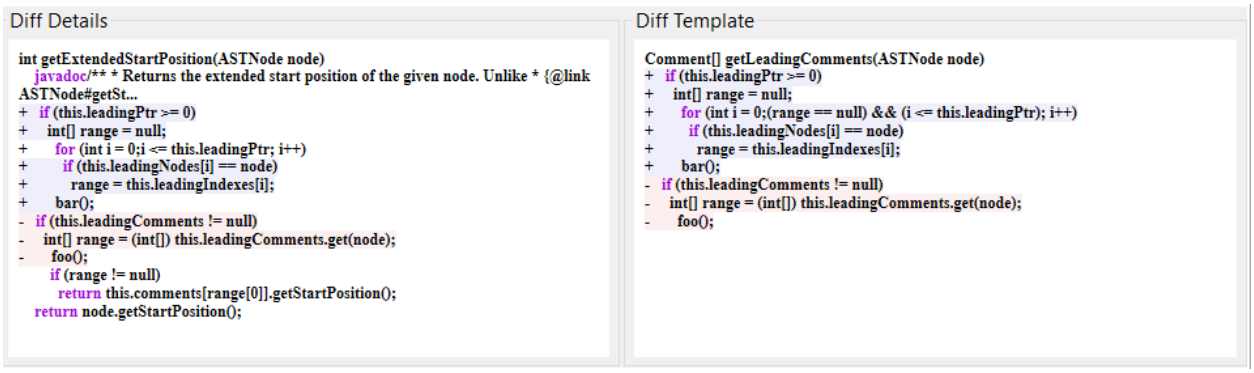
VI.1.4. Implement features to mitigate inconsistent anomalies

Overall, we proposed simple algorithms as shown in Algorithm 2 and Algorithm 3 to implement the main new features "Fix Incorrect Update" and "Apply Missing Update" in CRITICS2. Instead of visiting various specific ASTNode types in a ASTVisitor subclass to rewrite many different ASTNodes, we proposed a more sensible strategy that prepares a string representation of updated method body statements (*updatedMethodBodyStms*) from detected anomaly Nodes and their expected corrections and we only need to visit MethodDeclaration ASTNode in a ASTVisitor subclass and rewrite that specific MethodDeclaration to update the source code of the new revision. Both "Fix Incorrect Update" and "Apply Missing Update" are implemented in method addMenuToAnomalyTableViewer in file CriticsOverlayViewMenu.java. In the end it called ReplaceMethodBodyAnalyzer which finds CompilationUnit of the new revision and further calls ReplaceMethodBodyVisitor (Algorithm 4) to rewrite the new target method

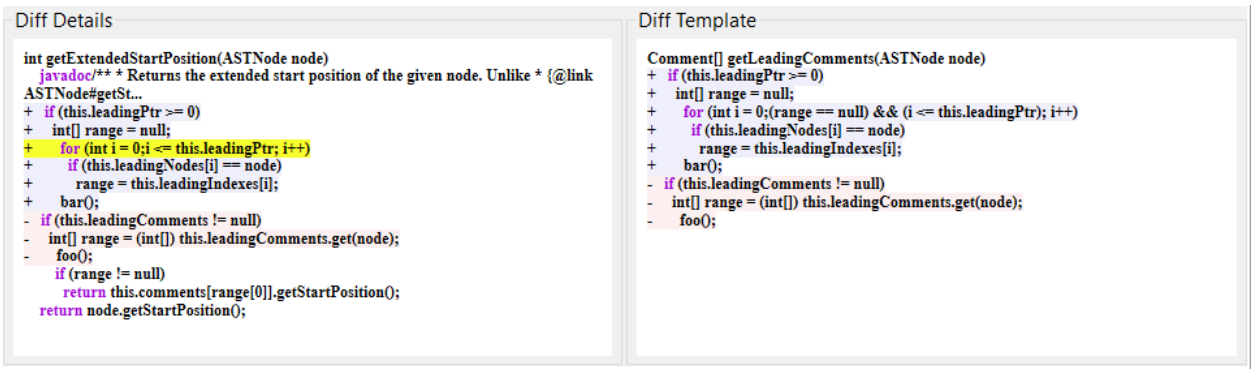
ASTNode. Class ReplaceMethodBodyAnalyzer is adapted from Class InsertMethodBodyAnalyzer in CSCI 8710 project-ex-0926-type-delete-ast-wang while Class ReplaceMethodBodyVisitor is adapted from Class InsertMethodBodyVisitor in the same project. A full demo to show newly implemented features "Fix Incorrect Update" and "Apply Missing Update" in CRITICS2 can be viewed at https://use.vg/WAWoDK.

VI.2. Analysis of the results

All the above results were evaluated using JDT 9800/9801diff patch (Patch 1 of dataset downloaded from https://www.dropbox.com/s/p2ikyu3iwm7lfww/DataSet.zip?dl=0). The resulting difference between original findNodeOfAnomaly method and the new one implementing Algorithm 1 is shown in Fig. 13.



(a) results from original findNodeOfAnomaly method

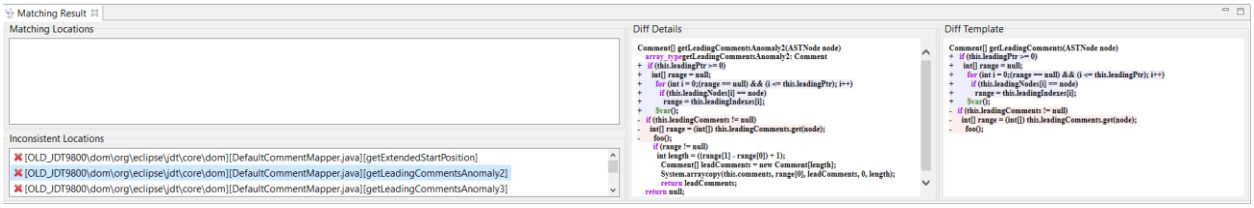


(b) results from new findNodeOfAnomaly method

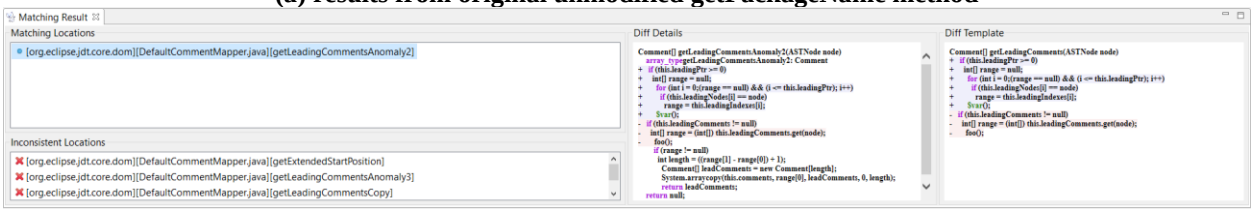
Fig.13 Comparison of results from old and new findNodeOfAnomaly method

As we can see, incorrect inserted nodes are highlighted with default white background while using original findNodeOfAnomaly method which did not clearly show their anomaly status. On the other hand, our new findNodeOfAnomaly method clearly highlighted them in yellow indicating they are anomaly.

In addition, the resulting difference between original unmodified getPackageName method and the new getPackageName method shown in Fig. 8 is shown in Fig. 14.



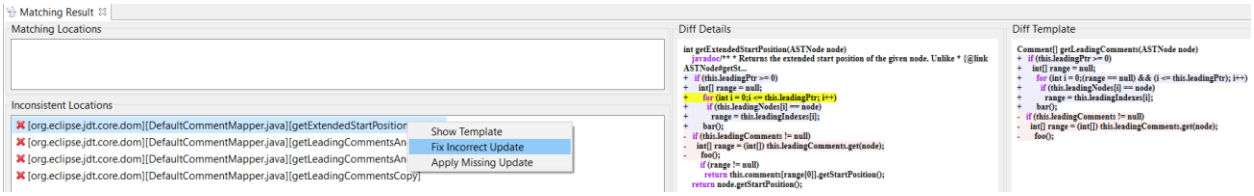
(a) results from original unmodified getPackageName method



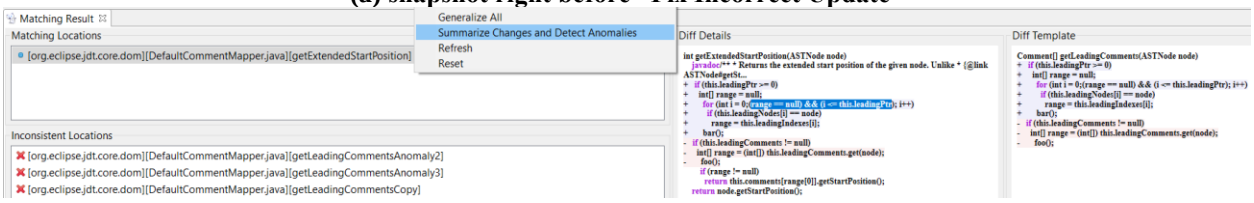
(b) results from new getPackageName method

Fig. 14 Comparison of results from old and new getPackageName method

As we can see, original getPackageName method could not tell the difference between consistent updates and inconsistent updates while the new one fixed such issue.



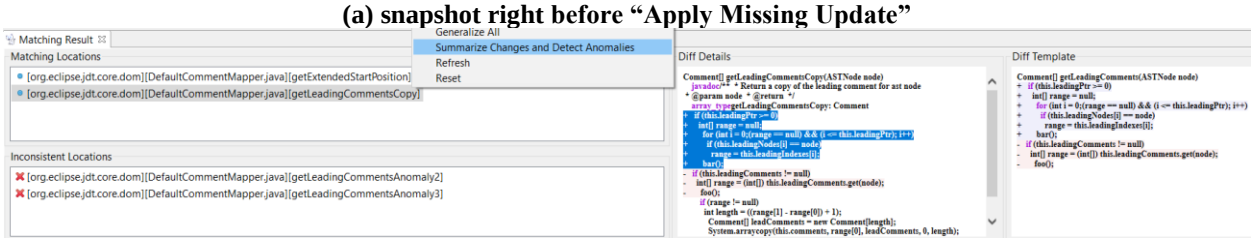
(a) snapshot right before "Fix Incorrect Update"



(b) snapshot after "Fix Incorrect Update" and "Summarize Changes and Detect Anomalies"

Fig. 15 Comparison of snapshots before and after "Fix Incorrect Update"

(a) snapshot right before "Apply Missing Update"



(b) snapshot after "Apply Missing Update" and "Summarize Changes and Detect Anomalies"
Fig. 16 Comparison of snapshots before and after "Apply Missing Update"

We have successfully implemented new "Fix Incorrect Update" feature as a pop-up menu item attached to "Inconsistent Locations" table viewer in CRITICS2 using a simple algorithm shown in Algorithm 2 and 4. The resulting snapshots before and after "Fix Incorrect Update" were shown in Fig. 15. As we can see, the incorrect insertion update with a wrong for-loop condition expression was clearly highlighted in "Diff Details" browser viewer as yellow in Fig. 15 (a). After we right-clicked on the selected getExtendedStartPosition method and chose/clicked "Fix Incorrect Update" pop-up menu, the selected method in the new version of source file DefaultCommentMapper.java was updated. After running "Summarize Changes and Detect Anomalies" again, getExtendedStartPosition method previously shown in "Inconsistent Locations" table viewer in Fig. 15 (a) was moved up to "Matching Locations" table viewer in Fig. 15 (b) as expected. Besides that, we can see the incorrect for-loop condition expression has been successfully transformed to the expected value as shown in "Diff Details" browser viewer of Fig. 15 (b).

We also successfully implemented new "Apply Missing Update" feature as a pop-up menu item attached to "Inconsistent Locations" table viewer in CRITICS2 using a simple algorithm shown in Algorithm 3 and 4. The resulting snapshots before and after "Apply Missing Update" are shown in Fig. 16. As we can see, all missing deletions were clearly highlighted in "Diff Details" browser viewer as yellow in Fig. 16 (a). After we right-clicked on the selected getLeadingCommentsCopy method and chose/clicked "Apply Missing Update" pop-up menu, the selected method in the new version of source file DefaultCommentMapper.java was updated. After running "Summarize Changes and Detect Anomalies" again, getLeadingCommentsCopy method previously shown in "Inconsistent Locations" table viewer in Fig. 16 (a) was also moved up to "Matching Locations" table viewer in Fig. 16 (b) as expected. Besides that, we can see the corresponding missing deletions have been successfully transformed to the expected insertions as shown in "Diff Details" browser viewer of Fig. 16 (b).

A short demo to show above two features can be viewed at https://use.vg/haQGwH.

VI.3. Discussions

VI.3.1. Problems and issues arisen during the research

One problem I have found during the research is that the anomaly Nodes found by method findNodeOfAnomaly as shown in Algorithm 1 are general tree Node type extending javax.swing.tree.DefaultMutableTreeNode defined by ChangeDistiller algorithm [Fluri et al. 2007] in **III.2.1**. However, usually org.eclipse.jdt.core.dom.ASTNode type is used to directly modify Java source codes. Therefore, we must find a way to convert type Node to type ASTNode or find corresponding ASTNode in a ASTVisitor subclass from the info provided by those anomaly Nodes. Though the ChangeDistiller Node type has a method called getMethodDeclaration which can be used to convert a Node type with label "METHOD" to MethodDeclaration type (a subclass of ASTNode type), in general not any arbitrary Node type can be converted to its corresponding ASTNode type. However, it is always possible to find corresponding ASTNode in a ASTVisitor subclass from the info provided by those anomaly Nodes. Eventually I have found method getASTNode(Node node, ASTNode root) in file CriticsGenerizationOld.java which can be used to convert an arbitrary Node type to its corresponding ASTNode type given an enclosing root ASTNode being searched. I have reused this getASTNode method in file CriticsOverlayViewMenu.java to handle special case for "FOR_STATEMENT" while implementing feature "Fix Incorrect Update" using Algorithm 2 and convert missingDeletionNodes to their corresponding ASTNodes to help locate their source code range while implementing feature "Apply Missing Update" using Algorithm 3.

Another useful Class which helped me fix the serious bug shown in Fig. 9 and 10 is the generic Class UTGeneralVisitor<T> in file UTGeneralVisitor.java. This generic Visitor class allows us to get a list of ASTNode with specified type <T>. For example, we can use it to find a list of MethodDeclaration as shown in method findMethods of file UTASTNodeFinder.java. I have implemented method findIfStatement in file UTChange.java to get a list of IfStatement while debugging the third bug as discussed in **VI.1.1.**

An important choice we need to make is which ASTNode type should be visited to apply ASTRewrite operations to correct those detected anomaly Nodes. In principle, we can visit more specific ASTNode types such as MethodInvocation, ForStatement, IfStatement, WhileStatement, VariableDeclarationStatement etc. since only those specific ASTNode types need to be updated. However, it is a challenge to visit all these potential specific ASTNode types in a ASTVisitor subclass since the types of anomaly Nodes can be any subclass of ASTNode depending on the target source codes being analyzed. Therefore, a more sensible strategy is to prepare a string

representation of updated method body statements from detected anomaly Nodes and their expected corrections as shown in method addMenuToAnomalyTableViewer implementing Algorithm 2 and 3. Then we only need to visit MethodDeclaration ASTNode in a ASTVisitor subclass and parse the string representation of updated method body statements to be a new method body block which can be used to replace the entire old method body block as shown Class ReplaceMethodBodyVisitor implementing Algorithm 4.

Another important discovery I have to mention again is that the string representation of the compilation unit *srcNewRev* obtained from source file is actually different from *srcNewViewer* obtained from org.eclipse.jface.text.source.ISourceViewer. This can be clearly seen from the debug info added to file CriticsOverlayViewMenu.java as shown in the following console output.

```
[DBG6] srcOldFile.length(): 19305
[DBG6] srcNewFile.length(): 23748
[DBG6] srcOldViewer.length(): 18714
[DBG6] srcNewViewer.length(): 23061
```

Fig. 17 Comparison of snapshots before and after "Apply Missing Update"

As we can see, the string representation *srcNewFile* of the compilation unit obtained from source file DefaultCommentMapper.java in the new revision has a length of 23748 which is clearly different than the length (23061) of *srcNewViewer* obtained from org.eclipse.jface.text.source. ISourceViewer. This required us to use either *srcNewFile*.substring() or *srcNewViewer*.substring () to get proper substring depending on whether getStartPosition() is based on *srcNewFile* or *srcNewViewer*. The same is true for the length difference between *srcOldFile* (19305) and *srcOldViewer* (18714) for the old revision.

VI.3.2. Limitations and constraints of the research (and results)

One limitation of the research is that the string representation of the detected anomaly Nodes should be unique in its method body since we used *updatedMethodBodyStms*.indexOf (incorrectInsertionNodeStr) to locate its index in updated method body statements. Otherwise, other regular Nodes with same string representation may be modified unintentionally as a collateral damage.

As mentioned in **1.4**, current CRITICS2 is still limited to be used in old Eclipse Luna IDE.

VII. Summary

In the field of code review the major difficulty is lack of tool support for understanding code changes. CRITICS's interactive and iterative customization of the change template allows code reviewers, especially novice programmers, to summarize similar changes and detect inconsistent change anomalies without a deep understanding of the codebase. However, CRITICS does not allow developers to automatically repair those inconsistent anomalies. In this work CRITICS has been extended to support automatic mitigation of detected inconsistent anomalies so that the developed new hybrid tool CRITICS2 will benefit both code reviewers and code developers. In particular we implemented a new method findNodeOfAnomaly with a new algorithm shown in Algorithm 1 which could detect all four types of special Nodes, namely incorrect inserted Nodes, incorrect deleted Nodes, missed insertion Nodes and missed deletion Nodes and clearly highlighted these anomaly Nodes with yellow background color in "Diff Details" browser view. We have successfully implemented new "Fix Incorrect Update" feature as a pop-up menu item attached to "Inconsistent Locations" table viewer in CRITICS2 using a simple algorithm shown in Algorithm 2. We also successfully implemented new "Apply Missing Update" feature as a pop-up menu item attached to "Inconsistent Locations" table viewer in CRITICS2 using a simple algorithm shown in Algorithm 3. We also proposed a sensible strategy that only needs to visit MethodDeclaration ASTNode in a ASTVisitor subclass ReplaceMethodBodyVisitor implementing Algorithm 4 where a ASTRewrite operation will be applied to correct those inconsistent anomalies. In the future we plan to replace outdated RTED tree matching algorithm with new APTED algorithm and adapt the tool for use in the latest version of Eclipse IDE.

References

A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," *35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, 2013, pp. 712-721, doi: 10.1109/ICSE.2013.6606617.

M. Barnett, C. Bird, J. Brunet and S. K. Lahiri, "Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets," *IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, 2015, pp. 134-144, doi: 10.1109/ICSE.2015.35.

B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, Volume 33, Issue 11, 2007, pp. 725 - 743, doi: 10.1109/TSE.2007.70731

S. Forrest, T. Nguyen, W. Weimer, and C. Goues, "A genetic programming approach to automated software repair," *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, New York, NY, 2009, pp. 947 – 954, doi: 10.1145/1569901.1570031

B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, 2010, pp. 1019 – 1028, doi: 10.1145/1753326.1753478

S. Herrmann, "JDT," https://wiki.eclipse.org/JDT (as of April 13, 2020).

K. Herzig and A. Zeller, "The impact of tangled code changes," *10th Working Conference on Mining Software Repositories*, San Francisco, CA, 2013, pp. 121-130, doi: 10.1109/MSR.2013.6624018.

D. Kim, J. Nam, J. Song and S. Kim, "Automatic patch generation learned from human-written patches," *35th International Conference on Software Engineering*, San Francisco, CA, 2013, pp. 802 - 811, doi: 10.1109/ICSE.2013.6606626.

M. Kim and D. Notkin, "Discovering and representing systematic code changes," *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, 2009, pp. 309 – 319, doi: 10.1109/ICSE.2009.5070531

C. Le Goues, M. Dewey-Vogt, S. Forrest and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," *34th International Conference on Software Engineering*, Zurich, 2012, pp. 3 - 13, doi: 10.1109/ICSE.2012.6227211.

N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, 2011, pp. 329 – 342, doi: 10.1145/1993498.1993537

N. Meng, M. Kim and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," *35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, 2013, pp. 502-511, doi: 10.1109/ICSE.2013.6606596.

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.

M. Pawlik and N. Augsten, "Tree edit distance: Robust and memory- efficient," *Information Systems* Volume 56, 2016, pp. 157-173, doi: 10.1109/ICSE.2013.6606596.

X. Ren, F. Shah, F. Tip, B. G. Ryder and O. Chesley. "Chianti: a tool for change impact analysis of Java programs," *Proceedings of the ACM SIGPLAN Conf. on Object Oriented*

Programming Languages and Systems, Vancouver, Canada, 2004, pp 432 - 448, doi: 10.1145/1028976.1029012.

R. Rolim, G. Soares, L. Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki and B. Hartmann, "Learning Syntactic Program Transformations from Examples," *IEEE/ACM 39th International Conference on Software Engineering*, Buenos Aires, 2017, pp. 404 - 415, doi: 10.1109/ICSE.2017.44.

Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, 2012, pp 1 – 11, doi: 10.1145/2393596.2393656.

Y. Tao and S. Kim, "Partitioning Composite Code Changes to Facilitate Code Review," *IEEE/ACM 12th Working Conference on Mining Software Repositories*, Florence, 2015, pp. 180-190, doi: 10.1109/MSR.2015.24.

L. Vogel, "Platform project," https://wiki.eclipse.org/Platform (as of March 24, 2016).

C. Windatt, "PDE/User Guide," https://wiki.eclipse.org/PDE/User_Guide (as of January 20, 2011).

Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, New York, NY, 2005, pp 54 – 65, doi: 10.1145/1101908.1101919

T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive Code Review for Systematic Changes," *IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, Italy, 2015, pp.111 – 122, doi: 10.1109/ICSE.2015.33.