Identifying the Within-Statement Changes to Facilitate Change Understanding

Chunhua Yang
School of Computer Science and Technology
Qilu University of Technology (Shandong Academy of Sciences)
Jinan, China
jnych@126.com

Jim Whitehead
Dept. of Computational Media
University of California, Santa Cruz
Santa Cruz, USA
ejw@soe.ucsc.edu

***Abstract*—As current tree-differencing approaches ignore changes that occur within a statement or do not present them in an abstract way, it is difficult to automatically understand revisions involving statement updates. We propose a tree-differencing approach to identifying the within-statement changes. It calculates edit operations based on an element-sensitive strategy and the longest common sequence algorithm. Then, it generates the metadata for each edit operation. Metadata include the type of operation, the type of entity and the name of the element part, the content, the content pattern and all references involved. We have implemented the approach as a free accessible tool. It is built upon ChangeDistiller and refines its statement-update type. Finally, to demonstrate how to use the proposed approach for change understanding, we studied the condition-expression changes in four open projects. We analyzed the non-essential condition changes, the `effective` changes that definitely affect the condition, and other changes. The results show that for revisions with condition-expression changes, nearly 20% contain non-essential changes, while more than 60% have effective changes. Furthermore, we found many common patterns. For example, we found that half of the revisions with effective changes were caused by adding or removing expressions in logical expressions. And, in these revisions, 47% enhanced the condition, while 49% weakened it.**

***Keywords*-code differencing; code changes; change understanding; software evolution**

I. Introduction

Code differencing approaches play an important role in helping people understand how software changes and evolves. By differencing two versions of the source code, they present a set of code changes, which alleviate users from the burden of checking the entire source code to find out where the code changes. In particular, tree differencing approaches such as ChangeDistiller [1] provide results in an abstract way from the syntax perspective, which makes automatic change analysis feasible. Therefore, they have been used by various researchers to promote their work, including analyzing the impact of changes [2], grouping the changes involved in the same task [3], predicting types of code changes [4], identifying the components that evolved together [5], and so on.

However, current tree differencing approaches focus on signature changes and statement-level changes. Changes that occur within a statement are ignored or indistinguishable from other changes. Approaches such as ChangeDistiller [1] adopt the former way. For each modified statement, ChangeDistiller returns a statement-update type, ignoring the internal change details. On the other hand, approaches such as GumTree [6], MTDIFF [7], JSync [8], and IJM [9] deal with the within-statement changes in the latter way. Although they generate edit operations for each statement update, they do not provide different types for the within-statement changes and other changes.

Updating statements are common in everyday software revisions. They act alone or in collaboration with other changes to achieve change tasks. Ignoring the within-statement changes or presenting them in the same way as other changes can hinder the understanding of changes in revisions that contain statement updates.

- First of all, ignoring the change details does not help to determine whether the statement-update is essential. For example, a statement can be modified by renaming multiple variables inside it, or simply updating an operator <= to <. The former is non-essential, whereas the latter usually fixes an important bug. For each change, ChangeDistiller returns a statement-update. Depending on the type itself, it is impossible to determine if the statement-update is essential. Non-essential changes can cause inaccuracies in high-level interpretations of software development effort [10]. Hence, it is important to distinguish between essential and non-essential changes.
- Secondly, returning indistinguishable script for both the within-statement changes and other changes makes it difficult to determine whether the script describes a within-statement change and where the change occurred. For example, as shown in Fig.1, GumTree returns a script of three operations for the if-statement update. Based on the script, it can be inferred that the first element part of a method invocation is updated with a class instance creation expression. But, based solely on the script, we can't know that the change occurred within the if-statement, since a method invocation can occur anywhere in a statement or declaration that allows an expression.

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00030

- Finally, ignoring within-statement changes or presenting them in the same way as other changes can hinder the analysis of dependencies between statement updates and other changes. Change dependency analysis is important for analyzing the impact of changes, grouping changes that accomplished the same task, or decoupling tangled code changes, etc. For example, adding a method parameter will cause all statements that call the method to be updated. To group the changes that accomplished this refactoring task, the dependency between the method declaration and all the changes to the related method calls is the basis. However, as the edit operations that describe the within-statement changes are entangled and indistinguishable from the operations that describe other changes, it is difficult to determine which parts of the script belong to a statement-update and which parts belong to a signature change, not to mention the difficulty of analyzing the dependencies among these changes.

To alleviate the issue, we make the following contributions in the paper:

- We propose a tree-differencing approach to identifying the within-statement changes and presenting them in an abstract way. For a statement update, it returns a set of changes by differencing two versions of the statement tree. Each change corresponds to an edit operation and is encapsulated as metadata. Metadata contains the type of operation, the entity and the element parts that have changed, all references involved, the content before and after the change, and content pattern (if any), etc.
- And, we have implemented the proposed approach as a free accessible tool[1]. It is based upon ChangeDistiller, extracting all the within-statement changes for each `statement-update` and outputting their metadata.

The target users of the proposed approach include researchers whose purpose is to analyze code changes in each submission, or to explore the laws in software evolution by mining software repositories. For the former researchers, they can reveal the dependencies between changes, by combining statement-level differencing approaches (such as ChangeDistiller) with metadata identified by the proposed approach for each statement update. For the latter researchers, each metadata identified from a statement update can be used to uncover the frequencies of the non-essential statement updates, filter out the non-essential statement-updates, and reveal the common statement-update patterns, etc. We will demonstrate how to apply the proposed approach to change understanding and its usefulness in the case study section. Besides, the proposed approach is beneficial to tool developers whose goal is to build tools that present changes in a commit. They can integrate the proposed algorithm into their work, providing detailed information on the changes within a statement from a syntax perspective.

[1]https://github.com/jinanych/StmtDiff

```
        // Try to connect to the server
-       if(portFile != null && portFile.exists())
+       if(portFile != null && new File(portFile).exists())
        {
```

(a) The statement update

```
DEL SimpleName: portFile
INS SimpleName: portFile to ClassInstanceCreation at 1
INS ClassInstanceCreation to MethodInvocation at 0
```

(b) The GumTree script

Figure 1. A sample statement update.

The remainder of the paper is organized as follows. In Section 2, we present the approach. In Section 3, we present the current implementation and the case study. In Section 4, we review the related work. We summarize the paper in Section 5.

II. The Approach

In this section, we first outline the approach. Then, we propose the differencing algorithm and metadata extraction.

A. Overview

The approach consists of two phases. In the first phase, it inputs two versions of the statement tree and executes a top-down match between the two trees to generate mappings of nodes. Then, in the second phase, it generates the edit operation for each mapping and extracts the metadata for each operation.

From the syntax perspective, a statement corresponds to a subtree of the AST of the source code. We call it a `statement tree`. For instance, the two versions of the statement tree of the if-statement in Fig.1 are shown in Fig.2.

In a statement tree, the root is the statement entity and the internal nodes are expression entities or reference entities. Each entity has one or more element parts. Each node in a statement tree has a type and a value. The type is an entity kind, while the value is the textual representation of the entity. In the remainder of the paper, we use the entity types defined in the eclipse jdt parser.

A `mapping` is a tuple <l_1, l_2>, where nodes l_1 and l_2 are in the old version and the new version tree, respectively, with similar value.

1) The Matching Phase: Let T_1 and T_2 be the roots of the old and new version of a statement tree, respectively, and T_1 matches T_2 by default. Starting with the matching tuple <T_1, T_2>, it compares their children to find matching tuples and then recursively checks the matching tuples. When it reaches a tuple with only one node or two different type of nodes, the matching process ends and the tuple is output as a mapping.

We adopt two strategies in the mapping phase: an element-sensitive strategy and a strategy based on the longest common subsequence algorithm.

Usually a statement tree is more complex than a declaration tree, because the elements of a statement are expressions and different types of expressions usually have different element parts. For example, a method invocation have an optional scope, a name, and optional arguments, while an assignment has a target and a value. Therefore, for a tuple <l_1, l_2>, we generally adopt the element-sensitive strategy to match their children. Specifically, assuming that $e_1, ..., e_n$ are the element parts of l_1 and l_2, then all children of l_1 in the e_1 part will be compared with all children of l_2 in the e_1 part, all children of l_1 in the e_2 part will be compared with all children of l_2 in the e_2 part, and so on. The matching process of the two versions of the statement in Fig.1 is shown in Fig.2. The green nodes in T and T' connected by dashed lines match each other. The tuple <n_1, n_2> is the mapping generated by the matching process.

However, if both nodes are recursive expressions, element-sensitive matching is inefficient. We have found recursive logic expressions with depths greater than 70. In this case, we use the longest common subsequence algorithm to align their children.

2) Script Generation and Metadata: Based on the mappings produced during the matching stage, a script is to be generated. The script is a sequence of edit operations. Current differencing approaches usually identify four kinds of edit operations: *add*, *delete*, *move*, and *update*.

A mapping <l_1, l_2> corresponds to an edit operation. If l_1 is null, it corresponds to *add*. If l_2 is null, it corresponds to *delete*. If both l_1 and l_2 are not null and their value are similar, then it corresponds to *update*. For the move operation, its definition varies in different approaches. In this paper, we define that the mapping <l_1, l_2> corresponds to a *move*, if l_1 and l_2 belong to the same element part and they have the same value but different order, or the value of l_1 is the same as a descendent of l_2, and vice versa. The former move is often called *order-change*. For the latter two, we call them *encapsulation* and *decapsulation* respectively. Fig.3 illustrates the six kinds of operations.

In the rest of the paper, we use the following terms to describe the edit operations. We call each node in the mapping <l_1, l_2> *terminate*, its parent *parent*, the element part it belongs to *element*, the path from the root to the parent *context*, the value of l_1 *old content*, and the value of l_2 *new content*. For example, for the mapping <n_1, n_2> in Fig.2, its root entity and element part is `IfStatement_condition`, its parent entity and element part is `MessageSend_scope`, its operation kind is `encapsulation`, and its content is `portFile]new File(portFile)`. Note that we separate the entity and element parts with _, and separate the old and new values with `]`.

B. The Differencing Algorithm

Algorithm 1 describes the differencing process. It inputs two trees T_1 and T_2 and outputs a set of mappings. Each mapping is of the form <$n_1, n_2, kind$>, where n_1 and n_2 are matching nodes, and $kind$ is the type of operation.

It first executes the $terminateCheck$ function to check if the termination condition is met. Then, if T_1 and T_2 are both recursive expressions, it will call $DiffRecursiveExpr$ to match their children. Otherwise, it calls $DiffElements$ to perform element-sensitive matching.

The terminate conditions are as follows:

(1) T_1 or T_2 is null or a leaf;
(2) T_1 and T_2 are different types of entities;
(3) T_1 and T_2 are in the same element part, have the same value but different order;
(4) T_1 has the same value as a descendent of T_2 and vice versa.

If the third condition is met, the mapping corresponds to order-change. If the fourth condition is met, it corresponds to encapsulation or decapsulation. If the first condition is met and one node is null, then it corresponds to add or delete. If the first condition is met and both nodes are not null and their relationship follows encapsulation or decapsulation, then it corresponds to encapsulation or decapsulation. Otherwise, it corresponds to update. If the second condition is met, the encapsulation and decapsulation will also be checked.

Algorithm 1: Differencing(T_1, T_2)

Input: trees T_1 and T_2
Output: the mapping set M

```
kind ← terminateCheck(T1, T2);
if kind ≠ "" then
    M ← M ∪ {< T1, T2, kind >}; return;
if isResuriveExpr(T1) ∧ isResuriveExpr(T2) then
    M ← DiffRecursiveExpr(T1, T2);
else
    M ← DiffElements(T1, T2);
```

1) The Element-Sensitive Differencing: The algorithm is described in Algorithm 2. It inputs two trees T_1 and T_2, and outputs the mappings. It first gets their children S_1 and S_2, respectively. The function $elementNames$ gets a set of element parts from each children set, and the function $nodesInElement(S, e)$ gets the nodes belonging to the element part e from the set S.

For each element part, if there are only nodes of S_1 in it, then the algorithm generates a delete mapping for each node. If there are only nodes of S_2, then the algorithm generate an add mapping for each node. Otherwise, it first removes nodes with the same value. The remaining nodes are then compared to each other to find similar nodes. Note that, if there is only one node in S_1 and there is only one node in S_2, they match by default. For matching nodes, the algorithm recursively calls $Differencing$ to get the mappings. And, for each unmatched node, it generates a delete or add mapping.

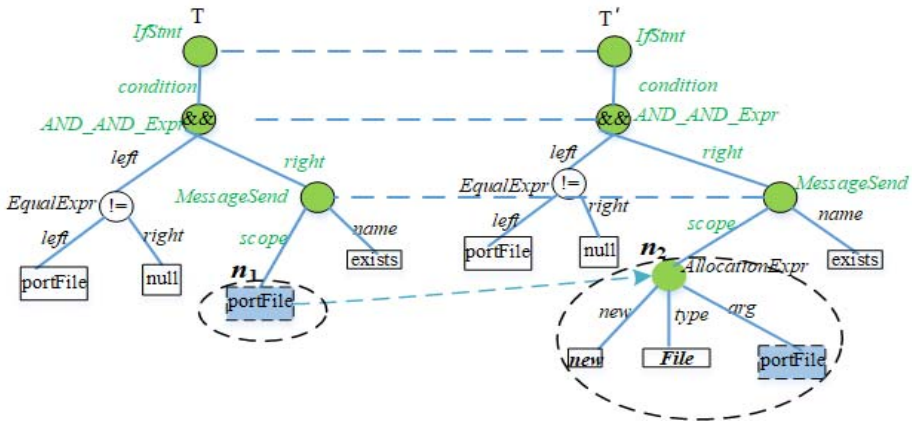


Figure 2. The process of matching two statement trees.

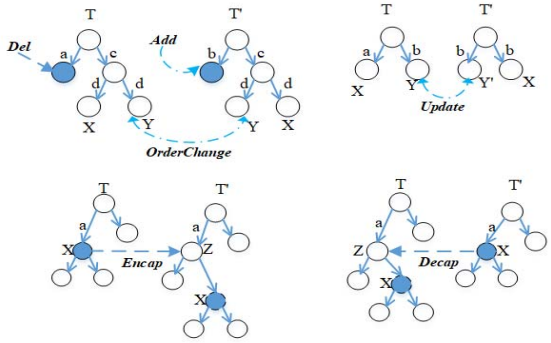


Figure 3. Six kinds of edit operations.

Fig.2 illustrates the element-sensitive differencing process. Firstly, root T_1 matches T' by default. As each if-statement has only one `and_and_`expression in its condition part, the two expressions match. Each `and_and_`expression has a $left$ part and a $right$ part. In the $left$ part, both versions of the node have the same value `portFile!=null`, so it didn't change. However, in the $right$ part, since the two versions of the method call have different values, they match. The two method invocations have the same name, but there is an encapsulation relationship between the two versions of the nodes in the scope part. Therefore, the matching process terminates. It returns an encapsulation mapping $<n_1, n_2>$.

Suppose N_1 and N_2 are the number of nodes of the old and new versions of AST, and h_1 and h_2 are depth, respectively. The differencing algorithm is based on level traversal. Therefore, its time complexity is $O(c_{avg} \times min(h_1, h_2))$, where c_{avg} is the average number of comparisons between nodes on each level. Assume that the comparison on each level is performed on child nodes of the same kind of entities, e_{arg} is the average number of element parts of the entity, and n_{avg} is the average number of nodes in each element part. Then, c_{avg} is $e_{arg} \times n_{avg}$. Expression entities and statement entities generally have only a few element parts, and the number of nodes in each element part is small. Therefore, the main cost of the algorithm depends

Algorithm 2: DiffElements(T_1, T_2)

Input: trees T_1 and T_2
Output: the set M of mappings

$S_1 \leftarrow T_1.children$, $S_2 \leftarrow T_2.children$, $M \leftarrow \varnothing$;
$E_1 \leftarrow elementNames(S_1)$;
$E_2 \leftarrow elementNames(S_2)$;
for *each name* $e \in E_1 - E_2$ **do**
  for *each node* $x \in nodesInElement(S_1, e)$ **do**
    $M \leftarrow M \cup \{(x, null, \text{“}Delete\text{”})\}$;
for *each name* $e \in E_2 - E_1$ **do**
  for *each node* $y \in nodesInElement(S_2, e)$ **do**
    $M \leftarrow M \cup \{(null, y, \text{“}Add\text{”})\}$;
for *each name* $e \in E_1 \cap E_2$ **do**
  $X \leftarrow nodesInElement(S_1, e)$;
  $Y \leftarrow nodesInElement(S_2, e)$;
  remove nodes with the same value from X and Y;
  if $|X| = 1 \wedge |Y| = 1$ **then**
    $M \leftarrow M \cup Differencing(x, y), x \in X, y \in Y$;
  else
    $N_{sim} \leftarrow findSimilarNodePairs(X, Y)$;
    for *each pair* $(x, y) \in N_{sim}$ **do**
      $M \leftarrow M \cup Differencing(x, y)$;
      remove x and y from X and Y respectively;
    for *each node* $x \in X$ **do**
      $M \leftarrow M \cup \{(x, null, \text{“}Delete\text{”})\}$;
    for *each node* $y \in Y$ **do**
      $M \leftarrow M \cup \{(null, y, \text{“}Add\text{”})\}$;

mainly on the depth of the tree. But, even in the worst case, $O(c_{avg} \times min(h_1, h_2))$ will not exceed $O(N_1 \times N_2)$, as the comparison is done independently within each element part.

2) The Recursive-Expression Differencing: If an element part of an expression is also an expression of the same type, we define it as a recursive expression. And, we set the threshold of recursion depth to 5. Usually, recursion occurs in method calls, arithmetic expressions, or logic expressions. For example, method invocations are often recursive in the *scope* part, which looks like $a(*).c(*)....d(*)$. Binary expressions are usually recursive in the $left$ part, which looks like $(...\&\&(x > 5))\&\&(c4 < 0))$.

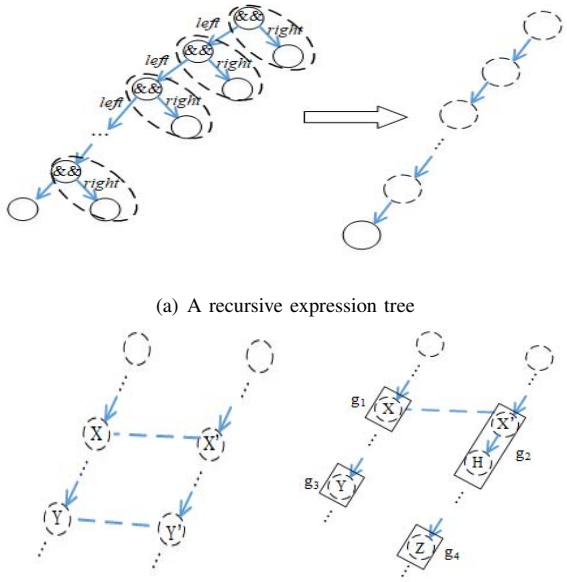
Take a binary expression as an example. It has a $left$, an *operator*, and a $right$ part. If we treat nodes in the $right$ part as a leaf, and the operator and nodes in the $right$ part as a composite node, then the recursive binary expression tree becomes a single branch tree, as shown in Fig.4(a). As mentioned before, the depth of a recursive expression can be very high, so the cost of the element-sensitive differencing is high. Since the two versions of the recursive expression have the same operator, the comparison between the operators is unnecessary. After ignoring the comparison of operators, differencing of the two trees becomes a comparison between the leaves.

The strategy for matching recursive expressions is as follows. Firstly, we get a sequence of leaves by post-traversing each version of the recursive expression tree. Then, we use the longest common subsequence algorithm to find the leaves that have not changed. For the remaining leaves, we match them according their level in the tree. Usually, the number of remaining leaves in the old version is the same as that in the new version, and each leaf in the old version is at the same level as the corresponding leaf in the new version, as shown in Fig.4(b). We call it `symmetric` pattern. However, sometimes some nodes do not have corresponding nodes at the same level, such as nodes Y and Z in Fig.4(c). We call it `asymmetric` pattern.

Symmetric-matching. In the case of the symmetric pattern, the nodes in the old version match the nodes of the same level in the new version, e.g. X and X' in Fig.4(b). Element-sensitive differencing is then performed on the matching nodes to find the mappings.

Asymmetric-matching. In the case of the asymmetric pattern, we first group successive leaves. Then, for each group in the old version and the new version, we execute a top-down matching. The process is as follows.

Let g_1 and g_2 be the current group in the old and new versions, respectively. If the relative level of group g_1 is equal to the relative level of group g_2, then an inner matching will be made on the nodes of the two groups. If the relative level of g_1 is less than that of g_2, then all nodes of g_1 are recognized as delete mappings. Otherwise, all nodes of g_2 are recognized as add mappings. Here the relative level of a node is the absolute level minus the number of nodes



(a) A recursive expression tree

(b) Symmetric matching

(c) Asymmetric matching

Figure 4. Recursive expression differencing.

previously removed or added. The relative level of a group is the relative level of its first node. For instance, in Fig.4(c), assuming that nodes X and X' are at level 5, their relative levels are 5. Assuming that node Y is at level 8 and Z is at level 11, then the relative level of Y is 7, while the relative level of Z is 9. Therefore, g_1 matches g_2, while g_3 and g_4 do not match.

The inner matching between groups g_1 and g_2 is an alignment. In this process, similar nodes match each other as an update mapping, while other nodes are identified as add or delete mappings. For example, as shown in Fig.4(c), the inner matching will identify a mapping $<X, X'>$, assuming X and X' are similar, and node H is identified as an add.

Algorithm 3: DiffRecursiveExpr(T_1, T_2)

Input: trees T_1 and T_2
Output: the mapping set M

```
1 L1 ← genList(T1), L2 ← genList(T2) ;
2 (D1, D2) ← lcsDiff(L1, L2);
3 if isSymmetricPattern(D1, D2) then
4 |   M ← M ∪ Differencing(x, y), x ∈ D1 ∧ y ∈
  |     D2 ∧ x.height = y.height;
5 else
6 |   M ← M ∪ innerGroupMatching(D1, D2)
```

Algorithm 3 implements recursive expression matching. In the algorithm, the function $genList$ gets a sequence of leaves, $lcsDiff$ gets the changed leaves, and $innerGroupMatching$ performs the inner group matching.

C. Metadata Extraction

The mappings are obtained through the matching phase, and each mapping corresponds to one operation. We extract the following metadata for each operation:

- `whichpart` that describes the parent entity and the element part;
- `rootwhichpart` that describes the root entity and the element part;
- `context` that describes the context;
- `action` that describes the operation;
- `left entity type` and `right entity type` that describe the types of terminate entities in the old and new versions respectively;
- `content` that describes the content;
- `content pattern` that describes the pattern between old and new values;
- `references` that describe all references that appear in the terminate entity tree.

Except for the content pattern and references, other metadata can be extracted directly from the terminate entity. For operations other than add, delete, and order-change, we extract their content patterns by replacing the common tokens between the old content and the new content with asterisks. A token corresponds to the value of the leaf in the terminate entity. For example, for content `portFile]new File(portFile)`, its content pattern is `*]new File(*)`.

Extracting the content pattern from encapsulation or decapsulation is easy. Take encapsulation for instance. Suppose x is encapsulated in Y as its descendant x'. We replace the old content with an asterisk. Then, we perform an in-order traversal on Y. During the process, when x' is reached, its token value is replaced with an asterisk. For other types of operations, we first obtain the token sequence for the old and new versions, respectively. Then, we use the longest common sequence algorithm to find the common tokens in the two sequences. Finally, we replace the common tokens with asterisks.

The references are extracted by traversing the tree of the terminate entity. A reference can be a field reference, a name reference, a type reference or a method reference. For each reference, we return the node and its type and name.

III. The Implementation and Case Study

We have implemented the proposed approach based on ChangeDistiller. Given two versions of the java source code file, ChangeDistiller generates the statement-level changes. And, for each statement update, we apply the proposed approach to extract the within-statement changes. Therefore, in addition to the original ChangeDistiller changes, the extended ChangeDistiller outputs the metadata for each operation extracted from each statement-update.

We choose ChangeDistiller to integrate the propose approach due to two reasons. First of all, ChangeDistiller is very efficient. And, since it returns classified changes, the statement update is easy to identify. Furthermore, based on the classification of ChangeDistiller, it is easy to find signature changes, from which we can search for referenced entities. Next, we'll demonstrate how to use the proposed approach and ChangeDistiller to understand and analyze the changes in a commit.

Condition-expression-change is a statement update that is typically used in code revisions. Furthermore, changes to the condition of an if-statement or a loop statement may affect its internal statements and sometimes affect software functionality, depending on how the expression is changed. Therefore, we conducted a case study on condition-expression-change, which is to answer the following two research questions:

- **RQ1:** Whether the value of the condition has changed?
- **RQ2:** How does the value change?

To answer the first research question, we examined the `Non-Essential` changes, including the common *add this* pattern and changes caused by referencing declarations or variables that were renamed, added, deleted, or moved.

To answer the second research question, we analyzed the rest of changes. We distinguish between changes that definitely affect the condition value and changes that may affect the condition value. We call the former `Effective` changes. For the latter, we classify them as `Dependent` or `Other` changes. A change is `dependent` if it replaces a reference with a literal or an expression but the referenced entity cannot be found in the change set or cannot be evaluated, otherwise it is an `other` change.

We studied common patterns in each category and analyzed their frequency and distribution in selected projects. The data set is taken from four open projects: jEdit[2], eclipse JDT Core[3], Apache maven[4], and google-guice[5]. Using a MininGit tool[6], we extracted the change history of the selected projects from their source repository and stored into a database. Information of the selected projects is shown in Table I. Changes in these projects are extracted using the current implementation. We only studied the revisions that contain condition-expression changes. As a result, a total of 7,135 revisions were selected.

The distribution of revisions containing each category of changes is shown in Table II. According to the results, an average of **1,326 revisions have `non-essential` condition changes, accounting for 19% of the total revisions. 4,509 revisions contain `effective` changes, accounting for 63%**. Revisions with `dependent` and `other` changes accounted for 13% and 54%, respectively.

[2]https://github.com/linzhp/jEdit-Clone
[3]https://github.com/eclipse/eclipse.jdt.core
[4]https://github.com/apache/maven/
[5]https://github.com/google/guice/
[6]MiniGit: https://github.com/SoftwareIntrospectionLab/MininGit

Table I
STUDIED PROJECTS AND THEIR TOTAL NUMBER OF REVISIONS CONTAINING CONDITIONAL EXPRESSION CHANGES IN THE TIME PERIOD

Project	Period	Count of Revisions
jEdit	1998-09-27∼2012-08-08	1,520
JDT Core	2001-06-05∼2013-10-16	4,390
Maven	2003-09-01∼2014-01-29	1,077
Guice	2006-08-22∼2013-12-11	148
Total		7,135

Note that since a commit may have multiple condition changes and they may belong to different categories, the sum of the revisions in the four categories in Table II is greater than the total number shown in Table I. This phenomenon also appear in other tables shown in the following subsections.

Table II
DISTRIBUTION OF REVISIONS WITH CONDITION-EXPRESSION CHANGES IN THE FOUR CATEGORIES

Category	jEdit	JDTCore	Maven	Guice	Total
Non-essential	242	869	176	39	1,326(19%)
Effective	1,049	2,717	671	72	4,509(63%)
Dependent	194	569	125	9	897(13%)
Other	823	2,531	457	76	3,887(54%)

Next, we will describe how to classify and examine the changes in each category and present common patterns and their frequencies.

A. Non-Essential Patterns

Kawrykow and Robillard [10] defined several common non-essential changes, including *Trivial Keyword Modifications* that inserts or deletes the this keyword, *Trivial Type Updates* that replaces a simple name with its fully-qualified name, for example: List to java.util.List, *Local Variable Extractions* that uses a temporary variable to store an expression and replaces the expression in a statement with the variable, and *Rename-Induced* that references a renamed entity.

In addition to these patterns, the following refactoring operations cause non-essential changes in condition expressions:

- Move or copy a declaration(field, method, or class) from an entity to another, which results in the insertion, deletion, or update of the scope of all references to the declaration;
- Add or remove parameters of a method, which causes the arguments to be added or removed from related method invocations;
- Update a class, or change the type of a method, field, parameter, or local variable, which causes the update of the type references.

1) Definition and Check: We define the following patterns to classify the non-essential changes in condition expressions:

- `AddThis`: adds this keyword to a reference or an expression;
- `DeleteThis`: removes this keyword from the scope or qualifier of a reference or an expression;
- `Rename`: replaces a reference with another one because the referenced entity is renamed;
- `Move`: changes the qualifier of a reference because of the referenced entity being moved;
- `VarExtraction`: replaces a reference with a literal, an expression, or another reference with the same value;
- `ArgAddDel`: adds or deletes arguments of a method invocation or an allocation expression because the method's parameters are added or deleted;
- `OrderChange`: changes the order of the arguments of a method invocation because the order of the parameters of the method has changed;
- `TypeAddDelete`: updates a type reference because the referenced class is inserted, removed, moved, or renamed.

Non-essential changes are identified by examining the metadata for each edit operation extracted from the condition expression update, in conjunction with searching for the related signature changes from the change set extracted by ChangeDistiller. Since ChangeDistiller only analyzes changes in modified files, for added and deleted files, we extracted all the declarations from their AST and add them to the checklist.

The identification of non-essential patterns is as follows:

- The `AddThis` or `DeleteThis` pattern is identified by checking if the content pattern is like `*]this.*` or `this.*]*`;
- The `ArgAddDel` or `OrderChange` pattern is identified through searching for all the PARAMETER_INSERT, PARAMETER_DELETE, and PARAMETER_ORDERING_CHANGE changes from the change set and matching the method name to the name of the updated method call;
- The `Rename` pattern is identified by searching for renamed signatures, parameters, and local variables and matching them to the references;
- The `Move` pattern is identified as follows: Firstly each reference whose scope or qualifier is changed is a candidate. Then, all added and deleted signatures, parameters, and local variables are checked to match their names to the references;
- The `VarExtraction` pattern is identified as follows: First, the entity type in each metadata is checked to find the added, deleted, or updated reference, and the value assigned to the reference is taken, assuming it is v. Then, every added, removed, updated variable,

field, or assignment is checked to find the entity whose name matches the reference. If the entity matches the reference, then the value that assigned to the entity is retrieved and its value is compared to v. If they are equal, then this is a `VarExtraction` pattern;
- The `TypeAddDelete` pattern is identified as follows: Firstly each updated type reference is a candidate. Then, all deleted or added classes will be checked to find the entity with the same name as the reference.

2) Results: Table III lists the results. As shown in the table, on average, **`Rename` is the most common non-essential pattern**, which occurred in 591 revisions, accounting for 45% of non-essential revisions (1,326). The `Move` and `VarExtraction` pattern ranked second and third, accounting for 25% and 19% respectively. Next, revisions with `AddThis` and `ArgAddDel` patterns accounted for 10% and 9%, respectively. However, as far as `AddThis` is concerned, its frequency in jDTCore is significantly higher than other projects, which implies that this pattern is not a common pattern across projects.

The other three kinds of patterns are not common, which means that changes to type and order of parameters are not often involved in condition expression updates.

Table III
NUMBER OF REVISIONS WITH NON-ESSENTIAL PATTERNS IN EACH PROJECT

Pattern	jEdit	JDTCore	Maven	Guice	Total
Rename	90	410	68	23	591 (45%)
Move	70	187	66	5	328 (25%)
VarExtraction	54	164	28	5	251 (19%)
AddThis	3	130	1	1	135 (10%)
DeleteThis	6	32	1	0	39(3%)
ArgAddDel	27	80	16	2	125 (9%)
TypeAddDel	2	6	1	5	14 (1%)
OrderChange	0	3	0	0	3

B. Effective Patterns

Logical operators play an important role in controlling condition values. Updating a logical operator may change the condition. For example, in the following update, the operator != is replaced with ==, which reverses the condition value:

```
(mode != null)](mode == null).
```

However, such an update may not change the condition if accompanied by other changes. For instance, in the following condition expression, the update of the operator is accompanied by an update of the operand: 1 to 0. The value of the condition have not changed.

```
((index%2)==1)](index%2)!=0)
```

We name the first pattern `OpUpdate`, and the second `MoreThanOpUpdate`.

Moreover, by browsing the condition expression changes extracted from the dataset, we found that many of them added or removed expressions from the logical expression. Two examples are shown as follows. The first one removes from an `and_and` expression `!newFile`, while the latter adds `adirty` to an `and_and` expression, respectively. The former weakens the condition while the latter enhances it.

```
((path == null)&&(!newFile))](path==null);
dirty](dirty && adirty).
```

Similar changes can occur in expressions such as OR_OR expressions and equations. They definitely affect the condition value. We classify them as `ExprPattern` pattern.

In addition, replacing a literal with another literal usually affects the value of the expression. We name it `LiteralToLiteral`. And, if there are different types of updates in multiple places in a condition expression, its value typically changes. We name it `MultipleChanges`. Lastly, if a reference is replaced with a literal, an expression, or another reference, and either of them is null or their values are not equal, then the condition value usually changes. We name it `RefNotEqual`.

1) Check: The identification of these patterns is as follows:

- The pattern `OpUpdate` is identified through checking if there is only an operator update.
- The pattern `MultipleChanges` is easily identified through checking if there are multiple different changes. For example, adding $this$ to multiple references is not a `MultipleChanges` pattern.
- The pattern `ExprPattern` is identified by examining whether the content pattern of a logic or arithmetic expression update is encapsulation or decapsulation.
- The pattern `LiteralToLiteral` is identified by checking if both entities are literal.
- The pattern `RefNotEqual` is identified in the same way as `VarExtraction`. If their values are not equal, then it is a `VarExtraction` pattern.

2) Results: Table IV describes the number of revisions that contain `effective` patterns in each project. On average, **`ExprPattern` is the most common `effective` pattern**, and revisions containing this pattern account for 51% of the total revisions with `effective` patterns. `MultipleChanges` ranked second, accounting for 38%. And, 17% of the total revisions with `effective` patterns are caused by replacing references with new values.

Table IV
NUMBER OF REVISIONS WITH `EFFECTIVE` PATTERNS IN EACH PROJECT

Pattern	jEdit	JDTCore	Maven	Guice	Total
ExprPattern	548	1,412	330	26	2,316
MultipleChanges	361	1,099	224	30	1,714
RefNotEqual	250	626	119	21	1016
LiteralToLiteral	209	245	73	3	530
OpUdate	153	232	44	3	432

As the `ExprPattern` pattern is so common, we are

curious about what these patterns are. We extracted their content patterns, classified them according to the type of target expression, and calculated their frequencies. The result is shown in the left part of Table V, where the operator column represents the type of the target expression, the `Encap` column represents the encapsulation, and `Decap` represents the decapsulation. For example, as shown in the first row of the table, 1,084 revisions contain the `ExprPattern` pattern that encapsulated an expression to an `and_and` expression, while 564 revisions contain `ExprPattern` patterns that decapsulated an expression from an `and_and` expression. The former accounted for 47% of the total 2,316 revisions with `ExprPattern` patterns. This means that **47% of `ExprPattern` changes enhanced the condition**.

Meanwhile, according to Table V, the other two common patterns are to decapsulate some expressions from an `and_and` expression (564 revisions) and to encapsulate some expressions into an `or_or` expression (567 revisions). Both of them actually weakened the condition. The total number is 1,131, accounting for 49%. This indicates that **49% of `ExprPattern` changes weakened the condition**.

In addition, we checked the changes in the `OpUpdate` category to find the updated operators. The result is shown in the right part of Table V. As shown in the table, the most common pattern is the update between operator != and ==. And, the next common pattern is the update between < or > and >= or <=.

Table V
NUMBER OF REVISIONS WITH EXPRPATTERN PATTERNS, AND NUMBER OF REVISIONS WITH OP−UPDATE PATTERNS

Expression Patterns			Op-Update patterns		
Operator	**Encap**	**Decap**	**Operator**	**Operator**	**Update**
&&	1,084	564	!=	==	322
||	567	247	<, >	<=, >=	108
!	121	120	<	>	6
+, −, /	38	29	+	−	5
&, ^, |	12	3	+	|	1
!=, ==, >	8	7			

C. Dependent Patterns

As pointed out in previous subsections, when a reference is replaced with a literal, expression, or other reference, if the referenced entity is moved or updated and the updated value is equal to its original value, then it is a $VarExtraction$. If it is not equal, then it is a $RefNotEqual$ pattern. Otherwise, there are two cases. One is that the referenced entity can be found in the change set, but its value cannot be obtained directly, such as a method invocation. Another case is that the referenced entity cannot be found in this commit. We classify such changes as `Dependent`. And, if the reference is a name reference, we name it `NameRefToEval` pattern. If the reference is a method reference, we name it `MethodRefToEval` pattern.

If an operator is updated but accompanied by other changes, we name it `MoreThanOpUpdate`.

We counted the number of revisions with patterns in the `Dependent` category, listed in Table VI. As shown in the table, revisions with `NameRefToEval` and `MoreThanOpUpdate` patterns are nearly half of the total revisions with `Dependent` patterns, respectively.

Table VI
NUMBER OF REVISIONS WITH DEPENDENT PATTERNS IN EACH PROJECT

Reason	jEdit	JDTCore	Maven	Guice	Total
NameRefToEval	86	277	73	4	440(46%)
MoreThanOpUpdate	116	314	39	2	471(49%)
MethodRefToEval	2	22	12	2	38(6%)

D. Discussion and Threats to Validity

Changes in the `Other` category are usually irregular. If we assume that irregular changes usually affect the value of the condition, the changes in this category should be treated as `Effective` pattern.

For the `MoreThanOpUpdate` pattern, if in most cases the changes in this category affect the value of the condition, they should be classified as `Effective`. However, in some cases, the condition expression changes from a style to another style. We are unable to determine the proportion of these two situations.

In addition, in Table V, we considered the `ExprPattern` that enhanced or weakened the condition. They take the following four forms, where * represents any logic expression and e represents a logic expression that is added or removed.

`*&&e]*; *||e]*; *]*&&e; *]*||e` .

There are certainly other patterns that can enhance or weaken the condition.

We now discuss the threats to the validity. Firstly, as the results shown in the cast study, many patterns replaced a reference with another reference, literal, or expression. We only searched for entities that have changed in the current revision. It may have been updated in previous revisions because sometimes a change task is scattered across consecutive commits. Or, we can search in the AST of the source code, although it may take time. Even so, the definition of an entity is not always accessible, for example, an entity in a library. Therefore, patterns such as `NameRefToEval` and `MethodRefToEval` may be inaccurately classified as `Dependent`. Moreover, we only analyzed patterns that occurred frequently in the studied projects. In practice, there may be other common condition changes.

In addition, in the case study, we only considered the condition-expression-change type. Actually, ChangeDistiller identifies a statement update based on the similarity between the textual values of the statement before and after the update. If the similarity is below the threshold set

by ChangeDistiller, then it will identify a statement insert and a statement delete. Therefore, the number of condition expression changes selected in the case study may not be as large as the actual amount. Furthermore, as a demonstration of how to use the proposed approach in change analysis, the selected projects were written in Java and the size of the dataset was not very large. Moreover, some patterns such as $AddThis$ are related to the developer's programming habits. Therefore, patterns found in the study may not be used frequently in other projects.

IV. Related Work

Textual Differencing. The well-known GNU diff [11] can return added or removed lines. LDiff [12] and LHdiff [13] improve the GNU diff by detecting moved lines. However, textual-differencing approaches only return textual differencing lines or tokens, not structural changes.

Tree Differencing. Tree-differencing approaches return structural changes by comparing two ASTs representing two versions of the source file. The most famous algorithm is ChangeDistiller [1]. It detects changes in classes, methods, and fields. Furthermore, it classifies changes according to tree edit operations [14]. However, its granularity is limited to the statement-level, therefore no further analysis of expression-level changes is made.

GumTree [6] analyzes both declaration changes and the within-statement changes. However, it only generates raw script. As the script is not abstract enough, it is not easy to determine where the actions occurred, especially if there are multiple updates in a statement.

JSync [8] first uses the longest common subsequence algorithm to get the initial mappings. It then maps the nodes at higher levels in a bottom-up manner. Finally, it goes top-down to align unmapped descendants. JSync does not limit the granularity. We adopt the longest common subsequence based strategy for recursive-expression differencing. For other expressions, we use the element-sensitive strategy.

IJM [9] can generate more accurate move and update operations than GumTree and ChangeDistiller. In the partial matching stage, it divides the AST into smaller parts that are individually matched. However, the division is limited to declaration trees. In the proposed approach, we divide the statement or expression trees into smaller element parts.

MTDIFF [7] generates shorter edit scripts than Gumtree, JSync, and ChangeDistiller by detecting more moves. Higo et al. [15] considered copy-and-paste as an edit operation. However, they did not pay attention to the within-statement changes. CLDiff [16] aims at generating code differences that are easier to understand. Therefore, it groups fine-grained code differences at the statement level or higher and describes the high-level changes in each group.

Change Pattern Analysis. Nguyen et al. [17] studied the repetitiveness of code changes in software evolution. They found that at the small sizes, repetitiveness of fixing changes was statistically higher than that of general changes, and repetitiveness of changes decreased exponentially as size increased. Moreover, they found many repetitive change types including method calls, field access, while statements, etc. Also they found that method calls, infix expressions, and if-statements have more changes than other expressions and statements.

Negara et al. [18] presented an approach to identifying unknown frequent change patterns from a fine-grained sequence of code changes. They found patterns such as *Convert Element to Collection*, *Add Null Check for a Parameter*, *Change and Propagate Field Type*, etc.

Kawrykow and Robillard [10] defined non-essential changes and gave several common non-essential patterns. We used the definition and made some extensions to find non-essential changes in the case study.

However, all of these studies did not analyze how expressions or statements change. We studied the conditional changes in the case study. The method used in the case study can be generalized to study other types of statement changes.

V. Conclusion

We present an approach to identify the changes that occur in a statement. Generally, it uses an element-sensitive strategy. However, when a recursive expression is encountered, it uses a longest common sequence based strategy to save time. We have designed the algorithms to implement the strategies. Currently, the approach has been implemented. It enhances ChangeDistiller by identifying the within-statement changes for each statement update and exposing the metadata for each change.

Through the case study on condition-expression changes in four open projects, we have demonstrated how to combine the proposed approach with ChangeDistiller for change analysis, including identifying non-essential changes and revealing some common patterns. In addition to this, the information contained in each metadata provides clues to find dependencies between statement updates and other changes. For instance, the operation type, the entity type, and element parts can be used to analyze control dependencies. And, the references can be used to reveal data dependencies, while the content and content pattern can be used for similarity dependency analysis. Besides, the proposed approach can be used in the impact analysis of statement updates as well as other related topics in change analysis and understanding.

References

[1] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," IEEE Transactions on software engineering, 2007, vol.33, no.11, pp.725-743.

[2] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang, "Change impact analysis based on a taxonomy of change types," In Proceedings of the 34th Annual Computer Software and Applications Conference (COMPSAC), 2010, pp. 373-382.

[3] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," In Proceedings of the 12th Working Conference on Mining Software Repositories, 2015, pp. 180-190.

[4] E. Giger, M. Pinzger, and H. C. Gall, "Can we predict types of code changes? an empirical analysis," In 9th IEEE Working Conference on Mining Software Repositories (MSR), 2012, pp. 217-226.

[5] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," IEEE Transactions on Software Engineering, 2015, Vol. 31, No.6, pp.429-445.

[6] J. R. Falleri, F. Morandat, X. Blanc, M. Martinez and M. Monperrus, "Fine-grained and accurate source code differencing," In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 313-324.

[7] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 660-671.

[8] H. A. Nguyen, T. T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen, "Clone Management for Evolving Software," IEEE Trans. Softw. Eng., 38(5):1008-1026, Sep. 2012.

[9] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating Accurate and Compact Edit Scripts Using Tree Differencing," In Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, 23-29 Sept. 2018, Madrid, Spain, pp. 264-274.

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

[11] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," Communications of the ACM, 1977, vol.20, no.5, pp.350-353.

[12] G. Canfora, L. Cerulo and M. Di Penta, "Ldiff: An enhanced line differencing tool," In Proceedings of the 31st International Conference on Software Engineering, 2009, pp. 595-598.

[13] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta, " LHDiff: A language-independent hybrid approach for tracking source code lines," In 29th IEEE International Conference on Software Maintenance (ICSM 2013), 2013, pp. 230-239.

[14] B. Fluri and H. C. Gall,"Classifying change types for qualifying change couplings," In Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC 2006, pp. 35-45.

[15] Y. Higo, A. Ohtani, and S. Kusumoto, "Generating simpler AST edit scripts by considering copy-and-paste," In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 532-542.

[16] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "CLDIFF: Generating Concise Linked Code Differences," In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, Montpellier, France, 2018, pp. 679-690

[17] H. A. Nguyen, A. T. Nguyen , T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, 2013, pp. 180-190.

[18] S. Negara, M. Codoban, D. Dig, and R. E.Johnson, "Mining fine-grained code changes to detect unknown change patterns," In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp.803-813.