Maletic Jonathan (Orcid ID: 0000-0001-5289-135X)

srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas

Michael John Decker[1], Michael L. Collard[2], L. Gwenn Volkert[3], Jonathan I. Maletic[3]

[1]Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403
mdecke@bgsu.edu

[2]Department of Computer Science
The University of Akron
Akron, Ohio 44325
collard@uakron.edu

[3]Department of Computer Science
Kent State University
Kent, Ohio 44242
{lvolkert, jmaletic}@kent.edu

***Abstract*—An efficient and scalable rule-based syntactic-differencing approach is presented. The tool, srcDiff, is built upon the srcML infrastructure. srcML adds abstract syntactic information into the code via an XML format. A syntactic difference of srcML documents is then taken. During this process the differences are further refined using a set of rules that model typical editing patterns of source code by developers. Thus, the resulting deltas models edits that are programmer centric versus a purely syntactic tree edit view. Other syntactic differencing approaches focus on obtaining an optimal tree-edit distance with the assumption that this will produce an accurate difference. While this may work well for small or simple changes, the differences quickly become unreadable for more complex changes. By contrast, the approach presented here, purposely deviates from an optimal tree-difference in order to create a delta that is both easier to understand and better models changes between the original and modified. To evaluate the approach, a comparison user study against a state-of-the-art syntactic differencing approach and two line-based differencing tools is conducted as an online within-participant study with about 70 subjects on fourteen sample changes. The results provide support that the rule-based syntactic-differencing produces more accurate and understandable deltas.**

***Keywords*—differencing; syntactic differencing; change comprehension; change visualization**

I. Introduction

Developing and maintaining software is an iterative process of change. That is, we constantly modify (add to and delete from) a code base. In a system modified by a solitary developer, understanding these changes is generally straightforward. However, when scaled to real-world, large-scale systems that involve large teams of individuals located in different geographical locations, understanding what exactly changed in a code base is often problematic and requires substantial effort by developers. It is essential for developers to quickly and easily comprehend changes made to source code. This is critical when understanding the impact of changes, merging changes from multiple developers, conducting reviews of commit requests, and tracking down errors arising from changes.

Differencing tools such as GNU diff have been in wide use by developers for many years. They provide an efficient means to determine the difference between two versions of a file (or set of files). These tools take a lexical line-based view and produce the set of lines that have been added and/or deleted between the two versions (i.e., a *delta*). Lexical differencing (*diff*) is the de facto standard for presenting changes of source code. It is widely integrated into IDEs (e.g., MS Visual Studio, Xcode) and version-control systems (e.g., git, svn). Line-based differencing approaches are very flexible (i.e., can be applied to any text file) and efficient (i.e., scale well to large files), however, they understand nothing about the syntax of the source code being differenced. As such, it is often difficult to understand the output in the context of the language syntax.

In practice, source-code changes are typically more fine-grained than a line (e.g., an identifier is changed) or cross multiple lines (e.g., an entire loop or block is deleted). Moreover, a single logical modification may span multiple non-contiguous lines, such as adding a block to an else-clause that previously had one statement. Another problematic issue is that modifications to a line are represented as a deletion and then an insertion. The related `–delete` and `+add` can occur many lines apart in the delta, making it very difficult to understand the (simple) change. In short, the line-based view does not respect syntactic boundaries. Thus, the deltas often do not sufficiently or clearly reflect the actual meaning of the change.

The alternative to line-based approaches is syntactic differencing [1-3]. Syntactic differencing requires some type comparison on the Abstract Syntax Tree (AST) of the source code. However, these approaches typically ignore changes to whitespace, comments, and preprocessor statements (as they are not in the AST or outside of the formal language syntax). Also, many syntactic differencing tools focus on optimal differences (i.e., edit distance) that may not produce a very understandable delta. This can lead to trying to map multiple unrelated parts from both versions together when in all actuality a complete replacement of code occurred. Lastly, while syntactic information is very useful it is not sufficient for generating easy to read and understandable deltas. For that, additional rules and idioms about how programmers edit and change code are needed. Moreover, tree-differencing approaches are generally slow (compared to diff) and thus do not scale well to large systems. These are some of the underlying reasons for why tree-differencing tools have not been widely accepted by developers.

Another approach is to save each keystroke and edit action of the developer(s) while the change is taking place, such as track-changes in a word processor. This information along with syntactic knowledge can produce a delta that reflects the actual syntactic change that occurred. However, this information is generally not available unless a specialized environment is used [4]. Furthermore, this information is completely unavailable in the case of two non-consecutive versions of a file or in the case of multiple versions from different sources (multiple developers editing the same file).

The goal of the work presented here is to produce a more understandable and readable delta then what is produced by current line-based and syntactic-differencing approaches. Our overarching hypothesis is that on its own, shortest edit script is a poor heuristic for syntactic differencing if the goal is to generate understandable deltas. To investigate this, we developed a novel approach and implementation named srcDiff. It produces differences in the srcDiff format [5], which is an extension to the srcML (www.srcML.org) format [6]. srcDiff is a syntactic-differencing approach that works on the AST of source code as represented by srcML. Thus, it respects syntactic boundaries and supports both coarse and fine-grained changes. Unlike pure syntactic-differencing approaches, srcDiff does not use a standard tree-based differencing approach. Instead, it is a hybrid approach that utilizes more efficient sequence-differencing, as provided by Myers' O(ND) Shortest Edit Script algorithm [7], to map changes onto the tree structure. Thus, it combines the efficiency of line-differencing with localized tree comparison. Several other existing approaches also employ a hybrid approach [5, 8, 9]. These approaches map the sequential deltas onto the tree structure. srcDiff differs by applying a sequential differencing approach [7] directly to the nodes of the tree structure.

In order to produce a more understandable delta, we use a set of empirically derived rules for selecting the most appropriate syntactic category (granularity) for a given change and to determine the scope of a set of changes within the code. The rules model how programmers make changes to code (domain knowledge) thus the resulting deltas better reflect their expectations in the context of changes. No other approach uses a set of rules like srcDiff to arrive at a better difference, and few incorporate any type of domain knowledge. An example of an approach that incorporates some type of domain knowledge (albeit, limited) is JDiff. For example, JDiff uses method signatures to match methods [10, 11]. By comparison srcDiff provides a much richer and fuller set of domain rules that are used to far greater extent and utility.

To evaluate the approach, a comparison study against a state-of-the-art syntactic-differencing approach and two line-based differencing tools is conducted as an online within-participant study with about 70 subjects on fourteen sample changes. The results give evidence that srcDiff produces more accurate and understandable deltas. The contributions of this work are as follows:

- Development of differencing rules based on knowledge of how developers change code and perceive changes
- Integration of these rules into an efficient syntactic-differencing tool
- First user study with a large number of participants (to our knowledge) on viewing and evaluating differencing approaches
- Development of a differencing approach that produces more readable, understandable, and accurate deltas compared to other approaches

The paper is organized as follows. Section II introduces the srcDiff approach, describes the algorithm along with the rules we developed. Section III presents the study and the results. Section IV presents threats to validity and section V discussion related work. Conclusion are given in section VI.

II. Syntactic Differencing via srcDiff

We developed an efficient syntactic differencing approach based on the srcML [6] infrastructure (www.srcML.org). This includes a syntactic difference output format [5] based on srcML and differencing approach (and tool) called srcDiff. Briefly, srcML is a XML source code representation that wraps lexical source code tokens with information from the Abstract Syntax Tree (AST). srcML completely preserves the original source code, such that removing the XML tags reproduces the original source code.

Original

```
void KisFilterOpSettings::setImage( KisImageSP image ) {
   m_optionsWidget->m_filterOption->setImage( image );
}
```

Modified

```
void KisFilterOpSettings::setImage( KisImageSP image ) {
   if (m_options) {
       m_options->m_filterOption->setImage( image );
   }
}
```

srcDiff

```
<unit xmlns="http://www.srcML.org/srcML/src" xmlns:cpp="http://www.srcML.org/srcML/cpp" \
xmlns:diff="http://www.srcML.org/srcDiff" revision="0.9.5" language="C++" filename=" original.cpp| modified.cpp">
<function><type><name>void</name></type> \
<name><name>KisFilterOpSettings</name><operator>::</operator><name>setImage</name></name><parameter_list>( \
<parameter><decl><type><name>KisImageSP</name></type> <name>image</name></decl></parameter> )</parameter_list> <block>{
<diff:insert><diff:ws>   </diff:ws><if>if<diff:ws> </diff:ws><condition>(<expr><name>m_options</name></expr>)</condition> \
<then><diff:ws> </diff:ws><block>{<diff:ws>
   </diff:ws><diff:common>   <expr_stmt><expr><call><name><name>~~<diff:delete type="replace">m_optionsWidget</diff:delete>~~ \
<diff:insert type="replace">m_options</diff:insert></name><operator>-&gt;</operator><name>m_filterOption</name><operator> \
-&gt;</operator><name>setImage</name></name><argument_list>( <argument><expr><name>image</name></expr></argument>) \
</argument_list></call></expr>;</expr_stmt>
</diff:common><diff:ws>   </diff:ws>}</block></then></if><diff:ws>
</diff:ws></diff:insert>}</block></function></unit>
```

Figure 1. Example source-code change. The top-left contains the original source code and the top-right contains the modified source code. The original and modified code contained within the srcDiff format is given. Deletions are in a light-red and with a line-though mark. Insertions are in green. All the original source code is placed in bold.

The srcDiff format[1] [5] extends the srcML format with the addition of four XML tags (`diff:common`, `diff:delete`, `diff:insert`, and `diff:ws`) to contain original and modified source code (i.e., any two versions) marked as the *delta* or the set of changes to the original source code (base version). Figure 1 gives an example of the format, as produced by the srcDiff tool, showing both the original and modified source code (simplified) of the function `setImage` from KOffice revisions 1026809-1026810. The changes include the statement in the function `setImage` being wrapped with an if-statement and the data member `m_optionsWidget` being renamed to `m_options`. The srcDiff subsequently has a `diff:insert` tag around the if-statement and a `diff:common` tag around the contents. The text of the renamed identifier (marked in srcML with a name tag) has a `diff:delete` tag around the old text and a `diff:insert` around the new text. In addition, the tags have an attribute `type` with the value `replace` to signify that the code is replaced (i.e., rename). If a modification is to an attribute of a srcML element, these values are versioned with "|", as in the attribute filename on the unit tag. Deleted/inserted whitespace is marked with `diff:ws` for easy processing/analysis. The srcDiff format completely preserves the original and modified srcML and therefore completely preserves both the original and modified source code (e.g., code, whitespace, comments, preprocessor). That is, the srcDiff format is a multi-version, single-document format that allows both the original and modified srcML/source-code to be extracted. As such, srcDiff supports both source-code change analysis, as well as, an efficient means of producing human-readable deltas.

Typically, syntactic-differencing methods support additional edits (e.g., update node value, move, etc). Because srcDiff marks up text directly (e.g., renamed identifier in Figure 1), it does not need a separate edit for an update. In srcDiff, moves are marked as a delete (moved from) and insert (moved to) tags with an attribute `move` and a unique identifier. Currently, srcDiff supports limited detection for moves within a file as part of the approach. Additional, stages can be added post-process to support additional move detection. Some experimentation has been done on this with success, however, we leave this for future work.

The srcDiff tool is designed to efficiently produce the srcDiff format from any two versions of a source-code document, i.e., two files, directories, or repository versions (Subversion or Git). In contrast to other syntactic differencers, the code does not need to be syntactically complete, and changes to whitespace and comments are marked up. srcDiff also attempts to produce results on syntactically incorrect code. The tool is very scalable as it handles 1,000 commits/versions (all changed files) in under 5 minutes (running on a standard desktop computer).

A. srcDiff Algorithm

At a high-level, srcDiff does a preorder traversal on both the original and modified Abstract Syntax Trees (AST), in srcML, to iteratively refine differences until the desired granularity is reached. A preorder traversal is required so that the resulting difference, in srcDiff format, completely preserves the original and modified code (and its order). Figure 2 presents the differencing algorithm for srcDiff. The algorithm applies a sequence differencing algorithm (i.e., Myers' [7]) to the original and modified children (including subtrees) of a node (see line 2). The result of the sequence differencing is a list of edits where each edit is one or more consecutive children of the same edit type. The edit types are:

- *common* - children identical in both versions,
- *delete* - children removed from the original AST,
- *insert* - children inserted into the modified AST, and

[1]The srcDiff format is a format to support automatic analysis of software changes. However, human-readable views are easily derived from the format and produced automatically by srcDiff.

- *change* - children from the original AST replaced with children from the modified AST[2].

Common and unique children are output directly (lines 5-9), however, changed children need to be further analyzed to determine which actions to take (lines 10-26). There are three possible situations:

1. *Match* - the child in the modified is a new version of the original;
2. *Nested* - if one child and its subtree in one version subsumes children in the other version;
3. *Deleted* or *Inserted* - if a child is unique to one of the versions.

[2]A change can also be described as a list of deleted children and a list of inserted children that occur at the same relative location.

```
struct ASTNode
  root            //Root of AST subtree
  children        //Children of AST subtree
  matchID         //markMoves sets if match found (pairing both matched).
                  //If set, used in remaining processing to remove/ignore

struct Edit
  originalList  //List of consecutive children from original
  modifiedList  //List of consecutive children from modified
  type          //Edit operation type common, delete, insert, or change

struct Match    //Struct holding results for determineMatchings
  isMatched     //Boolean indicating elements are a valid match
  originalList  //List with single matched node or consecutive unmatched nodes
  modifiedList  //List with single matched node or consecutive unmatched nodes

struct Nest     //Struct holding results for determineNestings
  type          //Indicates nest type: original, modified, none
  originalList  //Node to nest into (when type == original) or list of nodes
  modifiedList  //Node to nest into (when type == modified) or list of nodes
function srcDiff(original, modified)
  ListOfEdits ← computeShortestEditScript(original, modified)
  markMoves(ListOfEdits, original, modified)
 for each edit in ListOfEdits do
    if edit.type == common then
      output children as common, merging whitespace
    elseif edit.type == deleted OR edit.type == inserted then
      output deleted or inserted children marked appropriately
    else //a change
      matches ← determineMatchings(edit.originalList, edit.modifiedList)
      for each i in matches do
        if i.isMatched then
          output i.originalList[0].root
          srcDiff(i.originalList[0].children, i.modifiedList[0].children)
        else //Apply nesting rules to a sequence of unmatched children
            nestings ← determineNesting(i.originalList, i.modifiedList)
            for each j in nestings do
              if j.type == original then
                output j.originalList[0].root, marked as deleted
                srcDiff(j.originalList[0].children, j.modifiedList)
              elseif j.type == modified then
                output j.modifiedList[0].root, marked as inserted
                srcDiff(j.originalList, j.modifiedList[0].children)
              else //none
                output deleted and inserted nodes marked appropriately
              end if
           end for
        end if
      end for
    end if
  end for
```

Figure 2. The srcDiff algorithm. A preorder traversal is performed on both the original and modified trees. The input is the original and modified children of a node. A sequence differencing method is used to categorize the children as common, unique to one version, or changed. Common, inserted, and deleted children are output directly. Changed children are further analyzed in *determineMatchings* and *determineNestings* for the following situations: a child in the original is a previous version of a child in the modified, a child and its subtree in one version subsumes children in the other version, and a child is unique to one version. The functions *determineMatchings* and *determineNestings* use a set of rules to identify these situations.

Original	**Modified**
`int foo = 42;`	`extern int foo = ANSWER;`

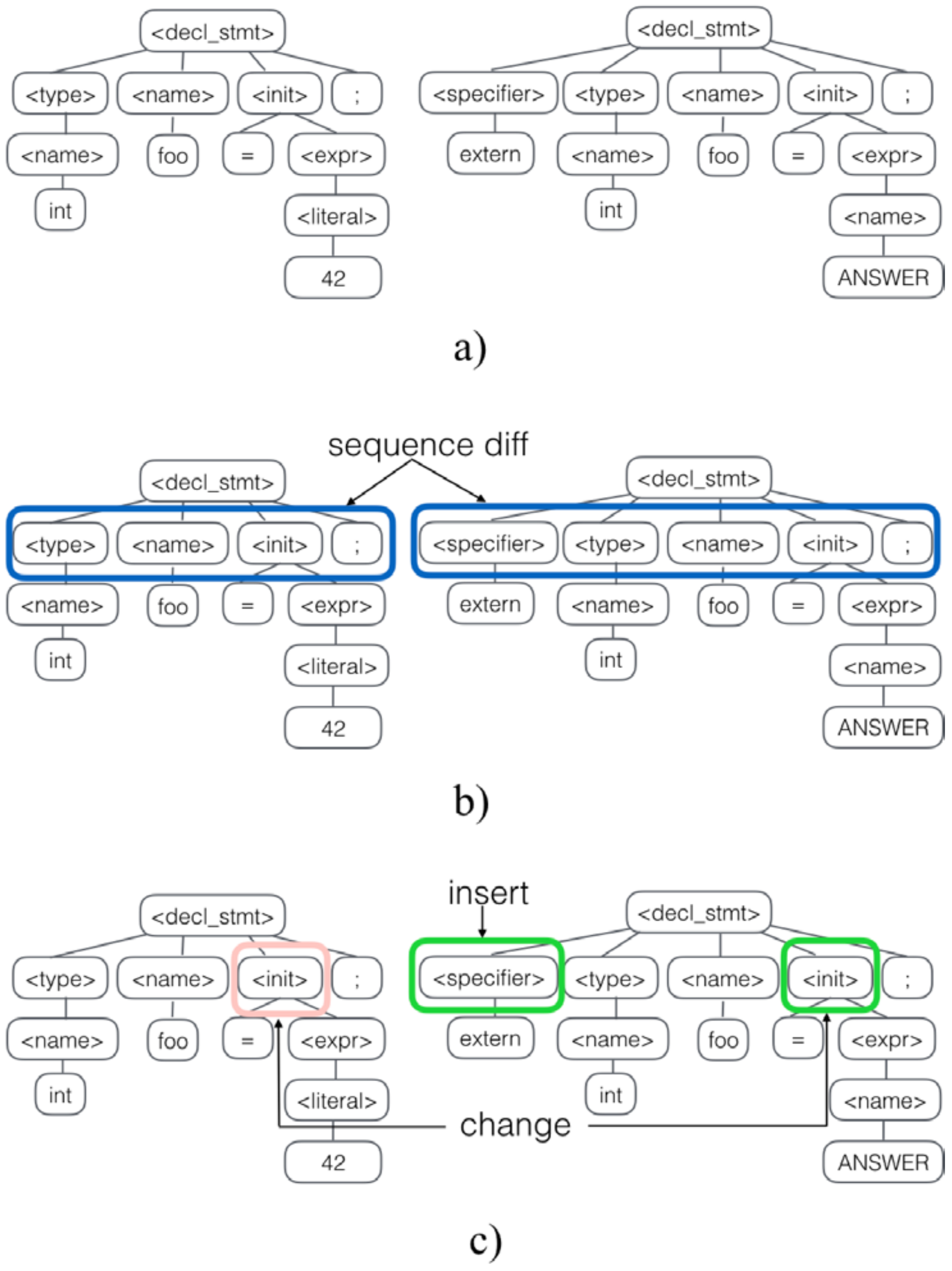


Figure 3. Example of the srcDiff process. a) The original and modified version of a declaration statement with their tree representations below each. b) With decl_stmt as common root, run a sequence diff on their child sequences (blue). Comparisons are between entire child subtrees. c) Output of sequential diff. red is delete edit, green is insert edit, remaining are common. <*specifier*> subtree is an insert and <*init*>'s are a change. Common and insert/delete require no further action. The change needs to be analyzed to see if Match, Nested, or Inserted/Deleted.

Figure 3 illustrates how srcDiff uses sequential differencing. Part a) gives the original and modified versions of a declaration statement, as well as, their tree representations. For the tree representation, only the declaration statement subtree is shown. That is, the root is the *decl_stmt* and the declaration statement's children are: type (*type*) and its subtree, identifier name (*name*) and its subtree, and initialization (*init*) and its subtree. The original declaration statement (source code) is modified in two ways: the addition of the *extern* keyword and the replace of the numeric literal *42* with the constant *ANSWER*. As such, the tree representation is modified in two ways: a new child of *decl_stmt* (*specifier* and its subtree) is added and the *literal* (descendent of *init*) is replaced with the constant *ANSWER* (i.e., *name*). srcDiff takes the children of a common node (*decl_stmt*), treats each as a sequence (one sequence for the children from the original code and a second for the children in the modified code), as shown in Figure 3 part b), and runs a sequential-diff algorithm (i.e., Myers' [Myers 1986]). Child comparisons are of the entire child subtrees. The result of this for Figure 3 is part c). The specifier and its subtree is reported as an insert edit. In addition, the *init* (and its subtree) is reported as a delete edit in the original tree and an insert edit in the modified tree (as the subtrees are not identical). As they occur in the same relative position in the sequence, the *init* insert and delete edits get aggregated into a change edit. The remaining children are then denoted as common. The insert of *specifier* and common children require no further action by the srcDiff algorithm. However, the changed children (i.e., *init*), are analyzed further to determine if they were Inserted/Deleted (i.e., original *init* was deleted and the modified *init* inserted), *Nested* (one *init* belongs inside the other), or a *Match* (i.e., deleted *init* is a previous version of the inserted). In this case, the deleted *init* will be identified as a previous version of the inserted *init*, and the algorithm will repeat itself on the children of *init*.

The process for determining which actions to apply are as follows. First, the changed children are checked to determine if any can be matched (i.e., using the function *determineMatchings*). If so, then finer-grained differencing is performed via a recursive call to the srcDiff algorithm on their children. The *determineMatchings* process uses a dynamic-programming approach that minimizes the number of unmatched children while utilizing a similarity metric as a tie breaker.

What can be matched is determined via a set of *differencing rules* (discussed in detail in the following section). The result of *determineMatchings* is a sequence of items that is either consecutive original and modified children with no corresponding matches (i.e., unmatched) or a pair of matched children.

Next, unmatched children are checked (via *determineNestings* at line 16) to determine if any of the children in the original or modified AST can be nested within the other. The result of *determineNestings* is a list that partitions all the children into groups of consecutive un-nested children or a group of children pertaining to a nesting. Similar to *determineMatchings*, a set of *differencing rules* is used to define the nestings. For each nesting, the deleted/inserted root is output and the srcDiff algorithm recursively called on the children and the span of elements that will be nested.

In the case that a child can be both matched and nested, *determineMatchings* uses the *differencing rules* to determine if nesting produces a better result. If so, the match is rejected and the children will be left as unmatched for *determineNestings*. *determineMatchings*, *determineNestings*, and the differencing rules will be elaborated on in the following sections.

Lastly, is the situation of a delete or insert; this can be output directly (see line 22). As a note, srcDiff provides limited support for moves (see line 3). Children of the original that are identified as deleted or changed by the sequence differencer are scanned to see if there is an exact match among the modified children identified as inserted or changed. If found, the children are marked as moves and ignored by *determineMatchings* and *determineNestings*. Further support for moves can be added by analyzing the resulting srcDiff, however we leave this for future work.

Typically, syntactic differencing approaches [3, 12] use a generic tree-differencing approach that consist of two steps: 1) a matching step where nodes from the original tree are paired/matched with nodes from the modified tree, and 2) a edit-script generation step that takes the matches/pairings and generates the shortest edit script. Basically, the more pairings identified during the matching step the shorter the edit script that is generated. The srcDiff algorithm is unique in that it blends the two steps together. That is, it generates an edit script as it walks through the tree and computes matches. The rules themselves, however, are specific to the matching part of the srcDiff algorithm and the rules are only checks on tree nodes (and information about descendent/ancestors) to determine potential matches. Most notably they identify when to reject pairings that will result in poor edit script generation. In other words, the srcDiff algorithm is unique and not applicable to other approaches, however the rules can be utilized by a typical matching procedure to test if two nodes can be paired/matched together, and most notably when they cannot be paired/matched together. As such, the rules themselves constitute a large contribution of this work.

Additionally, the rules are based on the grammar of the language along with developer knowledge of what will create a meaningful/understandable delta. No other syntactic differencing approach incorporates this level of syntactic knowledge or developer knowledge as the rules encompass. In fact, syntactic-differencing approaches are largely the application of tree differencing to a tree that happens to be an AST. They typically do not use language semantics/grammar or any domain knowledge in performing a difference. A counter example to the generic tree differencing is the work of Apiwattanapong et al. [11]. They use some limited syntactic knowledge, namely identifiers and method signatures in their algorithm to match classes and methods. As illustrated in the next section, the rules can be used to identify meaningful matches and avoid poor pairings to create more understandable deltas. We now discuss the differencing rules.

B. Differencing Rules

Pure syntactic tree differencing often produces results that are difficult to understand. Take the following example code from *hadoop* (revision: eee0d4563c62647cfaaed6605ee713aaf69add78):

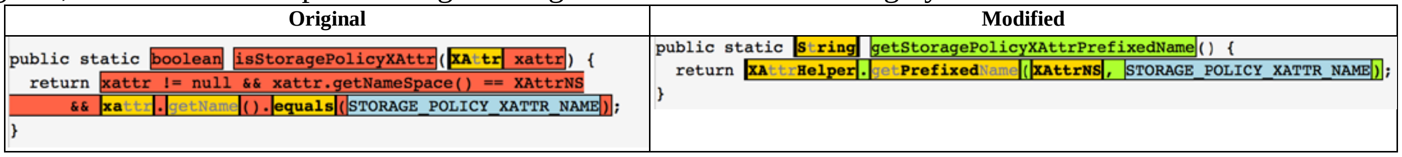
Original

```
public static boolean isStoragePolicyXAttr(XAttr xattr) {
  return xattr != null && xattr.getNameSpace() == XAttrNS
      && xattr.getName().equals(STORAGE_POLICY_XATTR_NAME);
}
```

Modified

```
public static String getStoragePolicyXAttrPrefixedName() {
  return XAttrHelper.getPrefixedName(XAttrNS,
STORAGE_POLICY_XATTR_NAME);
}
```

Textually, there is only a small amount of similarity between the two versions. Clearly, the method in the original version is completely replaced with a new method in the modified version that is, a delete and insert. Using a traditional syntactic differencing approach, it is more optimal (i.e., shorter edit script) to match any two given constructs (e.g., methods, if-statements, declaration-statements, etc.) because of common AST nodes that happen to be the same/similar. However, this approach fails miserably in this type of example and results in something very difficult to understand. The following is a side-by-side difference produced by GumTree [13] a current state of the art syntactic differencing tool. Deletes are marked in red, inserts in green, moves in blue. Updates are given in gold with common text in grey and deleted/inserted text in bold.



Original

```
public static boolean isStoragePolicyXAttr(XAttr xattr) {
  return xattr != null && xattr.getNameSpace() == XAttrNS
      && xattr.getName().equals(STORAGE_POLICY_XATTR_NAME);
}
```

Modified

```
public static String getStoragePolicyXAttrPrefixedName() {
  return XAttrHelper.getPrefixedName(XAttrNS, STORAGE_POLICY_XATTR_NAME);
}
```

A large number of meaningless matches between syntactic elements (e.g., XAttr and String) are made to achieve more optimal results (i.e., shorter edit script). While entire method replacement (as in this example) may not be very common, replacement of statements (or parts of statements) is a very common type of change. Anytime there is a coincidental similarity, pure syntactic differencing approaches will generate an inappropriate result.

Hence we take a different approach and use a set of customized heuristic rules instead of a single heuristic such as edit distance. We defined a set of rules that determine when it is acceptable to consider whether the change is a modification or a complete replacement of an existing statement. The rules are derived via analysis of the language syntax and our combined development experience (domain knowledge), as well, as by manual and statistical analysis of over 1,000 revisions/version of changes to several open-source C++ projects[3]. Many of the rules are based on the syntax and development experience. For example, if a declaration with the same name appears in the original and modified code, then they are considered as matching declarations. Some of the rules employ constant values which are derived through statistical analysis and fine-tuning (via trial and error). These values can be further adjusted based on preference or additional empirical studies. In addition to the empirical analysis, we tested srcDiff on a suite of over 500 examples (simple to very complex) that includes changes from various open-source systems. These rules are general enough that they could be incorporated as part of the matching processes employed by other syntactical-differencing techniques.

The differencing rules employed by srcDiff are organized into three categories:

- *Match* rules - identifies when two syntactic elements of the same type (e.g., two if statements) can be matched. For example, a developer adds additional statements to the body of an if-statement, the *match* rules report that the if-statement in the original version can be matched with the corresponding if-statement in the modified version. These are presented in Section II.D.
- *Convertibility* rules - identifies when one syntactic element can be converted into another syntactic element of differing type. For example, a developer changes a for-statement into a while-statement. Here, *Convertibility* rules report that the for-statement in the original version can be converted into the while-statement in the modified version. These are presented in Section II.E.
- *Nesting* rules - identifies when a syntactic element can be placed within another. For example, a developer adds an if-statement as a guard around an existing statement. In this case *Nesting* rules report that the existing statement in the original version can be nested within the if-statement in the modified version. These are presented in Section II.F.

As part of all three categories of rules, a set of metrics and rules based off of similarity (i.e., *Similarity* rules) is used. The *Similarity* rules and metrics are presented in Section II.C.

As *determineMatchings* and *determineNestings* from Figure 2 (srcDiff Algorithm) encompass how the rules are used, we now elaborate further on *determineMatchings* (Figure 4) and *determineNestings* (Figure 5). The dynamic-programming algorithm *determineMatchings* (Figure 4) maximizes the number of matched syntactic elements while utilizing max similarity as a tie breaker. We implemented *determineMatchings* using the standard LCS algorithm [14]. The only difference being that maximum number of matched elements/similarity is used instead of string equality on all three ways for moving through the matrix (from Cormen et al. [14], c[i - 1, j], c[i, j - 1], and c[i - 1, j - 1]). What can be matched is determined via the *Match* and *Convertibility* rules (Section II.D and Section II.E) and the similarity metric is calculated via the *textual-similarity* metric (Section II.C). In addition, when the *Match* and *Convertibility* rules report a potential match, a check provided as part of the *Nesting* rules (Section II.F) is conducted to see if the syntactic element is better nested. If so, then the syntactic elements are treated as if the *Match* and *Convertibility* rules failed.

[3]This includes bug fix commits to KOffice (commit message's contain BUG: <bugnumber>, several release versions of GCC, etc. Additionally, srcDiff was actively used/refined during the development of srcML and srcDiff (this includes time as a full-time developer held previously by one of the authors), where the changes to the projects were used to empirically fine-tune the approach.

As a note, moved children (identified by *markMoves*) are removed from lists before the dynamic programming algorithm is applied.

The algorithm *determineNestings* (Figure 5) tests iteratively each unmatched child to see what original elements can be nested into a modified element and what modified elements can be nested into an original element using the *Nesting* rules. Lists are created for each (an empty list means no valid nesting is found). When only one list is non-empty, it is chosen. When both produce non-empty lists, if the two lists are disjoint (no elements overlap each other), the one that has children that comes first sequentially is chosen. Otherwise, the list that spans the most elements is chosen. The *determineNestings* process is repeated for the remaining children until no new nesting are found (both lists are empty). To illustrate this, consider having the following list of original and modified children:

Original Children	**Modified Children**
A	V (A & B pass nesting rules)
B	W (C passes nesting rules)
C (W & Y pass nesting rules)	X
D	*Y*
E (Z passes nesting rules)	*Z*

Both lists contain children (each consecutive) which are not matched during *determineMatchings.* With the first iteration of the nesting rules, it will identify that *C* can have the list of children *W* and Y nested within and *V* can have the children *A* and *B* nested within. As *A* and *B* occur before *C* (and *V* before *W*), *A* and *B* will be nested into V. In the next iteration, *C* is identified again to be able to have the list of children *W* and *Y* nested within and *W* can have *C* nested within. Both contain overlapping elements (not disjoint), so the list spanning the most children (*C* covering *W*, *X*, and Y) is chosen. Although, *X* is not nestable, since a latter element is included, it is also included (to maintain document order in output). The subsequent recursive call(s) to the srcDiff algorithm will mark *X* correctly as deleted. In the final iteration, only *E* is identified as being able to have *Z* nested within it and is chosen. Note, all identified move children (via *markMoves*) are skipped and will not appear in either list.

```
struct TableData
    isMatch // if the associated nodes are a match
    totalMatches // total number of matches along path taken
    totalSimilarity // total similarity among number of matches along path taken
    pathTaken // The path taken to reach this entry

function determineMatchings(originalChanged, modifiedChanged)
    table ← new TableData[originalChanged.length][modifiedChanged.length]
    for i in 1 to originalChanged.length do
        for j in 1 to modifiedChanged.length do
            similarity = textual similarity of (originalChanged[i], modifiedChanged[j])
            isMatch = passes Match/Convertibility rules
                      AND !isBetterNested(originalChanged[i], modifiedChanged[j],
                                          originalChanged, modifiedChanged)
            Update table using isMatch and similarity
                Check table[i][j - 1], table[i - 1][j], table[i - 1][j - 1]
                Pick one with most matches or if matches equal, highest similarity
                Set table[i][j] with new total matches/similarity and record path taken

        end for
    end for

    Process table[originalChanged.length][modifiedChanged.length] to
    construct list of struct Match (Figure 2)

```

Figure 4. *determineMatchings* algorithm. Procedure is a standard dynamic programming algorithm that uses *Match* and *Convertibility* rules, as well, as a check if nesting produces a better result to see what elements can be matched. Algorithm maximizes the number of matched children as primary operation and then maximizes *text-similarity* (e.g., as a tie-breaker).

Although this process is greedy, we found it to work well, except in the case when there is a single child deleted or inserted and nothing can be nested into it, but it can be nested (possibly into multiple candidates). In this case, srcDiff does an exhaustive search to determine the best nesting match with textual similarity (Section II.C) used to select the best candidate.

Each of the rule categories is now discussed separately and illustrated in the same manner as Figure 1, with deleted text in red/light grey with a line-through (~~deleted~~), inserted text in a green/dark grey (inserted), and the original text left unchanged in normal font. The following is the result of srcDiff on the previous method example (as side-by-side diff[4]).

Original	Modified

`public static boolean isStoragePolicyXAttr(XAttr xattr) {`
`  return xattr != null && xattr.getNameSpace() == XAttrNS`
`      && xattr.getName().equals(STORAGE_POLICY_XATTR_NAME);`
`}`
	`public static String getStoragePolicyXAttrPrefixedName() {`
	`  return XAttrHelper.getPrefixedName(XAttrNS,`
	`STORAGE_POLICY_XATTR_NAME);`
	`}`

srcDiff	

~~**public static boolean isStoragePolicyXAttr(XAttr xattr) {**~~
~~**return xattr != null && xattr.getNameSpace() == XAttrNS**~~
~~**&& xattr.getName().equals(STORAGE_POLICY_XATTR_NAME);**~~
~~**}**~~
	public static String getStoragePolicyXAttrPrefixedName() {
	return XAttrHelper.getPrefixedName(XAttrNS,
	STORAGE_POLICY_XATTR_NAME);
	}

C. Similarity Metrics and Rules

The *Match*, *Convertibility*, and *Nesting* rules use a set of *Similarity* metrics and *Similarity* rules (which use the metrics). We define them here.

There are two categories of similarity metrics: *syntactic* and *textual*. For each, category, a metric is defined for both similarity and dissimilarity (e.g., *textual-similarity* and *textual-dissimilarity*). Both similarity and dissimilarity are computed using Myers' Shortest Edit Script Algorithm [7]. In turn, we can use Levenshtein distance [14]. However, with Levenshtein distance, substitutions are treated as a single operation, instead of a delete and an insert. As substitutions are not required by the metrics, we use Myers' algorithm. Similarity and dissimilarity metrics and are defined as:

$$similarity = min(\#original - \#deletions, \#modified - \#inserts)$$
$$dissimilarity = \#deletions + \#inserts$$

where *#original* and *#modified* is the number of textual tokens in the case of the textual metric, and the number of child elements in the case of the syntactic metric. The *#deletions and #inserts* are the number of inserted and deleted lexical tokens/child elements.

[4] srcDiff also produces a unified-view. For a direct comparison to GumTree, a side-by-side view is shown here. Latter examples show a unified view of the changes.

Additionally, the maximum and minimum number of token/children are used and are calculated as:

maxSize = max(#original, #modified)
minSize = min(#original, #modified)

Two *Similarity* rules are formed using the *syntactic* metrics and *textual* metrics respectively: *Syntactic-match* rule and *Textual-match* rule.

```
1.  function determineNestings(originalChanged, modifiedChanged)
2.      originalNestable ← []
3.      for each i in originalChanged do
4.          for each j in modifiedChanged do
5.              isNestable ← Nesting rules for j into i
6.              if isNestable then
7.                  Append (i, j) to originalNestable
8.                  for k in j + 1 to modifiedChanged.length do
9.                      isNestable ← Nesting rules for k into i
10.                     if isNestable then
11.                         Append (i,k) to originalNestable
12.                     end if
13.                 end for
14.                 break to Line 18
15.             end if
16.         end for
17.     end for
18.     Check for Nestings for original changed into modified changed
19.     Choose list coming first (with children as ordered list)
20.     If lists share children, choose list covering most elements
21.     Repeat Lines 2-19 on remaining children until no new nestings found
```

Figure 5. *determineNestings* algorithm. Lines 2-17 apply the *Nesting* rules to determine what modified (unmatched) children can be placed inside original (unmatched) children. The process for checking if original children can be nested in modified children (Line 18) is similar to Lines 2-17 and so is left out for simplicity. If the originalNestable and modifiedNestable are disjoint, the one with children that come first sequentially is chosen (if other is still disjoint in next iteration it will be chosen). Otherwise, the list that spans the most elements is chosen. This process is repeated (Line 20) until no more possible nesting are found.

The *Syntactic-match* rule is computed on the child elements of two candidate elements after stripping out immediate child literals, modifiers, operators, empty argument lists, and text tokens (e.g., the keyword "if" when applied to an if-statement). If similarity is greater or equal than half the minimum number of children between both sets and the amount of dissimilarity is not greater than this minimum, then the two candidate elements are a match. Additionally, if both elements have a block as the final child or an if-statement with only a then-clause, and the previous check fails, the same syntactic check is also applied on children of the block.

The *Textual-match* rule is comprised of nine sub-rules and is shown in Figure 6. With exception of sub-rule 2 (in Figure 6), it is computed on the textual leaf tokens after stripping out non-operator punctuating characters such as "(", ")", "{", "}", "[", "]", ":", ";", and ",", and operators/modifiers part of a name (except "::"). Sub-rule 2 uses measures without stripping out any tokens.

```
1. if ((similarity == minSize) && (dissimilarity < 2*minSize)) return true;
2. if (similarityNoStrip == minSizeNoStrip)                   return true;
```

```
3. if (dissimilarity > maxSize)                               return false;
4. if ((minSize < 30) && (dissimilarity > 1.25*minSize))      return false;
5. if ((minSize ≥ 30) && (dissimilarity > 1.05*minSize))      return false;
6. if (minSize ≤ 2)                                           return similarity ≥ ½*minSize;
7. else if (minSize ≤ 3)                                      return similarity ≥ 2/3*minSize;
8. else if (minSize ≤ 30)                                     return similarity ≥ 7/10*minSize;
9. else                                                       return similarity ≥ ½*minSize;
```

Figure 6. *Textual-match* rule. The sub-rules use similarity, dissimilarity and size measures to test if two elements are similar enough textually. The measures, with exception to sub-rule 2, are calculated after stripping out punctuation characters and other tokens that negatively affect performance. Sub-rule 2 uses all tokens to test for an exact match. It is not applied to *Nesting* rules.

The first two if-statements (sub-rules 1 and 2) handle complete matches (i.e. one child is a complete subset). The first applies to all the rules, while the second does not apply to *Nesting* rules. The 3rd - 5th sub-rules encode that a potential match should not have an excessive amount of dissimilarity. The remaining rules constitute the main *Textual-match* tests. When the list is two or three tokens, a majority of them need to be the same. Finally, during the empirical analysis of changes we found that a relatively small structure (e.g., < 30 tokens) needs a larger amount of similarity to be considered a match, while a larger structure can be a simple majority. That is, an if-statement of two or three lines (smaller number of tokens) requires a closer match than an if-statement of say 50 lines (large number of tokens).

TABLE I. MATCH RULES AND THE ASSOCIATED SYNTACTIC CATEGORIES (SRCML ELEMENTS). MATCH CONDITIONS ARE ALSO PROVIDED. THESE CONDITIONS ARE IN ADDITION TO SIMILARITY RULES WHICH ARE APPLIED AFTERWARDS IF THESE FAIL.

Rule	Match Conditions	Syntactic Category
Always Match	Elements are always matched	name (single identifier), type, then, condition, control, init, default, comment, block, private, protected, public, parameter_list, argument_list, member_init_list, argument, range, literal, operator, modifier
Any Similarity Match	*textual-similarity > 0* ***AND*** *textual-dissimilarity ≤ textual-maxSize*	expr, expr_stmt
Name Match	*original-name == modified-name*	call, decl, decl_stmt, parameter, function, function_decl, constructor, constructor_decl, destructor, destructor_decl, class, struct, union, enum
Condition Match	*original-condition == modified-condition*	while, switch, for
If-Statement Match	*original-then == modified-then* ***OR*** *(original-condition == modified-condition* ***AND*** *(original-has-block == modified-has-block* ***OR*** *original-has-else == modified-has-else* ***OR*** *(original-has-block* ***AND*** *!modified-has-else)* ***OR*** *(modified-has-block* ***AND*** *!original-has-else)))*	if

D. Match Rules

Match rules are applied to two of the same syntactic elements, such as two declaration statements or two if-statements to determine if they are a match. More explicitly, the *Match* rules are used to determine if a) two elements are similar enough to break down into constitute parts and perform additional fine-grained differencing, or b) if it is better to produce a delta showing a replacement of the one element with the other. The goal here is to generate a delta that is easily understood by the developer. Currently, the *Match* rules are divided into five different types of actions as shown in TABLE I.

The *Always Match* rule includes syntactic categories that will always be broken down, no matter how different. There are two reasons for this. The first is to break down small nodes, which typically contain only text (e.g., names, operators, etc.). In the following example, the only item changed between the original and modified code is the variable name. Instead of marking the entire identifier as deleted and a separate identifier inserted (i.e. the node and its text is deleted and a new node with separate text inserted), the subparts of the name are marked as deleted and inserted (i.e., it is the same node, but the text is changed) to indicate a rename.

Original	**Modified**
`KisPenWidget *m_optionsWidget;`	`KisPenWidget *m_options;`
srcDiff	
`KisPenWidget *`~~`m_optionsWidget`~~`m_options;`	

The second reason is when it does not make sense for a syntactic element to be repeated (e.g., an argument list). In the following example, there is no similarity between the original argument list and the modified argument list, besides the parenthesis. Thus, without the *Always Match* rule, the argument list will not be broken down, leaving the original argument list as deleted and the modified argument list as inserted. However, with this rule, the argument list is broken down and the inserted argument can be correctly marked.

Original	**Modified**
`update();`	`update(true);`
srcDiff	
`update(true);`	

The *Any Similarity Match* rule applies to expressions and expression statements, and is similar to *Always Match*, except if the two have no *textual-similarity* at all or there is too much *textual-dissimilarity* (see TABLE I). In both situations a match does not occur otherwise any amount of *textual-similarity* constitutes a match. The elements in this rule are ubiquitous and can have very complex contents, which makes it valuable in many contexts not to perform finer-grained differencing thus obtaining more understandable deltas. In the following, there is no matching text between the two expressions, so the entire statement can be marked deleted and inserted.

Original	**Modified**
`a + b;`	`c - d;`
srcDiff	
~~`a + b;`~~`c – d;`	

In this next example, the expressions have some *textual-similarity* without excessive *textual-dissimilarity*. This rule, followed by the use of *Always Match*, has the operator marked as inserted and the identifier renamed.

Original	**Modified**
`a + b;`	`a + -d;`
srcDiff	
`a + -`~~`b`~~`d;`	

The rule *Name Match* applies to elements which contains a name, e.g., functions and variable declarations, where a matching name constitutes a match no matter what other differences exist. In its simplest form, a class in the original and modified code with the same name is considered the same, even if the contents are massively different. The following example contains a number of textual changes between the original and modified code, such that the inserted declaration can be considered a replacement if not for the matching variable name.

Original	Modified
`QTextDocument * m_document;`	`QTextDocumentPtr m_document;`
srcDiff	
~~QTextDocument~~QTextDocumentPtr~~ *~~ m_document;	

A small extension is needed for method calls of an object, e.g., `str.size()` and the call to `setImage` in Figure 1. If the variable is a rename, then we still want a match to occur. srcDiff currently considers it a match if any of the names match.

The *Condition Match* rule applies to loop statements and the switch statement and produces a match if the conditional is the same. For instance, the following while-statement has a large number of changes to its block. However, as it still has the same condition, it is considered to be the same statement.

Original

```
while(args[0]) {
    ++args;
}
```

Modified

```
while(args[0]) {
    if(strcmp(args[0], OUT_FLAG) == 0) {
        output_file = args[1];
        ++args;
    }
    ++args;
}
```

srcDiff

```
while(args[0]) {
    if(strcmp(args[0], OUT_FLAG) == 0) {
        output_file = args[1];
        ++args;
    }
    ++args;
}
```

In the case of checking if for-statements match, the initialization and increment are considered in addition to the condition (i.e., the whole control group). In the following, the contents of the block are completely different. However, since the control group is the same, it is treated as the same for-statement. In contrast, if any part of the control group does not match, then it will not pass this rule.

Original

```
for(i=n.begin();  i != n.end(); ++i) {
    std::cout << i->treeID() << '\n';
}
```

Modified

```
for(i=n.begin();  i != n.end(); ++i) {
    int id = i->treeID();
    if(id == treeID)
      return *i;
}
```

srcDiff

```
for(i=n.begin();  i != n.end(); ++i) {
~~    std::cout << i->treeID() << ‘\n’;~~    int id = i->treeID();
    if(id == treeID)
      return *i;
}
```

A special, and more complex, rule (summarized in TABLE I) is applied to if statements, the *If-Statement Match*. The simplest situation is when both if-statements have identical then-clauses. In this case, they are considered the same if-statement. The following is an example:

Original	Modified
`if (p >= count()) break;`	`if (already_exist(chain, p)) break;`
srcDiff	
if (~~p >= count()~~already_exist(chain, p)) break;	

The conditions of the two if-statements have very little similarity and the amount of change between the statements is greater than the similarity. However, since the then-clauses are identical, it is considered the same if-statement.

The other rules associated with an if-statement are related to when both conditions are identical. First, the rule produces a match if either of the following is true about the original and modified if-statements: 1) both contain a block-statement (“{}”) in the then-clause; or 2) both do not have a block-statement in the then-clause. In the following, both if-statements have the same condition, but very different statements in the block. Despite this, they should be considered the same if-statement.

Original	Modified

if(name == null) {
  return false;
}
	if(name == null) {
	  throw new IllegalException("...");
	}

srcDiff	

if(name == null) {
~~ return false;~~ throw new IllegalException("...");
}	

Second, the rule produces a match when any of the following is true about the original and modified if-statements: 1) both contain an else-clause; 2) both do not contain an else-clause; or 3) only one has a block-statement, while the other if-statement does not have an else-clause. This complexity is needed in cases such as when an if-statement is taken from around code and an else-clause added. The following two examples illustrate the condition/block/else situation. In the first example, the modified if-statement has a block and the other does not. The if-statement appears to be a previous version of the other, so matching is expected.

Original	Modified

if(!paraStyles.contains(root.ID()))
    tree_root << root.ID();
	if(!paraStyles.contains(root.ID())) {
	    int id = root.ID();
	    if(id != tree_rootID)
	        tree_root.insert(id);
	}

srcDiff	

```
if(!paraStyles.contains(root.ID()))
~~    tree_root << root.ID();~~ {
    int id = root.ID();
    if(id != tree_rootID)
        tree_root.insert(id);
}
```

In the next example, the original if-statement has a block and the other does not. Here, they do not match so the second modified if-statement can be placed in the original if-statement. The main difference is the addition of an else-clause. With the additional else-clause logic, the correct delta is achieved.

Original	**Modified**
`if (shape.has("anchor")) {`	`string type;`
`    string type = shape.get("anchor");`	`if (shape.has("anchor"))`
`    if(type != "page") {`	`    type = shape.get("anchor");`
`        // several lines common code`	`else`
`    }`	`    type = "as-char";`
`}`	`if(type != "page") {`
	`    // several lines common code`
	`}`

srcDiff

```
~~if (shape.has("anchor")) {~~
~~    ~~string type~~ = shape.get("anchor")~~;
if (shape.has("anchor"))
    type = shape.get("anchor");
else
    type = "as-char";
~~    ~~if(type != "page") {
~~    ~~    // several lines common code
~~    ~~}
~~}~~
```

This rule is derived from examining the change history of existing projects and fails when the extracted if-statement has no else-clause. However, if an else-clause is not needed, then it does not need extraction. For code that does not pass one of the rules above, the two *Similarity* rules (Section II.C: *Syntactic-match* and *Textual-match*) are applied in succession (first, *Syntactic-match*, then, *Textual-match)* as a set of fallback rules. If anyone passes, then the two elements will be considered a match. Depending on how the AST is formed, the *Syntactic-match* rule can subsume the *Condition Match* rule (i.e., a for, while, or switch consist only of control group/condition and block statements). In these instances, the *Condition Match* rule may be considered a quick pre-check to avoid a costlier rule. Additionally, if both elements have a block as the final child or an if-statement with only a then-clause, and the previous check fails, the same *Syntactic-match* rule is also applied on children of the block. If it passes a match will be produced.

TABLE II. CONVERTIBILITY RULES ORGANIZED BY TYPE. EACH TYPE HAS THE ASSOCIATED SYNTACTIC CATEGORIES. MATCH CONDITIONS ARE ALSO PROVIDED. THESE CONDITIONS ARE IN ADDITION TO SIMILARITY RULES WHICH ARE APPLIED AFTERWARDS IF THESE FAIL.

Rule	Match Conditions	Syntactic Category
Class	*original-name == modified-name*	class, struct, union, enum
Access Regions	-	private, protected, public
Logical	*original-condition == modified-condition*	if, while, for
Else	-	else, elseif
Statement	*original-has-expression* ***AND*** *modified-has-expression* ***AND*** *original-expression and modified-expression pass similarity rules* ***AND*** *expression-textual-similarity > ½ * expression-MaxSize* ***AND*** *expression-dissimilarity < ½ * expression-MaxSize* ***AND*** *((!original-is-decl-stmt* ***AND*** *!modified-is-decl-stmt)* ***OR*** *max(original-expression-size, modifed-expression-size) <* *4 * min(original-expression-size, modifed-expression-size)* *)*	expr_stmt, decl_stmt, return
Cast	-	const_cast, dynamic_cast, reinterpret_cast, static_cast

E. Convertibility Rules

Convertibility rules occurs between two syntactic elements of different types and determines if one of the elements can be converted to the other, such as converting a for-statement to a while-statement. When identified, srcDiff marks the elements with delete and insert tags each with attribute `type` and value `convert` and performs fine-grained differencing on their children via recursively applying the srcDiff algorithm (Figure 2) on the children. The explicit identification and recoding of conversions is unique to srcDiff. There are currently six types as listed in TABLE II.

All the convertible rules use the Similarity rules as a check to see if one can be converted to the other. *Class*, *Logical*, and *Statement* have additional conditions that will result in passing. For the *Class* rule, a conversion can occur if they both have the same name. The *Logical* rule passes if both have the same condition (similar to the *Condition Match*, except only the condition of a for is used*)*. The following is an example for-statement converted to a while-statement.

Original	Modified

```
int itemCount = d.items.count();
for(int i = itemCount-1; i >= 0; --i) {
    Item &sbItem = d.items[i];
    if (sbItems.widget() == widget) {
        // several common lines
    }
}
```

```
int itemCount = d.items.count() - 1;
while(i >= 0) {
    Item &sbItem = d.items[i];
    if (sbItems.widget() == widget) {
        // several common lines
    }
    --i;
}
```

srcDiff

```
int itemCount = d.items.count() - 1;
~~for~~while~~(int i = itemCount-1; ~~(i >= 0~~;~~) ~~ --i) ~~{
    Item &sbItem = d.items[i];
    if (sbItems.widget() == widget) {
        // several common lines
    }
    --i;
}
```

For *Statement*, the top-level expression in each is used to determine if they can be converted, i.e., if the expressions can be matched (initialization in declaration-statement, returned expression in return-statement, and the complete expression in expression-statement). To be converted, as specified in TABLE II, they both must have expressions, the expressions must pass either of the *Similarity* rules, *textual-similarity* must be greater than half the size of the largest expression, and *textual-dissimilarity* must be less than half the size of the largest expression. In the case of matching a declaration statement to something, then the size of the expression trees must also not be greater than four times each other (a simple check to avoid poor matches). The following is a simple, but real change example.

Original	Modified
`bool ok = openUrl(url);`	`openUrl(url);`
srcDiff	
~~bool ok = ~~openUrl(url);	

F. Nesting Rules

Nesting rules occur between syntactic elements with the goal to determine if one or more of the elements can be safely nested (placed) within another. That is, mark the one element as deleted/inserted and recursively apply the srcDiff algorithm on the nest-able elements and that element's children to determine if they can be matched or further nesting is required. For example, an if-statement is deleted, however, the statements in the if-statement's body are left unchanged. The nesting rules determine that the statements in the modified document can be nested within the deleted if-statement where they can later be marked as unchanged.

The *Nesting* rules (Figures 7-11) are applied to unmatched elements, i.e., those that failed *Match*/*Convertibility* rules, and are applied as specified in Figure 5. However, *Nesting* rules are also checked during the *Match*/*Convertibility* process (i.e., *determineMatchings*) to see if an element can and will create a better difference by nesting it into the potential match/conversion or a subsequent element (Figure 11). If so, the match/conversion is rejected to allow for the nesting.

We will first discuss the core test for nestability (Figure 7) and associated algorithm in Figure 8. Figure 9 (which also uses Figure 10) detail how this is applied in Lines 5 and 9 of Figure 5, and Figure 11. gives the algorithm used during *Match*/*Convertibility* process to determine if it is better to nest elements.

```
function isNestableCore(nestElement, otherElement)
    // check if nestElement can be nested in otherElement
    grammarNestable ← table lookup if language grammar allows nesting
    if !grammarNestable then
        return FALSE
    end if

    bestMatchElement ← getBestMatch(nestElement, otherElement)
    if bestMatchElement == None then
        return FALSE
    end if

    isMatch ← Match Rules between nestElement and bestMatchElement
              except then-clause, block, comment, literal, operator
              modifier, name, and expression-statement use Similarity rules
    if !isMatch then
        return FALSE
    end if

    if nestElement.name == "name"
            AND firstNonNameParent(nestElement)
              != firstNonNameParent(otherElement) then
        return FALSE
    end if

    if nestElement.name == otherElement.name then
        matchMetrics ← textual metrics of nestElement and otherElement
        nestMetrics ← textual metrics of nestElement and bestMatchElement
        if (nestMetrics.similariy < matchMetrics.similarity
                OR nestMetrics.dissimilarity > matchMetrics.dissimilarity)
              AND (min(matchMetrics.minSize, nestMetrics.minSize) ≤ 50
                OR nestMetrics.minSize / nestMetrics.similarity
                    ≥ 0.9 * (matchMetrics.minSize / matchMetrics.similarity)) then
            return FALSE
        end if
    end if

    return TRUE
```

Figure 7. Core algorithm that looks for a nestable element and returns a boolean value indicating if nestElement can be nested within otherElement. Various checks are made to test if one element is nestable. This includes check if it is allowed via the grammar, a version of the match rules and check based on similarity/dissimilarity.

Figure 7 gives the core algorithm that checks if an element can be nested into other. As such, it encodes the conditions for nesting. For this, we constructed rules for syntactic elements (statement and sub-statement) that can contain nested elements (based on language grammar). For example, an if-statement can have other if-statements, blocks, while loops, etc. If allowable by the language grammar, four additional checks are made. 1) the best internally matching element is found (Figure 8). The best internally matching element must be the same element, both must be no greater than four times the size of the other (a simple pre-check to eliminate poor matches), and have the highest *textual-similarity* of all possible candidates. In the case of a node having descendants with the same matching node name, only the ancestor is taken (a subsequent nesting will nest further if required). If no item is found, then that item will not be nested. 2) the nesting candidates are checked using rules similar to the *Match* rules except, then-clause, block, comment, literal, operator, modifier, expression, expression-statement, and name are no longer *Always Matched* or *Any Similarity Match*, but use the *Similarity* rules (i.e., a nest can occur if either passes). The remaining details are the same as the *Match* rules. 3) if the element that will be nested is a name (identifier), then the first non-name ancestor of each element must be the same type.

Lastly, 4) if the elements being nested and nested into have the same type (i.e., two if-statements), an additional check is made comparing the item to nest and the the best internally matching element (from condition 1. They must have greater or equal *textual-similarity* and less than or equal amount of *textual-dissimilarity* (i.e., it is more optimal to nest), or the smallest element (in terms of textual nodes) must be sufficiently large (> 50) and the ratio of minimum size of text nodes to *textual-similarity* when nested must be less than 90% of the minimum size/*textual-similarity* ratio when not being nested. The second condition catches false negatives. The best nesting may have slightly less total *textual-similarity* (due to other changes) and still produce a more meaningful difference. However, to reduce false positives, it must have a sufficiently better ratio and is bounded to work for larger structures (we picked > 50 after tuning) where ratios are more stable. The following is an example with the > 50 relaxed due to space restrictions. In this example, the parent if-statement is deleted and its child declaration statement moved into the child if-statement. Because of the move of the declaration statement there is less *textual-similarity* between the the child if-statement in the original and the matching modified if-statement. However, since the original child if-statement is smaller the ratio is better by a notable margin.

Original

```
if(root.ID()) {
    node left = root.left;
    if(child.ID() < root.ID()) {
        this->insert(left, child);
    }
}
```

Modified

```
if(!root.ID()) return;
if(child.ID() < root.ID()) {
    node left = root.left;
    this->insert(left, child);
}
```

srcDiff

```
if(!root.ID()) return;~~if(root.ID()) {~~
~~    node left = root.left;~~
    if(child.ID() < root.ID()) {
    node left = root.left;
        this->insert(left, child);
    }
}
```

```
function getBestMatch(nestElement, otherElement)
    candidateElements ← get elements from otherElement of same type as nestElement
    maxSimilarity ← 0
    matchedElement ← None
    for each i in candidateElements do
        maxLength ← max(nestElement.length, i.length)
        minLength ← min(nestElement.length, i.length)
        if maxLength > 4 * minLength then
            continue
        end if
        similarity ← textual-similarity of nestElement to i
        if similarity > maxSimilarity then
            maxSimilarity ← similarity
            matchedElement ← i
        end if
    end for
    return matchedElement
```

Figure 8. Algorithm that, if exists, finds the best matching element for nestElement which is nested with otherElement. Best is chosen based on max textual similarity.

```
function isNestable(nestElement, otherElement, maxNumberElements)
    if !isNestableCore(nestElement, otherElement) then
        return FALSE
    end if

    if maxNumberElements > 1 AND nestElement.name == "name"
            AND nestElement.parent.name == "expr"
            AND otherElement.parent.name == "expr" then
        return FALSE
    end if

    nestMatchElement ← getBestMatch(nestElement, otherElement)
    nestMetrics ← textual metrics for nestElement and nestMatchElement
    if        !isBetter(nestMetrics, otherElement, nestElement)
          AND !isBetter(nestMetrics, nestElement, otherElement.next)
          AND !isBetter(nestMetrics, otherElement, nestElement.next) then
        return TRUE
    end if
    return FALSE
```

Figure 9. Rules as applied by Figure 5. Applies core nestability checks (Figure 7) and checks if a other possibly better ways of nesting can occur. An additional corner case is checked in the case of trying to nest a name when both parents are expressions. In this case, there can be no other elements that are candidates for nesting.

Figure 9 shows how the core nestability tests are applied in Lines 5 and 9 of Figure 5. First, the core testability checks area applied (Line 2) to check if the item to be nested (*nestElement*) can be placed inside *otherElement*. Then, if the item to be nested is a identifier (name) and parent of both the identifier and what it may potentially be nested into are expressions (expr), then these must be the only elements in *originalChanged* and *modifiedChanged* (i.e., *maxNumberElements* of *originalChanged* and m*odifiedChanged* is 1). Next, checks are made to see if a better nesting is possible if other elements are considered. First, it is checked if *otherElement* is better nested within *nestElement*. Then, *nestElement*, is checked if it will be better nested in the element after *otherElement*. Lastly, *otherElement* is checked if it is better nested in the element after *nestElement*. If either of these will produce a better match, then *nestElement* will not be nested into *otherElement*.

Figure 10 gives the algorithm to determine if a better nesting will occur by using textual metrics. First, the best internally matching element (Figure 8) of the nest element (*nestElement*) and the element we are checking if it is better to nest into (*nestIntoCandidate*) is found. The textual metrics are computed between *nestElement* and the best internally matching element (*candidateNestMatch*) and compared to the baseline textual metrics (*currentMetrics*). *currrentMetrics* is the textual metrics of any pairing (e.g., another nest or a match from *Match*/*Conversion* process). The nest is better if it has greater or equal *textual-similarity* and less than or equal *textual-dissimilarity*, or it has a better ratio of minimal text node size to *textual-similarity*.

Figure 11 gives the algorithm that checks if elements matched during *Match*/*Convertibility* process (matched elements) are better nested. In summary, the algorithm checks if nesting either matched element in the other would produce a better match (Figure 10). Additionally, any subsequent original/modified elements are checked to see if any of the possible matched elements can be nested into them.

The following is an example where the modified code introduced an if-statement that wraps the statement from the original. In this example, the delete-statement is iteratively nested through recursive application of the srcDiff algorithm. The initial application places the delete-statement in the if-statement, a second application places it in the then-clause, and a final recursive application placed the delete-statement in the block.

Original	**Modified**

`delete m_document;`	`if (m_frames.isEmpty()) {`
	`        delete m_document;`
	`}`

srcDiff	

`if (m_frames.isEmpty()) {`
`        delete m_document;`
`}`	

The following is an example of a nest where the element being nested and the element being nested into are the same elements. Here, the original if-statement is removed and the statements in the block are kept. Subsequently, the statements in the modified code are nested into the if-statement in the original.

Original	**Modified**

```
if (shape.has("anchor")) {
    string type = shape.get("anchor");
    if(type != "page") {
        Anchor *anchor = new
            Anchor(shape);
        anchor->loadShape(element);
    }
}
```

```
string type = shape.get("anchor");
if(type != "page") {
    Anchor *anchor = new
        Anchor(shape);
    anchor->loadShape(element);
}
```

srcDiff

```
~~if(shape.has("anchor")) {~~
~~    ~~string type = shape.get("anchor");
~~    ~~if(type != "page") {
    ~~    ~~Anchor *anchor = new
        ~~    ~~Anchor(shape);
    ~~    ~~anchor->loadShape(element);
~~    ~~}
~~}~~
```

```
function isBetter(currentMetrics, nestElement, nestIntoCandidate)
    if !isNestableCore(nestElement, nestIntoCandidate) then
        return FALSE
    end if

    candidateNestMatch ← getBestMatch(nestElement, nestIntoCandidate)
    candidateMetrics ← textual metrics for nestElement and candidateNestMatch
    if (candidateMetrics.similarity ≥ currentMetrics.similarity
          AND candidateMetrics.dissimilarity ≤ currentMetrics.dissimilarity)
        OR candidateMetrics.minSize / candidateMetrics.similarity
           < currentMetrics.minSize / currentMetrics.similarity then
       return TRUE
    end if
    return FALSE
```

Figure 10. Algorithm that checks if one match/nesting is better based on textual metrics. Nesting must be possible via core algorithm (Figure 7).

```
function isBetterNested(original, modified, originalChanged, modifiedChanged)
    matchMetrics ← textual metrics of elements original and modified
```

```
3.     // if original or subsequent (from originalChanged) is better nested into
4.     for each i in originalChanged.slice(original, originalChanged.last)
5.         if isBetter(matchMetrics, modified, i) then
6.             nestMatchElement ← getBestMatch(modified, i)
7.             nestMetrics ← textual metrics of modified element/nestMatchElement
8.             if !isBetter(nestMetrics, i, modified) then
9.                 return TRUE
10.            end if
11.        end if
12.    end for
13.
14.    Repeat lines 2-12 for modifiedChanged.slice(modified, modifiedChanged.end)
15.
16.    return FALSE
```

Figure 11. Algorithm that checks if there is an element that is better nested than the current match. A better match is determined via Figure 10. None checks left out for simplicity.

G. Complexity

The following presents the worst-case time complexity of the srcDiff algorithm. For illustration of its expected/practical purposes, a timing comparison is performed as part of the evaluation in Section III. Let *N* be the maximum number of children for a node, *D* be the maximum number of individual edits (inserts and deletes) between children (which is always less than or equal to *2N*), *C* be the cost of comparing two child nodes, and *R* be the cost of the srcDiff rule computation. Myers' algorithm (Line 2) then runs in *2CND* or *O(CND)* time. In Line 3, the D edits are checked for identical matches of the opposite edit type using a hash map. In the worst case, when everything hashes to the same value and using child node comparison to disambiguate, this is $O(CD^2)$ (which is less than *CND*). Line 4 iterates through the groupings of consecutive edits, consecutive common children, and changes, so at most *O(N)*. Common, deleted, and inserted children are output (lines 5-8), which over all recursive calls to the srcDiff algorithm and in combination with lines 13, 19, and 22, results in the output of all nodes. Let *M* be the maximum number of nodes between both trees, so output takes at most *2M* or *O(M)*. Line 10 uses a quadratic dynamic programming algorithm. At worst all D edits will belong to a single change or $O(RD^2)$ (in this case Line 10 will be *O(1)*). Line 16 (at worst when there are no matches) checks if each node can be nested in the other for $O(RD^2)$. Combined, these will result in at worst *D* recursive calls. Finally, let *K* be the maximum depth of the srcML tree. Lines 10 and 16 will result in *O(DK)* total recursive calls. So, the runtime complexity of the srcDiff algorithm is $O(DK(CND + RD^2) + M)$. In practice, *K*, *N*, and *D* are not typically large numbers. In addition, recursion is stopped (recursive subtrees pruned) when children are determined to be common, deleted, or inserted. Since, a large part of the srcDiff rules rely on children being similar enough, excessively dissimilar children will not result in a recursive call and thus, expected runtime is much less.

For the srcDiff rules (i.e, *R*), t*extual-similarity* and s*yntactic-similarity* are the costliest as they both employ Myers' algorithm on the text nodes and child nodes, respectively. The two times these are used most heavily (as part of the rules) is when checks are made in *determineMatchings* to see if it is better to nest a child in one of the subsequent others of the other edit type and when searching for the best (highest similarity) descendant of a certain type. In the first case, t*extual-similarity* and s*yntactic-similarity* might be applied *O(D)* times. In the latter, t*extual-similarity* will be applied once to each found descendant. Of note, the srcDiff rules are not run on every child part of an edit. *Match* rules will never be applied to two elements of separate type, so, *R* is constant for those children. Similarly, *Convertibility* rules and *Nesting* rules first check if an element can be converted to or nested in the other through simple lookups.

In practice, from various experiments on applying the approach to the complete history of software systems, the bottleneck is primarily *textual-similarity* (and other Myer's shortest edit script usages). The only time this is encountered is in the case of extremely large auto-generated files. For this reason, optional approximation is added in place of Myer's algorithm when one of the sequences is greater than 2048.

Due to the novel approach srcDiff takes, the complexity of srcDiff is not directly comparable to other approaches. That is, srcDiff's complexity is mainly terms of number of children, number of edits, and depth of the tree, while other approaches are generally in terms of the number of nodes in the trees. However, for reference, we provide the complexity of major syntactic differencing algorithms. Namely, we provide the complexity for ChangeDistiller, GumTree, and MTDIFF. These approaches work in two phases: node matching phase and edit-script generation phase. They each have a edit-script generation phase of $O(N^2)$. The node-matching phase complexity of ChangeDistiller [3] is reported as $O(CL^2 + L^2 \log L^2 + CI + IL)$, where C is cost of comparison, L is the maximum number of leaf nodes between the two trees, and I is the number of inner nodes. GumTree [12] reports a worst case $O(M^2)$, where M is the maximum number of nodes in both trees . MTDIFF [15] has a node-matching phase complexity reported as $O(L^2 I^2 \log L^2 + I^2 \log I^2 + L^3)$ where I and L are the number of inner and leaf nodes, respectively.

H. Language Extensibility

The current implementation of srcDiff is not tied to a particular language. It is tied to the srcML (XML) representation. Since the srcML is near identical between the languages it supports (with differences being language specific extensions), it is simple to extend srcDiff to any language supported by srcML (up to C11, C++ 14, Java SE 8, and C# ECMA-334). That is, the only typical modification (addition) to srcDiff and its rules is providing support for the language specific extensions. For example, in order to provide support for Java, Java-specific markup needed to be incorporated into the rules (e.g., static block). The process took only a few hours.

The drawback is the amount of time to add support for a language not yet supported by srcML. In the authors experience, a new language (with syntax similar to those already supported) can be added in about a one-two work week (i.e., 40-80 hours). In addition, there are plans for the srcML infrastructure to support plug-in grammars which will make adding additional languages much easier.

The list of languages srcML supports are all currently procedural/object-oriented and all derive from the C family. As such, srcML is able to produce near identical markup, which allows srcDiff to be used with little to no modification. The current srcDiff implementation's ability to be used with non-C languages or languages of different paradigms depends on how consistent with the typical srcML markup those languages can be produced. The rules are more language independent. For example, conditionals having conditions is language agnostic. However, the srcDiff implementation expects this to be in particular positions in the tree, and if the markup of where these elements occur differs, then the srcDiff implementation will have to undergo a greater amount of updates.

I. Implementation

The current implementation of srcDiff is a command-line tool (Windows, Linux, and Mac) in C++ using the srcML infrastructure [16]. The tool supports several different outputs: srcDiff (XML) for analysis or generation of personal visual outputs and side-by-side views and unified views for humans (in both text and html). As a command-line tool, the approach is easily integrated into software development environments. For example, a short bash script is all that is needed to integrate the srcDiff implementation into the version control system Git. Likewise, a plugin can be written for any IDE (supporting plugins) to utilize srcDiff's built-in views or to generate their own.

III. Evaluation

For evaluation we conduct a user study, where srcDiff and its rules are compared to GumTree [13], GitHub's difference view [17] and Mergely [18]. Timing test (presented in Section III.E) against GumTree [12] are also presented. For the user study, three research questions are formulated.

- **RQ1:** Does srcDiff produce changes that are easier to read and understand than the other approaches?
- **RQ2:** Is srcDiff preferred over the other approaches?
- **RQ3:** Does srcDiff result in a delta that is more accurate than the other approaches?

GumTree is a syntactic-differencing approach. As computing an optimal edit script while considering moves is known to be NP-hard, GumTree uses heuristics to approximate the optimal-edit script. For details of the algorithm please see [13]. Like most syntactic-differencing approaches, GumTree is unable to provide changes to comments and whitespace. GumTree can output a side-by-side view of changes by overlying the AST edits onto the source code. Several other non-visual representations of the edits are also provided. GumTree, is compared to several other differencers [13]. The results show that GumTree produces a more optimal-edit script. In addition, the authors performed a manual investigation to compare the visual output of GumTree to that of Mergely. Once again, the results show GumTree as the best approach compared. As GumTree represent the best research tool currently available, GumTree is selected for this study.

GitHub's difference view is a line-based differencing approach to source-code differencing that also highlights within-line changes. It supports both a side-by-side view and a unified view of changes. It is limited in the fact that it can only be used to view changes in repositories hosted by GitHub. However, as it is used by GitHub, it sees heavy use and is therefore selected for study.

Mergely is another line-based differencing approach that supports highlighting of within-line changes. The within-line changes are computed in a separate manner than GitHub's difference view. It supports a side-by-side view of changes. As Mergely is used in a comparison with GumTree [13], it provides an excellent baseline and point of comparison between these studies.

As previous approaches aim at an optimal edit script, edit script size and number-of-mappings are often used to evaluate different syntactic-differencing approaches [13, 19]. However, such an evaluation is not appropriate here. Edit script size does not take into account the understandability of the resultant delta as we have demonstrated previously. Additionally, edit script cannot be used for comparing syntactic approaches to line-differencing approaches. Lastly, the AST used in srcDiff and GumTree differ drastically (e.g., different number of nodes, different syntactic categories) and a comparison of edit script is therefore not very meaningful.

Since the goal of srcDiff is to produce a delta that is both accurate and understandable to programmers, a within-participant user study is performed using the online survey tool Qualtrics. In terms of a usability study, the participants are provided the original and modified source-code and the output of the four tools. They are tasked with evaluating the approaches on several criteria related to each differencing approach's ability to recover the changes present between the original and modified source code. In order to accomplish this task, participants need to perform some combination of manual inspection of the original and modified source code and utilization of the differencing approaches' outputs in order to determine the changes present. They then evaluate the approach along the selected criteria and provide any additional comments in the form of an open-ended question.

In order to provide for a fair comparison with GumTree, which only supports a side-by-side view, the study consists only of side-by-side views of source-code changes. srcDiff and GitHub also support a unified view, and thus can be the subjects of future studies. A replication package and research data for the study can be found at http://www.sdml.cs.kent.edu/srcDiff/jsep2019.

A. Data Collection and Preparation

Since, GumTree is evaluated on the Java [13] programming language, Java is used as the language for samples in the study. In total, fourteen samples of source-code changes are collected from four separate Java systems: Elasticsearch, Google Guava, SLF4J, and jEdit. Separate systems were used to add support for Java. The following explains how change samples were collected for the study. In general, the collection procedure aims at getting a wider variety of changes to increase the generalizability of the results.

First, 100 revisions are selected at random for each system from all revisions with modified files. From these, two shorter methods, 5 to 30 lines, with smaller amounts of change, 2 to 10 lines as reported by a whitespace-ignoring line diff, and one larger method, 20 to 50 lines, with larger amounts of changes, 10 to 25 lines, are selected at random from the systems Elasticsearch, Google Guava, and SLF4J. These criteria are used to provide for a wide coverage of change complexity while keeping the size of the methods and amount of change manageable for a survey. One sample was redrawn, as the change is too simple to produce variation in the output of each tool. The remaining five samples were chosen to supplement the randomly-drawn methods with different change types and selected by querying the 100 random revisions (via srcML on srcDiff documents). They are: a replaced method (Google Guava), a heavily modified class (Google Guava), and three examples of changes to conditions and conditionals (jEdit and Google Guava).

Ideally, a larger number of samples would have been included in the study, however, any larger of a size is prohibitive and greatly decrease the number of participants and quality of the results [20, 21]. As such, jEdit is not used for the shorter and longer methods, as this generates too many examples[5] for a survey. jEdit is used as part of the supplemental changes to provide both an additional system and a system using Subversion version control system (others utilized Git).

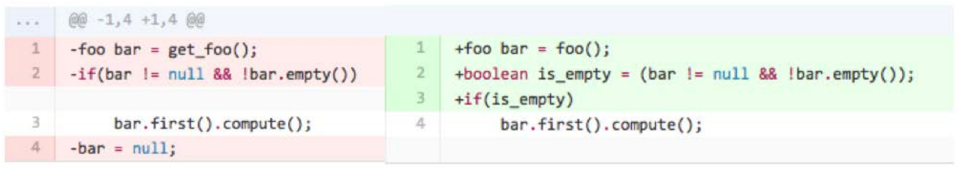
In the survey, GumTree is presented in HTML to allow for the interaction it supports via Javascript. To be comparable, srcDiff is also presented in HTML, although no Javascript interaction is needed. For GitHub and Mergely, we attempted to present them in HTML, but as both are tied to a web service, there was difficulty extracting them, and reproduction would be error prone. So, they are best presented as images. To control for any possible problems, directions are given in the survey to compensate for participants with small screens/resolutions. This worked well, as only three participants noted having trouble viewing GitHub/Mergely. One cannot view them at all, so it was removed from the pool of results, and the other two reported only inconvenience. For two with inconvenience, the results were analyzed with and without them. Since, removal of these two participants negatively affected GitHub and Mergely (clearly showing that having GitHub and Mergely as images is not a threat to validity) they were kept in the study. Several user responses about how much they liked GitHub's view also shows that this is acceptable.

GumTree is unable to report changes to comments, so, comments are stripped from the samples. All the tools are run to ignore whitespace when computing a difference. Additionally, since GumTree is unable to report changes to whitespace, srcDiff is set to not report changes to whitespace. Unfortunately, GitHub and Mergely always show changes to whitespace and have no option to disable output of whitespace changes. As such, participants are instructed to ignore changes to whitespace when answering questions. With exception of GumTree, all the tools are run on only the code samples. As GumTree requires syntactically-complete code, the code is wrapped with a class before it can be run. This is stripped out in the survey so that all tools show the same code. GumTree is also run on the full classes to check the results. There is no difference on ten of the fourteen samples. However, on the four remaining complete files, GumTree produced a slightly worse (sample 3, 5, 10) and far worse (sample 4) delta. On sample 13, GumTree reported two different

[5] Alternatively, we can cut some of the supplementary examples. However, this will not test the important cases contained within the supplementary examples and negatively impact the generalizability of the survey results.

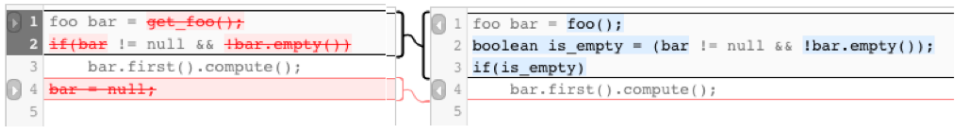
changes depending on the run. The best is chosen. The latest release of GumTree (2.0.0) is used. The previous release (1.0.0) did not perform as well. That is, GumTree is shown in the best possible light for the study.

Since, each tool outputs the differences with a slightly different representation (e.g., highlighting colors), the participants are given instructions on how the representations show the same concepts. Each sample is presented separately and the participant is given the output of the four approaches along with the original and modified code. Participants are instructed to answer in any order and only for the current sample. The samples are shown in random order. The study is blind with regard to the tools. That is, the approaches output is explained to the participants, however, the names of the approaches are not given. Examples of the output of GitHub, Mergely, srcDiff, and GumTree on a simple example are provided in Figures 12-15, respectively.



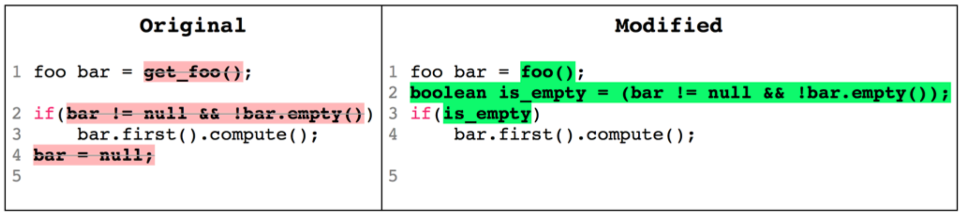
```
@@ -1,4 +1,4 @@
-foo bar = get_foo();                      +foo bar = foo();
-if(bar != null && !bar.empty())           +boolean is_empty = (bar != null && !bar.empty());
                                           +if(is_empty)
    bar.first().compute();                     bar.first().compute();
-bar = null;
```

Figure 12. GitHub output. Deleted code is shown on the left side with the line(s) in a red background and inserted code is shown on the right side with the line(s) in a green background. Deleted lines are also preceded with a minus (-) and inserted lines preceded with a plus (+). When identified, within line changes are highlighed with a darker shade of red/green respectively. Line numbers are provided.



```
foo bar = ~~get_foo();~~                  foo bar = foo();
~~if(bar~~ != null && ~~!bar.empty())~~   boolean is_empty = (bar != null && !bar.empty());
    bar.first().compute();                if(is_empty)
~~bar = null;~~                               bar.first().compute();
```

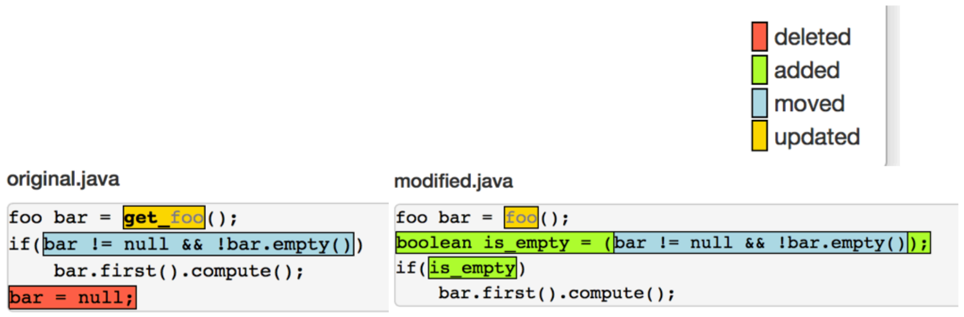
Figure 13. Mergely output. Deleted code is shown on the left side in red with a red background and a line-through. Inserted text is shown on the right side with a blue background.



```
Original                                  Modified

foo bar = ~~get_foo()~~;                  foo bar = foo();
                                          boolean is_empty = (bar != null && !bar.empty());
if(~~bar != null && !bar.empty()~~)       if(is_empty)
    bar.first().compute();                    bar.first().compute();
~~bar = null;~~
```

Figure 14. srcDiff output. Deleted code is shown on the left side with a red background, line-through, and in bold and inserted code is shown on the right side with a green background in bold.



```
foo bar = get_foo();
if(bar != null && !bar.empty())
    bar.first().compute();
bar = null;
```

```
foo bar = foo();
boolean is_empty = (bar != null && !bar.empty());
if(is_empty)
    bar.first().compute();
```

Figure 15. GumTreee output. Deleted code is shown on the left with a red background and inserted code is shown on the right with a green-yellow background. The tool displays updated code as a gold background in both versions with common characters in grey and deleted/inserted characters in bold black. Moved code is in blue on both sides. A legend is also provided for GumTree.

As mentioned previously, each tool outputs slightly different representations of differences. This presents a possible bias. However, we decided not to normalize the outputs to a common format for several reasons. First, normalizing to a common format loses information contained by a representation or add some that is not actually present. This presents another bias. For example, GumTree supports Javascript interaction. This is used, for example, to highlight matching text tokens (i.e., updates) from the original and modified code. Without this feature, it becomes extremely difficult to understand the GumTree output. This could be added to other views to normalize them, but adding this feature to the output of the textual differences would be arbitrary based on our interpretation of the results. It may have improved the readability of srcDiff, though. Similarly, the textual views show change hunks (groups of consecutive changed lines). This is not something directly supported by the other views, and so there is not a good way to normalize them without losing or arbitrarily adding information. Secondly, normalization of all formats is an error-prone process which can result in a major threat to validity. Thirdly, this is the approach to the best of our knowledge was applied by Falleri et al. [12] for which we are comparing results. The fact that we have the same output comparison allows us to compare our results to them. Lastly, the goal of the experiment is to evaluate the accuracy, readability, and understandability of the changes themselves (what is changed), and not how they are marked up. As such, the style in which it is marked up is agnostic. The survey description and explanation tried to emphasize this (e.g., "You ARE evaluating the tools with respect to changes to code"), but it is of course still the threat that the participant misinterpreted or forgot.

B. Measures

The survey evaluated the approaches along four separate criteria: understandability (RQ1), readability (RQ1), personal preference (RQ2), and accuracy (RQ3). TABLE III contains a listing of the questions. The first two questions for accuracy use a standard unipolar rating scale of *None at all*, *A slight amount*, *A moderate amount*, *A large amount*, and *A very large amount* (coded as 6-10). The remaining are constant sum questions, selected to allow for ties and to allow the participants to provide the extent for which a single approach is better than the other. All data is ordinal scale, that is, the median is calculated, while the mean is not valid [15]. In addition, the participants were given the option to provide comments. The questions are asked individually for each sample and after answering the questions for all samples, the participants are asked the same questions about the approaches overall.

C.Results and Discussion

The survey was given as a project in two software engineering classes and to a number of graduate students. Sixty-nine students (including 5 experts) finished the survey. TABLE IV shows the results. As there is almost no variation in the first question, it is not reported. In preference, the participants are also asked to rank the Original and Modified source code. As this received a median of zero in all cases, this is left out. All the research data and full charts are available in the replication package.

TABLE III. SURVEY QUESTIONS.

Criteria	Question
Accuracy - Unchanged	How much of the code is marked as unchanged that should be marked otherwise?
Accuracy - Changed	How much of the code is marked as changed that should be marked otherwise?
Accuracy	Rank the tools by distributing 100 points among the tools relative to how well each tool does at capturing the differences between the original and modified code.
Understandability	Rank the tools by distributing 100 points among the tools relative to how easy each tool's output is to understand.
Readability	Rank the tools by distributing 100 points among the tools relative to how readable each tool's output is.
Preference	Rank the following by distributing 100 points among each relative to your preference.

For understandability/readability questions, srcDiff is rated the best approach, most frequently (most points including ties), while GitHub is ranked second. This provides direct support for RQ1, that is, srcDiff is easier to read and understand. For the preference question, srcDiff is also rated best, most frequently (by a large margin). This also provides direct support for RQ2: srcDiff is preferred over the other approaches. In terms of accuracy, GumTree and srcDiff are tied. We feel this is because the survey instructs participants to compare the approaches to what they determined is actually changed by the developer. Without enough experience examining changes to software, abstracting out what actually changed is a difficult task. That is, a large portion of the participants do not have significant experience with differencing tools and looking at changes relative to what a developer actually performed. While these participants are fully capable of evaluating the readability, understandability, and preference criteria, they did not have enough expertise to evaluate the accuracy. Results on experts (those rated as excellent skills using differencing tools) for the ranking questions, rate srcDiff as the most accurate 15/15 times, most understandable 15/15 times, most readable 15/15 times, and most preferred 14/15 times. No other tool is ranked the best more than 2 times on the combined samples and overall. In essence, GumTree appears more accurate to less skilled developers, possibly due to the additional fine-grained markup on different code, however, since these changes are not meaningful, they are unable to understand the changes. This supports RQ3. srcDiff generally produces a delta that is equal or more accurate that the other approaches. The authors invite the reader to view the changes produced by the approaches in the provided replication package.

In [8], GumTree is compared to Mergely via a manual investigation by the authors. Our study also supports that GumTree is better than Mergely. However, for preference, understandability, and readability, GitHub is more often rated higher than GumTree. GumTree is often ranked lower than all other approaches when presented with more complicated changes (e.g., sample 3, 5, 6), while in the GumTree study, only simple changes are used (those that ChangeDistiller reported as a single change). In contrast, srcDiff is ranked on par with or as the highest. This lends supports our main hypothesis: optimal tree-differences produce poor results on more complicated changes, while srcDiff performs quite well.

The data was investigated for integrity and comments read. Nothing was suspicious about the expert's data. From reading the comments on all participants, two people claimed inability to see code on the overall questions (which had no code, so possibly answered randomly), two had small inconvenience in viewing GitHub and GumTree images, and two may have made up their minds in the middle of the exam and thus stopped answering additional samples wholeheartedly (i.e., given up). The impact each had individually and combined was verified on the data. Removal of those who may not have answered overall correctly changes the results slightly in favor of srcDiff. Those with some inconvenience with GitHub and Mergely images have a slight negative effect of them when removed. Those who may have given up had mostly slight affects. The result on the conclusions (all participants) is that srcDiff is slightly less accurate than GumTree, but GumTree and GitHub are even less preferable and understandable. This makes GumTree slightly more preferable than GitHub and GitHub roughly on par with Mergely. Looking at the data when removing all of these participants slightly benefits srcDiff. As these involve some amount of speculation, and the effects are minor, as they do not affect the conclusion of srcDiff being the most accurate (which comes from expert data) and the other criteria on either data set, these participants were left in with the presented data.

Also, of note the completion time was investigated for all participants. Two non-experts had a somewhat quick, but not impossibly quick response time. The data was also investigated without these two participants. As the results without them were more in favor of srcDiff and do not affect the conclusions, they are also left in the presented data.

Here we provide a summarization of potential biases/confounding variables due to choices with experimental setup and design. Further experiments are needed to address these and further validate the results. As discussed earlier, the tool outputs are not normalized and participants may have evaluated based on this property instead of on the changes themselves as intended. Secondly, images were shown for the textual differences, although we found no indication of this hindering them, this may have been the case. Additionally, wording of questions may be interpreted differently by different participants and may impact results. To mitigate this, preliminary studies were conducted and questions reworded based on feedback. The textual differences reported whitespace-only changes, while the syntactic differences did not, as the textual differences did not have the option to not report and GumTree is incapable of reporting such changes. Lastly, although attempts were made at trying to get a diverse collection of changes, only fourteen samples were present in order to get quality results from participants. This cannot generalize to all changes. Some sample change types are added to try and bolster generalizability, however, section of these types of changes may introduce a selection bias. Additional, experiments are required to further verify results for generalizability and to address potential biases/confounding variables. Further discussion is provided as part of Threats to Validity (Section IV).

Table IV. Results for each of the samples and post-questionnaire (Overall). All reported measures are medians with GitHub (GH), Mergerly (M), srcDiff (SD), and GumTree (GT). Last line shows a count of the number of samples (and Overall) for which the approaches achieved the best result (including ties) out of all the approaches. For Changed, smaller number is better, others larger is better.

	Changed				**Accuracy**				**Understandability**				**Readability**				**Preference**			
Samples	***GH***	***M***	***SD***	***GT***	***GH***	***M***	***SD***	***GT***	***GH***	***M***	***SD***	***GT***	***GH***	***M***	***SD***	***GT***	***GH***	***M***	***SD***	***GT***
Sample 1	6	7	6	6	25	23	27	25	25	24	29	22	25	25	30	20	25	20	25	20
Sample 2	6	7	6	6	25	20	25	25	25	20	25	25	25	20	26	25	25	20	25	25
Sample 3	7	7	6	7	25	25	25	25	30	25	30	15	25	24	30	15	25	20	25	15
Sample 4	8	8	6	6	17	15	30	35	20	20	25	30	25	20	25	30	20	15	28	30
Sample 5	6	6	6	7	25	25	25	25	26	25	30	18	27	25	30	16	25	24	30	15
Sample 6	7	6	6	6	25	25	25	25	25	20	30	15	30	20	30	15	25	20	30	15

Sample 7	7	7	6	6	25	20	25	25	25	24	25	25	25	21	27	25	20	20	25	22
Sample 8	6	7	6	6	20	20	25	30	25	20	26	25	25	20	30	25	20	20	25	25
Sample 9	6	6	6	6	25	25	27	25	25	25	28	25	25	20	30	25	22	20	25	25
Sample10	7	7	6	6	25	20	25	25	25	20	30	20	28	20	26	24	25	20	30	20
Sample11	6	7	6	6	24	21	25	26	25	20	28	25	25	20	28	25	25	20	25	25
Sample12	6	6	6	6	25	22	30	25	25	20	30	22	25	20	30	20	22.5	20	30	20
Sample13	7	7	6	6	25	20	25	25	30	20	27	20	25	20	30	20	25	20	30	20
Sample14	7	6	7	6	23	25	25	25	25	25	25	25	25	25	25	20	20	22	20	20
Overall	7	7	7	7	25	20	25	25	25	20	28	20	25	20	30	20	25	20	25	20
Count	8	6	**14**	13	8	4	**12**	**12**	5	1	**13**	4	3	1	**13**	1	5	1	**13**	5

D. Statistical Significance

The data for each criteria is tested for statistical significance using multiple tests. As more than two approaches are compared (i.e., four), first the Friedman test is applied to see if any of the approaches is different. Then, all pairs of approaches are tested using the Wilcoxon signed-rank test (non-parametric test for paired data) with a Bonferroni correction[6]. Effect size is calculated using the following equation where r is the resulting effect size, Z is the statistic produced by the Wilcoxon signed-rank test and N is the total number of observations (or cases X 2).

$$r = \frac{Z}{\sqrt{N}}$$

To simplify discussion, statistical significance is applied to the responses to all samples as a whole. That is, for each criteria the responses for each of the fourteen samples for all sixty-nine participants (14 x 69 = 966 total responses) are used as the input for the statistical test. SPSS is used for the Friedman test and for the Wilcoxon signed-rank test. We now report the results here using a p-value of 0.05 and applying a Bonferroni correction. When statistically significant, the approach with the highest sum of ranks is chosen as the better approach for the data. In all cases, the approach with the highest sum of ranks also has the most ranks in its favor. For effect size, we use $r \geq 0.1$ for small, $r \geq 0.3$ for medium, and $r \geq 0.5$ for large. If effect size is less than 0.1, it is not reported.

The Friedman test indicates that all criteria are significant. The all-pairs significance (Wilcoxon signed-rank) determinations for Unchanged, Changed, Accuracy, Understandability, and Readability are in Table V, and Preference is in Table VI. For each table, the all pairs Wilcoxon signed-rank test is given. If the null-hypothesis cannot be rejected, then *Null* is listed in that entry. Otherwise, the approach that the statistical test indicates to have performed better for the data is given.

For the Unchanged criteria and for the experiments data, after the Bonferroni correction, nothing is statistically significant. For the Changed criteria and for the experiments data, srcDiff and GumTree are statistically significantly better than GitHub and Mergely, but there is no statistical significance between each other. There is also no statistical significance between GitHub and Mergely. The Accuracy criteria for the experiments data is almost identical to the Changed criteria, except GitHub is not statistically significantly better than Mergely. For the Understandability and Readability criteria and for the experiments data, srcDiff is statistically significantly better than all the other approaches, and GumTree is statistically significantly worse than srcDiff and GitHub. There is no statistical significance between Mergely and GumTree.

[6] Results are from two-tailed analysis and so are a bit conservative.

For the Preference criteria and for the experiments data, all other approaches are statistically significantly better than the original and modified view, srcDiff is statistically significantly better than all other approaches, and GumTree is statistically significantly worse than GitHub and better than Mergely.

TABLE V. STATISTICAL SIGNIFICANCE USING WILCOXON SIGNED-RANK TEST FOR UNCHANGED, CHANGED, ACCURACY, UNDERSTANDABILITY, AND READABILITY CRITERIA. FOR EACH PAIR, *NULL* INDICATES NO STATISTICAL SIGNIFICANCE, OTHERWISE THE APPROACH THAT IS STATISTICALLY BETTER FOR THE DATA IS LISTED. EFFECT SIZE IS INDICATED IN PARENTHESIS WHEN STATISTICALLY SIGNIFICANT WITH SMALL, MEDIUM, AND LARGE. AN EFFECT SIZE LESS THAN SMALL IS NOT REPORTED.

	Unchanged			**Changed**			**Accuracy**			**Understandability**			**Readability**		
	Mergely	***srcDiff***	***Gum Tree***	***Mergely***	***srcDiff***	***Gum Tree***	***Mergely***	***srcDiff***	***Gum Tree***	***Mergely***	***srcDiff***	***Gum Tree***	***Mergely***	***srcDiff***	***Gum Tree***
GitHub	Null	Null	Null	Null	srcDiff (small)	Gum Tree (small)	GitHub	srcDiff (small)	Gum Tree (small)	GitHub (small)	srcDiff (small)	GitHub (small)	GitHub (small)	srcDiff (small)	GitHub (small)
Mergely	-	Null	Null	-	srcDiff (small)	Gum Tree (small)	-	srcDiff (small)	Gum Tree (small)	-	srcDiff (small)	Null	-	srcDiff (medium)	Null
srcDiff		-	Null		-	Null		-	Null		-	srcDiff (small)		-	srcDiff (small)

TABLE VI. STATISTICAL SIGNIFICANCE USING WILCOXON SIGNED-RANK TEST FOR PREFERENCE CRITERIA. FOR EACH PAIR, *NULL* INDICATES NO STATISTICAL SIGNIFICANCE, OTHERWISE THE APPROACH THAT IS STATISTICALLY BETTER FOR THE DATA IS LISTED. EFFECT SIZE IS INDICATED IN PARENTHESIS WHEN STATISTICALLY SIGNIFICANT WITH SMALL, MEDIUM, AND LARGE. AN EFFECT SIZE LESS THAN SMALL IS NOT REPORTED.

	Preference			
	GitHub	***Mergely***	***srcDiff***	***GumTree***
Original/Modified	GitHub (medium)	Mergely (medium)	srcDiff (large)	GumTree (medium)
GitHub	-	GitHub (small)	srcDiff (small)	GitHub
Mergely		-	srcDiff (medium)	GumTree
srcDiff			-	srcDiff (small)

The Friedman test also indicates that all criteria are significant when only considering experts. Table VII contains the all-pairs statistical significance for the Unchanged, Changed, Accuracy, Understandability, and Readability criteria for the experts. The expert's preference criteria all-pairs statistical significance is in Table VIII. For the Unchanged criteria and for the experiment data, there is no significant difference between GitHub, Mergely, srcDiff, and GumTree (with the Bonferroni correction). For the Changed criteria and for the experiment data, srcDiff is statistically significantly better than all other approaches, Mergely is statistically significantly better than GitHub. For the Accuracy, Understandability, and Readability and for the experiment data, srcDiff is statistically significantly better than all other approaches. Also for all three criteria, there is no statistically significant difference between GitHub, Mergely, and GumTree. For the Preference criteria and for the experiment data, all approaches are statistically significantly more preferred than the original and modified view, srcDiff is statistically significantly more preferred than all other approaches, and there is no statistical significance between any of the other approaches.

TABLE VII. STATISTICAL SIGNIFICANCE FOR EXPERTS USING WILCOXON SIGNED-RANK TEST FOR UNCHANGED, CHANGED, ACCURACY, UNDERSTANDABILITY, AND READABILITY CRITERIA. FOR EACH PAIR, *NULL* INDICATES NO STATISTICAL SIGNIFICANCE, OTHERWISE THE APPROACH THAT IS STATISTICALLY BETTER FOR THE DATA IS LISTED. EFFECT SIZE IS INDICATED IN PARENTHESIS WHEN STATISTICALLY SIGNIFICANT WITH SMALL, MEDIUM, AND LARGE. AN EFFECT SIZE LESS THAN SMALL IS NOT REPORTED.

	Unchanged			**Changed**			**Accuracy**			**Understandability**			**Readability**		
	Mergely	***srcDiff***	***Gum Tree***	***Mergely***	***srcDiff***	***Gum Tree***	***Mergely***	***srcDiff***	***Gum Tree***	***Mergely***	***srcDiff***	***Gum Tree***	***Mergely***	***srcDiff***	***Gum Tree***
GitHub	Null	Null	Null	Mergely (small)	srcDiff (medium)	Null	Null	srcDiff (medium)	Null	Null	srcDiff (medium)	Null	Null	srcDiff (medium)	Null
Mergely	-	Null	Null	-	srcDiff (medium)	Null	-	srcDiff (medium)	Null	-	srcDiff (medium)	Null	-	srcDiff (medium)	Null
srcDiff		-	Null		-	srcDiff (small)		-	srcDiff (medium)		-	srcDiff (medium)		-	srcDiff (medium)

TABLE VIII. STATISTICAL SIGNIFICANCE FOR EXPERTS USING WILCOXON SIGNED-RANK TEST FOR PREFERENCE CRITERIA. FOR EACH PAIR, *NULL* INDICATES NO STATISTICAL SIGNIFICANCE, OTHERWISE THE APPROACH THAT IS STATISTICALLY BETTER FOR THE DATA IS LISTED. EFFECT SIZE IS INDICATED IN PARENTHESIS WHEN STATISTICALLY SIGNIFICANT WITH SMALL, MEDIUM, AND LARGE. AN EFFECT SIZE LESS THAN SMALL IS NOT REPORTED.

	Preference			
	GitHub	***Mergely***	***srcDiff***	***GumTree***
Original/Modified	GitHub (large)	Mergely (medium)	srcDiff (large)	GumTree (medium)
GitHub	-	Null	srcDiff (medium)	Null
Mergely		-	srcDiff (large)	Null
srcDiff			-	srcDiff (medium)

In essence, the statistical significance provides additional evidence for the claims made in sub-section C. That is, for the fourteen samples utilized, srcDiff is more understandable, readable, and preferable (all participants and experts). For the Accuracy criteria: with all participants, srcDiff is on par or better than the other approaches, while with only experts, srcDiff is the most accurate. Additional studies on more changes are needed to see if this generalizes.

E. Timing

To illustrate srcDiff's efficiency, we ran timings on srcDiff and GumTree using changes between 1.509.4 and 1.532.2 of the continuous integration server Jenkins (i.e., same releases as in GumTree [13]). However, we did not discard any files, running both tools on all modified files in revisions between releases. In total, git log reports 1309 modified Java files. In contrast to [13] and to better reflect a user's expectations, we time entire runs of each using the time command. For srcDiff, a unified diff is computed (i.e., typical user request). As GumTree's human readable output is to a URL requiring human interaction, its JSON output (latest revision March 2016 develop build) is used. The minimum of three runs on each file is taken. srcDiff took a mean wall clock time of 0.1 seconds, while GumTree took 1.17 seconds[7]. Although GumTree's

[7] srcDiff uses a denser (more fine-grained) tree. The difference depends on the particular piece of code, however, a difference between 2-4 times as many nodes has been seen. As a consequence, srcDiff produces an even greater efficiency relative to Gumtree on a tree of equal size/density.

human readable output is setup to listen to changes (i.e., subsequent comparisons faster than reported here), it is significantly slower in initial startup and thus in computing a delta for analysis[8]. A plot of the timings versus file size is shown in Figure 16. srcDiff appears to be linear with regards to file size and GumTree is mostly linear with more noticeable deviations. Of note, as mentioned in section II.G, srcDiff complexity is also based off of the number of changes. A large number of changes (not typical) can impact execution time, however, this trend does not show in the timing data. All timings run on iMac running El Capitan with 2.7GHz i5/8GB RAM.

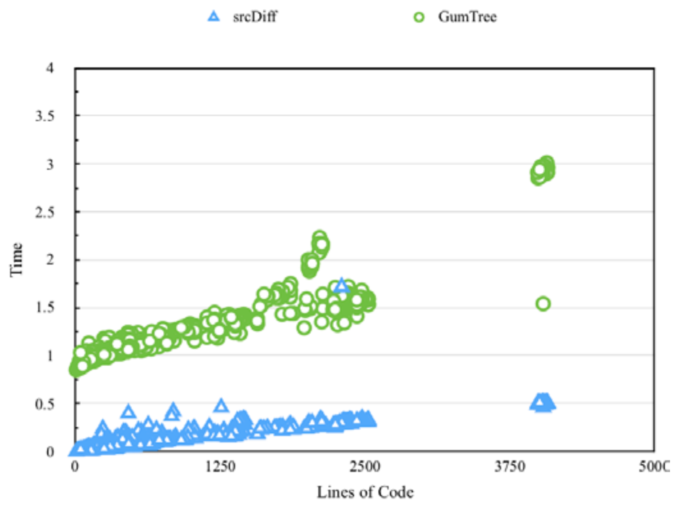


Figure 16. Timing comparison between srcDiff and GumTree. Blue is srcDiff and green is GumTree. srcDiff is largely linear, while GumTree largely appears to be linear, however, there are more noticeable deviations, especially with the larger file sizes.

IV. Threats to Validity

In this section we present the following threats of validity to: construct validity in section A, internal validity in section B, and external validity in section C.

A. Construct Validity

Construct validity is the degree at which the research instrument (i.e., survey) correctly measure what it is designed to measure. One threat may be that a single representation of changes is not used for all four approaches. If all the formats were unified into the GumTree view, then as the GitHub and Mergely have no equivalent markup or report moves and updates, the conversion of such is arbitrary (biased). If all are unified into one of the other simpler views, then the possible accuracy that GumTree is afforded by the additional markup is lost (bias) and may make the output more difficult to read and understand (bias). Additionally, to the best of our knowledge this is applied by Falleri et al. [13] in their study. As such, we feel that the explanation of the different approaches is the best course of action.

In addition, accuracy is asked in regard to how well the developers changes are recovered and since the differences in markup of each type is explained, it is largely agnostic to how it is stylistically marked up.

[8]The goal is to show the actual expected runtimes. However, some of the slowness of GumTree is related to its implementation in Java. That is, although out of the scope of this work, a reimplementation of GumTree into a more efficient language may improve its runtime, however, the extent is not known. In either case, the timing shows that the srcDiff approach can be effectively used with low expected runtimes.

Understandability is asked in relation to how easy it is to understand the output. By definition, to understand is "to perceive the meaning" [22]. So, the question asks how easily they can perceive the meaning of the output. As the participants were instructed in how each tool presents changes (i.e., they are equally capable of understanding how each change type is stylistically marked up), this is largely what is marked up and as what operation. For many, understandability and readability are the same measure (at least one comment in the data attests to this), to others they are not. As the results are close, they are regarded as mostly the same. Preference is based on the individual and their weighting of the other criteria.

Another threat may be that images were used for the textual differencers, while HTML was provided for the syntactic differencers. As explained previously, we found no evidence that this impacted the results.

B. Internal Validity

Internal validity is the degree to which changes in our response variables (e.g, accuracy.) are caused by our causal variables (i.e., type of tool). One aspect that may affect ability to evaluate the approaches is color blindness, which is controlled for in the study. The removal of all color-blind participants is negligible and does not affect the conclusions.

GumTree and srDiff use different parsers and thus operate on different trees . As each operate on resulting trees of different granularity, this may affect the accuracy of what they produce. In particular, srcDiff operates on a denser (more fine-grained) tree. In our experience, traditional approaches such as GumTree begin to give more inaccurate results as the tree becomes more fine-grained as it give the opportunity for matches based on coincidental similarities in the tree. That is, it is we believe that applying GumTree to a denser tree (such as srcDiff) will most likely degrade its performance, while srcDiff is applicable to such a fine-grained tree.

C. External Validity

External validity is the degree to which results may generalize. As a survey is used, only fourteen samples are chosen so that the survey can be conducted in a manageable amount of time necessary to obtain responses and quality answers [20, 21]. Adding questions makes the survey too long and drastically lowers the response rate. To mitigate potential problems with response rate/quality, subjects are informed that they can do the survey in multiple sittings and given proper incentive. While all possible change situations cannot be covered, nine of the fourteen are drawn at random. The remaining are used to supplement the randomly drawn in order to make sure certain types of changes are covered, e.g., changes to conditions and conditionals which we regard as very important for differencing. Even still, these selected changes represent a possible bias and threat to validity. In summation, we can claim that srcDiff is better in both the random cases and important cases selected for the survey, but not cannot do so in the general case.

One threat to validity may be the experience of the participants. A broader set of industry professionals may give different results. This threat is mitigated as some participants have expert-level programming/differencing experience (self-assessment has been shown to be reliable [23]), and/or have worked in industry. Additionally, the graduate students (who generally had more programming experience) are from the same lab as a few of the authors. Despite the study being conducted blind, they may have been able to identify the tools. The results are analyzed with and without the graduate students. The only difference with the conclusions is that with all participants, GumTree was seen more accurate on a few samples. Results with the remaining criteria with all participants and the experts on all criteria (with and without graduate students) remains the same. As there is little difference, we presented the data together.

The rules are derived from examining real-world software projects and our experience as developers. Situations may exist for which no current rule applies (i.e., results will not generalize to those cases). We designate this future work and note that srcDiff is highly adaptable (i.e., new rules easily added) and not fixed on immutable algorithms. We are confident in the rules presented, as shown by the user study.

srcDiff is developed primarily to support C/C++. Minimal effort went into adding support for Java. Different systems are used for investigation (android libcore, lucene-solr, and hadoop) and multiple systems for study. As such, there is no over-fitting present.

Comparison to other approaches may yield different results. However, since we compared multiple approaches (line and state-of-the-art syntactic) we are confident results generalize.

People commit to different version control systems (VCS) in different granularities, especially centralized/decentralized systems. Both Subversion and Git are used in the survey/investigation, although Subversion is used to a lesser extent. Further validation is needed to show results generalize to both VCS types.

V. Related Work

To the best of the authors' knowledge, no work has presented a syntax-based differencer that completely preserves the underlying source code and thus can show changes to whitespace, comments, preprocessor, and code, works on syntactically incomplete and incorrect code, nor has any work been done for source-code differencing that uses rules based on the source-code syntax as presented here. In addition, to the best of the author's knowledge, no previous work has done a full user study on viewing source-code changes. An exception is GumTree, which performed a manual investigation with the authors and is related to the manual investigation used to develop srcDiff's rules and Dotzler and Philippsen [15] that performed a questionnaire with few participants (eight participants) that are not all computer scientists and only using optimal tree-differencers.

The work most related to srcDiff is that of syntactic differencing. In general, previous work on syntactic differencing emphases obtaining a minimal edit script, which as shown can lead to incorrect deltas in some cases. srcDiff differs by purposefully deviating from a minimal edit script in order to obtain a more understandable delta. In addition, previous work on syntactic differencing does not incorporate rules based on the syntax of language and domain knowledge of differencing. The following summarizes previous related work on syntactic differencing. Fluri et al. [3] presents ChangeDistiller, a tree-differencer, which computes an approximate minimal edit script (based on computing similarities) that is used to report and classify the tree edit operations according to their taxonomy [24]. ChangeDistiller is designed to work on a simplified tree where leaf nodes are statements. GumTree compared itself to ChangeDistiller both on the ChangeDistiller tree and a fuller-tree and is shown to produce a shorter edit script in most to nearly all cases. In addition, GumTree is compared to RTED [25], a tree-differencer that produces an optimal edit-distance when moves are not considered, and was found to produce a shorter edit script in a majority of the cases. Dotzler and Philippsen [15] improve GumTree, ChangeDistiller, and other differencers to produce even shorter edit scripts. They also present MTDIFF which is based on ChangeDistiller but has better move detection. The work of Dotzler and Philippsen [15] was not yet published at the time of our user study or original write up. However, as their goal is to create more optimal edit script, this will only exasperate the problems identified in this paper. diff/TS [19] is a syntactic differencing approach that like GumTree aims to approximate an optimal tree-edit distance. It provides a view where edit operations are overlaid onto the source code text. As GumTree is more recent, and it compared itself to other well-known source-code differencers, GumTree was chosen. As diff/TS also approximates an optimal tree edit distance, the results of our study generalize. JSync [9] is a clone management tool for evolving software that incorporates an approximate optimal tree edit algorithm of their own design and LTDIFF [8] simulates tree differencing for use with their semantic merging. As part of both approaches (as well as a very early version of srcDiff [5]), they first use textual differencing and map/align the changes to text with an AST to come up with a tree-difference. Both approaches may possibly be used to report a human readable difference. However, it is not their original intent and since, the approaches are doing a form of approximation of optimal tree-differencing the results of this paper still apply.

Smart Differencer [26], based on Baxter's Design Maintenance System (DMS) [27], is a heavy-weight compiler-based approach to source code parsing and syntactic differencing of source code which describes the changes in terms of abstract editing operations and gives the code fragments affected. It reports changes to preprocessor and comments, but not to whitespace. As Smart Differencer is propriety software, the details of how it works are unknown.

Dex is a syntax and semantic source code difference tool that uses graph differencing on Abstract Semantic Graphs (ASG) created utilizing patches between two systems. The output is a properties file describing the changes in a patch [2]. JDiff is a diff tool for Java that understands object-oriented features and outputs a pair of nodes sets and their status (modified/unchanged) [1]. As these are semantic approaches and what they provide as output is different, they are not directly comparable with srcDiff.

Ldiff [28] is an extension to GNU diff that adds enhancements to identify fragment changes and moves. Its output is similar to GNU diff. Both GNU diff and Ldiff can be used in conjunction with a diff viewer such as TKDiff, which will highlight within line changes. spiff [29] produces results similar to GNU diff, but highlights tokens within a line that changed. Google-diff-match-patch [30] computes changes on a character basis and colors the changed tokens. Eclipse's Structure Compare incorporates both a Java Structure Compare that reports changes to top-level structures and a Text Compare and Java Source Compare which reports changes to source via textual differencing [31]. As all these tools are textual, the differences they report do not always make sense syntactically. The study included two popular textual difference tools and thus, the results also generalize to these tools.

Other related work includes, Semantic Diff which performs a difference using the change of dependence relations between inputs and outputs of a procedure and results in a textual description [32]. LSdiff infers systematic structural differences as logic rules [33]. These approaches report differences in a separate manner than what is presented here, and can be used in conjunction with a tool like srcDiff.

Omori and Maruyama [34] propose an IDE that records edit operations as they occur. Similarly, SpyWare is a change-aware development toolset that captures changes as they occur and can provides many views of source-code change, including a session inspector that allows changes in a session to be replayed [4]. Dias [35] gives another example of an approach that listens and records developer actions taken in an IDE. The main difference is these approaches record changes as they are made, while srcDiff, is able to recover changes without having a specially designed development environment.

Lastly, there is a considerable number of approaches for differencing non-source code. Since, srcDiff works on an XML representation of source-code, the most relevant are XML differencing techniques [36-39]. XML differencing techniques are designed for general XML documents, that is, unlike srcDiff, they have no knowledge of the underlying source-code or AST and its significance for differencing. The most recent version of srcDiff appears in a Master's thesis [40] and dissertation [41] and is similar only in name to [5].

VI. Conclusion and Future Work

Based on the results of the study presented here we feel that previous work on syntactic differencing has incorrectly assumed that producing a more optimal edit script produces a difference that is more accurate and understandable for developers. Furthermore, many syntactic differencers that output a visual diff, do so by overlaying the edits on the text and typically do not address whitespace or comments/preprocessor changes which are vital [42]. Our work presents several rules to enhance the presentation of deltas for syntactic-based differencing of source code. These rules are incorporated into the tool srcDiff, which runs directly on un-preprocessed text and AST and can selectively display changes to whitespace and comments. The rules are validated in by a user study administers as a survey, where the output of srcDiff is compared against a state-of-the-art syntactic differencer GumTree and well-known line-differencing approaches. srcDiff is generally shown to be the most accurate, readable, understandable, and preferable.

To the best of the authors' knowledge, this paper presents the first full user study on viewing source code changes. While results are promising, we are only able to compare to tools that produce a side-by-side difference for a fair comparison. In the future, we will compare srcDiff to tools that produce a unified difference, however, we know of no other syntactic difference approach that produces a unified diff.

While this work is limited to changes to C++/Java source-code, the rules are pervasive across languages. srcDiff can be applied easily to the other languages supported by srcML (i.e., C and C#) with only minimal effort to investigate if un-handled syntactic constructs present in those languages need additional rules. Since, srcML uses a near identical AST for each language, srcDiff can also do a difference between languages.

We do not claim that this is a complete set of rules. The identification of additional rules needs to be investigated and validated through exploration of changes to existing source code projects. Finally, srcDiff currently only detects pure moves (moves with no modifications) that occur in the same file and are siblings in the AST. Extending pure move detection within a file to any part of the AST (and not just siblings) is fairly simple to do as a post processing step. However, the detection of moves with modifications and moves across files is a much more difficult task [43]. Further analysis and investigation is needed to see if this support for move detection is warranted.

ACKNOWLEDGMENTS

This work was supported in part by a grant from the National Science Foundation CNS 13-05292/05217.

REFERENCES

[1] Apiwattanapong, T., Orso, A., and Harrold, M., "JDiff: A differencing technique and tool for object-oriented programs", *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, vol. 14, no. 1, 2007, pp. 3-36.

[2] Raghavan, S., Rohana, R., Podgurski, A., and Augustine, V., "Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11 - 14 2004, pp. 188-197.

[3] Fluri, B., Wursch, M., Pinzger, M., and Gall, H. C., "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction", *IEEE Transactions on Software Engineering*, vol. 33, no. 11, 2007, pp. 725-743.

[4] Robbes, R. and Lanza, M., "SpyWare: a change-aware development toolset", in *30th International Conference on Software Engineering (ICSE'08)*. Leipzig, Germany: ACM, 2008, pp. 847-850.

[5] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11-17 2004, pp. 210-219.

[6] Collard, M. L., Decker, M., and Maletic, J. I., "Lightweight Transformation and Fact Extraction with the srcML Toolkit", in Proceedings of 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11), Williamsburg, VA, USA, Sept 25-26 2011, pp. 10 pages.

[7] Myers, E. W., "An O(ND) difference algorithm and its variations", *Algorithmica*, vol. 1, 1986, pp. 16.

[8] Hunt, J. J. and Tichy, W. F., "Extensible Language-Aware Merging", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'02), Montreal, Canada, October 3-6 2002, pp. 511-520.
[9] Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J., and Nguyen, T. N., "Clone Management for Evolving Software", *IEEE Transactions on Software Engineering*, vol. 38, no. 5, 2012, pp. 1008-1026.
[10] Apiwattanapong, T., Orso, A., and Harold, M. J., "A Differencing Algorithm for Object-Oriented Programs", in *19th IEEE International Conference on Automated Software Engineering (ASE '04)*. Linz, Austria: IEEE, 2004.
[11] Apiwattanapong, T., Orso, A., and Harrold, M. J., "JDiff: A Differencing Technique and Tool for Object-oriented Programs", *Automated Software Engineering*, vol. 14, no. 1, 2007, pp. 3-36.
[12] Falleri, J.-R. m., Morandat, F. a., Blanc, X., Martinez, M., and Montperrus, M., "Fine-grained and accurate source code differencing", in *29th ACM/IEEE International Conference on Automated software engineering (ASE'14)*. Vasteras, Sweden: ACM, 2014, pp. 313-324.
[13] Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M., "Fine-grained and accurate source code differencing", in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Vasteras, Sweden: ACM, 2014, pp. 313-324.
[14] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to algorithms*, MIT press, 2009.
[15] Dotzler, G. and Philippsen, M., "Move-optimized source code tree differencing", in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. Singapore, Singapore: ACM, 2016, pp. 660-671.
[16] Collard, M. L., Decker, M. J., and Maletic, J. I., "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration", in Proceedings 2013, pp. 516-519.
[17] GitHub, "GitHub", Date Accessed: March 25, 2016, https://github.com, 2016.
[18] Peabody, J., "Mergely - Online diff, merge documents", Date Accessed: March 25, 2016, http://www.mergely.com, 2016.
[19] Hashimoto, M. and Mori, A., "Diff/TS: A Tool for Fine-Grained Structural Change Analysis", in Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08), 15-18 Oct. 2008 2008, pp. 279-288.
[20] Deutskens, E., de Ruyter, K., Wetzels, M., and Oosterveld, P., "Response Rate and Response Quality of Internet-Based Surveys: An Experimental Study", *Marketing Letters*, vol. 15, no. 1, 2004/02/01 2004, pp. 21-36.
[21] Galesic, M. and Bosnjak, M., "Effects of questionnaire length on participation and indicators of response quality in a web survey", *Public opinion quarterly*, vol. 73, no. 2, 2009, pp. 349-360.
[22] Dictionary.com, L., "Dictionary.com | Meanings and Definitions of Words at Dictionary.com", Date Accessed: July 14, 2017, http://www.dictionary.com, 2017.
[23] Feigenspan, J., Kästner, C., Liebig, J., Apel, S., and Hanenberg, S., "Measuring Programming Experience", in Proceedings of IEEE 20th International Conference on Program Comprehension (ICPC), Passau, Germany, 11-13 June 2012 2012, pp. 73-82.
[24] Fluri, B. and Gall, H. C., "Classifying Change Types for Qualifying Change Couplings", in Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06), 0-0 0 2006, pp. 35-45.

[25] Pawlik, M. and Augsten, N., "RTED: a robust algorithm for the tree edit distance", *Proceedings of the Very Large Database (VLDB) Endowment*, vol. 5, no. 4, 2011, pp. 334-345.
[26] Semantic-Designs, "Smart Differencer Tool", Date Accessed: December 2, 2014, www.semdesigns.com/Products/SmartDifferencer/, 1995-2012.
[27] Baxter, I. D., Pidgeon, C., and Mehlich, M., "DMS: Program Transformations for Practical Scalable Software Evolution", in Proceedings of 26th International Conference on Software Engineering (ICSE04), Edinburgh, Scotland, UK, May 23 -28 2004, pp. 625-634.
[28] Canfora, G., Cerulo, L., and Penta, M. D., "Ldiff: An enhanced line differencing tool", in Proceedings of 31st International Conference on Software Engineering (ICSE'09), 2009, pp. 595-598.
[29] "dontcallmedom/spiff", github.com/dontcallmedom/spiff,
[30] "google-diff-match-patch - Diff, Match and Patch libraries for Plain Text - Google Project Hosting", code.google.com/p/google-diff-match-patch/,
[31] The-Eclipse-Foundation, "eclipse", Date Accessed: June 22, 2018, https://eclipse.org, 2018.
[32] Jackson, D. and Ladd, D. A., "Semantic Diff: A Tool for Summarizing the Effects of Modifications", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'94), Victoria, British Columbia, Canada, September 19-23 1994, pp. 243-252.
[33] Loh, A. and Kim, M., "LSdiff: a program differencing tool to identify systematic structural differences", in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, vol. 2. Cape Town, South Africa: ACM, 2010, pp. 263-266.
[34] Omori, T. and Maruyama, K., "A change-aware development environment by recording editing operations of source code", in *International Working Conference on Mining Software Repositories*. Leipzig, Germany: ACM, 2008, pp. 31-34.
[35] Dias, M., Supporting Software Integration Activities with First-Class Code Changes, Laboratoire d'Informatique Fondamentale de Lille, 2015.
[36] INRIA, "XyDiff", http://www-rocq.inria.fr/~cobena/cdrom/www/xydiff/eng.htm,
[37] IBM, "XML TreeDiff", Date Accessed: May 16, http://alphaworks.ibm.com/tech/xmltreediff, 1998.
[38] DeltaXML, "Diff, Compare, & Merge XML | DeltaXML", Date Accessed: March 24, 2016, http://www.deltaxml.com, 2000-2016.
[39] Wang, Y., DeWitt, D. J., and Cai, J.-Y., "X-Diff: An Effective Change Detection Algorithm for XML Documents", in Proceedings of 19th International Conference on Data Engineering (ICDE'03), Bangalore, India, 2003, pp. 519-530.
[40] Decker, M. J., Structural Analysis of Source-code Changes in Large Software Through srcDiff and DiffPath, The University of Akron, Akron, Ohio, 2012.
[41] Decker, M. J., srcDiff: Syntactic Differencing to Support Software Maintenance and Evolution, Kent State University, Dissertation, 2017.
[42] Hunt, J. J., "Fast Semi-Semantic Differencing and Merging", Web page, Date Accessed: 02/01/2004, http://wwwswt.fzi.de/cocoon/mount/swt/mitarbeiter/jjh/, 2004.
[43] Godfrey, M. W. and Lijie, Z., "Using origin analysis to detect merging and splitting of source code entities", *IEEE Transactions on Software Engineering*, vol. 31, no. 2, 2005, pp. 166-181.
[44] Levenshtein, V. I., "Binary codes capable of correcting deletions, insertions and reversals", in Proceedings of Soviet physics doklady, 1966, pp. 707.