# Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction

Beat Fluri, *Student Member*, *IEEE*, Michael Würsch, *Student Member*, *IEEE*, Martin Pinzger, *Member*, *IEEE*, and Harald C. Gall, *Member*, *IEEE*

**Abstract**—A key issue in software evolution analysis is the identification of particular changes that occur across several versions of a program. We present *change distilling*, a tree differencing algorithm for fine-grained source code change extraction. For that, we have improved the existing algorithm by Chawathe et al. for extracting changes in hierarchically structured data [8]. Our algorithm extracts changes by finding both a match between the nodes of the compared two abstract syntax trees and a minimum edit script that can transform one tree into the other given the computed matching. As a result, we can identify fine-grained change types between program versions according to our *taxonomy of source code changes*. We evaluated our change distilling algorithm with a benchmark that we developed, which consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects. We achieved significant improvements in extracting types of source code changes: Our algorithm approximates the minimum edit script 45 percent better than the original change extraction approach by Chawathe et al. We are able to find all occurring changes and almost reach the minimum conforming edit script, that is, we reach a mean absolute percentage error of 34 percent, compared to the 79 percent reached by the original algorithm. The paper describes both our change distilling algorithm and the results of our evaluation.

**Index Terms**—Source code change extraction, tree-differencing algorithms, software repositories, software evolution analysis.

## 1 Introduction

Since Lehman's Laws of Program Evolution from the 1980s [25], it has been well understood that software has to be adapted to changing requirements and environments or it becomes progressively less useful. Change is broadly accepted as a crucial part of a software's life cycle. As a consequence, in recent years, several techniques and tools have been developed to aid software engineers in maintaining and evolving large complex software systems. For instance, Ying et al. or Zimmermann et al. developed approaches that guide programmers along related changes by telling them "programmers who changed these functions also changed..." [45], [47]. The Hipikat tool of Čubranić et al. used project history information to provide recommendations for a modification task [9]. Gall et al. detected possible maintainability hot spots by analyzing cochange relationships of modules [13].

We argue that such techniques and tools are valuable but suffer from the low quality of information available for changes. Typically, such information, in particular for source code, is stored by versioning systems (for example, CVS or Subversion). They keep track of changes by storing the text lines *added* and/or *deleted* from a particular file. Structural changes in the source code are not considered at all.

More sophisticated approaches are able to narrow down changes to the method level, but fail in further qualifying changes such as the addition of a method invocation in the else branch of an if-statement. Furthermore, a classification of changes according to their impact on other source code entities is missing. In particular, the latter information is important to improving the quality of software evolution results and, as a consequence, to providing better support for programmers and system analysts.

Since source code can be represented as abstract syntax trees (ASTs), tree differencing can be used to extract detailed change information. This approach is promising because exact information on each entity and statement is available in an AST. In our previous work [12], we built a *taxonomy of source code changes* that defines source code changes according to tree edit operations in the AST and classifies each change type with a *significance level*. The level expresses how strongly a change may impact other source code entities and whether a change may be functionality modifying or functionality preserving. In our taxonomy, we focus on object-oriented programming languages (OOPLs) and Java in particular. By adjusting the change type extraction, the taxonomy can also be used for other OOPLs. In total, our taxonomy defines 35 change types.

In this paper, we present *change distilling*, a tree-differencing algorithm for fine-grained source code change extraction. For that, we improved the existing algorithm for extracting changes in hierarchically structured data by Chawathe et al. [8]. This algorithm finds changes according to basic tree edit operations such as *insert*, *delete*, *move*, or *update* of tree nodes.

• *The authors are with the Department of Informatics, University of Zurich, Binzmühlestrasse 14, CH-8050 Zürich, Switzerland. E-mail: {fluri, wuersch, pinzger, gall}@ifi.uzh.ch.*

*Manuscript received 15 Jan. 2007; revised 13 July 2007; accepted 23 July 2007; published online 3 Aug. 2007.*
*Recommended for acceptance by H. Muller.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0012-0107.*
*Digital Object Identifier no. 10.1109/TSE.2007.70731.*

The contributions of this paper are twofold: 1) our change distilling algorithm and 2) a *benchmark* to evaluate source code change extraction algorithms.

Our change distilling algorithm uses the *bigram string similarity* to match source code statements (such as method invocations, condition statements, and so forth) and the *subtree similarity* of Chawathe et al. to match source code structures (such as if statements or loops). To further improve the matching, we use a best match algorithm for all leaf nodes and inner node similarity weighting. To overcome mismatch propagation in small subtrees, we use dynamic thresholds for subtree similarity.

The second contribution of this paper, the benchmark we developed, consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects to evaluate our change distilling algorithm. Compared to the original change extraction algorithm by Chawathe et al., we perform 45 percent better. We almost reach the minimum conforming edit script, that is, we reach a mean absolute percentage error of 34 percent. With this knowledge about source code, changes in existing software evolution analysis tools can be improved. For instance, the Hatari tool rates the risk of changing a method according to the frequency of method changes that caused a bug [33]. Detailed information about the changes is not taken into account, for instance, whether a bug is caused by the insertion of a method invocation statement or by the insertion of a whole else-if-statement. With the information obtained from CHANGEDISTILLER such a differentiation would be possible: Hatari could inform software developers which change types in which parts of the method body are risky to apply.

The remainder of the paper is organized as follows: In Section 2, we present the original algorithm by Chawathe et al. and describe inadequacies concerning the extraction of source code changes. Section 3 presents string and tree similarity measures and our improved algorithm. We discuss our implementation including the generation of the tree representation in Section 4. In Section 5, the benchmark and our results are described. Section 6 reviews the related work. We conclude the paper in Section 7.

## 2 Change Extraction in Tree-Like Data Structures

Since source code is represented in a tree-like data structure, that is, in an AST, we can use tree differencing algorithms to extract changes between two versions of a Java class. We use basic tree edit operations to describe changes applied to source code.

Our algorithm to extract changes is based on the work by Chawathe et al. in [8]. We discuss the reasons why it is adequate to build upon this algorithm in the related work section (Section 6). In the following, we introduce the terminology and outline their original algorithm, which outputs an edit script of basic tree edit operations transforming an original into a modified tree. Then, we illustrate why the original algorithm is not adequate for source code and discuss how we improved it to handle source code changes.

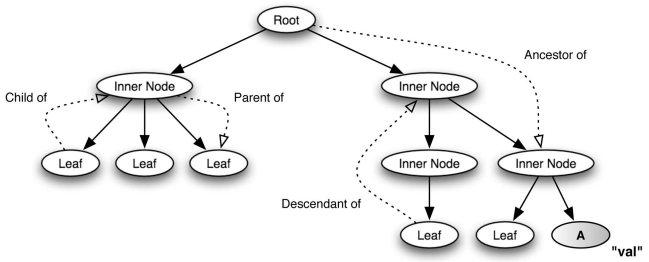


Fig. 1. A generic tree structure. The rightmost leaf shows how we annotate labels and values of nodes.

### 2.1 Terminology

Speaking in terms of graph theory, a tree is a directed acyclic graph consisting of nodes interconnected by edges representing a parent-child relationship. According to the notation used by Chawathe et al., a node $n$ is the *parent node* of a node $m$, $n = p(m)$ if $m$ is a child of node $n$. Nodes along the path to the top of the tree are called *ancestors of* $m$. In return, $m$ is called their *descendant*. The node in a tree that has no parent is called the *root node* or *root*. Nodes that have no children are called *leaf nodes* or *leaves*. Nodes in between are *inner nodes*. Whenever the distinction between *root*, *inner node*, and *leaf* does not add to our discussion, we will talk about *nodes* in general. A node $n$ has a label, $l(n)$, and a value, $v(n)$. In our graphical tree representation, node labels are put inside a node, for example, $A$, and node values left or right beside the node, for example, "*val.*" Fig. 1 illustrates this terminology with an example tree. Leaves in the tree are noncompound statements, for example, *method invocation* or *assignment*. For all nodes, the label is the type of the statement, for example, *MI* for a method invocation or *IF* for an if-statement. The value of an inner node depends on its label, for instance, the condition expression for if-statements: "$a < b$." For leaves, the value is the textual representation of the statement, for example, the method invocation statement "*x.foo(arg);*".

Changes are detected between two trees $T_1$ and $T_2$. In general, $T_1$ denotes the original tree and $T_2$ the modified tree.

### 2.2 Basic Algorithm

Our change detection relies on the algorithm presented in [8]. Their algorithm detects changes in hierarchically structured data represented in tree-like data structures. To extract the changes, the algorithm splits the problem into two tasks:

- Finding a "good" matching between the nodes of the trees $T_1$ and $T_2$.
- Finding a minimum "conforming" edit script that transforms $T_1$ into $T_2$, given the computed matching.

Finding a "good," that is, correct and accurate, matching between the nodes is *crucial* to the outcome of the edit script task. The more nodes that can be matched, the better the minimum conforming edit script.

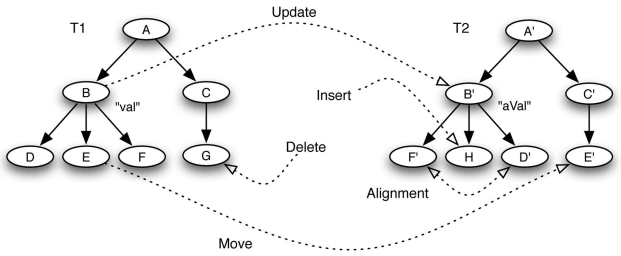


Fig. 2. The five tree edit operations extracted by the edit script generation algorithm by Chawathe et al. Nodes with the same letter are intended to match (example: a matches A′). Node values have been omitted unless they changed from $T_1$ to $T_2$.

We first outline the calculation of the edit script and then describe the matching procedure in detail to highlight the parts to be adapted for detecting changes in the source code.

### 2.2.1 Calculating an Edit Script

The matching set of node pairs is passed to the edit script generation that runs through five phases. Each phase is designed to detect one of the following basic tree edit operations, also illustrated in Fig. 2:

- **Insert.** $\mathrm{INS}((l,v),y,k)$; insert a new leaf node with label $l$ and value $v$ as the $k$th child of node $y$, for example, in Fig. 2, $H$ is inserted as a child of $B'$: $\mathrm{INS}((H,''\,''),B',2)$.
- **Delete.** $\mathrm{DEL}(x)$; delete node $x$ from its parent $p(x)$, for example, in Fig. 2, $G$ is deleted: $\mathrm{DEL}(G)$.
- **Alignment.** $\mathrm{MOV}(x,p(x),k)$; node $x$ becomes the $k$th child of $p(x)$, for example, in Fig. 2, $F'$ becomes the first child of its parent $B'$: $\mathrm{MOV}(D,B',3)$.
- **Move.** $\mathrm{MOV}(x,y,k), p(x) \neq y$; node $x$ becomes the $k$th child of $y$ and is deleted from $p(x)$, for example, in Fig. 2, $E$ is moved from $B$ to $C'$: $\mathrm{MOV}(E,C',1)$.
- **Update.** $\mathrm{UPD}(x,val)$; update $v(x)$ with $val$, that is, $val = v_{new}(x)$ and $v_{old}(x) \neq v_{new}(x)$, for example, in Fig. 2, the value of $B$ is updated: $\mathrm{UPD}(B,''aVal'')$.

### 2.2.2 Matching Procedure

The matching procedure finds an appropriate matching set of pairs of nodes from $T_1$ and $T_2$. Chawathe et al. define two fundamental matching criteria necessary for the algorithm to produce a "good" matching set with which a minimum conforming edit script is achieved.

**Matching Criterion 1 (Leaves):**

$$match_1(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and} \\ & sim(v(x),v(y)) \geq f \\ false & \text{otherwise.} \end{cases}$$

Leaves match if their labels are equal and their values (as strings) are similar according to a given string similarity measure, $sim(x,y)$. The value $f$ is the threshold for the string similarity. Pretesting the labels for equality is important to prevent the matching of different node types.

**Matching Criterion 2 (Inner Nodes):**

$$match_2(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and} \\ & \frac{|common(x,y)|}{max(|x|,|y|)} \geq t \\ false & \text{otherwise,} \end{cases}$$

where $|x|$ denotes the number of leaves contained by $x$.

The inner node matching does not use similarities for the node values. Instead, it uses a measure of how many leaves the subtrees have in common:

$$common(x,y) = \{(w,z) \in M | w \text{ is a leaf of } x, \text{and } z \text{ is a leaf of } y\}, \text{where } M \text{ is the set of matched node pairs.}$$

The number of common leaves is put into proportion to the maximum number of leaves in either subtrees. The value $t$ is the threshold for the inner node similarity. Matching Criterion 2 puts a strong focus on the leaves and is therefore good for LaTeX documents, where leaves (words or sentences of natural language) cover most of the text semantics.

Since the approach presented by Chawathe et al. is used for detecting changes in hierarchically structured documents, they use an assumption to make a *unique maximal* matching:

**Assumption 1.** *For any leaf $x \in T_1$, there is at most one leaf $y \in T_2$ such that $sim(v(x),v(y)) > 0$.*

The assumption that there is at most one leaf in the right tree that can match a corresponding leaf in the left tree (and vice versa) is a necessary precondition for the algorithm to consequently produce an optimal matching and a minimal conforming edit script. Even if the assumption fails, Chawathe et al. apply a postprocessing step to improve the solution. For source code comparisons, Assumption 1 is one of the main reasons why the approach by Chawathe et al. produces suboptimal results. In Section 2.3.3, we discuss the assumption and the postprocessing step, as well as the circumstances under which the postprocessing step is insufficient for our concerns.

## 2.3 When Matching Fails

When applied to source code, the shortcomings of the basic algorithm impact the matching set—in these cases, the matching fails. However, *failing* does not mean that the algorithm yields incorrect results, that is, leading to an edit script that does not transform the original into the modified tree correctly. The edit script is always correct, but, if the matching is inadequate, the solution may not be minimal.

The quality of the *sim*-function and the associated threshold $f$, introduced in the first matching criterion, are crucial for an optimal matching on the leaf-level. When Assumption 1 does not hold, a mismatch on the leaves can be propagated to the inner nodes, leading to a mismatch on higher levels. This can happen whenever a certain number of children of an inner node violate Assumption 1; this is particularly prominent for small subtrees. In the following, we discuss issues concerning leaf-matching based on node values and illustrate mismatch propagation.

```
if (a > b) {                    if (a > b) {
   foo.getHuga();                  foo.getHuga();
   foo.doNothing();                foo.bar();
}                               }

        (a)                             (b)
```

Fig. 3. (a) The original if-statement; (b) The modified if-statement: The method invocation `foo.doNothing();` was replaced by `foo.bar();`.

### 2.3.1 Node Values

Matching leaves is based on two conditions: First, the leaves have to be of the same kind, which we can verify by testing their labels for equality. The second condition applies to the values of the leaves and is evaluated using the function introduced in Matching Criterion 1. In terms of the AST that we use, values correspond to statements (or to the condition in the case of an if-statement) that are strings. Consider the two strings *verticalDrawAction* and *drawVerticalAction*, which can be found, for example, in method invocation statements. From a human's point of view, we intuitively see that they can be considered as an original and a modified version of the same statement, especially when they were found in the same context, that is, in subsequent versions of the same method of a class.

Considering common string similarity measures, context semantics are missing. As we observed in our case studies, common renaming of identifiers during refactoring often involves changing the word order. To allow these strings to match, we have to lower the string similarity threshold, $f$, significantly, possibly resulting in false negatives in other places.

### 2.3.2 Small Subtrees

A mismatch on a single leaf pair does not have a noteworthy impact on the quality of the outcome of the algorithm; we find additional insert and delete-operations instead of update-operations in the edit script. However, these mismatches can be propagated to higher levels of the tree, leading to a complete mismatch of a whole subtree and, therefore, to many unnecessary tree edit operations.

We discuss the propagation of mismatches using small trees as an example: Between the code snippets in Figs. 3a and 3b, a single statement was deleted and a new one was inserted. The surrounding code did not change at all and the threshold $t$ of Matching Criterion 2 is set to 0.6.

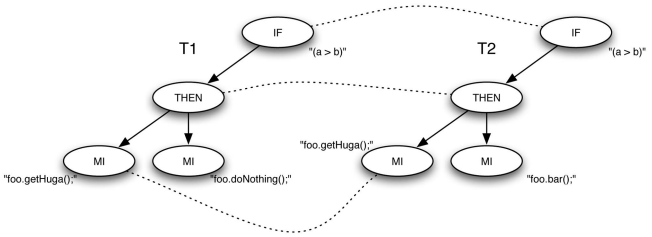


Fig. 4. An example of two similar trees, $T_1$ and $T_2$, for which the algorithm fails to calculate a minimal edit script.

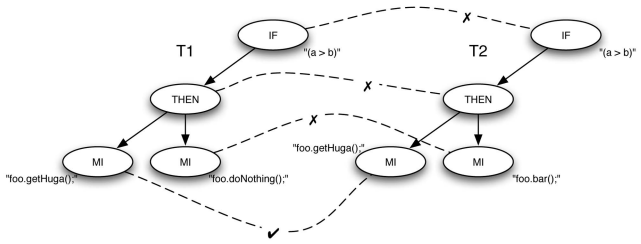


Fig. 5. The whole subtree is considered as mismatched.

Fig. 4 visualizes the same source code using an AST representation. The node with label *IF* denotes an if-statement. Its value corresponds to the if-condition. The node with label *THEN* denotes the then-block. The node with label *MI* denotes method invocation statements that are listed as values.

For the matching, we traverse the trees bottom-up, that is, in a depth-first manner from left to right. The leaves representing the method invocation `foo.getHuga();` in $T_1$ and $T_2$ match according to Matching Criterion 1. They are added to the matching set and marked as *matched*. Although the labels of both right leaves are the same, the values `foo.doNothing();` and `foo.bar();` cannot be matched. We proceed to the next level in the tree and reach the inner node representing the then-block in $T_1$. Inner nodes being matched in accordance with Matching Criterion 2, we count the number of common leaf-descendants of both nodes and divide them by the maximum number of leaves in either trees, leading to the tree similarity of 0.5 and, therefore, to a mismatch of the two then-blocks: $\frac{|common(x,y)|}{max(|x|,|y|)} = \frac{1}{2} = 0.5$.

We proceed to the root of the subtree, the if-statement, which is not matched due to the inner node similarity of 0.5. The final (mis)matchings are shown in Fig. 5.

Although the trees in Fig. 4 show a potential matching set of three node pairs, the algorithm fails—only one node can be matched using the matching criteria and a threshold of 0.6.

### 2.3.3 When Assumption 1 Does Not Hold

Considering source code, similar statements can occur frequently. For instance, statements that print out a particular string on the console are commonly used for debugging. In such cases, there is more than one matching partner for a single node $x \in T_1$, leading to a violation of Assumption 1.

Fig. 6 shows the consequences that a single statement insert (Node 3) can have: There is more than one possible counterpart in the right tree for Node 1, namely, Nodes 2 and 3. Since the tree is traversed in a bottom-up manner, Nodes 1 and 3 are put into the matching set, whereas the better match, that is, the pair of identical Nodes 1 and 2, is not considered to match.

In $T_1$, the root is the only node that remains. Due to the simplicity of our example, we are able to catch mismatching

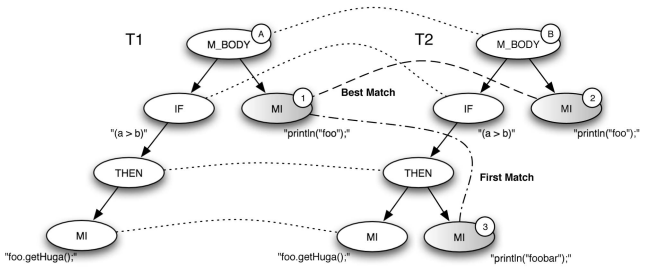


Fig. 6. Suboptimal results are very likely to occur whenever Assumption 1 does not hold.

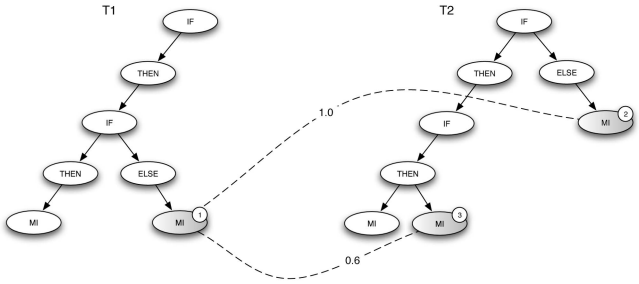


Fig. 7. A trivial example of two trees where the postprocessing step will not be able to improve matching.

propagation on this third level: According to Matching Criterion 2, the roots match because they have two common leaves divided by a maximum of three leaves in $T_2$, leading to a similarity of $\frac{2}{3}$, which lies above threshold $t = 0.6$. Even for our trivial example, the algorithm found nine changes—eight more than we expected. We expected the insert operation $\text{INS}((MI, "println("foobar");"), THEN, 2)$, but the changes found are as follows:

1. $\text{INS}((IF, "(a > b)"), M\_BODY, 1)$,
2. $\text{INS}((MI, "println("foo");"), M\_BODY, 2)$,
3. $\text{INS}((THEN, " "), IF, 1)$,
4. $\text{INS}((MI, "foo.getHuga();"), THEN, 1)$,
5. $\text{MOV}((MI, "println("foo");"), THEN, 2)$,
6. $\text{UPD}((MI, "println("foo");"),$ $"println("foobar");")$.
7. $\text{DEL}((MI, "foo.getHuga();"))$,
8. $\text{DEL}((THEN, " "))$, and
9. $\text{DEL}((IF, "(a > b)"))$

In cases where Assumption 1 does not hold, a postprocessing step is applied. For each matching pair $(x, y)$, where $x \in T_1$ and $y \in T_2$, it is checked whether the matching partner of a child node $c$ of $x$ is a child node of $y$. If not, it is checked whether a child $c'$ of $y$ can be found such that $match(c, c')$ holds. In this case, the old matching pair is replaced by $(c, c')$. For further details, we refer to [8]. In the example above, the postprocessing improves the matching set: For the matching pair (Node A, Node B), we check whether the matching partner of Node 1 is a child node of Node B. This is not the case. Therefore, we search for an unmatched child $c'$ in Node B so that $match_1(\text{Node } 1, c')$ holds. Node 2 is such a $c'$ in Node B. We replace the matching pair (Node 1, Node 3) with (Node 1, Node 2). The expected node is matched, which reduces the previous edit script by the changes 2, 5, and 6, but adds $\text{INS}((MI, "println("foobar");"), THEN, 2)$.

There are a number of tree constellations in which the postprocessing step does not improve the matching. In Fig. 7, we show an example of such a constellation. Node 1 has been moved between $T_1$ and $T_2$ to a new position: It has been moved two levels up and is represented by Node 2 in $T_2$. Postprocessing is not possible under these circumstances; the parent of Node 1 has no partner (corresponding) node in $T_2$.

During our research on source code taken from open source projects such as ArgoUML,[1] we encountered mismatch propagations over two or three levels, for example, in nested if-then-else and try-catch statements. The levels of propagation seem to correlate with the nesting depth of, for example, if-statements or loops and the number of involved statements.

Despite their low frequency, these propagations can have huge implications on the size of the edit script and the classification of the occurred source code changes. In Section 3, we present how we overcome these inadequacies and customize the matching algorithm for detecting source code changes.

In summary, the shortcomings of the original algorithm for extracting source code changes are 1) inadequate matching of node values, 2) using the *first* match instead of finding the *best* match, and 3) the propagation of mismatches in small subtrees. We have addressed these shortcomings and, next, we present a solution to improve the extraction of source code changes.

## 3 Change Distilling Algorithm

We stated that the hierarchical change detection algorithm by Chawathe et al. needs to be adapted to take source code characteristics into account. In addition, we have discussed the circumstances under which the assumptions made for hierarchically structured text documents do not hold to compute a minimal edit script transforming an original AST into a modified AST (see Section 2.3). In this section, we discuss which parts of Chawathe et al.'s matching algorithm need to be customized for source code change extraction. Based on the desired improvements, we describe what measures and techniques overcome the inadequacy of the matching criteria discussed in the previous section.

To meet the requirements of source code change characteristics, we improve the original matching procedure with the following steps:

1. *Customize node value matching*. Since leaf matching is crucial to minimize the edit script, we aim at finding an adequate string similarity measure to match source code statements.

1. http://argouml.tigris.org.

2. *Customize inner node matching*. We aim at finding a tree similarity measure that flexibly matches the inner nodes even if some unintended mismatches occur on the leaf level.
3. *Introduce best match*. Chawathe et al.'s Assumption 1 does not apply to source code because, often, multiple matching candidates for an original node are found. To address multiple matches, we select the leaf pair with the highest similarity.
4. *Use dynamic thresholds for inner node matching*. Propagation of mismatches leads to an enormous amount of unintended deletions and insertions. This is especially prominent for small subtrees—independent of the accuracy of the selected string similarity measure. Thus, we aim at finding a solution for matching small trees more adequately.

We proceed by developing similarity measures to reach the desired improvements. In the following, we discuss existing string and tree similarity measures that are adequate for source code and introduce our change distilling algorithm.

## 3.1 Matching of Leaves

Mismatches at the leaf level have tremendous impact on the size of the edit script. They can lead to mismatch propagation to higher levels in the tree and, consequently, to unnecessary node insert, delete, and move operations. String similarity measures that are robust to detecting common source code changes, as well as techniques to reduce the amount of false first matches, are crucial to overcome mismatch propagation.

We have evaluated string similarity measures provided by SIMPACK, a generic Java library for similarities and ontologies [5]. In this evaluation, two measures were shown to be suitable for source code change extraction.

### 3.1.1 The Levenshtein String Similarity Measure

The Levenshtein Distance [26] denotes the minimum number of operations needed to transform one string into the other. These operations are 1) insert a character, 2) delete a character, or 3) substitute a character. The algorithm is based on the problem of the *longest common subsequence*. A larger distance means that the strings are less similar, that is, that more operations are necessary to transform one string into another, whereas a distance of 0 operations denotes that the strings are equal. The runtime-complexity is $O(n \cdot m)$, where $n$ is the number of characters in $s_a$ and $m$ in $s_b$.

For our concerns, distances are less useful than similarities since we cannot state that a distance of 3 is generally better than a distance of 4. It depends on the lengths of the compared strings. To overcome this situation, we normalize and convert the distance using a distance-to-similarity conversion:

$$sim_{\text{Lev}}(s_a, s_b) = 1.0 - \frac{D(s_a, s_b)}{D_{\text{worstcase}}(s_a, s_b)}.$$

The denominator $D_{\text{worstcase}}$ is equal to the maximum costs experienced under the assumption that the longest common subsequence of $s_a$ and $s_b$ has a length of 0, that is, that they have no characters in common: $D_{\text{worstcase}} = \max(m, n)$.

The Levenshtein Distance is susceptible to changes in word or character order. Consider the strings $s_1 = verticalDrawAction$ and $s_2 = drawVerticalAction$. If they are found at the same position in two versions of a source code entity, then it is very likely that someone has performed a refactoring, for example, by unifying identifier nomenclature. The Levenshtein Distance does not recognize this similarity, as our example illustrates: The longest common subsequence is "*verticalAction*." The remaining characters cause four insertions and four deletions, that is, a total of eight change operations and a distance of 8, respectively, leading to a string similarity of $sim_{\text{Lev}}(s_1, s_2) = 1 - \frac{8}{18} \approx 0.56$.

The Levenshtein Distance is inadequate in this case. Since we noticed during prototyping that a lot of unintentional mismatches on the leaf level were actually based on the deficiencies of the string similarity measure, we were eager to find an algorithm showing more robustness.

### 3.1.2 String Similarity Measures Using $n$-Grams

A family of string similarity measures is based on the *Dice Coefficient* [11]—a modification of the *Jaccard Coefficient* [19]. Adamson and Boreham used the Dice Coefficient to rate the similarity of strings by setting their $n$-grams into relation [1]. $n$-grams are bags and constructed by putting a sliding window of length $n$ over a string and extracting at each position the $n$ underlying characters. For instance, the trigrams of the string "vertical" are

$$3\text{-}grams(\text{vertical}) = \{''\text{ver}'', ''\text{ert}'', ''\text{rti}'', ''\text{tic}'', ''\text{ica}'', ''\text{cal}''\}.$$

The $n$-gram similarity measure defined by Adamson and Boreham is the ratio of twice the number of shared $n$-grams and the total numbers of $n$-grams in two strings:

$$sim_{\text{ng}}(s_a, s_b) = \frac{2 \times |n\text{-grams}(s_a) \cap n\text{-grams}(s_b)|}{|n\text{-grams}(s_a) \cup n\text{-grams}(s_b)|}.$$

The Dice Coefficient with bi and trigrams is a popular word similarity measure. In combination with source code, bigrams have been used by Xing and Stroulia for their UMLDiff approach [42] and trigrams by Weidl and Gall for their CORET approach [38].

To illustrate the applicability of the $n$-gram similarity measure for source code change detection, we calculate the similarities for strings on which the Levensthein measure fails. As before, the strings to use are $s_1 = verticalDrawAction$ and $s_2 = drawVerticalAction$. The similarities for bi, tri, and four-grams are

$$sim_{2g}(s_1, s_2) = \frac{2 \times 14}{34} \approx 0.82,$$
$$sim_{3g}(s_1, s_2) = \frac{2 \times 12}{32} \approx 0.75,$$

and $sim_{4g}(s_1, s_2) = \frac{2 \times 10}{30} \approx 0.67$. Using a hash-table to store the $n$-grams of both strings, the runtime complexity of the

$n$-gram similarity measure is in $O(n+m)$—one order of magnitude faster than Levenshtein.

The $n$-gram similarity measure is more robust to changes to the word order since it does not rely on the longest common subsequence. It primarily focuses on common characters and secondarily on word order. Regarding source code in general and source code identifiers in particular, the measure allows a more intuitive similarity scoring. During our experiments, the measure performed worse than Levenshtein only under rare circumstances (rare in conjunction with source code): It seems to be more susceptible to substitutions, including misspellings due to phonetic reasons that are common in natural language but not so in source code. The strings *Levenshtein* and *Levnshtain*, for example, score with a similarity $\sim 0.72$ when Levenshtein is used, but with only 0.5 when bigrams are used. Furthermore, the measure is limited to strings of a certain maximum length since the given number of different characters is finite. As a string gets longer, it will become more likely that most permutations between characters are covered. The number of character pairs in the intersection will therefore increase, leading to an imprecise similarity. However, we were not yet able to prove this expectation experimentally, but, instead, we were able to confirm the effectiveness of the $n$-gram similarity measure to the source code on the statement-level in our evaluation (see Section 5).

### 3.2 Similarity Rating for Best Match

As we have discussed in Section 2.3.3, Assumption 1 does not hold for source code represented in an AST. The postprocessing step proposed by Chawathe et al. does not succeed either. Consequently, a *first* match cannot become a *best* match using Assumption 1 and the postprocessing step.

In general, a first match that is not the best match is formalized as follows:

Let $x$ be a leaf in $T_1$ and $y$ be its matching partner in $T_2$. Furthermore, let $z$ be another leaf in $T_2$ and $f$ be the threshold, so that

$$sim(v(x), v(y)) \geq f \text{ and } sim(v(x), v(z)) \geq f \text{ but } sim(v(x), v(y)) > sim(v(x), v(z)).$$

Whenever $z$ will be visited before $y$ during postorder traversal, a suboptimal matching will be calculated.

Accordingly, we can derive a solution for that: Let $x$ be a leaf in $T_1$. Furthermore, let $mp_i$ be its $i$th possible matching partner in $T_2$, such that $i \in N$ and

$$sim(v(x), v(mp_i)) \geq f.$$

We mark $(x, mp_i)$ as *best match* until we find another possible partner $mp_{i+\epsilon}$ such that $\epsilon \in N$ and

$$sim(v(x), v(mp_{i+\epsilon})) > sim(v(x), v(mp_i)).$$

In this case, we mark $(x, p_{i+\epsilon})$ as a *best match*. We repeat until we have tried to match all possible partners in $T_2$ to $x$.

The solution involves finding the matching partner $y \in T_2$ that matches $x \in T_1$ best. There are combinations of statements so that $x$ in $T_1$ has more than one possible partner, for example, when one and the same statement can be found over and over again in a block of code (for example, print outs for debugging). In this case, we apply the heuristics that unchanged statements stay in situ between subsequent versions of a source code entity: The first "best" match, that is, the matching pair with the highest similarity score that has been visited during postorder traversal first, will make it into the final matching set.

So far, we have developed an approach for finding the best partner $y \in T_2$ for leaf $x \in T_1$. However, this relationship is not always a two-way optimum, that is, $x$ is not always the best partner for $y$. We can overcome this by calculating the similarity of each leaf pair $(x_i, y_j) \in T_1 \times T_2$ and add those pairs to the final matching set that show highest similarity.

### 3.3 Matching of Inner Nodes

Leaf matching propagates to inner nodes as similarity on inner nodes is calculated by the number of matching leaves. A measure for inner nodes that takes leaf matching into account and is robust to potential mismatches or small subtrees is important for a maximal matching set. Chawathe et al. presented a simple but adequate tree similarity measure for inner nodes (Matching Criterion 2). In this section, we discuss the suitability of this measure and other measures in terms of source code characteristics and small subtrees.

#### 3.3.1 Tree Similarity Used by Chawathe et al.

The tree similarity measure used by Chawathe et al. (Matching Criterion 2) takes only descending leaves into account when deciding whether two nodes should match. Inner node descendants are ignored completely. This is an adequate approach for similarity analysis of structured text documents such as those that are written in LaTeX, where the inner nodes are used for structuring means and do not hold any semantics. For source code, inner nodes are more important since some of them cover fundamental constructs, such as iterations and alternatives or exception handling.

Since, for instance, an else-block may contain an if-statement, matching between descendants can occur. During our studies, it happened that an else-block matched with a descendant else-block. This matching resulted in a nonapplicable move operation since a parent node cannot become a child node of one of its descendants. To overcome such situations, we added the check that the string similarity of the value of inner nodes must also satisfy the threshold $t$. Whenever a node does not have its own value, it inherits that of its parent to emphasize their affiliation.

#### 3.3.2 Dice Coefficient for Inner Nodes

By using the Dice Coefficient, we get a measure taking inner nodes into account. In conjunction with code clone detection, Baxter et al. used the Dice Coefficient to calculate

```
if (cancelled()) {
   close();

}
```
(a)

```
if (cancelled()) {
   close();
   logger.debug("user has cancelled action");
}
```
(b)

Fig. 8. (a) A small if-statement. (b) A logging statement has been added.

the similarity of two ASTs [4]. For our purpose, we apply the same measure to inner nodes:

$$sim_{\text{Dice}}(T_a, T_b) = \frac{2 \times |nodes(T_a) \cap nodes(T_b)|}{|nodes(T_a) \cup nodes(T_b)|},$$

where $nodes(T_x)$ denotes all nodes of $T_x$, including the root.

Taking inner nodes of the subtrees into account does not impact the value of the similarity measure because the matching of leaves propagates to inner nodes. A more important aspect of the Dice Coefficient is that common nodes of $T_a$ and $T_b$ are weighted more than mismatches. When two trees share most of their nodes, but $T_b$ differs in structure from $T_a$ by a few changes, the Dice Coefficient is more robust than the measure used by Chawathe et al. Overall, our evaluation showed that the algorithm by Chawathe et al. including inner node similarity weighting and dynamic threshold (see the next sections) performs better than the Dice Coefficient.

#### 3.3.3 Inner Node Similarity Weighting

According to our adapted Matching Criterion 2, the similarity of inner node values and the similarity of their subtrees influence the similarity for inner nodes likewise. Therefore, two inner nodes do not match either because their node values mismatch or they have too few leaves in common. Regarding if-statements or loops, a value, that is, condition expression, mismatch may cause a tremendous amount of unnecessary changes. We overcome this situation by weighting the common leaves function more than the similarity of values between inner nodes.

#### 3.3.4 Inhibiting Propagation of Mismatches in Small Subtrees

The similarity measures for strings and for trees introduced in the previous sections reduce mismatching of single nodes but do not reduce them for small subtrees. Consider the code snippets in Fig. 8. According to Matching Criterion 2, the similarity between the two then-blocks of the if-statements is 0.5 (one shared node, two leaves), causing a mismatch of the then-blocks and the if-statements.

To weaken the high impact that small changes can have on small subtrees, we dynamically lower thresholds for small subtrees; dynamically, meaning in regard to the size of the subtrees under investigation. We experienced adequate matching results for $t = 0.6$ if $n > 4$ and $t = 0.4$ if $n \leq 4$, where $n$ is the number of leaf descendants of the inner node.

Lowering thresholds for all inner nodes, no matter how many leaf descendants they count, injects undesired behavior into the algorithm: The amount of similar inner nodes increases by lowering the threshold leading to false matches.

### 3.4 Our Matching Algorithm Used for Change Distilling

In this section, we present our improved tree-differencing algorithm suitable to extract changes in source code. To recall, our improvements are

1. using bigrams as a robust string similarity measure that is able to cover common changes of source code identifiers,
2. adding a similarity check of node values to Chawathe et al.'s tree similarity measure to solve the problem of descendant subtree matching,
3. using inner node similarity weighting to reduce inadequate mismatches of condition expressions,
4. introducing the best match algorithm to reduce the impact of Chawathe et al.'s Assumption 1, and
5. using dynamic thresholds to reduce the propagation of mismatches in small subtrees.

We evaluated combinations of the discussed string and tree similarity measures as well as best match, dynamic threshold, and inner node similarity weighting with our benchmark (see Section 5 for a detailed discussion). The following combination of measures and techniques performed best for extracting source code changes:

- For Matching Criterion 1 (Leaves), we use the bigram string similarity measure:

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

  where $f = 0.6$.
- In addition to Matching Criterion 1, we take the *best* match for a leaf $x$ instead of the *first* match.
- Matching Criterion 2 (Inner nodes) is extended by the check as to whether the values of the inner nodes are similar:

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

  where $f = 0.6$ and $t = 0.6$.
- We add the inner node similarity weighting: If the string similarity of inner node values, for example, the condition of an if-statement, is less than the threshold $f$, but $\frac{|common(x,y)|}{max(|x|,|y|)} \geq 0.8$ holds, $match_2(x, y)$ is true.
- The threshold for the inner node similarity measure is adjusted dynamically for small subtrees: $n \leq 4 \rightarrow t = 0.4$.

1: **Input**: trees $T_1$, $T_2$
2: **Result**: final matching set: $M_{\text{final}}$
3: $M_{\text{final}} \leftarrow \phi$, $M_{\text{tmp}} \leftarrow \phi$
4: Mark all nodes in $T_1$ and $T_2$ "unmatched"
5: **for all** leaf $x \in T_1$ and leaf $y \in T_2$ **do**
6: **if** $match_1(x, y)$ **then**
7: $M_{\text{tmp}} \leftarrow M_{\text{tmp}} \cup (x, y, sim_{2g}(v(x), v(y)))$
8: **end if**
9: **end for**
10: Sort $M_{\text{tmp}}$ into descending order, according to the leaf-pair-similarity
11: **for all** leaf-pair-similarity $(x, y, sim_{2g}(v(x), v(y))) \in M_{\text{tmp}}$ **do**
12: $M_{\text{final}} \leftarrow M_{\text{final}} \cup (x, y)$
13: Remove all leaf-pairs from $M_{\text{tmp}}$ that contain $x$ or $y$
14: Mark $x$ and $y$ "matched"
15: **end for**
16: Proceed post-order on trees $T_1$ and $T_2$:
17: **for all** unmatched node $x \in T_1$, if there is an unmatched node $y \in T_2$ **do**
18: **if** $match_2(x, y)$ (incl. dynamic threshold and inner node similarity weighting) **then**
19: $M_{\text{final}} \leftarrow M_{\text{final}} \cup (x, y)$
20: Mark $x$ and $y$ "matched"
21: **end if**
22: **end for**

Fig. 9. Our matching algorithm used for change distilling.

The final algorithm is presented in Fig. 9. The inputs to the algorithm are two labeled and valued trees, $T_1$ and $T_2$. The algorithm first calculates a complete matching of all leaves (Lines 5-9). The leaf pairs are sorted (Line 10) according to their similarity and the best matches are added to the final matching set (Lines 11-15). At the end, the inner nodes are matched using dynamic thresholds (Lines 17-22). The output of the algorithm is a set of matching node pairs that is used by the edit script algorithm to compute the tree edit operations.

The runtime analysis of the matching algorithm by Chawathe et al. has to be extended by the additional computation steps. Assume $n = \max(|T_1|, |T_2|)$, where $|T|$ is the number of leaves. The cost to compare two leaves is denoted by $c$. The matching of all leaves is in $O(n^2c)$, that is, $O(n^2)$, since we have to compare each possible leaf pair. Sorting the generated $O(n^2)$ matching pairs is in $O(n^2 \log n^2)$. For each pair that is added to $M_{\text{final}}$, the whole $M_{\text{tmp}}$ has to be traversed at most once to remove all corresponding leaf pairs. Thus, building $M_{\text{final}}$ for the leaves is proportional to $n^2(1 + c + \log n^2)$. The runtime complexity of inner node matching can be derived from the original work by Chawathe et al.: The number of inner nodes in $T_1$ and $T_2$ is denoted by $m$. Matching Criterion 2 can be computed for all inner nodes in $O(mn)$ (we refer to [8] for more details). In addition, the value comparison of the inner nodes is in $O(mc)$. The overall runtime of inner node matching is $O(m(c + n))$. In summary, the total time of the matching algorithm is proportional to

$$n^2(c + 1 + \log n^2) + m(c + n).$$

Compared to the original algorithm by Chawathe et al., our runtime is $O(\log n^2)$ slower. We describe in Section 4 how we mitigate the impact of this additional factor to optimize the runtime performance of our change distilling algorithm.

## 4 Implementation

We built the Eclipse plugin CHANGEDISTILLER that implements our change distilling algorithm. Our current implementation relies on the CVS capabilities, Java Development Tools (JDT),[2] and compare functionality of Eclipse. The extracted source code changes are stored in a Hibernate[3] mapped database.

We have automated the process of change distilling within Eclipse. Starting with an Eclipse project, CHANGE DISTILLER is able to extract changes from the version chain of a single class, packages, or a whole project.

### 4.1 Fine-Grained Change Extraction Process

Fig. 10 depicts the change extraction process of CHANGE DISTILLER. From a project under CVS control, revisions of Java classes are checked out using the CVS capabilities of Eclipse. For two subsequent revisions of a Java class, we use the compare plug-in to extract the methods and attributes that have changed. This prefiltering step leads to smaller trees for comparison. Assume a class has about 1,000 lines of code, but only a single method with 20 lines of code has changed. Using the compare plug-in[4] reduces the input to our change distilling algorithm significantly. Recalling the runtime complexity of the matching algorithm, this is a considerable performance gain as the input trees are kept small.

For both versions of a changed method or attribute, intermediate ASTs are created using the AST visitor from JDT. Creating intermediate trees is necessary since the matching algorithm expects labeled and valued nodes as well as a uniquely defined parent-child relationship between hierarchically situated nodes. This expectation is not covered by ASTs created by JDT. For instance, an if-statement may have two children—a then and an else-block. Depending on the AST implementation, the access from the if-statement (parent) to the two blocks (children) is not available through "getChildren" but through "getThen Block" and "getElseBlock." Leaves in the intermediate AST are normal statements with the statement kind as label and the statement itself as value. For instance, the leaf of statement `foo.bar();` has the label *MI* and the value "*foo.bar();*".

The intermediate ASTs $T_1$ and $T_2$ are then fed into our change distilling algorithm. The algorithm can be configured with different string and tree similarity algorithms and thresholds, as described in Section 3. The output is a set of basic tree edit operations that are classified to change types and stored into the Hibernate mapped database.

2. http://www.eclipse.org/jdt.
3. http://www.hibernate.org.
4. The complexity of the compare plug-in is in $O(n^2)$, where $n$ is the number of members of a Java class.

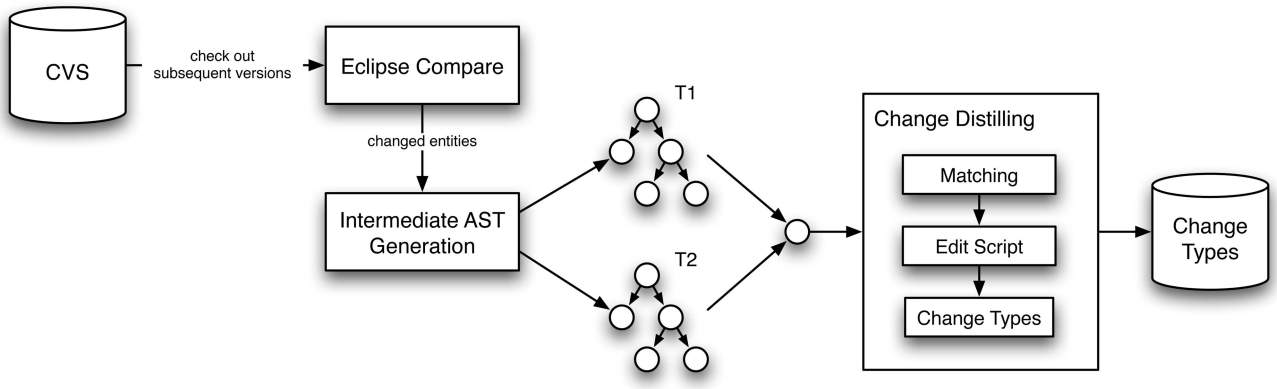


Fig. 10. Fine-grained change extraction process.

### 4.2 Classifying Tree Edit Operations

In [12], we have assigned basic tree edit operations to change types. For instance, the tree edit operation for *Statement Ordering Change* is $\mathrm{MOV}(s, p(s), k)$, meaning that the statement $s$ is moved to position $k$ in the children of its parent $p(s)$.

Sometimes, we can infer that an update took place even if the similarity between the two strings under comparison is too low. Consider the methods `foo(Object myParam)` in revision $n-1$ and `foo(Figure myParam)` in revision $n$. A parameter type change from "Object" to "Figure" happened, but the similarity of the two strings "Object" and "Figure" is below the threshold $f = 0.6$ and, hence, is not matched. Therefore, by classifying the tree edit operation without further check, a new parameter would be inserted and an old one would be deleted. Since the parameter name did not change, the classifier is able to classify the two operations as a *Parameter Type Change* by checking whether the parents of "Object" and "Figure" are equal.

## 5 Evaluation

In Section 3.4, we have described our change distilling algorithm. To investigate the quality of our improvements, we developed an extensive benchmark. The benchmark consists of a set of special test cases and of a large data set of manually classified changes. The data set is taken from three different open source case studies: ArgoUML,[5] Azureus,[6] and JDT.[7] With the benchmark, we show that our improvements approximate the minimum conforming edit script more closely than Chawathe et al.'s change detection algorithm. Although the ChangeDistiller is able to detect changes on the class level as well, our benchmark focuses on changes on the method level. Since our major interest lies in the tree-differencing part of our algorithm, changes on the method level are sufficient—they cover all tree structures that may occur in an AST.

5. http://www.argouml.org.
6. http://azureus.sourceforge.net.
7. http://www.eclipse.org/jdt.

### 5.1 Preliminaries

The final step of ChangeDistiller is to analyze, consolidate, and classify the tree edit operations into *change types* [12]. Change types are the most suitable data set for benchmarking our change distilling algorithm because they are an adequate measure for the quality of our algorithm and straightforward to implement and validate manually. Change types represent the kind of changes that a human will intuitively find when she compares two subsequent versions of a Java method. For example, she will recognize that a *method invocation* has been inserted into a method rather than thinking of the corresponding tree edit operation.

Taking two versions $(n-1, n)$ of a Java method, we count the occurrences of each particular change type manually. We, then, run the ChangeDistiller on the same pair of versions. For each version pair $(n-1, n)$ and each change type $t$, we calculate the mean absolute error $\epsilon_t$ and the mean absolute percentage error $\delta_t$:

$$\epsilon_t = \frac{1}{k}\sum_{i=1}^{k} |x_i(t) - \tilde{x}_i(t)|, \quad \delta_t = \frac{1}{k}\sum_{i=1}^{k} \left|\frac{x_i(t) - \tilde{x}_i(t)}{x_i(t)}\right|,$$

where $x_i(t)$ is the expected number of occurrences of change type $t$, $\tilde{x}_i(t)$ is the found number of occurrences of change type $t$, and $k$ is the number of version pairs in which $t$ was expected or found. The smaller the difference between the number of change types classified manually and found by ChangeDistiller, the smaller the error and the better we consider the performance of our algorithm.

For each version pair $(n-1, n)$, we calculate the mean absolute error $\epsilon$ and the mean absolute percentage error $\delta$ for the edit script:

$$\epsilon = \frac{1}{m}\sum_{i=1}^{m} |x_i - \tilde{x}_i|, \quad \delta = \frac{1}{m}\sum_{i=1}^{m} \left|\frac{x_i - \tilde{x}_i}{x_i}\right|,$$

where $x_i$ is the expected length of the edit script, $\tilde{x}_i$ is the found length of the edit script, and $m$ is the number of version pairs.

Before applying these measures to our change distilling algorithm, we have to discuss one shortcoming in terms of counting change types for the benchmark: We cannot

evaluate exactly where the change occurred since we do not store its exact location in the benchmark, but rather in which version and method it was found. This means that we can tell that, for example, two *statement inserts* were found in the method `foo()` between Versions 1.11 and 1.12, but not whether the statements were, for example, inserted into a particular then-block or somewhere else. Performing a manual qualitative analysis on the whole data set instead of restricting ourselves to a quantitative evaluation is barely feasible; we would have to determine the exact location in the AST for each change by hand to compare it to the output of our algorithm. For a sufficiently large set of changes, this is too time consuming and error prone.

To show that counting the occurrence of change types is sufficient nonetheless, we performed a qualitative evaluation on a randomly selected sample of the data in our benchmark. For this, we have calculated precision and recall as follows:

$$Precision = \frac{\#\text{ relevant changes found}}{\#\text{ changes found}},$$
$$Recall = \frac{\#\text{ relevant changes found}}{\#\text{ changes expected}}.$$

The selected sample contains 13 pairs of Java method versions comprised of 120 expected changes. We compared each of the 151 changes found by CHANGEDISTILLER with the expected changes manually and obtained a precision of $\frac{118}{151} = 0.78$ and a recall of $\frac{118}{120} = 0.98$. Furthermore, we observed that the found edit scripts always transform the old into the new version of the Java methods correctly. Consequently, a recall $< 1.0$ denotes that our algorithm found changes that replace the ones that we expected. For instance, the method invocation `mParameter.setKind(MParameterDirectionKind.IN)`[8] was updated with `ModelFacade.setKindToIn(mParameter)`, but CHANGEDISTILLER found a corresponding *statement delete* and *statement insert* instead. A precision $< 1.0$ denotes that our algorithm found a nonminimal conforming edit script with *virtual* changes, that is, pairs of changes in the same edit script, of which the second reverts the first one and vice versa. Consider the example of source code in Fig. 11, taken from our benchmark. For this, we manually classified four statement inserts (one if-statement insert and three method invocations). For this particular case, our change distilling algorithm extracts five statement inserts, one statement delete, and two statement parent changes, leading to an absolute error $\epsilon$ of 4 and a percentage error $\delta$ of 50 percent of the length of the edit script. Since the topmost if-statements (Line 1) share only two out of five leaves (Lines 2 and 3 in (a) with Lines 2 and 8 in (b)), Matching Criterion 2 is not satisfied, that is, they do not match. Therefore, the edit script contains the insert and delete operations of the topmost if-statement and move operations of the first and the last statement from the deleted to the reinserted if-statement. Applying these four changes does not transform the source code, but leads to a nonminimal conforming edit script.

8. This is in method `addOperation(...)` in the class `org.argouml.uml.reveng.java.Modeller` between Revision 1.45 and 1.46.

```
1  if (matches.length == 0) {
2     fElements =
3        growAndAddToArray(fElements, type);
4     return;
5  }
```

(a)

```
1  if (matches.length == 0) {
2     fElements =
3        growAndAddToArray(fElements, type);
4     if(SelectionEngine.DEBUG){
5        System.out.print(
6           "SELECTION - accept type("
7        );
8        System.out.print(type.toString());
9        System.out.println(")");
10    }
11    return;
12 }
```

(b)

Fig. 11. (a) The original if-statement. (b) The modified if-statement of method `acceptSourceMethod(..)` of class `jdt.internal.core.SelectionRequestor`.

Regarding the high recall, we claim that our algorithm at least finds the changes we expect. However, in certain cases, it finds a conforming edit script that is not minimal. If it finds fewer than expected changes, such as statement updates, a set of corresponding changes is found instead (for example, in case of statement update: statement insert and delete).

With our benchmark, we show that the output of our change distilling algorithm approximates the minimum conforming edit script more closely than Chawathe et al.'s algorithm. Therefore, we only benchmark with the error measures.

## 5.2 Our Benchmark for Change Distilling

For the benchmark, we use a combination of dedicated test cases and data from three different case studies. We discuss how we have chosen the data and what preparation steps they have undergone.

### 5.2.1 Test Cases

The test cases serve as a validation for our improvements. We focused on testing string similarity measures, matching of small subtrees, and special issues on ordering changes. Test cases that failed with the original algorithm had to pass with the customized algorithm. For that, we have hard coded exact tree edit operations and their classification between two source code version of one class. For an in-depth discussion of these test cases, we refer to [40].

#### 5.2.2 Collecting Changes from Existing Software

Special test cases are well suited to investigating specific or theoretical issues. They are insufficient for claiming whether an approach applies to real-world problems or not. Therefore, we decided to integrate data from the open source projects ArgoUML ($\sim 1,500$ classes, $\sim 272$ kLOC), Azureus ($\sim 2,300$ classes, $\sim 432$ kLOC), and JDT of Eclipse ($\sim 1,100$ classes, $\sim 388$ kLOC). Choosing representative test data among the approximately 4,900 classes was a challenge. We fed the projects into CHANGEDISTILLER with the original change extraction configuration and applied the following criteria to find appropriate Java classes:

- A lot of changes over time, few changes between revisions. We preferred classes that have 100 to 200 revisions and contain methods that show 10 to 20 changes per revision.
- Method size. We have chosen methods with 50 to 500 lines of code.
- Nesting. Methods that have nested if and loop-statements are most interesting in terms of the small-subtree-problem.
- Diversity of changes. We preferred classes with different change types since we want to benchmark our algorithm in a broad variety of source code structures.

According to the above criteria, we located eight candidate methods in total—each one in a different class—that we integrated into our benchmark. We performed a checkout of every revision in which the selected methods experienced changes. Preparation of the classes was done by deleting all fields and methods except the chosen ones. During manual inspection, we finally classified 1,064 changes in a total of 219 revisions. To reduce evaluator bias, two of the authors of this paper classified the changes independently and consolidated their findings.

### 5.3 Results and Discussion

In Section 3, we claimed that our algorithm is better suited to source code changes than the original algorithm by Chawathe et al. In this section, we present and discuss selected comparisons between different configurations of our change distilling algorithm, that is, we show how the different configurations perform against each other. We benchmark different combinations of the following:

- The original *first* match algorithm for leaves or our *best* match algorithm.
- Either the tree similarity measure suggested by Chawathe et al. or the Dice Coefficient is used for inner node comparisons.
- We dynamically lower the threshold $t$ for inner nodes to 0.4 whenever the left and the right tree roots have four or fewer descendants.
- We either turn on or off inner node similarity weighting.
- We use either the Levenshtein or the $n$-grams similarity measures to match node values.

For the string similarity measures, we use $f$ as the threshold variable and $t$ as the inner node similarity threshold.

#### 5.3.1 Benchmarking

We have conducted four runs with different configurations:

(a) Chawathe et al.'s original algorithm, Levenshtein as string similarity measure, $f = 0.7$ and $t = 0.6$, dynamic thresholds as well as inner node similarity weighting disabled.
(b) Chawathe et al.'s original algorithm, bigrams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds as well as node similarity weighting disabled.
(c) Our best match, bigrams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds as well as inner node similarity weighting disabled.
(d) Our best match, bigrams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds as well as inner node similarity weighting enabled.

The minimum conforming edit script is comprised of 1,064 changes and the smaller the mean absolute error $\epsilon$ and the mean absolute percentage error $\delta$ are, the better the performance of the algorithm. Table 1 depicts the results from Runs (a), (b), (c), and (d) in the respective columns. Additionally, we provide more detailed tables for each run, including root mean squared absolute error and root mean squared percentage error in the Appendix.

*Run (a).* In the first run, we found fewer *statement updates* and *condition expression changes* than expected with a mean absolute error $\epsilon$ of 0.96 and 1.02 between each pair of versions. In other words, the algorithm has missed, on average, approximately one statement update and condition expression change per pair of versions. As indicated by the $\epsilon$ values of statement inserts and deletes, the missed statement update and condition expression change are replaced by a pair of statement inserts and deletes. The accuracy of finding *statement updates* depends on the accuracy of the string similarity measure. The fewer *statement updates*, the more *statement inserts* and *deletes* are found. Besides the string similarity measure, the accuracy of finding *condition expression changes* relies on the matching of inner nodes. Two if-statements match if their conditions (that is, values) match and if the inner node similarity satisfies the threshold $t$. Thus, matching small trees has an impact on condition expression changes. A mismatch leads to deletes of if-statements and alternative parts with additional insert and ordering/parent changes. On the other hand, when their conditions do not match but their subtrees do, a mismatch occurs as well. The original algorithm is not able to match nodes accurately, leading to a mean absolute percentage error $\delta$ of 0.79 with additional 3.27 changes per version pair, as depicted in Column (a).

TABLE 1
Benchmark Results of the Four Runs (a)-(d) Including the Runtime Performance in Seconds, $\epsilon$ and $\delta$ per Change Type and Edit Script for Each Configuration

| | | (a) | | | (b) | | | (c) | | | (d) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Change Type | $x$ | $\tilde{x}$ | $\epsilon$ | $\delta$ | $\tilde{x}$ | $\epsilon$ | $\delta$ | $\tilde{x}$ | $\epsilon$ | $\delta$ | $\tilde{x}$ | $\epsilon$ | $\delta$ |
| Alternative Part Del. | 9 | 32 | 1.04 | 0.08 | 28 | 1.17 | 0.11 | 25 | 1.06 | 0.12 | 14 | 0.78 | 0.22 |
| Alternative Part Ins. | 15 | 40 | 0.86 | 0.06 | 36 | 0.88 | 0.06 | 33 | 0.78 | 0.07 | 22 | 0.41 | 0 |
| Cond. Exp. Change | 91 | 51 | 1.02 | 0.58 | 64 | 0.89 | 0.44 | 58 | 0.92 | 0.47 | 85 | 0.58 | 0.24 |
| Method Renaming | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| Param. Delete | 9 | 12 | 0.43 | 0.07 | 12 | 0.43 | 0.07 | 11 | 0.33 | 0.08 | 11 | 0.33 | 0.08 |
| Param. Insert | 16 | 20 | 0.29 | 0.04 | 20 | 0.29 | 0.04 | 19 | 0.23 | 0.04 | 19 | 0.23 | 0.04 |
| Param. Ord. Change | 0 | 19 | 2.71 | 0 | 19 | 2.71 | 0 | 19 | 2.71 | 0 | 17 | 3.4 | 0 |
| Param. Renaming | 3 | 1 | 0.67 | 0.67 | 2 | 0.75 | 0.5 | 1 | 0.67 | 0.67 | 1 | 0.67 | 0.67 |
| Param. Type Change | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| Return Type Change | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| Return Type Insert | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| Stmt. Delete | 144 | 371 | 2.28 | 0.3 | 283 | 1.96 | 0.29 | 264 | 1.84 | 0.32 | 225 | 1.77 | 0.34 |
| Stmt. Insert | 391 | 640 | 2.15 | 0.4 | 552 | 1.89 | 0.42 | 533 | 1.76 | 0.38 | 494 | 1.54 | 0.32 |
| Stmt. Ord. Change | 14 | 105 | 2.26 | 0.12 | 82 | 2.23 | 0.19 | 83 | 2.08 | 0.11 | 73 | 2.23 | 0.08 |
| Stmt. Parent Change | 86 | 185 | 1.84 | 0.2 | 194 | 1.87 | 0.21 | 162 | 1.56 | 0.11 | 118 | 1.14 | 0.21 |
| Stmt. Update | 282 | 216 | 0.96 | 0.34 | 318 | 0.72 | 0.19 | 259 | 0.41 | 0.14 | 260 | 0.41 | 0.14 |
| Runtime | | $\sim$ 12 s | | | $\sim$ 5 s | | | $\sim$ 9 s | | | $\sim$ 9 s | | |
| Total | 1064 | 1696 | 3.27 | 0.79 | 1613 | 2.91 | 0.72 | 1471 | 2.2 | 0.52 | 1343 | 1.64 | 0.34 |

*Run (b).* While evaluating the results of the initial run, we found that the outcome mainly relies on the string similarity measure and on the chosen threshold. We therefore lowered the threshold to $f = 0.6$ and used bigrams as the string similarity measure instead of Levenshtein. Column (b) in Table 1 illustrates that the number of statement updates increased tremendously compared to the number of condition expression changes—it even exceeded the expected number of statement updates. The reason for this increase is the flexibility of the bigram similarity measure, leading to statement updates instead of inserts and deletes. Configuration (b) reduced the overall $\epsilon$ from 3.27 to 2.91. This decreased the $\delta$ by 7 percent down to 72 percent.

*Run (c).* To further improve the result, in particular to reduce the number of statement updates, we used our best match algorithm with bigrams. Column (c) in Table 1 shows the corresponding results. Using the best match algorithm reduces the number of statement updates and increases the condition expression changes. The advantage of the best match is that it is less likely that correct statement inserts/deletes are replaced by updates because better matches are taken for the matching set. Using the best match improved the output of the algorithm significantly. We achieved a $\delta$ of 52 percent; thus, we further reduced the $\epsilon$ by 0.71 to 2.2.

*Run (d).* The results of the last and most influencing improvement, that is, our matching algorithm, are shown in Column (d) in Table 1. In particular, the *inner node similarity weighting* and *dynamic threshold* increased the number of condition expression changes. The number of statement inserts, deletes, and ordering changes as well as the alternative part inserts and deletes were reduced. The reason for the decrease in those changes was that more if-statements matched and, therefore, fewer statements were moved to a new if-statement.

Using the dynamic thresholds, we are able to get rid of the mismatch propagation in small subtrees. This led to an improvement in the overall $\delta$ by 8 percent. Enabling the weighting of the inner node similarity derived a further decrease of the $\delta$ by 10 percent.

Concerning the runtime, we observed a decrease between Runs (a) and (b) as well as an increase between (b) and (c) or (d). The Levenshtein similarity measure used in Run (a) is an order of magnitude slower than the bigram similarity measure used in Run (b). The best match algorithm used in Runs (c) and (d) is slower than the first match used in Run (a).

Our change distilling, in particular the configuration we used in Run (d), reduced the mean absolute percentage error $\delta$ by 45 percent from 79 percent to 34 percent compared to the original algorithm. The number of additional changes found was reduced by 2.08 from 3.27 to 1.64 per pair of versions.

#### 5.3.2 Further Benchmark Runs

We performed further benchmarking using the Dice Coefficient and other $n$-grams. We do not discuss these results in detail as they were not as promising as our configuration used in Run (d), but summarize them briefly: Using tri or four-grams instead of bigrams resulted in a $\delta$ of 38 percent and 40 percent. Since tri and four-grams are less flexible than bigrams, fewer statement updates occurred. The Dice Coefficient for inner node matching combined with the various configurations resulted in a minimum $\epsilon$ of

43 percent, which is lower than the one that was achieved with the inner node similarity of Chawathe et al.

### 5.4 Limitations

Coming back to the results in Table 1, our algorithm is still limited in finding the appropriate number of move operations. In particular, the performances of *parameter ordering changes* and *statement ordering changes* are modest. After an in-depth inspection of the benchmark results, we found that the method `acceptSourceMethod(...)`[9] was responsible for these outliers. Removing this method from the benchmark yielded a $\delta$ of 30 percent; this is a further improvement of 4 percent. The number of parameter changes was decreased to one and all declaration changes were extracted correctly.

Concerning body changes, the main reason for the few additional errors was also due to this method because it mainly consists of small nested if and loop-statements. Although we used our dynamic threshold approach, these small blocks were not matched because 1) the node similarities of those blocks fall below 0.4 and 2) the depths of their subtrees are mostly bigger than 4.

Furthermore, the best match approach may match reoccurring statements that are not at the same position in the method body. For instance, consider that the first statement of a method changed, but the same statement reoccurs at the end of the method and stays unchanged. The best match approach will match the first with the last statement, leading to a mismatch for the first statement. Such a mismatch can have, as in this particular case, tremendous impact on the extraction of other changes. We noticed that such mismatches led to replacements of nested if and loop-statements. Currently, we are investigating postprocessing steps that take the position of statements into account to remove inappropriate matches.

The declaration changes, in particular the parameter ordering changes, are also an implication of the small tree problem. The parameter changes in Fig. 12 happened from Revision 1.35 to 1.39 of the `acceptSourceMethod(...)` described above; three new parameters were inserted. The similarity between the parameter-list nodes is 0.57 ($\frac{4}{7}$); thus, the nodes do not match. This mismatch yields the changes:

1. deletion of the old parameter list,
2. insertion of a new parameter list,
3. insertion of the three new parameters, and
4. moving of the existing parameters to the new list.

Besides the three parameter insertions, four additional parameter-ordering changes are classified—the parameter list insert and delete are omitted.

As we have selected the methods for the benchmark randomly and the $\delta$ of our algorithm is for all methods about 30 percent, except for the method described above, we claim that the unsolvable small tree problems occur relatively seldom. However, further investigations of this issue are needed and are the subject of future work.

9. In `org.eclipse.jdt.internal.core.SelectionRequestor`.

```
protected void acceptSourceMethod(
    IType type,
    char[] selector,
    char[][] parameterPackageNames,
    char[][] parameterTypeNames)
(Revision 1.35)
protected void acceptSourceMethod(
    IType type,
    char[] selector,
    char[][] parameterPackageNames,
    char[][] parameterTypeNames,
    boolean isDeclaration,
    int start,
    int end)
(Revision 1.39)
```

Fig. 12. Parameter changes from Revision 1.35 to 1.39 from `acceptSourceMethod(...)`.

### 5.5 Summary

To validate our improvements, we established an extensive benchmark comprised of 1,064 manually classified changes. Compared to the original algorithm of Chawathe et al., we approximate the minimum conforming edit script with a mean absolute error of 1.64 and a mean absolute percentage error of 34 percent per version pair, that is, an improvement of 45 percent. This means that, on the average, we find less than two additional change types, whereas the original algorithm finds more than three additional change types between two versions. The results showed that the combination of our best match algorithm with bigrams, Chawathe et al.'s node similarity measure, dynamic thresholds, and the inner node similarity weighting achieved the best benchmark results.

Although our dynamic thresholds noticeably inhibit mismatch propagation in small subtrees, we consider the problem as not fully solved yet as the changes in method `acceptSourceMethod(...)` showed.

## 6 Related Work

Source code differencing has proven itself to be a long-term research topic fundamental to multiversion program analyses, as pointed out by Kim and Notkin [22]. Existing approaches rely on either lexical, syntactical, or semantical differencing techniques. A further classification can be done with respect to the granularity of the algorithms, that is, whether they perform coarse-grained or fine-grained change extraction and analyses. Our algorithm identifies fine-grained syntactical changes. In [15], Hassan and Holt propose evolutionary code extractors in general. They discuss the need for such tools and the level of source code extraction granularity.

The algorithm presented by Chawathe et al. and our change distilling algorithm are closely related to tree differencing in general and to the tree edit distance problem in particular. The tree edit distance problem is to compute the edit distance based on a corresponding edit script

between two labeled ordered or unordered trees [6]. The edit operations used are 1) change the label of a node (relabel), 2) delete a nonroot node, and 3) insert a node. One of the first nonnaive algorithms for the tree edit distance problem was introduced by Tai [35]. The quadratic upper bound of this general approach has been improved by Shasha and Zhang [32], [46]. These algorithms are inappropriate for our concerns because

1. they do not act on labeled, ordered, and valued trees,
2. the operation *relabel* cannot be used for source code, as, for instance, a method invocation must not become an assignment,
3. they do not support move operations,
4. they do not support updates of values.

The algorithm by Chawathe et al. addresses these issues and, additionally, is faster than these general tree edit distance algorithms.

Existing differencing tools such as the well-known *GNU diff* [18] deal with flat, rather than with hierarchical, information. They are usually based on the longest common subsequence algorithm and calculate textual changes, that is, a list of lines that were changed, inserted, or deleted. *GNU diff* cannot, for example, distinguish between changes applied to license information or documentation and changes applied to a method body. In contrast to *diff*, our algorithm can detect changes more precisely and is able to assign a particular change to a concrete source code entity (such as the declaration or body part of a method), rather than just to a line number.

In [27], Maletic and Collard present a language independent approach for detecting syntactic differences between source files using an intermediate representation of the source code in XML. The output provided by *GNU diff* is mapped to an XML representation to locate changed entities. Our approach does not rely on textual differences and is able to detect changes due to move operations. Recently, Canfora et al. reconstructed changes from differencing results provided by CVS or Subversion *diff* to track the evolution of source code lines [7]. For that, they used Vector Space Models and the Levenshtein string similarity measure.

Yang describes an algorithm based on a branch-and-bound implementation of the largest common subtree problem [43]. The output of the algorithm is sets of matching and modified AST nodes, but it is not reported what operations transform the original into the modified tree.

Horwitz's approach computes semantic and textual differences between two programs [16]. The approach partitions a program according to its behavior extracted from the program representation graph. Similarly to our approach, Horwitz builds a matching set between such partitions to extract the differences. The approach is limited to programs written in a language that supports a subset of traditional programming languages. Furthermore, our approach provides a more complete set of tree edit operations and additionally classifies changes into change types. The algorithm presented by Jackson and Ladd reports semantic changes in procedural programs [20]. They analyze the input-output behavior of two procedures to detect changed behavior.

Apiwattanapong et al. [2] use enhanced control flow-graphs to model semantic behavior of methods of object-oriented programs. Identifying modified and unmodified methods is based on graph isomorphism. Their discussion of the impact of path changes caused by exception handling can be used to extend our work. Furthermore, we claim that both approaches, the one presented in [2] and our work, are complementary and that semantic differencing can be used to extend and refine our work.

Raghavan et al. implemented *Dex* [30], a tool for extracting changes between C source files. They use change information provided by patch files to locate the changed parts in source files. These parts are fed into their tree-differencing algorithm that outputs the edit operations. *Dex* can be used with our taxonomy to classify source code changes in C programs.

Tu and Godfrey used their *BEAGLE* tool to detect structural evolution of software systems [36]. With origin analysis, *BEAGLE* detects old functions as the "origin" of new ones based on software metrics and code clone detection. Origin analysis was also used to detect merging and splitting [14] and method renaming [24].

Recently, Xing and Stoulia presented their *UMLDiff* tool in [42]. *UMLDiff* tracks changes on the interface (logical design) of classes. In contrast to our work, they are able to track when entities are moved among different classes. However, *UMLDiff* focuses on *recovering higher level design knowledge evolution*, that is, changes on the interface level, whereas our work additionally allows fine-grained differencing on the implementation level, that is, changes on single statements inside of method bodies. Similarly to *UMLDiff*, *SiDiff* by Kelter et al. extracts differences between UML models [21]. The models are stored in XMI files. They use a combined top-down and bottom-up approach for matching model parts. The matching is then used to classify differences of UML models.

Kim et al. presented an approach to automatically infer likely changes at or above the method level [23]. They use a simple matching algorithm with the Levenshtein string similarity measure. Compared to *UMLDiff* or *SiDiff*, the Kim et al. inference approach represents the changes concisely as first-order relational logic rules. Each of them combines a set of similar low-level transformations and describes exceptions that capture anomalies to a general change pattern.

Approaches discussed next are in the field of change analysis and classification. They are related to our taxonomy of source code changes.

Xing and Stoulia [41] use their UMLDiff to classify interface changes. For each class version, they assign a volatility level, for example, "intense evolution" or "rapidly developing," according to the number of changes that occurred. In contrast, Kelter et al. focus on special differences of UML models (for example, attribute or reference differences) instead of general insert, delete,

TABLE 2
Benchmark Results of Runs (a) and (b)

| | | (a) | | | | | (b) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Change Type | $x$ | $\tilde{x}$ | $\epsilon$ | $\epsilon^2$ | $\delta$ | $\delta^2$ | $\tilde{x}$ | $\epsilon$ | $\epsilon^2$ | $\delta$ | $\delta^2$ |
| Alternative Part Delete | 9 | 32 | 1.04 | 1.24 | 0.08 | 0.29 | 28 | 1.17 | 1.47 | 0.11 | 0.33 |
| Alternative Part Insert | 15 | 40 | 0.86 | 1.13 | 0.06 | 0.22 | 36 | 0.88 | 1.27 | 0.06 | 0.23 |
| Condition Expression Change | 91 | 51 | 1.02 | 1.53 | 0.58 | 0.76 | 64 | 0.89 | 1.43 | 0.44 | 0.68 |
| Method Renaming | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Parameter Delete | 9 | 12 | 0.43 | 0.65 | 0.07 | 0.19 | 12 | 0.43 | 0.65 | 0.07 | 0.19 |
| Parameter Insert | 16 | 20 | 0.29 | 0.53 | 0.04 | 0.13 | 20 | 0.29 | 0.53 | 0.04 | 0.13 |
| Parameter Ordering Change | 0 | 19 | 2.71 | 3.09 | 0 | 0 | 19 | 2.71 | 3.09 | 0 | 0 |
| Parameter Renaming | 3 | 1 | 0.67 | 0.82 | 0.67 | 0.82 | 2 | 0.75 | 0.87 | 0.5 | 0.71 |
| Parameter Type Change | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Return Type Change | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Return Type Insert | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Statement Delete | 144 | 371 | 2.28 | 3.36 | 0.3 | 0.9 | 283 | 1.96 | 2.72 | 0.29 | 0.93 |
| Statement Insert | 391 | 640 | 2.15 | 3.41 | 0.4 | 1.08 | 552 | 1.89 | 3.01 | 0.42 | 0.96 |
| Statement Ordering Change | 14 | 105 | 2.26 | 3.52 | 0.12 | 0.49 | 82 | 2.23 | 3.43 | 0.19 | 0.58 |
| Statement Parent Change | 86 | 185 | 1.84 | 2.9 | 0.2 | 0.58 | 194 | 1.87 | 3.05 | 0.21 | 0.63 |
| Statement Update | 282 | 216 | 0.96 | 1.74 | 0.34 | 0.57 | 318 | 0.72 | 1.32 | 0.19 | 0.51 |
| Total | 1064 | 1696 | 3.27 | 6.44 | 0.79 | 1.66 | 1613 | 2.91 | 6.21 | 0.72 | 1.91 |

move, and update of UML diagram parts [21]. Compared to their work, we classify individual changes.

Śliwerski et al. classify changes according to whether they induced a fix [34], that is, changes that lead to problems. Their Eclipse plugin Hatari [33] extracts and visualizes such changes. With our classification, we can detect frequent fix-inducing change types.

Small changes are also investigated by Purushothaman and Perry in [29]. In a large case study, they found that there is less than a 4 percent probability that a one-line change will introduce a fault. This result implies that the significance level of a one-line change is low.

The area of code clone detection and software merging, although not directly related to our work, relies on source code differencing.

Sager et al. [31] used several tree matching algorithms for detecting similar Java classes. First, they converted the AST as generated by Eclipse into the language independent metamodel FAMIX [10]. In a second step, they transformed the model into a generic tree format. The generic tree representations of all classes of a software system were then matched against each other to find similar classes. Sager et al. evaluated three different tree similarity algorithms for this purpose, derived from a *bottom-up maximum common subtree isomorphism*, a *top-down maximum common subtree isomorphism*, and an *edit distance of two given trees*, all three originally presented in [37]. These algorithms can be used to replace the tree similarity measure calculated in our approach.

Baxter et al. describe *CloneDr*, a tool for code clone detection [4] that relies on AST but categorizes subtrees by hashing. This significantly reduces the number of comparisons needed since only subtrees with the same hash values have to be compared. Classification using hash values works well for exact duplicates, but fails for locating near-miss clones, that is, code duplicates that are very similar. They are able to overcome this shortcoming by choosing an artificial bad hash function, that is, a function that ignores identifier names. For determining the similarity of two ASTs, Baxter et al. have used the Dice Coefficient [11].

Mens has conducted a survey on existing software merging techniques in [28]. For example, the approaches presented in [3], [17], [39], [44] rely on tree-based differencing in order to perform merging. All of them have some limitations with regard to our concerns; as far as we know, neither of them detects *moves* or outputs an edit script.

## 7 Conclusions

A key issue in software evolution analysis is the identification of particular changes that occur across several versions of a program. Current approaches that investigate source code changes rely on information provided by versioning systems such as CVS. They track changes of source code files on a *text basis* without storing detailed information. In particular, the granularity, the type, and the significance level of changes between two versions of a source code entity are not tracked at all. To improve change analysis results, it is necessary to differentiate change types. Only in this way can we provide better support for programmers, designers, and project managers to develop and maintain software systems and control their evolution.

To overcome the imprecise results of textual differencing, we presented *change distilling*, an approach for *fine-grained source code change extraction*. We enhanced the existing tree differencing algorithm by Chawathe et al. to classify source

TABLE 3
Benchmark Results of Runs (c) and (d)

| | | (c) | | | | | (d) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Change Type | $x$ | $\tilde{x}$ | $\epsilon$ | $\epsilon^2$ | $\delta$ | $\delta^2$ | $\tilde{x}$ | $\epsilon$ | $\epsilon^2$ | $\delta$ | $\delta^2$ |
| Alternative Part Delete | 9 | 25 | 1.06 | 1.33 | 0.12 | 0.34 | 14 | 0.78 | 1.11 | 0.22 | 0.47 |
| Alternative Part Insert | 15 | 33 | 0.78 | 1.14 | 0.07 | 0.23 | 22 | 0.41 | 0.8 | 0 | 0 |
| Condition Expression Change | 91 | 58 | 0.92 | 1.5 | 0.47 | 0.67 | 85 | 0.58 | 1.01 | 0.24 | 0.48 |
| Method Renaming | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Parameter Delete | 9 | 11 | 0.33 | 0.58 | 0.08 | 0.2 | 11 | 0.33 | 0.58 | 0.08 | 0.2 |
| Parameter Insert | 16 | 19 | 0.23 | 0.48 | 0.04 | 0.14 | 19 | 0.23 | 0.48 | 0.04 | 0.14 |
| Parameter Ordering Change | 0 | 19 | 2.71 | 3.09 | 0 | 0 | 17 | 3.4 | 3.61 | 0 | 0 |
| Parameter Renaming | 3 | 1 | 0.67 | 0.82 | 0.67 | 0.82 | 1 | 0.67 | 0.82 | 0.67 | 0.82 |
| Parameter Type Change | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Return Type Change | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Return Type Insert | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Statement Delete | 144 | 264 | 1.84 | 2.54 | 0.32 | 0.92 | 225 | 1.77 | 2.37 | 0.34 | 0.79 |
| Statement Insert | 391 | 533 | 1.76 | 2.84 | 0.38 | 0.89 | 494 | 1.54 | 2.65 | 0.32 | 0.83 |
| Statement Ordering Change | 14 | 83 | 2.08 | 3.34 | 0.11 | 0.38 | 73 | 2.23 | 3.61 | 0.08 | 0.24 |
| Statement Parent Change | 86 | 162 | 1.56 | 2.67 | 0.11 | 0.43 | 118 | 1.14 | 1.64 | 0.21 | 0.53 |
| Statement Update | 282 | 259 | 0.41 | 0.9 | 0.14 | 0.34 | 260 | 0.41 | 0.9 | 0.14 | 0.34 |
| Total | 1064 | 1471 | 2.2 | 4.89 | 0.52 | 1.45 | 1343 | 1.64 | 3.93 | 0.34 | 1.09 |

code changes according to our *taxonomy of source code changes* with the following substantial improvements:

- Using bigrams as a robust string similarity measure that is able to cover common changes on source code identifiers.
- Adding a similarity check of node values to Chawathe et al.'s tree similarity measure to solve the problem of descendant subtree matching.
- Using inner node similarity weighting to reduce inadequate mismatches of condition expressions.
- Introducing the best match algorithm to reduce the impact of Chawathe et al.'s Assumption 1.
- Using dynamic thresholds to reduce the propagation of mismatches in small subtrees.

Furthermore, we introduced an extensive benchmark to evaluate source code change extraction algorithms. The benchmark consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects. By applying the benchmark to the CHANGE DISTILLER, the implementation of our change distilling algorithm, we achieved significant improvements in extracting change types: Our algorithm approximates the minimum edit script 45 percent better than the original change extraction approach by Chawathe et al. We were able to find all occurring changes and almost reach the minimum conforming edit script, that is, we reach a mean absolute percentage error of 34 percent, compared to the 79 percent reached by the original algorithm.

Although our dynamic thresholds significantly inhibit mismatch propagation in small subtrees, we consider the problem not fully solved yet. In our benchmark, we experienced inadequacies with one particular method that is deeply nested and has major declaration changes. Since further improvements of string similarity measures are limited, we will investigate postprocessing steps to filter further inadequate matches.

## APPENDIX

For each of the four Runs (a)-(d) described in Section 5.3, the detailed results are listed in Tables 2 and 3. The tables contain the expected number of occurrences of each change type $x$, the found number of occurrences of each change type $\tilde{x}$, the mean absolute error $\epsilon$, the root-mean-squared absolute error $\epsilon^2$, the mean absolute percentage error $\delta$, and the root-mean-squared absolute percentage error $\delta^2$.

## ACKNOWLEDGMENTS

This work was supported by the Swiss National Science Foundation as part of the Controlling Software Evolution project (COSE) and the Hasler Foundation as part of the ProMedServices—Proactive Software Service Improvement and EvoSpaces—Multidimensional Navigation Spaces for Software Evolution projects. The authors would like to thank Abraham Bernstein, Michele Lanza, Peter Vorburger, and the reviewers for their insightful suggestions that greatly helped to improve the paper.

## REFERENCES

[1] G.W. Adamson and J. Boreham, "The Use of an Association Measure Based on Character Structure to Identify Semantically Related Pairs of Words and Document Titles," *Information Storage and Retrieval*, vol. 10, nos. 7-8, pp. 253-260, July-Aug. 1974.

[2] T. Apiwattanapong, A. Orso, and M.J. Harrold, "JDiff: A Differencing Technique and Tool for Object-Oriented Programs," *Automated Software Eng.*, vol. 14, no. 1, pp. 3-36, Mar. 2007.

[3] U. Asklund, "Identifying Conflicts during Structural Merge," *Proc. Nordic Workshop Programming Environment Research,* pp. 231-242, June 1994.
[4] I.D. Baxter, A. Yahin, L.M. de Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance,* pp. 368-377, Nov. 1998.
[5] A. Bernstein, C. Kiefer, and E. Kaufmann, "SimPack: A Generic Java Library for Similarity Measures in Ontologies," technical report, Dept. of Informatics, Univ. of Zürich, Switzerland, 2005.
[6] P. Bille, "A Survey on Tree Edit Distance and Related Problems," *Theoretical Computer Science,* vol. 337, nos. 1-3, pp. 217-239, June 2005.
[7] G. Canfora, L. Cerulo, and M.D. Penta, "Identifying Changed Source Code Lines from Version Repositories," *Proc. Int'l Workshop Mining Software Repositories,* p. 14, May 2007.
[8] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change Detection in Hierarchically Structured Information," *Proc. ACM Sigmod Int'l Conf. Management of Data,* pp. 493-504, June 1996.
[9] D. Čubranić, G.C. Murphy, J. Singer, and K.S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Trans. Software Eng.,* vol. 31, no. 6, pp. 446-465, June 2005.
[10] S. Demeyer, S. Tichelaar, and P. Steyaert, *Famix—The Famoos Information Exchange Model,* 1999.
[11] L.R. Dice, "Measures of the Amount of Ecologic Association between Species," *ESA Ecology,* no. 26, pp. 297-302, July 1945.
[12] B. Fluri and H.C. Gall, "Classifying Change Types for Qualifying Change Couplings," *Proc. Int'l Conf. Program Comprehension,* pp. 35-45, June 2006.
[13] H. Gall, K. Hayek, and M. Jazayeri, "Detection of Logical Coupling Based on Product Release History," *Proc. Int'l Conf. Software Maintenance,* pp. 190-198, Nov. 1998.
[14] M.W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. Software Eng.,* vol. 31, no. 2, pp. 166-181, Feb. 2005.
[15] A.E. Hassan and R.C. Holt, "Studying the Evolution of Software Systems Using Evolutionary Code Extractors," *Proc. Int'l Workshop Principles of Software Evolution,* pp. 76-81, Sept. 2004.
[16] S. Horwitz, "Identifying the Semantic and Textual Differences between Two Versions of a Program," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 234-245, June 1990.
[17] J.J. Hunt, "Extensible, Language-Aware Differencing and Merging," PhD dissertation, Univ. of Karlsruhe, Germany, 2001.
[18] J.W. Hunt and T.G. Szymanski, "A Fast Algorithm for Computing Longest Common Subsequences," *Comm. ACM,* vol. 20, no. 5, pp. 350-353, May 1977.
[19] P. Jaccard, "The Distribution of the Flora in the Alpine Zone," *New Phytologist,* vol. 11, no. 2, pp. 37-50, Feb. 1912.
[20] D. Jackson and D.A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," *Proc. Int'l Conf. Software Maintenance,* pp. 243-252, Sept. 1994.
[21] U. Kelter, J. Wehren, and J. Niere, "A Generic Difference Algorithm for UML Models," *Software Eng.,* pp. 105-116, Mar. 2005.
[22] M. Kim and D. Notkin, "Program Element Matching for Multi-Version Program Analyses," *Proc. Int'l Workshop Mining Software Repositories,* pp. 58-64, May 2006.
[23] M. Kim, D. Notkin, and D. Grossman, "Automatic Inference of Structural Changes for Matching Across Program Versions," *Proc. Int'l Conf. Software Eng.,* pp. 333-343, May 2007.
[24] S. Kim, K. Pan, and J.E. Whitehead, "When Functions Change Their Names: Automatic Detection of Origin Relationships," *Proc. Working Conf. Reverse Eng.,* pp. 143-152, Nov. 2005.
[25] M.M. Lehman, "Programs, Life Cycles and Laws of Software Evolution," *Proc. IEEE,* pp. 1060-1076, Sept. 1980.
[26] V.I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady,* vol. 10, pp. 707-710, Feb. 1966.
[27] J.I. Maletic and M.L. Collard, "Supporting Source Code Difference Analysis," *Proc. Int'l Conf. Software Maintenance,* pp. 210-219, Sept. 2004.
[28] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. Software Eng.,* vol. 28, no. 5, pp. 449-462, May 2002.
[29] R. Purushothaman and D.E. Perry, "Toward Understanding the Rhetoric of Small Source Code Changes," *IEEE Trans. Software Eng.,* vol. 31, no. 6, pp. 511-526, June 2005.
[30] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Base," *Proc. Int'l Conf. Software Maintenance,* pp. 188-197, Sept. 2004.
[31] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer, "Detecting Similar Java Classes Using Tree Algorithms," *Proc. Int'l Workshop Mining Software Repositories,* pp. 65-71, May 2006.
[32] D. Shasha and K. Zhang, "Fast Algorithms for the Unit Cost Editing Distance between Trees," *J. Algorithms,* vol. 11, no. 4, pp. 581-621, Dec. 1990.
[33] J. Śliwerski, T. Zimmermann, and A. Zeller, "Hatari: Raising Risk Awareness," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 107-110, Sept. 2005.
[34] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" *Proc. Int'l Workshop Mining Software Repositories,* pp. 24-28, May 2005.
[35] K.-C. Tai, "The Tree-to-Tree Correction Problem," *J. ACM,* vol. 26, no. 3, pp. 422-433, July 1979.
[36] Q. Tu and M.W. Godfrey, "An Integrated Approach for Studying Architectural Evolution," *Proc. Int'l Workshop Program Comprehension,* pp. 127-136, June 2002.
[37] G. Valiente, *Algorithms on Trees and Graphs.* Springer, 2002.
[38] J. Weidl and H.C. Gall, "Binding Object Models to Source Code: An Approach to Object-Oriented Re-Architecting," *Proc. Computer Software and Applications Conf.,* pp. 26-31, Aug. 1998.
[39] B. Westfechtel, "Structure-Oriented Merging of Revisions of Software Documents," *Proc. Int'l Workshop Software Configuration Management,* pp. 68-79, June 1991.
[40] M. Würsch, "Improving ChangeDistiller—Improving Abstract Syntax Tree Based Source Code Change Detection," master's thesis, Univ. of Zürich, 2006.
[41] Z. Xing and E. Stoulia, "Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software," *IEEE Trans. Software Eng.,* vol. 31, no. 10, pp. 850-868, Oct. 2005.
[42] Z. Xing and E. Stoulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing," *Proc. Int'l Conf. Automated Software Eng.,* pp. 54-65, Nov. 2005.
[43] W. Yang, "Identifying Syntactic Differences between Two Programs," *J. Software—Practice and Experience,* vol. 21, no. 7, pp. 739-755, July 1991.
[44] W. Yang, "How to Merge Program Texts," *J. Systems and Software,* vol. 27, no. 2, pp. 129-135, Nov. 1994.
[45] A.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Trans. Software Eng.,* vol. 30, no. 9, pp. 574-586, Sept. 2004.
[46] K. Zhang, "Algorithms for the Constrained Editing Distance between Ordered Labeled Trees and Related Problems," *Pattern Recognition,* vol. 28, no. 3, pp. 463-474, June 1995.
[47] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Eng.,* vol. 31, no. 6, pp. 429-445, June 2005.

**Beat Fluri** received the MSc degree in computer science from the Federal Institute of Technology (ETH), Switzerland, in 2004. He is currently working toward the doctorate degree and works as a research assistant in Professor Harald C. Gall's Research Group at the University of Zurich, Switzerland. His main research interest is software evolution, focusing on source code change analysis and changeability assessment. He is a student member of the IEEE, the IEEE Computer Society, and the ACM. More information is available at seal.ifi.uzh.ch/fluri.

**Michael Würsch** received the MSc degree in computer science from the University of Zurich in 2007. He is currently working toward the doctorate degree and is a research assistant in the Software Engineering Group in the Department of Informatics at the University of Zurich. His current research interests include software design, software evolution analysis, and service-centric software engineering. He is a student member of the IEEE and the IEEE Computer Society. More information is available at seal.ifi.uzh.ch/wuersch.

**Martin Pinzger** received the doctorate degree (Dr. techn.) in computer science from the Vienna University of Technology, Austria, in June 2005. He is a senior research associate in the Software Engineering Group in the Department of Informatics at the University of Zurich, Switzerland. His research interests are in software engineering, focusing on software evolution analysis and software design and quality analysis. He is a member of the IEEE, the IEEE Computer Society, and the ACM. More information is available at seal.ifi.uzh.ch/pinzger.

**Harald C. Gall** received the MSc and PhD (Dr. techn.) degrees in informatics from the Technical University of Vienna, Austria. He is a professor of software engineering in the Department of Informatics at the University of Zurich, Switzerland. Prior to that, he was an associate professor in the Distributed Systems Group at the Technical University of Vienna. His research interests include software engineering, focusing on software evolution, software quality analysis, software architecture, reengineering, collaborative software engineering, and service-centric software systems. Recently, he was the program chair of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE) 2005 and the International Workshop on Mining Software Repositories (MSR), colocated with the International Conference on Software Engineering (ICSE) in 2006 and 2007. He is a member of the IEEE, the IEEE Computer Society, and the ACM. More information is available at seal.ifi.uzh.ch/gall.