Fine-grained and Accurate Source Code Differencing

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, Martin Monperrus

Fine-grained and Accurate Source Code Differencing

Jean-Rémy Falleri
Univ. Bordeaux,
LaBRI, UMR 5800
F-33400, Talence, France
falleri@labri.fr

Floréal Morandat
Univ. Bordeaux,
LaBRI, UMR 5800
F-33400, Talence, France
fmoranda@labri.fr

Xavier Blanc
Univ. Bordeaux,
LaBRI, UMR 5800
F-33400, Talence, France
xblanc@labri.fr

Matias Martinez
INRIA and University of Lille,
France
matias.martinez@inria.fr

Martin Monperrus
INRIA and University of Lille,
France
martin.monperrus@inria.fr

ABSTRACT

At the heart of software evolution is a sequence of edit actions, called an *edit script*, made to a source code file. Since software systems are stored version by version, the edit script has to be computed from these versions, which is known as a complex task. Existing approaches usually compute edit scripts at the text granularity with only *add line* and *delete line* actions. However, inferring syntactic changes from such an edit script is hard. Since moving code is a frequent action performed when editing code and it should also be taken into account. In this paper, we tackle these issues by introducing an algorithm computing edit scripts at the abstract syntax tree granularity including move actions. Our objective is to compute edit scripts that are short and close to the original developer intent. Our algorithm is implemented in a freely-available and extensible tool that has been intensively validated.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms: Algorithms, Experimentation

Keywords: Software evolution, Program comprehension, Tree differencing, AST.

1. INTRODUCTION

The first law of software evolution states that almost all software systems have to evolve to be satisfactory [19]. Since this law was formulated, many studies have been performed to better understand how software systems evolve, and forms what is called the *software evolution* research field [21].

There is global software evolution (e.g. evolution of requirements, of execution environments, ...) and local software evolution (evolution of source code files). In this paper, we focus on the latter, that is on understanding how source code files evolve. In particular, we focus on *edit scripts*, that are sequences of edit actions made to a source code file. Usually, since software is stored in version control systems, edit scripts are computed between two versions of a same file. The goal of an edit script is to accurately reflect the actual change that has been performed on a file.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642982.

Edit scripts are used by developers on a daily basis. For example, the Unix `diff` tool takes as input two versions of a source code file and performs the Myers algorithm [24] at the text line granularity and returns an edit script indicating which lines have been added or deleted. However, the limitations of `diff` are twofold. First, it only computes additions and deletions and does not consider other kinds of edit actions such as update and move. Second, it works at a granularity (the text line) that is both coarse grain and not aligned with the source code structure: the abstract syntax tree.

To overcome this main limitation, there are algorithms that can work at the abstract syntax tree (AST) level [13]. The main advantage in using the AST granularity is that the edit script directly refers to the structure of the code. For instance, if an edit action is the addition of a new `function` node, it clearly means that a new function has been added in the code. Despite several key contributions (e.g. [13]), the problem of computing AST edit scripts is still open, with two main challenges: handling move actions, and scaling to fine-grained ASTs with thousands of nodes[1]. This is where this paper makes a contribution.

To design our novel algorithm, we take the viewpoint of the developer: she is never interested in the theoretical shortest edit script. She is rather interested in having an edit script that reflects well the actual changes that happened. Thus our objective is not to find the shortest sequence of actions between two versions, but a sequence that reflects well the developer intent. Consequently, we devise an algorithm based on heuristics that contain pragmatic rules on what a good edit script is, and as importantly, that is efficient and scales to large ASTs. This algorithm has been implemented within a freely-available and extensible tool[2].

To sum up, our contributions are:

- a novel efficient AST differencing algorithm that takes into account move actions, and its implementation;

[1]The best known algorithm with add, delete and update actions has a $O(n^3)$ time complexity with n being the number of nodes of the AST [27]. Computing the minimum edit script that can include move node actions is known to be NP-hard [4]

[2]`github.com/jrfaller/gumtree`

- an automated evaluation of the implementation performances on real data;
- a manual evaluation of the results of the algorithm through the manual assessment of 144 differencing scenarios;
- a large-scale automated evaluation of 12 792 differencing scenarios showing that the results of our algorithm are more accurate than the related work, even on fine-grained ASTs.

The rest of this paper is structured as follows: Section 2 presents what is AST differencing. Section 3 describes our new AST differencing algorithm. Section 4 presents our tool that implement this new algorithm and its performances. Section 5 presents an empirical evaluation of our tool. Section 6 presents the related work. Finally, Section 7 concludes and presents the future work.

2. AST DIFFERENCING

Prior to presenting AST differencing, we briefly introduce the main concepts defining the AST structure. We consider that an AST is a labeled ordered rooted tree where nodes may have a string value. Labels of nodes correspond to the name of their production rule in the grammar, i.e., they encode the structure. Values of the nodes correspond to the actual tokens in the code.

More formally, let T be an AST. T is a set of nodes. A tree T has one node that is root (denoted by $root(T)$). Each node $t \in T$ has a parent $p \in T \cup \emptyset$. The only node that has $\emptyset$ for parent is the root. The parent of a node is denoted by $parent(t)$. Each node $t \in T$ has a sequence of children ($children(t)$). Each node has a label in a alphabet $l \in \Sigma$ ($label(t) = l$). Each node has a string value $v \in String$ that is possibly empty ($value(t) = v \vee \epsilon$).

As an example, we consider a simple JAVA source code and its corresponding AST (see the bottom-left of Figure 1). The AST of this JAVA source code contains 19 nodes that correspond to the structure of the JAVA programming language. Each node of the AST has therefore a label, which maps to structural elements of the source code (such as `MethodDeclaration` or `NumberLiteral`), and a value that corresponds to the actual tokens in the code (such as `NumberLiteral` associated to `1`). Some values do not encode information and are therefore discarded, for instance `MethodDeclaration` has no interesting token associated to it and thus no value.

ASTs can have different granularities, a node can encode a whole instruction or finer grain expressions. We believe than fine-grained ASTs are best for the developers. For instance the `return "Foo!";` statement, can be encoded with a single node of type `Statement` and value `return "Foo!"`, or with two nodes like in our example (see node b). If this statement is changed to `return "Foo!" + i;`, only the fine-grained representation enables to see that the `InfixExpression:+` and `SimpleName:i` nodes are added.

AST differencing is based upon the concept of AST edit actions. It aims at computing a sequence of edit actions that transform an AST into another. This sequence is called an *edit script*. Regarding our definition of an AST, we consider the following edit actions:

- $updateValue(t, v_n)$ replaces the old value of t by the new value v_n
- $add(t, t_p, i, l, v)$ adds a new node t in the AST. If t_p is not null and i is specified then t is the i^{th} child of t_p. Otherwise t is the new root node and has the previous root node as its only child. Finally, l is the label of t and v is the value of t.
- $delete(t)$ removes a leaf node of the AST.
- $move(t, t_p, i)$ moves a node t and make it the i^{th} child of t_p. Note that all children of t are moved as well, therefore this actions moves a whole subtree.

As there are many possible edit scripts that perform the same transformation, the edit script quality depends on its length: the shorter the transformation, the better the quality. Note that finding the shortest transformation is NP-hard when the *move* action is taken into consideration.

We then consider in this paper that the AST differencing problem inputs two ASTs and aims at identifying a short edit script σ of edit actions (including *move*) that transforms a first AST (named t_1) into a second one (name t_2). The existing algorithms that perform such an AST differencing use heuristics to return a short edit script σ. Moreover, they usually follow a two steps process. First, they establish mappings (pairs of nodes) between the similar nodes of the two ASTs. There are two constraints for these mappings: a given node can only belong to one mapping, and mappings involve two nodes with identical labels. Second, based on these mappings, they deduce the edit script that must be performed on the first AST to obtain the second one. The first step is the most crucial one because quadratic optimal algorithms exist for the second step [6, 15]. In the next section, we will present a new algorithm to compute mappings between two ASTs.

3. THE GUMTREE ALGORITHM

As explained in the previous section, AST differencing algorithms work in two steps: establishing mappings then deducing an edit script. Since an optimal and quadratic algorithm has already been developed for the second step [6], we only explain in this section how we look for the mappings between two ASTs. The output of this algorithm can be then used by the algorithm of Chawathe et al. [6] to compute the actual edit script. Our algorithm to compute the mappings between two ASTs is composed of two successive phases:

1. A greedy top-down algorithm to find isomorphic subtrees of decreasing height. Mappings are established between the nodes of these isomorphic subtrees. They are called *anchors mappings*.
2. A bottom-up algorithm where two nodes match (called a *container mapping*) if their descendants (children of the nodes, and their children, and so on) include a large number of common anchors. When two nodes matches, we finally apply an optimal algorithm to search for additional mappings (called *recovery mappings*) among their descendants.

This algorithm is inspired by the way developers manually look at changes between to files. First they search for the biggest unmodified pieces of code. Then they deduce which container of code can be mapped together. Finally they look at precise differences in what is leftover in each container. To better illustrate our algorithm, we introduce the example shown in Figure 1.

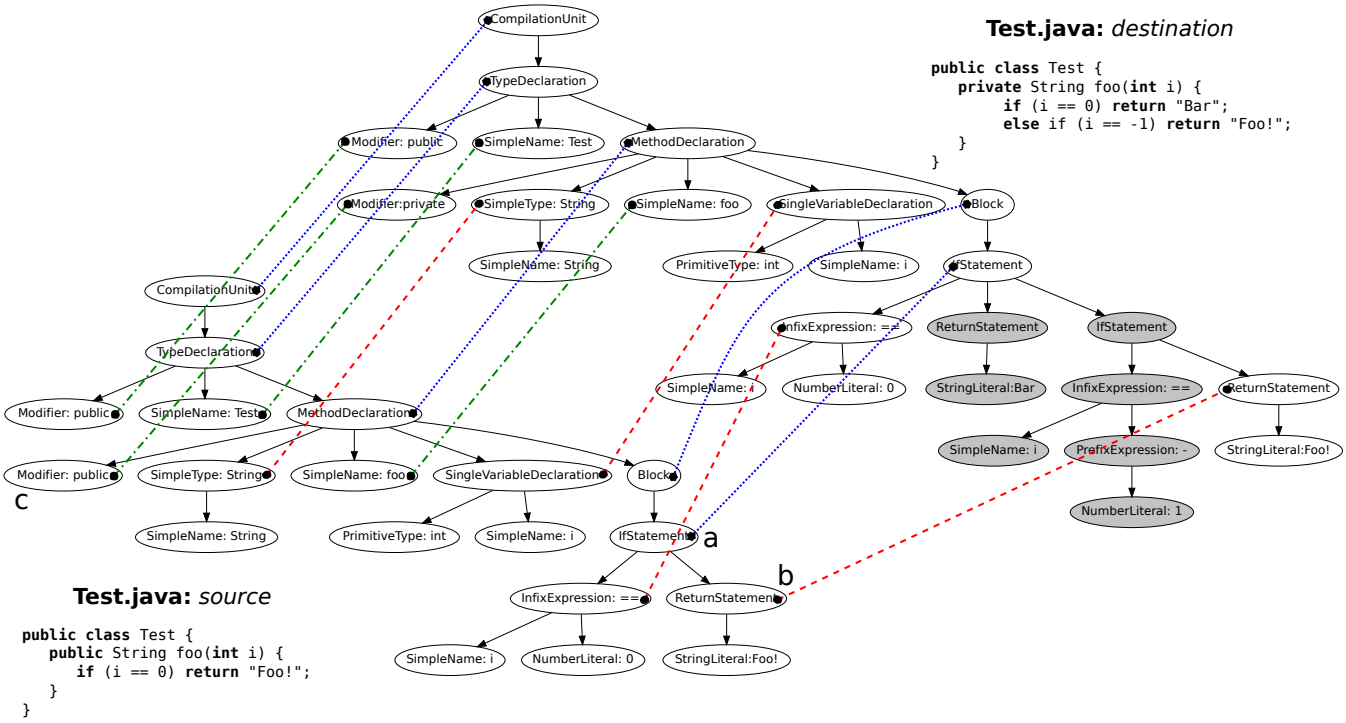

Figure 1: Two sample Java files with their corresponding ASTs and mappings. Nodes with a label only are denoted: `Label`, nodes with a label and a value are denoted: `Label: value`. Mappings from the top-down phase are depicted with long-dotted lines (descendants of these nodes are also mapped but it is omitted to enhance readability). Mappings from the bottom-up phase are depicted using short-dotted lines (container mappings) or alternate-dotted lines (recovery mappings). Unmatched nodes are greyed.

3.1 Top-down Phase

The first step of GumTree is a top-down greedy search of the greatest isomorphic subtrees between T_1 and T_2. Before explaining how we proceed, we introduce the notion of *height* in a tree. The height of a node $t \in T$ is defined as: 1) for a leaf node t, $\text{height}(t) = 1$ and 2) for an internal node t, $\text{height}(t) = \max(\{\text{height}(c) | c \in children(t)\}) + 1$. The algorithm uses an auxiliary data structure called *height-indexed priority list*. This list contains a sequence of nodes, ordered by decreasing height. The following functions are associated with this data-structure. $push(t, l)$ inserts the node t in the list l. $peekMax(l)$ returns the greatest height of the list. $pop(l)$ returns and removes from l the set of all nodes of l having a height equals to $peekMax(l)$. $open(t, l)$ inserts all the children of t into l. We also define the dice function that measure the ratio of common descendants between two nodes given a set of mappings $\mathcal{M}$ as $\text{dice}(t_1, t_2, \mathcal{M}) = \frac{2 \times |\{t_1 \in s(t_1) | (t_1, t_2) \in \mathcal{M}\}|}{|s(t_1)| + |s(t_2)|}$, with $s(t_i)$ being the set of the descendants of node t_i. The dice coefficient ranges in the $[0, 1]$ real interval, a value of 1 indicates that the set of descendants of t_1 is the same as the set of descendants of t_2. The algorithm of the top-down phase of GumTree is shown in Algorithm 1.

In this algorithm, we map the common subtrees of T_1 and T_2 with the greatest height possible. The principle is to start with the roots (since they have the greatest heights) and to check if they are isomorphic. If they are not, their children are then tested. A node is matched as soon as an isomorphic node is found in the other tree. When a given node can be matched to several nodes, all the potential mappings are kept in a dedicated *candidate mappings* list. This list is processed after all nodes that are uniquely matched have been processed; those nodes being directly placed into the mappings set. The algorithm considers only nodes with a height greater than $minHeight$. To process the candidate mappings, we use the dice function on the parents of each candidate mapping. The values of this function are used to sort the candidate mappings list, mappings with greater values being first. Then, until the candidate mappings list is empty, we remove the first element, add it in the mappings set, and we remove from the candidate mappings list the mappings involving a node of this mapping. On the sample trees of Figure 1 with a $minHeight = 2$, Algorithm 1 finds the mappings shown with dashed lines.

3.2 Bottom-up Phase

Algorithm 2 shows the bottom-up phase, where the mappings produced during the top-down phase are taken as input. First we look for container mappings, that are established when two nodes have a significant number of matching descendants. For each container mapping found, we look for recovery mappings, that are searched among the still unmatched descendants of the mapping's nodes. To find the container mappings, the nodes of T_1 are processed in post-order. For each unmatched non-leaf node of T_1, we extract a list of candidate nodes from T_2. A node $c \in T_2$ is a candidate for t_1 if $label(t_1) = label(c)$, c is unmatched, and t_1 and c have some matching descendants. We then select the candidate $t_2 \in T_2$ with the greatest $\text{dice}(t_1, t_2, \mathcal{M})$ value. If

Algorithm 1: The algorithm of the top-down phase.

```
Data: A source tree T1 and a destination tree T2, a minimum
      height minHeight, two empty height-indexed priority
      lists L1 and L2, an empty list A of candidate
      mappings, and an empty set of mappings M
Result: The set of mappings M
1  push(root(T1), L1);
2  push(root(T2), L2);
3  while min(peekMax(L1), peekMax(L2)) > minHeight do
4      if peekMax(L1) ≠ peekMax(L2) then
5          if peekMax(L1) > peekMax(L2) then
6              foreach t ∈ pop(L1) do open(t, L1);
7          else
8              foreach t ∈ pop(L2) do open(t, L2);
9      else
10         H1 ← pop(L1);
11         H2 ← pop(L2);
12         foreach (t1, t2) ∈ H1 × H2 do
13             if isomorphic(t1, t2) then
14                 if ∃tx ∈ T2 | isomorphic(t1, tx) ∧ tx ≠ t2
                   or ∃tx ∈ T1 | isomorphic(tx, t2) ∧ tx ≠ t1
                   then
15                     add(A, (t1, t2));
16                 else
17                     add all pairs of isomorphic nodes of s(t1)
                       and s(t2) to M;
18         foreach t1 ∈ H1 | (t1, tx) ∉ A ∪ M do open(t1, L1);
19         foreach t2 ∈ H2 | (tx, t2) ∉ A ∪ M do open(t2, L2);
20 sort (t1, t2) ∈ A using dice(parent(t1), parent(t2), M);
21 while size(A) > 0 do
22     (t1, t2) ← remove(A, 0);
23     add all pairs of isomorphic nodes of s(t1) and s(t2) to M;
24     A ← A \ {(t1, tx) ∈ A};
25     A ← A \ {(tx, t2) ∈ A};
```

$\text{dice}(t_1, t_2, \mathcal{M}) > minDice$, t_1 and t_2 are matched together. To search for additional mappings between the descendants of t_1 and t_2, we first remove their matched descendants, and if both resulting subtrees have a size smaller than $maxSize$, we apply an algorithm denoted *opt* that finds a shortest edit script without move actions. In our implementation we use the RTED algorithm [27]. The mappings induced from this edit script are added in $\mathcal{M}$ if they involve nodes with identical labels.

On the sample trees of Figure 1, with $minDice = 0.2$, Algorithm 2 finds the container mappings shown using short-dotted lines. From these container mappings, with $maxSize = 100$, several recovery mappings are found, shown with alternate-dotted lines. Finally, the edit script generated from these mappings is as follows (nodes a, b and c are shown in Figure 1, nodes t_i are new nodes):

$$
\begin{aligned}
&add(t_1, a, 1, ReturnStatement, \epsilon)\\
&add(t_2, t_1, 0, StringLitteral, Bar)\\
&add(t_3, a, 2, IfStatement, \epsilon)\\
&add(t_4, t_3, 0, InfixExpression, ==)\\
&add(t_5, t_4, 0, SimpleName, i)\\
&add(t_6, t_4, 1, PrefixExpression, -)\\
&add(t_7, t_6, 0, NumberLiterral, 1)\\
&move(b, t_3, 1)\\
&updateValue(c, private)
\end{aligned}
$$

We recommend the following values for the three thresholds of our algorithm. We recommend $minHeight = 2$ to avoid single identifiers to match everywhere. $maxSize$ is used in the recovery part of Algorithm 2 that can trigger a cubic algorithm. To avoid long computation times we recommend to use $maxSize = 100$. Finally under 50% of common nodes, two container nodes are probably different. Therefore we recommend using $minDice = 0.5$

3.3 Complexity Analysis

Our algorithm has a worst-case complexity of $O(n^2)$ where $n = \max(|T_1|, |T_2|)$. Indeed, Algorithm 1 performs in the worst-case a Cartesian product of nodes with identical heights. Since the isomorphism test we use is in $O(1)$ thanks to hashcodes proposed in [7], the whole algorithm is $O(n^2)$. Moreover with real ASTs this worst-case is very unlikely to happen. Algorithm 2 also performs a Cartesian product of unmatched nodes in the worst-case. This operations is also $O(n^2)$ because all sub-operations are bounded, even the cubic algorithm *opt* which is only applied on trees smaller than a fixed size. Finally the algorithm that computes the edit script from the mappings, described in [6], also has a $O(n^2)$ worst-case complexity.

4. TOOL

The algorithm described in the previous section has been implemented in a freely-available and extensible tool. AST differencing requires parsers (that produce the AST representation) to support a given programming language. This is clearly a constraint, since new languages do not work out of the box. Another interesting challenge faced by such a tool is that it is used by different actors with different expectations, such as a developer that wants a neat graphical display of the edit script to quickly understand it, or a researcher that wants the results in a structured format that can be processed automatically. In this section we present our AST differencing tool, that allows to integrate new programming languages, differencing algorithms and ways of providing results.

4.1 Architecture

Our tool uses a *pipe and filter* architecture shown in Figure 2. Two input files are transformed into two ASTs by a parser. Since parser is an abstract module, several concrete

Algorithm 2: The algorithm of the bottom-up phase.

```
Data: Two trees T1 and T2, a set M of mappings (resulting
      from the top-down phase), a threshold minDice and a
      maximum tree size maxSize
Result: The set of mappings M.
1  foreach t1 ∈ T1 | t1 is not matched ∧ t1 has matched
   children, in post-order do
2      t2 ← candidate(t1, M);
3      if t2 ≠ null and dice(t1, t2, M) > minDice then
4          M ← M ∪ (t1, t2);
5          if max(|s(t1)|, |s(t2)|) < maxSize then
6              R ← opt(t1, t2);
7              foreach (ta, tb) ∈ R do
8                  if ta, tb not already mapped and
                   label(ta) = label(tb) then
9                      M ← M ∪ (ta, tb);
```

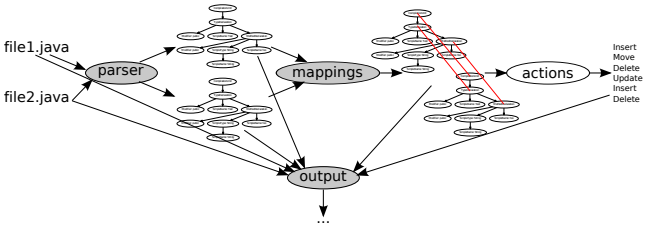

Figure 2: Our pipe and filter architecture. Abstract modules are greyed.

implementations can be furnished (such as Java or C). These two ASTs are then given to an abstract mappings module that computes as output a set of mappings. Since this module is also abstract, several concrete algorithms (such as GumTree or ChangeDistiller [13]) can be provided. Finally this set of mappings is given to an actions module that computes the actual edit script. The input files, ASTs, mappings, and edit script are finally given to an abstract output module. Since this module is abstract, several outputs can be provided (e.g., XML, JSON, ...). Note that all the data structures are given to the output module; it can therefore operate on any of them (for instance it can produce the XML of an AST or of an edit script).

Using this architecture, we have been able to integrate the Java (using the Eclipse JDT parser), JavaScript (using the Mozilla Rhino parser), R (using the FastR parser [17]) and C (using the Coccinelle parser [26]) programming languages. We have also integrated the GumTree, ChangeDistiller [13], XYDiff [8] and RTED [27] algorithms. Finally we can produce the following outputs: a graphviz representation of an AST, a XML representation of an AST, a web-based view of an edit script (shown in Figure 3), and a XML representation of an edit script.

4.2 Runtime Performances

In this section, we want to assess the runtime performances of our tool on real data. As explained in the previous section, our tool applies the differencing algorithm on ASTs parsed from two versions of a source code file. We have integrated several parsers into our tools. We use the Java and JavaScript parsers in this section.

Test.java (source)

```
public class Test {
        public String foo(int i) {
                if (i == 0) return "Foo!";
        }
}
```

Test.java (destination)

```
public class Test {
        private String foo(int i) {
                if (i == 0) return "Bar";
                else if (i == -1) return "Foo!";

        }
}
```

Figure 3: The web-based diff view of our tool.

To gather representative data to assess our tool, we selected arbitrarily two mature, popular and medium-to-large sized projects. For the Java language, we use Jenkins (a continuous integration server) and for JavaScript we use JQuery (a DOM manipulation library). We arbitrarily selected a complete release of each project, and extracted each file modification performed in the commits corresponding to this release. In Jenkins, we use the release 1.509.4 → 1.532.2 where we extracted 1 144 modifications. In JQuery, we use the revision 1.8.0 → 1.9.0 where we extracted 650 modifications. Each modification consists in a pair of files (previous version and next version). They have been extracted thanks to the Harmony platform [12].

In this performance study, we want to assess two important aspects: running time and memory consumption. We use a MacBook Pro retina with a 2.7GHz Intel Core i7 with 16 Gb of RAM. To have reference measures, we use three other tools in addition to our tool. The complete list of tools we use is:

- A classical text diff tool, that computes an edit script with add and delete actions on text lines. As explained in Section 6, this tool is very fast and therefore represents the lower bound for a code differencing algorithm. In our experiment, we use the Google implementation[3].
- The parser included in GumTree which only parse the two files involved in the modification without applying AST differencing algorithms. As parsing the files is mandatory to perform AST differencing, it represents the lower bound for an AST differencing algorithm. In our experiment, we use Eclipse JDT parser to parse Java files, and Mozilla Rhino parser to parse JavaScript files.
- The GumTree algorithm (including parsing), with the following thresholds: $minHeight = 2$, $minDice = 0.5$ and $maxHeight = 100$.
- The RTED algorithm (including parsing), that computes an edit script on an AST with add, update and delete actions. As explained in Section 6, RTED has a cubic worst-case complexity (n^3). Therefore it represents an upper bound for AST differencing. In our experiment we use the implementation provided by Pawlik et al.[4] in our framework.

We only compare GumTree to text diff and RTED because we have re-implemented the other algorithms included in our tool by following the description of the articles, but with no particular care for optimization. Therefore, reporting memory consumption or running times for these algorithms would not be fair.

For the memory consumption, we ensure that the tools can run using 4Gb of RAM, a common amount of memory in modern computers. To that extent, we use a Java virtual machine bound to 4Gb of memory. We run each tool on each modification, and count the number of modifications leading to an out of memory error. In this experiment the only tool that underwent out of memory errors is RTED with 82 errors

[3] `code.google.com/p/google-diff-match-patch`
[4] `www.inf.unibz.it/dis/projects/tree-edit-distance`

(around 5% of the modifications). Even though this number is not so high, it still shows that the complexity of RTED lead to a very expensive memory consumption in some cases.

For the running time we perform two experiments. In the first experiment, we investigate if the tools are capable of computing an edit script of a modification in less than 10 seconds. After 10 seconds, we believe that the tools will not be used interactively by developers. To that extent, we run each tool on each modification and count the number of cases where the execution lasted more than 10 seconds. In this experiment, only RTED underwent such cases, 206 times (around 12% of the cases with no out of memory error). Therefore, on our data, RTED is not capable of computing an edit script for around 17% of the cases, which is a large number of cases. It clearly shows that the complexity of this algorithm is not suitable to real data.

In the second experiment, we compare the running times of the tools. To compute the running times, we compute the edit scripts ten times for each modification, and we retain the median of these values. To avoid noise in the measures, we ensure that the JAVA virtual machine is hot by running each algorithm a hundred times on a random modification, i.e., that no more dynamic loading is involved and that the JIT compiler has compiled and installed the code corresponding to hot-spots. We also pre-load all files involved in the modifications to avoid IO latencies. To be able to compare the tools on the same dataset, we discarded all the modifications that led to a out of memory error or an execution timeout (execution lasting more than 10 seconds) for at least one tool. To present the values, we use the running time of text diff as a reference value, since it is the faster existing tool. Therefore for each modification, we divide the running time of Parsing, GumTree and RTED tools by the running time of the text diff tool. This ratio represent the number of times that the tool is slower than performing a text differencing. We then present the boxplots of the distributions of these resulting ratios.

Figure 4 shows the results of the second experiment. The first interesting conclusion is that just parsing the files is significantly longer than performing a text differencing: the median of parsing time ratios is 10. Additionally, we see that computing an edit script with GumTree is only slightly slower than just parsing the files (median at 18 for Jenkins and 30 for JQuery). The difference between Jenkins and JQuery medians indicates that JAVASCRIPT ASTs are likely to contain more nodes than JAVA ASTs. Finally we see that RTED is significantly slower than just parsing the files (median at 298 for Jenkins and 2 654 for JQuery). The difference between the two medians is also observed for the RTED tool.

As a conclusion, we clearly see that text diff tool is by far the fastest. However, performing AST differencing with GumTree induces only a small overhead over parsing the files. It means that our algorithm is fast and can therefore be applied on real data. The mean running times of GumTree are 20 milliseconds on Jenkins and 74 milliseconds on JQuery. Our experiments also confirm that using RTED on real data induces a huge overhead compared to text diff.

5. EVALUATION

We now present the empirical evaluation of GumTree. Our goal is to answer the following research questions: RQ1) Does GumTree produce tree differences that are correct and better than Unix diff (5.1)? RQ2) Does GumTree maximize the number of mappings and minimize the edit script size compared to the existing algorithms (5.2)? RQ3) Does GumTree detect move operations better than ChangeDistiller (5.3)? We discuss the threats to the validity of our results in 5.4.

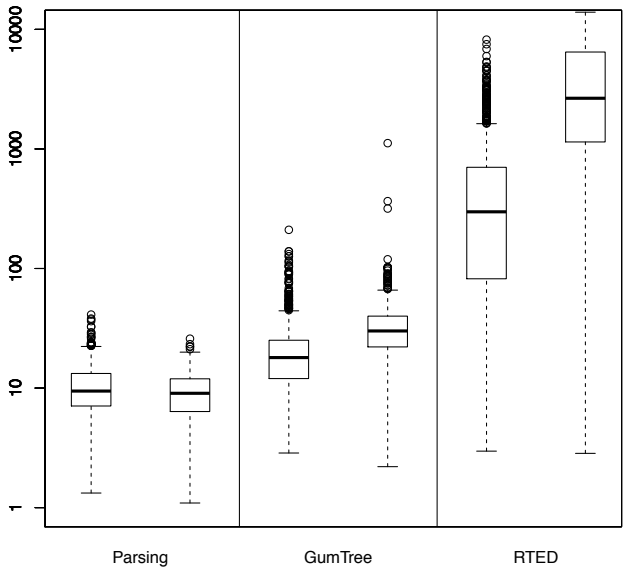

Figure 4: Distribution of the running time ratios of the tools. For each tool, the left boxplot shows the ratios for Jenkins, and right one for JQuery. The running time ratio is the running time of the tool divided by the running time of text diff. It represents the number of times it is slower than text diff.

5.1 Manual Evaluation

First, we consider the viewpoint of the developer. For her, what matters is that the computed edit script is good to understand the essence of a commit. In this perspective, we have developed an experiment that has two main goals: 1. to evaluate the correctness of GumTree, and 2. to compare GumTree against text line-based differencing.

5.1.1 Overview

The experiment consists in the manual evaluation of file differences, i.e. on the manual assessment of a file pair difference (the difference between the version before and the version after the commit). These file differences are computed using two techniques; GumTree and a state of the art text differencing tool[5]. We will refer to the text differencing tool as diff, and to GumTree as GT. For each file pair of a dataset, the outputs from both approaches are given to a human evaluator. He/she compares both outputs and then answers to the following questions: *Is the GumTree output correct?* and *Which technique yield the most understandable differencing information: GumTree or diff?*.

For example, the revision 1.15 of the class `ARecord` of the DNSJava project[6] introduces a new parameter (`int index`) in the method called `rrToWire`. The diff output is a source code hunk pair: the left hunk is composed of the line cor-

[5] `mergely.com`

[6] `sf.net/projects/dnsjava`

responding of the previous method signature—the entire line—from the revision 1.14. The right hunk corresponds to the updated method signature from revision 1.15. The GumTree output states that there are one new parameter type and one new parameter name. In this case, the evaluator may decide that GumTree more precisely localizes and describes the change introduced by the revision 1.15 compared to Unix diff.

5.1.2 Experiment Setup

A commit is composed of a set of modified files. After a commit, each modified file is said to have a new *Revision*. In our experiment, each file pair corresponds to consecutive file revisions. We used stratified sampling to randomly select revisions from the software history of 16 open source projects (from [23]). We only consider revisions with few source code changes (those revisions for which the ChangeDistiller differencing algorithm states that there is only one single source code change). We pick 10 items (file pairs) per project (if 10 simple revisions are found otherwise less). In total, the dataset contains 144 transactions.

Then, we create an *evaluation item* for each pair of files of the evaluation dataset. An evaluation item contains: the GumTree output between the revision pair of the transaction, the diff output between the same pair, and the commit message associated with the transaction.

The diff output shows two files (i.e. called left and right) and highlights the changes made per line. In particular, it highlights the lines deleted from the left file, the lines added from right file. Note that we have configured diff to discard whitespaces. The GumTree output (shown in Figure 1) highlights the added, deleted, updated and moved AST nodes. The commit message describes the intention of the change, it sometimes help to meaningfully assess the relevance of both differencing algorithms.

The 144 evaluation items were independently evaluated by three authors of this paper called the *raters*. All 3 raters evaluated all the edit scripts of 144 file pairs at the AST and line level (i.e. 288 outputs). This makes a total of $3 \times 2 \times 144 = 864$ ratings. The rater has to answer to the following questions:

- Question #1: *Does GumTree do a good job?*
  The possible answers are:
  1. GumTree does a good job: it helps to understand the change.
  2. GumTree does a bad job.
  3. Neutral.
- Question #2: *Is GumTree better than diff?*
  The possible answers are:
  1. GumTree is better.
  2. diff is better.
  3. GumTree is equivalent to diff.

Optionally, the rater could write a comment to explain his decision. Those comments are used to identify buggy or corner cases where GumTree could be improved.

		Full (3/3)	Majority (2/3)
#1	GT does good job	122	137
	GT does not good job	3	3
	Neutral	0	1
#2	GT better	28	66
	Diff better	3	12
	Equivalent	45	61

Table 1: Agreements of the manual inspection of the 144 transactions by three raters for Question #1 (top) and Question #2 (bottom).

5.1.3 Experiment Result

Table 1 (top) presents the number of agreements for the first question. Let us consider question #1, the three raters fully agreed for 122/144 (84.7%) file pairs to say that GumTree does a good job in explaining the change. If we consider the majority (at least 2/3 agree), it has been assessed that GumTree has a good output for 137/144 file pairs (95.1%) .

Table 1 (bottom) presents the number of agreements for the second question. In 28/144 (19,4%) evaluation items, there was a full agreement to say that GumTree better highlights the changes between two files. In 45/144 (31%) items the raters fully agreed the GumTree's output is as good as the one of diff to explain the change. This shows that intuitively, GumTree is a tool that has added value compared to diff. Beyond those raw numbers let us now measure the statistical level of agreement.

5.1.4 Statistics

Let us assume that p_i measures the degree of agreement for a single item (in our case in $\{\frac{1}{3}, \frac{2}{3}, \frac{3}{3}\}$. The overall agreement $\bar{P}$ [9] is the average over p_i where in $i \in \{1, \ldots, 144\}$. The coefficient κ (Kappa) [9, 16] measures the confidence in the agreement level by removing the chance factor[7]

1. For Question #1:
   We have $\bar{P} = 0.905$. Using the scale introduced by [18], this value means there is an *almost perfect agreement*. The κ degree of agreement in our study is 0.321, a value distant from the critical value, that is 0. The null hypothesis is rejected, the observed agreement was not due to chance.

2. For Question #2:
   We have $\bar{P} = 0.674$. Using the mentioned scale, this value means there is a *substantial* overall agreement between the rates. The κ degree of agreement in our study is 0.426, far higher that the critical value. The null hypothesis is rejected, the observed agreement was not due to chance.

5.1.5 Conclusion

The manual rating of 144 revisions by 3 independent raters shows that 1) GumTree can be used to compare two JAVA files in order to understand the essence of the change and 2) its output is sometimes more understandable than the one from diff. There is a statistically significant level of agreement between raters for both results.

[7]Some degree of agreement is expected when the ratings are purely random [9, 16].

5.2 Automatic Evaluation

We are now confident that GumTree is good from the viewpoint of a human developer. We now assess whether GumTree maximizes the number of mappings and minimizes the edit script. Compared to the previous manual evaluation, this evaluation is fully automatic. Consequently, it can be performed on a large scale.

5.2.1 Goal and measures

The goal of this experiment is to measure the performance of tree differencing algorithms with respect to:

1. the number of mappings;
2. the edit script size;

We compare GumTree against the two major differencing algorithms (as of today): ChangeDistiller [13] and RTED [27]. Other algorithms exist but they have less impact compared to those two (in terms of publication visibility or citations). For a description of ChangeDistiller and RTED, please refer to Section 6.

As explained in Section 6, ChangeDistiller uses a simplified ASTs where the leaf nodes are code statements. Therefore, we compute the metrics for both simplified ASTs (as described in [13]) and raw ASTs generated by the Eclipse JDT parser. In the remainder of the section these granularities are called respectively CDG (ChangeDistiller granularity) and JDTG (Eclipse JDT granularity). Our motivation is to compare GumTree and ChangeDistiller algorithms, even on CDG ASTs: it would have been unfair to claim anything on ASTs that would be different from those for which ChangeDistiller is designed and optimized. Since the goal of GumTree is to work on fine-grained ASTs, we evaluate the performance metrics on this granularity as well. Finally for the sake of comparison, we also evaluate RTED on fine grain AST, since that is the granularity GumTree is designed for.

5.2.2 Procedure

The experiment consists in comparing source code file pairs using several AST differencing algorithms. We take a sample of 1 000 revision pairs from 16 JAVA open source projects of the CVS-Vintage dataset [23]. For revision pairs, we create 4 tree representations (before and after the commit and for both kinds of ASTs we consider). Then, we run the competing tree differencing algorithms for each pair of ASTs. Finally, we measure the execution time, we count the mapped nodes, and we save the edit scripts. For sake of performance, we discard the revision pairs of large source code file, i.e. whose tree representations have more than 3 000 nodes (mainly because the ChangeDistiller algorithm is too slow for large ASTs). In total, this results in 12 792 evaluation cases.

Replication information: For GumTree, we use the following thresholds: $minHeight = 2$, $minDice = 0.5$ and $maxSize = 100$. For ChangeDistiller, we use a label similarity threshold of 0.5. For unmatched nodes, we use two similarity thresholds: 0.6 for inner nodes with more than 4 children and 0.4 for the rest.

5.2.3 Experiment Result

The results are presented in Table 2. The upper part of the table presents the performance comparison of ChangeDistiller and GumTree at CDG granularity; while the middle part shows them at the JDTG granularity (finer grain, more AST nodes). Finally the lower part compares GumTree and RTED differencing algorithms at the JDTG granularity. Each cell of these tables presents the number of cases where an approach is better than the other for a given AST granularity and measure. We now analyze the experimental results by metric.

		GT better	CD better	Equiv.
CDG	Mappings	4007 (31.32%)	542 (4.24%)	8243 (64.44%)
	ES size	4938 (38.6%)	412 (3.22%)	7442 (58.18%)
		GT better	CD better	Equiv.
JDTG	Mappings	8378 (65.49%)	203 (1.59%)	4211 (32.92%)
	ES size	10358 (80.97%)	175 (1.37%)	2259 (17.66%)
		GT better	RTED better	Equiv.
	Mappings	2806 (21.94%)	1234 (9.65%)	8752 (68.42%)
	ES size	3020 (23.61%)	2193 (17.14%)	7579 (59.25%)

Table 2: Number of cases where GumTree is better (resp. worse and equivalent) than ChangeDistiller (top, middle) and RTED (bottom) for 2 metrics, number of mappings and edit script size (ES size), at the CDG granularity (top) and JDTG granularity (middle, bottom).

Mappings.

As explained in Section 2, finding the mappings is the most important step in an AST differencing algorithm. Finding more mappings increases the odds of deducing a short edit script. Considering the CDG granularity, in 4 007 (31.32%) cases, GumTree matches more nodes than ChangeDistiller. Then, in 8 243 cases (64.44%) both approaches find the same number of mappings. At the JDTG granularity (finer grain), in 8 378 (65.49%) cases, GumTree matches more nodes than ChangeDistiller. In 4 211 cases (32.92 %) the number of mappings is the same. At both granularities, GumTree matches more nodes than ChangeDistiller.

When comparing GT against RTED, in most of the cases 8 752, (68.42%) the same number of mappings is found. However, GumTree finds more mappings, which is twice many better than the opposite 2 806 (21.94%) vs 1 234 (9.65%).

Edit Script Size.

Once the mappings are found, an edit script is computed. The length of the edit script is a proxy to the cognitive load for a developer to understand the essence of a commit. Hence, the goal is to minimize the size of edit scripts.

Considering the CDG granularity, the size of edit scripts of GT and ChangeDistiller are the same in 7 442 cases (58.18%). In 4 938 (38.6%) cases, the edit script from ChangeDistiller is longer than the one of GumTree (i.e. GumTree is generally better). For the JDTG granularity, it is the same, ChangeDistiller often produces bigger scripts: in 10 358 (80.97%) cases (versus 175 cases (1.37%) where it performs better than GumTree).

The comparison between GT and RTED shows in most of the cases (59.25%) the edit script size is the same, and in 23.6% of the cases GT produces shorter edit script than RTED.

According to our dataset, GumTree systematically produces shorter edit scripts, which is better to understand the meaning of a commit.

5.3 Analysis of Move Actions

Both GumTree and ChangeDistiller are able detect move node actions. This section presents an analysis of move actions found by the GumTree and ChangeDistiller matching algorithms. The goal of this experiment is to check how these algorithms detect move actions.

The evaluation metric can not be an absolute number of move detected operations. The reason is twofold. On the one hand, one wants to maximize the number of moves (instead of having additions and deletions). On the other hand, one wants to minimize the number of spurious detected move actions that have nothing to do with the conceptual change.

Consequently, we need a more subtle evaluation scenario. We propose to compare the number of moves by stratifying over the results of both algorithms. For instance, if ChangeDistiller is able to completely explain a commit with only move actions, GumTree should also find an edit script that is uniquely composed of moves. In this case, one can reasonable think that the edit script with the smallest number of moves is the best. So we compare the results for a number of different sub cases.

5.3.1 Procedure

We analyze move actions from the differencing of JAVA file pairs of the dataset introduced in Section 5.2. We focus on move actions from edit scripts produced by ChangeDistiller (CD) and GumTree (GT). In this experiment we do not consider RTED because this algorithm does not identify move actions. We select those JAVA file pairs for which the edit script from ChangeDistiller *or* GumTree is only composed of move actions. From the initial dataset of 12 792 file pairs, this results in 130 elements.

Then, to compare the edit scripts, we classify each pair of edit script (ChangeDistiller versus GumTree) in the following categories.

1. ChangeDistiller and GumTree produce only moves. Both have the same number of actions, i.e. they are equivalent (top-left).
2. ChangeDistiller and GumTree produce only moves, but in different numbers (top-left).
   (a) GumTree with more moves.
   (b) ChangeDistiller with more moves.
3. ChangeDistiller produce only move actions, and GumTree other actions which can include moves (top-right).
4. GumTree produce only move actions, and ChangeDistiller other actions (bottom-left).

The analysis of the number of items in each category enables us to settle the question of the effectiveness of the detection of move actions.

5.3.2 Experiment Result

The results are presented in Table 3. There are 77 comparisons for which both matching algorithms produce only move actions, 58 out of these 77 cases correspond to the case 1 where both algorithms exactly produce the same edit script (same number of moves). Then, there are 18 instances where ChangeDistiller has more move actions (case 2-b). It remains one case where GumTree produce more moves (case 2-a). This shows that GumTree edit scripts are *more concise* to describe move actions than the ones of ChangeDistiller.

	GT only move op	GT other op
CD only move op	77	1
CD other op	52	12662

Table 3: Comparison of the number of move operations from GumTree and ChangeDistiller for 12 792 file pairs to be compared.

Moreover, there are 52 differencing scenarios where GumTree produces only move actions while ChangeDistiller produces other kinds of actions (case 4). In these cases ChangeDistiller has other actions (e.g. one node addition and one node deletion) in addition to a move. This means that GumTree is *more precise* to represent changes involving move actions. To sum up, according to our dataset, GumTree is better than ChangeDistiller at detecting move actions; it is both more concise and more precise.

5.4 Threats to Validity

We now discuss the threats to the validity of our evaluation set up. We first discuss those that are specific to a research question and then the generic ones.

For the manual analysis, the main threat to validity is that the raters are also authors of this paper. To reassure the reader, the evaluation dataset is made publicly available[8].

For the comparative analysis of the number of mappings and move operations between different tools, the main threat is a potential bug in our implementation. In particular, we have re-implemented ChangeDistiller because the original implementation needs an Eclipse stack. Our new implementation may not reflect all specific implementation decisions of the original implementation or even introduce new bugs. For sake of future analysis and replication, our implementation of the competitors is in the same repository as GumTree. We also note that we have experimented with the original ChangeDistiller in several experiments (for instance [20]) and have confidence that our implementation reflects the original one.

Our experiments only consider edit scripts of Java files. This is a threat to the external validity. Despite unlikely (our algorithm is independent of any Java specificity), we can not conclude on whether GumTree performs so well on other programming languages.

Finally the three thresholds of GumTree have been fixed to the following values: $minHeight = 2$, $maxSize = 100$ and $minDice = 0.5$. These values have been chosen according to our expertise. However, other values could perform differently. More experiments are needed to evaluate their impact on the runtime efficiency and the algorithm results.

6. RELATED WORK

In this section we present the related work on code differencing, from text to graph granularity.

Text Differencing.

Computing differences between two versions of a source file is most commonly performed at the text line granularity [24, 22]. Within this granularity, the edit actions are insertion,

[8] `www.labri.fr/~falleri/dist/articles/GumTree`

deletion or update of a text line. The more advanced algorithms are even capable of detecting moved lines [29, 5, 3]. The algorithms using this granularity are usually very fast and completely language independent, compared to our approach which requires a parser that slows down the whole process. The main issue with these algorithms is that they cannot compute fine-grained differences. Indeed in many languages (such as JavaScript or even Java) a text line can contain many programming constructs. Additionally the output of these algorithms is very difficult to process automatically since it composed of source code lines that might not be parsed, since they might be incomplete. It is therefore difficult to extract automatically the syntactic modifications using these approaches.

Tree and AST Differencing.

The tree differencing problem has been largely investigated when considering only the add node, delete node and update node actions [4]. For this problem, many optimal algorithms have are described in the literature. The fastest algorithms of this family [27] run in $O(n^3)$, which can result in significantly long edit script computation time for large source code files. The other issue faced by these algorithms is their inability to uncover moved nodes, which is a frequent action in source code files. It results in unnecessarily big edit scripts which are hard to understand.

When considering move node actions, the problem of finding the shortest edit script between two trees becomes NP-hard. However, several algorithms from the document engineering or software engineering research fields that use practical heuristics exists in the literature. One of the most famous is the algorithm of Chawathe et al. [6] that computes edit scripts (containing move actions) on trees representing LaTeX files. Unfortunately, this algorithm has constraints (acyclic labels and leaf nodes containing a lot of text) that do not apply to fine grained ASTs of general purpose programming languages. Several algorithms have also been designed specifically for XML documents [8, 1]. Unlike the algorithm of Chawathe et al., they do not have any particular constraint. However these algorithms put a particular emphasis on the edit script computation time because they are mostly used for automatic on-the-fly compression. Regarding our objective, the most important thing is to compute an edit script that reflects well the developer intent, computation time is only secondary. GumTree is inspired from the algorithm of Cobena et al. [8], because we apply a very similar first phase. The major difference is that they are not interested in having fine-grained differences since the differencing is computed only for compression purpose. Our algorithm is much more accurate since it also performs a second phase that increase the number of mappings found, and therefore produces shortest edit scripts at the expense of the running time.

The most famous algorithm that works on ASTs is ChangeDistiller [13]. It is largely inspired by the one of Chawathe et al. but tuned to work better on ASTs. However, this algorithm is still based on the assumption that leaf nodes contain a significant amount of text. Therefore, the authors use simplified ASTs where the leafs are in fact code statements, rather than raw ASTs. Therefore ChangeDistiller will not compute fine-grained edit scripts on languages that can have a lot of elements in statements (such as JavaScript). The Diff/TS algorithm [15] is able to work on raw ASTs. The automatic experiment performed in the orginal article shows that it produces efficiently short edit scripts. However the results of this algorithm have not been validated by humans. The VDiff algorithm [10] generates edit scripts from Verilog HDL files. It is slightly similar to the first phase of GumTree, but it uses also the lexical similarity of the code. However, the generated edit scripts are specific to the VHDL language. Finally the JSync [25] algorithm is also able to compute edit scripts that include move actions. However it relies on a classical text differencing algorithm applied on the unparsed ASTs as first step. It therefore limits its ability to find moved nodes. Additionally, it is focused on producing information on clones rather than edit scripts.

Graph Differencing and Origin Analysis.

There are several algorithms that go beyond the AST structure and compute edit scripts on graphs representing the source code [28, 31, 2, 11]. Although these algorithms can uncover semantic differences, they are significantly harder to use in practice. Mainly because they require much more semantic information on the source code (such as program dependency graphs, class models, meta-models or control-flow graphs) which is very hard if not impossible to obtain in many languages (such as JavaScript or C), or when considering only program fragments (e.g., plug-ins).

Finally there are also many algorithms that perform the so-called origin analysis (e.g., [14, 30]). These algorithms output the matching program elements between two versions. They usually use lexical as well as structural similarities between the elements. However they only consider a few kind of program elements (usually class, functions and attributes) and do not output edit scripts.

7. CONCLUSION AND FUTURE WORK

In this article, we present a novel algorithm that computes fine-grained edit scripts on ASTs, including move node actions. Our algorithm is implemented in a freely-available and extensible tool. We have evaluated the running time and memory consumption of our tool, and shown that it is reasonable on real data and can be used on a daily basis. We also have performed an empirical evaluation of the results of our tool. This evaluation shows that the results of our algorithm are good, and often more comprehensible than the result of a classical text diff.

As future work, we plan to extend our tool to extract modifications that are performed *across files*. We also plan to introduce new algorithms that can automatically process an edit script of GumTree to produce higher-order edit scripts (for instance to identify refactorings). Finally as the bottleneck of this approch is the parsing, we consider moving to more fuzzy parsers, in order to accept not well formed file and reduce the parsing time.

8. REFERENCES

[1] R. Al-Ekram, A. Adma, and O. Baysal. diffX: an algorithm to detect changes in multi version XML documents. In *CASCON*, page 1–11, 2005.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th International Conference on Automated Software Engineering*, ASE '04, page 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[3] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. D. Penta. LHDiff: a language-independent hybrid approach for tracking source code lines. In *International Conference on Software Maintenance, Eindhoven*, pages 230–239, 2013.

[4] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[5] G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: A language-independent approach. *Software, IEEE*, 26(1):50–57, Jan. 2009.

[6] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 International Conference on Management of Data*, pages 493–504. ACM Press, 1996.

[7] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *The 17th International Conference on Program Comprehension*, pages 243–247. IEEE Computer Society, 2009.

[8] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In R. Agrawal and K. R. Dittrich, editors, *Proceedings of the 18th International Conference on Data Engineering*, pages 41–52. IEEE Computer Society, 2002.

[9] J. Cohen et al. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[10] A. Duley, C. Spandikow, and M. Kim. Vdiff: a program differencing algorithm for verilog hardware description language. *Automated Software Engineering*, 19(4):459–490, 2012.

[11] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In *Proceedings of the 11th International MoDELS Conference*, pages 326–340. Springer, 2008.

[12] J.-R. Falleri, C. Teyton, M. Foucault, M. Palyart, F. Morandat, and X. Blanc. The harmony platform. *CoRR*, abs/1309.0456, 2013.

[13] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[14] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[15] M. Hashimoto and A. Mori. Diff/TS: a tool for fine-grained structural change analysis. In A. E. Hassan, A. Zaidman, and M. D. Penta, editors, *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 279–288. IEEE, 2008.

[16] F. L. Joseph. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378–382, 1971.

[17] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A fast abstract syntax tree interpreter for R. In *VEE*, pages 89–102. ACM Press, 2014.

[18] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, Mar. 1977.

[19] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, Sept. 1984.

[20] M. Martinez and M. Monperrus. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Software Engineering*, Online First, Sept. 2013. Accepted for publication on Sep. 11, 2013.

[21] T. Mens and S. Demeyer. *Software Evolution*. Springer, 1 edition, 2008.

[22] W. Miller, Eugene, and W. Myers. A file comparison program. *Software: Practice and Experience*, page 1040, 1985.

[23] M. Monperrus and M. Martinez. CVS-Vintage: A Dataset of 14 CVS Repositories of Java Software.

[24] E. W. Myers. An o(ND) difference algorithm and its variations. In *Algorithmica*, pages 251–266, 1986.

[25] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Trans. Software Eng.*, 38(5):1008–1026, 2012.

[26] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: ten years later. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, page 305. ACM Press, 2011.

[27] M. Pawlik and N. Augsten. RTED: a robust algorithm for the tree edit distance. *PVLDB*, 5(4):334–345, 2011.

[28] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *20th International Conference on Software Maintenance*, pages 188–197. IEEE Computer Society, 2004.

[29] S. Reiss. Tracking source locations. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 11–20, May 2008.

[30] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 325–334, New York, NY, USA, 2010. ACM.

[31] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, page 54–65, New York, NY, USA, 2005. ACM.