Imprecisions Diagnostic in Source Code Deltas

Guillermo de la Torre*
DCC, University of Chile
Santiago, Chile
gdelator@dcc.uchile.cl

Romain Robbes
SwSE Research Group, Free
University of Bozen-Bolzano
Bolzano, Italy
rrobbes@unibz.it

Alexandre Bergel
DCC, University of Chile
Santiago, Chile
abergel@dcc.uchile.cl

ABSTRACT

Beyond a practical use in code review, source code change detection (SCCD) is an important component of many mining software repositories (MSR) approaches. As such, any error or imprecision in the detection may result in a wrong conclusion while mining repositories. We identified, analyzed, and characterized impressions in GumTree, which is the most advanced algorithm for SCCD. After analyzing its detection accuracy over a curated corpus of 107 C# projects, we diagnosed several imprecisions. Many of our findings confirm that a more language-aware perspective of GumTree can be helpful in reporting more precise changes.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; Software maintenance tools;

KEYWORDS

source code change detection, differencing, quality, GumTree

ACM Reference Format:
Guillermo de la Torre, Romain Robbes, and Alexandre Bergel. 2018. Imprecisions Diagnostic in Source Code Deltas. In *MSR '18: 15th International Conference on Mining Software Repositories, May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196398.3196404

1 INTRODUCTION

Adequately characterizing source code changes across multiple software revisions is an activity essential both in software development and in MSR research. For instance, developers routinely review source code changes before deciding whether to integrate them, using tools such as pull requests in Git [12]. In MSR research, approaches based on change data are too numerous to list (Kagdi provides a summary of early MSR research [16]).

The version control systems commonly used by developers treat source code as textual files and thus discard the semantics of the programming language [23]. As a consequence, a versioning system such as Git can only express changes in terms of textual content whereas a code change may express a complex structural changes, such as a code refactoring. To address that limitation, the modern approaches to SCCD [9, 10] instead operate on the source code's abstract syntax tree (AST): these approaches compute the differences between two successive versions of the code (two ASTs) and express changes in terms of an edit script, a sequence of operations to transform the first AST into the second.

While these approaches are a vast improvement over text-based change detection, even the most advanced techniques of AST-based SCCD suffer from imprecisions, of which some examples are found in Section 2. As SCCD is an early step in many MSR approaches, imprecisions during that step have the potential to carry over, leading to variations in their results [15, 17, 28]. To improve on the state of the art, developing an understanding of the issues affecting SCCD is crucial (for additional background on SCCD, see Section 3).

The goal of this paper is two-fold: 1) to gain a better understanding of the kind of imprecisions that affect SCCD algorithms (Section 4), and 2) to estimate the impact of the said imprecisions by developing heuristics to detect these imprecisions in a large corpus of source code (Sections 5–8). The subject of our study is GumTree [9], a state of the art SCCD algorithm, applied to a corpus of 107 C# projects. This paper investigates two research questions:

Research Question #1: *What are the issues affecting the GumTree algorithm?* In order to better understand the imprecisions that affect GumTree, we first analyzed 86 file revision pairs originating from a single project. Out of these 86 pairs, we found 23 cases (27%) where GumTree produces incorrect or sub-optimal results. We also categorized the imprecisions found in 4 categories: Spurious changes, Arbitrary Changes, Redundant Changes, and Ghost Changes. This initial evidence suggests that there is still lots of room to improve upon the state of the art in SCCD.

Research Question #2: *What is the extent of the issues identified during RQ1?* While the initial evidence gathered in the first part of the paper is enough to give us an intuition on the issues that affect SCCD algorithms and GumTree in particular, it is not enough. An extensive diagnostic is needed in order to gauge the relative importance of the issues encountered. Without such a diagnostic the improvements depend more of the intuition and will not necessarily be tackling the most relevant issues. To do so, we developed heuristics to detect the imprecisions that we identified, and ran them on a large source code corpus (107 systems, 143,419 file revision pairs). We then manually evaluated the accuracy of the heuristics on a subset of the result. This allowed us to: 1) confirm our preliminary finding that there is room for improvement in SCCD, and 2) estimate the impact of each issues in the corpus and get a sense of their relative importance.

Section 9 concludes our diagnostic by summarizing the issues found and commenting on possible solutions that deserve to be

*Guillermo de la Torre is supported by CONICYT-PFCHA/Doctorado Nacional/2018-21181919 (Chile).

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196404

studied in a future research effort. One of the key reasons for the issues that we observed is that GumTree under exploits the syntax and the semantic of the source code. In the same manner that versioning systems treat source code as "just text", GumTree treats source code as "just an AST". Adding more knowledge about the syntax and semantics of the programming language to an SCCD algorithm should allow it to make better decisions.

Finally, we close this paper by documenting the limitations of this study (Section 10), and presenting related work (Section 11), before concluding (Section 12).

2 MOTIVATING EXAMPLE

One reason for the suboptimal performance of GumTree is that it considers the AST as "just a tree", and ignores most of the domain knowledge. This domain knowledge includes, for instance, the type of changes that are likely to occur for each type of source code element, as well as common programming conventions.

While conducting our exploratory study (Section 4), we encountered the example shown in Listing I. It highlights parts of pull request #123 made to project `AjaxControlToolkit` [1]. In this listing as well as in Listing II, we colored the code according to the AST changes: deletions, insertions, updates, and moves.

Using GumTree to calculate the differences of file `AjaxFileUpload.cs` yields several imprecisions (Listing I, top), compared to the ideal behavior (Listing I, bottom). In particular, GumTree concludes that `OnInit(EventArgs)` was renamed to `AreFileUploadParamsPresent()`. The latter is in fact a regular method, while the former is an *event handler* which is conceptually very different. In fact, the code in `AreFileUploadParamsPresent` describes only a substep of the original event processing logic. To make matters worse, GumTree infers that a new `OnInit(EventArgs)` event handler was inserted, instead of preserving the identity of the event handler accross versions. The reason for this is that GumTree gives too much weight to the content of the methods, as opposed to their conceptual type.

GumTree's behavior (top) contrasts with the expected behavior (bottom): method `AreFileUploadParamsPresent` is created, and the logical step is moved to it. The consequences are that GumTree found some changes that were unnecessary (the ones colored at top—such as `IsDesignMode` being deleted then inserted—, but in black at bottom, lines 1–3), while others were reported incorrectly (the ones with different colors at top and bottom). The reduced number of changes in the expected behavior makes the code actually added (lines 5–6, bottom right), much easier to spot. Describing the change in this way might have made the pull request reviewer's job easier. Indeed, while reviewing the pull request, the reviewer did mention the code in lines 5–6 (see the second comment of pull request #123 [2]). This clearly indicates that the distinction between regular methods and event handlers is important; failing to take this into account leads to a suboptimal description of the changes. Note that Listing II (Section 4) contains additional examples of imprecisions.

3 BACKGROUND

Source code change detection generally works at the file level. Consider two file version pairs (*file revision pairs*), the first describing an older version (the *original*) and a newer version (the *modified*). We refer to the ASTs of the original and modified versions, as T_1 and T_2, respectively.

A *conceptual element* t_i may exist both in T_1 and T_2 (e.g., the method call on lines 2 at left and right, Listing I), only in T_1 (e.g., the `IF` of line 4 at left), or only in T_2 (e.g., the two statements on lines 5 and 6 at right). The original version of t_i (denoted t_{i1}) is its occurrence inside T_1, while t_{i2} denotes the modified version of t_i inside T_2. That is, $t_i{=}\langle t_{i1}, t_{i2}\rangle$, $t_{i1}{\in}T_1$ (or $t_{i1}{=}\emptyset$ due to t_i does not occur in T_1), and $t_{i2}{\in}T_2$ (or $t_{i2}{=}\emptyset$ due to t_i does not occur in T_2).

If t_i does not occur in T_1 (i.e., $t_i{=}\langle\emptyset, t_{i2}\rangle$) its original version never existed, then t_i was *inserted*. Similarly, if t_i does not occur in T_2 (i.e., $t_i{=}\langle t_{i1}, \emptyset\rangle$) its modified version will never exist again, then t_i was *deleted*. If t_i exists both in T_1 and T_2 (i.e., $t_{i1}{\neq}\emptyset$ and $t_{i2}{\neq}\emptyset$), then t_{i1} may be exactly or approximately equal to t_{i2}. For example, lines 2 at left and right are exactly equal, while the IF's condition in lines 3 at left and right are approximately equal. In the first case, the conceptual element t_i is *unmodified*, so no change should transform t_{i1} into t_{i2}. Otherwise, t_i was *modified*, and part of the reported changes should transform t_{i1} into t_{i2}. In addition, when t_{i1} and t_{i2} belong to different parents or belong to a same parent but have different positions, t_i was *moved* (e.g., the condition on line 4, left, moves to line 9, right).

An *edit script* is made of element insertions, element deletions, and transformations (updates, moves) among elements. The edit script describes how to transform T_1 into T_2 and consequently the changes that occurred. The actions that make up the edit script depend on the approach of change detection used. Classical text-based approaches delete, insert and update textual units, such as lines (e.g. unix diff), characters, or tokens. Tree differencing approaches delete, update and insert nodes, but also move entire subtrees. We use the tree differencing actions defined in Chawathe et al. [4].

GumTree. GumTree [9] is a SCCD approach oriented to the common sense of the developers. GumTree locates the larger modified contexts first, and later identifies concrete changes. To do that, the algorithm prioritizes the matches among bigger subtrees in a top-down pass (comparing their hashed values and using Dice as a tie-breaker). Subsequently, it looks for smaller matches in a bottom-up fashion, in which it also integrates the generic tree differencing algorithm RTED [22] on the smallest subtrees. GumTree works on any source code represented as an AST, for which it needs an ad hoc parser. The support of GumTree[3] for C# relies on SrcML [5].

4 EXPLORATORY STUDY

4.1 Methodology

Project selection. In order to get a better understanding of the issues that affect GumTree, we started this study by analyzing the results of 86 file revision pairs of the `AjaxScriptToolkit` project. The analysis of the differences produced by GumTree is a time intensive process, as it requires a degree of understanding of the project's specificities. In essence the person reviewing the differences has to acquire some domain knowledge about a project to be able to review it. This is why we limited our exploratory study to a single project. Preserving domain knowledge is also the reason why this

[1] the entire pull request is available at: http://tinyurl.com/ybbuh67d
[2] https://tinyurl.com/y9rebqnp

[3] We use a snapshot compiled on 25-may-2017

Listing I: MOTIVATING EXAMPLE (original at left, modified at right)

GumTree's Behavior: It mistakenly matches one method that handles a web page event ("OnInit") with other method that is not an event handler ("AreFileUploadParamsPresent"). This because they share a lot of original code moved there through refactorings. GumTree does not know about conventions as the event handlers, but it relies on the heuristic that the subtrees are very similar.

```
1 protected override void OnInit(EventArgs e){
2   base.OnInit(e);
3   if(!IsDesignMode){
4     if(!string.IsNullOrEmpty(Page.Request.QueryString["contextkey"])
          && Page.Request.QueryString["contextkey"] ==
          ContextKey && Page.Request.QueryString["controlID"]
          == ClientID)
5       IsInFileUploadPostBack = true;
6     }
7 }
```

But, "OnInit" is still present in the version modified! It could have been correctly matched by a simple mapping among same named methods.

```
1 protected override void OnInit(EventArgs e){
2   base.OnInit(e);
3   if(IsDesignMode || !AreFileUploadParamsPresent()) return;
4   IsInFileUploadPostBack = true;
5   var processor = new UploadRequestProcessor {...};
6   processor.ProcessRequest();
7 }
8 bool AreFileUploadParamsPresent() {
9   return
        !string.IsNullOrEmpty(Page.Request.QueryString["contextkey"])
        && Page.Request.QueryString["contextkey"] ==
        ContextKey && Page.Request.QueryString["controlID"] ==
        ClientID;
10 }
```

Expected Behavior: The second `IF` (line 4, left) was deleted, but its condition was moved to a new method (line 8, right). The first `IF` (line 3, left) expanded its condition with an `OR` expression where the new method is called. The line 5 (left) moved to the line 4 (right). The lines 5-6 (right) were inserted.

```
1 protected override void OnInit(EventArgs e){
2   base.OnInit(e);
3   if(!IsDesignMode){
4     if(!string.IsNullOrEmpty ... ["controlID"] == ClientID)
5       IsInFileUploadPostBack = true;
6     }
7 }
```

"OnInit" did not modify its signature, moved the parameter list and the line 2, nor inserted and deleted the body's punctuations.

```
1 protected override void OnInit(EventArgs e){
2   base.OnInit(e);
3   if(IsDesignMode || !AreFileUploadParamsPresent()) return;
4   IsInFileUploadPostBack = true;
5   var processor = new UploadRequestProcessor {...};
6   processor.ProcessRequest();
7 }
8 bool AreFileUploadParamsPresent(){
9   return !string.IsNullOrEmpty ... ["controlID"] == ClientID;
10 }
```

work was carried out only by the first author of this paper, who in addition to being an expert in C#, possesses expertise that the other authors lack.

Corpus selection. `AjaxControlToolkit` has 466 file revision pairs, so we selected a subset of these for inspection. To guide our selection, we used the intuition that file revision pairs where GumTree produces "larger than normal" or "smaller than normal" edit scripts may highlight cases where the algorithm has issues. This intuition relies on the existence of a "normal", i.e., a baseline for comparison. A natural baseline is a textual difference, as opposed to the AST edit script. Thus we computed the Levenshtein distance between the 466 file revision pairs as a baseline. We then computed the ratio of the edit script size divided by the Levenshtein distance to find out which file revision pairs had AST edit scripts considerably larger or smaller than their textual edit distance. We then inspected the 24 outliers of this distribution (9 high, 15 low).

We complemented this data with a set of 14 revision pairs that were within 10% of the median ratio value, and with an additional random sample of 10% of the file revision pairs (48 pairs), for a total of 86 file revision pairs (18% of all revision pairs).

Analysis. The first author then analyzed the edit script produced by GumTree, using the swingdiff interface supported by GumTree and going change by change, to determine whether the edit script produced was optimal ("good") or if it could be improved ("bad"). The first author also categorized the issue in four categories described next, and made notes of the observations about each case.

Preliminary results. Our first observation is that in more than a quarter of the cases (23 out of 86: 27%), GumTree produced suboptimal results. The issues varied in severity, from minor issues to major issues such as the example in Listing I.

We further determined that the vast majority of the issues affecting GumTree were due to *mismatches*, in which GumTree's matching step would fail to recognize that a source code entity in the original version was still present in the new version of the code, for a variety of reasons (a *missed match*). Another, less frequent type of mismatches is due to GumTree inferring that two distinct entities are the same entity across versions, while they are not (a *spurious match*).

While we have very limited evidence of the effectiveness of the intuition described above, note that the category of high outliers

(i.e., the edit script was comparatively larger than the textual distance) is the one for which the proportion of sub-optimal results was highest (4 out 9, 44%), while the low outlier category had the lowest proportion of sub-optimal results (2 out of 15, 13%). Further study is however needed to confirm whether this intuition holds.

4.2 Imprecisions: Causes and Effects

Missed and spurious matches. As mentioned above, the main reason for GumTree's issues is due to *mismatches*. Recall (Section 3) that t_{i1} denotes a conceptual element in the original version T_1, while t_{i2} denotes the same conceptual element in the modified version T_2. A *missed match* happens when GumTree does not associate t_{i1} with t_{i2} as it should. In this case, the algorithm ignores the existence of the conceptual element t_i in T_2. A *spurious match* associates version t_{i1} of conceptual element t_i with t_{j2} that belongs to a different conceptual element t_j ($t_i \neq t_j$). This means that GumTree confuses one conceptual element (t_i) with another (t_j).

Isolated mismatches. The simplest cause of imprecision is an isolated missed match where the algorithm partitions t_i in two versions wrongly disconnected. So, GumTree deletes t_{i1} and inserts t_{i2} (e.g., Listing I, the conceptual `IF` in the lines 3). The spurious matches also cause imprecisions themselves. Listing II, case A, shows a `Width` property that was spuriously matched to a different property (`UseShadow`). As result, several updates are detected instead of the expected deletion of `Width` and insertion of `UseShadow`.

Compound mismatches. However, in many cases, the mismatches combine with one another. A conceptual element t_i can be both the source of a missed match, and be at the same time spuriously matched. Since GumTree failed to match t_{i1} with t_{i2}, it may still seek to match t_{i1} or t_{i2} with a different conceptual element $t_j = \langle t_{j1}, t_{j2} \rangle$. This can lead to a variety of outcomes:

- If GumTree matches t_{i1} with t_{j2}, it will infer that t_{i1} was updated to t_{j2} and/or moved to the t_{j2}'s position, and that t_{i2} was inserted.
- If GumTree matches t_{j1} with t_{i2}, it will infer that t_{j1} was updated to t_{i2} and/or moved to the t_{i2}'s position, and that t_{i1} was deleted—e.g., Listing I, the method `OnInit`.
- If GumTree matches t_{i1} with t_{j2} and t_{j1} with t_{i2}, GumTree will issue multiple updates and/or moves.

In the worst cases, GumTree may confuse both versions of t_i with two conceptually different elements $t_j = \langle t_{j1}, t_{j2} \rangle$ and $t_k = \langle t_{k1}, t_{k2} \rangle$; this leads it to update t_{i1} to t_{j2} and/or move t_{i1} to the t_{j2}'s position, and to update t_{k1} to t_{i2} and/or move t_{k1} to the t_{i2}'s position.

4.3 Categories of Imprecisions

The imprecisions are caused by isolated missed matches, isolated spurious matches or the combination of missed and spurious matches. By analyzing the effect of the mismatches at a higher level, we found four categories of imprecisions that we detail below.

Redundant Changes. Redundant changes are in principle caused by missed matches. The consequences are a group of changes that redo what another group of changes undo. A single missed match may be the cause of a rather large set of redundant changes. Listing II, case A shows an example. The missed match of `Combine` leads to: 1) a spurious match between `GetFullPath` and `Combine`, 2) a missed match for `outPutDir`, and 3) mismatches between parentheses. This leads to a large number of redundant changes, obscuring the real ones (e.g. the insertion of `Replace`).

Spurious Changes. These are caused by spurious matches. They take the shape of changes that transform two different conceptual elements into one another. Listing II, case B, shows an example. Two C# properties were spuriously matched, due to their high source code similarity. However the similarity is due to the verbose way that these properties were defined. The most relevant aspects are the name and type of the property, which are markedly different. The correct behavior in this case is to treat these conceptual elements as different, meaning that the first one should be deleted, and the second one inserted. Note that this would likely result in a larger edit script. This conflicts with a common goal in evaluating SCCD algorithms, which is to compare the size of their edit scripts (the shortest one being assumed to be the best).

Arbitrary Changes. These are spurious matches that trigger transformations that are extremely unlikely, as they update very different source code elements into one another. We see two examples in Listing II, case C. The first is caused by the algorithm matching two unrelated assignment operators. As a result, GumTree infers that the operator has moved from one line to another, although it could not reconcile the very different operator arguments. The other arbitrary match is between a string literal (a complex regular expression) and a boolean, resulting in an unlikely update from the expression to the boolean.

Ghost Changes. These changes involve conceptual elements that were not modified. They are side-effects of other changes. Ghost changes are the black portions in the expected detection, that are colored in the algorithm's output (e.g., Listing I and Listing II case A). Since they are side effects of the other imprecisions, the remainder of this paper focuses on spurious, arbitrary and redundant changes.

This characterization is not exclusive. In many missed matches at least one conceptual version is spuriously matched. Arbitrary changes are spurious by definition, while some ghost or redundant changes are additionally spurious. However, we separated these kinds of imprecision to analyze them following a divide-and-conquer approach. The next section presents a battery of heuristics to recognize the effects of imprecisions, and evaluate their impact.

5 DETECTING AND QUANTIFYING ISSUES WITH HEURISTICS

The Corpus. We built our corpus based on several sources. We started with the projects developed by Microsoft from the *.NET Foundation* [4], selecting those that were hosted on Github. We complemented these by a selection of projects from three other sources: *GitHub C# Trending Projects*[5]; *Up-for-grabs*[6]; and *Open Source Microsoft* [7]. In all three cases, we restricted our selection to projects having at least 1,000 commits, and 1 year of development.

Our corpus contains 143,419 file revision pairs over 107 projects. From 292,935 unique pairs initially extracted, we considered those having real source code modifications: 147,945 (50.50%). For instance, simple path renames may originate new but unmodified file

[4] https://dotnetfoundation.org/
[5] https://github.com/trending/csharp: pp 1–5, order by stars
[6] http://up-for-grabs.net, tagged "C#" or ".NET"
[7] https://opensource.microsoft.com/, first 9 pages with tag "C#"

Listing II: IMPRECISION EXAMPLES (original at left, modified at right)

Case A (*Redundant changes due to missed matches*): In the first line, the call to "Combine" failed to match, leading to a spurious match between "GetFullPath" and "Combine", and several entities being deleted and inserted ("Combine", "outPutDir"), and others moved ("filePrefix", parentheses). **Expected behavior**: second lines, left and right. This makes the actual changes (e.g., "Replace") much easier to see.

```
1 var staticFilesDirName = Path.GetFullPath(Path.Combine(outputDir,
      filePrefix));
1 var staticFilesDirName = Path.GetFullPath(Path.Combine(outputDir,
      filePrefix));
```

```
1 var staticFilesDirName = Path.Combine("../" + outputDir,
      filePrefix).Replace("/",@"\");
1 var staticFilesDirName = Path.Combine("../" + outputDir,
      filePrefix).Replace("/",@"\");
```

Case B (*Spurious changes due to spurious matches*): GumTree matches the properties "Width" and "UseShadow" since they involve several lines of rather repetitive code. Domain knowledge is necessary to recognize which are the most important parts of the AST that describe a property (i.e., its name). **Expected behavior**: The Property "Width" was deleted and the property "UseShadow" was inserted.

```
1 public int Width {
2   get { return GetPropertyValue("Width", 300); }
3   set { SetPropertyValue("Width", value); }
4 }
```

```
1 public bool UseShadow {
2   get { return GetPropertyValue("UseShadow", true); }
3   set { SetPropertyValue("UseShadow", value); }
4 }
```

Case C (*Arbitrary changes due to spurious matches*): Operators "=" (line 2 at left, line 4 at right) should not match, neither the string literal (line 4 at left) and the boolean literal (line 2 at right). Some elements cannot simply match everywhere. **Expected behavior**: The lines 2 and 3 were completely deleted at the left. The lines 2-4 were completely inserted at the right.

```
1 ... CleanAttributeValues(HtmlAttribute attribute){...
2   attribute.Value = Regex.Replace(...); ...
3   ... Regex.IsMatch(attribute.Value, "\s*e\s*x\s*p\s*r\s*e\s*s\s*s"
        "\s*i\s*o\s*n\s*", ...);
4 ...}
```

```
1 ... CleanAttributeValues(HtmlAttribute attribute){
2   var hasMatch = true;
3   if(Regex.IsMatch(attribute.Value, ...)
4     hasMatch = true;
5 ...}
```

revision pairs. However, 4,526 (3.06%) additional revision pairs were later filtered out because they had only modifications of comments. The curated corpus represents 48.96% of the initial one.

Heuristics. For each of the 3 categories of imprecisions we described, we developed detection heuristics that we ran on the entire corpus. This allows us to estimate how wide-spread each of the issues are. We describe the heuristics in the next 3 sections.

Manual Rating. Imprecise heuristics could severely overestimate the magnitude of the issues. To increase the precision of our estimates, we manually investigate a random sample of the issues highlighted by the heuristics, to gauge their accuracy. This revision is performed by the first author of the paper (a C# expert). The rater analyzes the revision pairs of the random sample, looking at the matches of the heuristics, rating them as:

- **Correct:** The source code location singled out by the heuristic shows a sub-optimal behavior by GumTree, whether directly or indirectly related to the heuristic.
- **Incorrect:** The source code location singled out by the heuristic does not show a sub-optimal behavior by GumTree.

Sampling. We calculate sample sizes according to a standard formula [27]. We draw samples from the population of file revision pairs where each heuristic found matches, aiming for a confidence interval of 10% with a confidence level of 95%. We use a 10% interval due to the large amount of heuristics to check, to keep the workload manageable (the rating process took several months).

Diagnostic. The first author took notes of observations about each case; these are briefly discussed in the quantitative results.

6 REDUNDANT CHANGES

6.1 Redundancy Checking

To recognize redundancy changes, it is necessary to identify the conceptual element t_i that was implicitly missed by the SCCD algorithm. However, figuring out the matches $\langle t_{i1}, t_{i2} \rangle$ is precisely one of the major challenges in change detection. We simplified the problem: we check for conceptual elements that kept the same name across the original and the modified versions.

Algorithm 1 checks for redundant changes produced by GumTree. The entry point is `FindRedundantRenames`. The algorithm searches for potentially missed matches in all the changes (the delta) produced by a SCCD algorithm on a file version pair (lines 11–12). Then it checks combinations of changes. The intention is to recognize names that were deleted, moved out, or updated (i.e., overwritten), but were later (re)inserted, moved back, or (re)updated with an incorrect new name (the incorrect new names being rather old names that should not have been updated). Each combination defines one of 10 redundancy patterns (lines 13–22, see next subsection), that is checked by `MissedNames`. In addition, each combination may have a different way to test name equality, this is the third argument passed to `MissedNames`'s calls on lines 13–22.

Given two changes and a way to test for equality of names, `MissedNames` performs the actual test. Two conceptual element versions e_{i1} and e_{j2}, could describe the same (and hypothetically missed) conceptual element if:

- (a) *they both are of one of the types in Table 2*;

- (b) *their names of interest are equal*. We assume that developers rarely eliminate one element type and introduce another semantically different element with an equal name;
- (c) *their names are related by one redundancy pattern* presented in the next subsection;
- (d) *they have one common ancestor*. The name is unique in all the well delimited syntactical scopes.

The recognition starts from lower ancestors and goes increasing the levels until it finds the first symptom of redundancy (line 7). Matched ancestors (line 6) estimate the conceptual scopes. As they are traversed bottom-up (line 3), it is expected that the redundancy symptoms will be encountered on the lower matches (i.e., the smallest conceptual scopes) delimited by GumTree. If two names are equal and they were suspiciously changed under a same scope (line 4), they probably represent a missed match for the same conceptual element. This fact itself is reported as a symptom of imprecision.

Algorithm 1 Redundancy checking

Require: Original and modified subject versions (E_1 and E_2), $equal(x, y)$ determines if two conceptual versions are compatible.

```
1:  function MissedNames(E1, E2, equal)
2:    R ← {}
3:    A(z) ← z's ancestors ordered bottom-up
4:    for each {⟨e1∈E1, e2∈E2⟩ : equal(e1, e2)} do
5:      for each a2∈A(e2) do
6:        if ∃a1∈A(e1) such that a1 matches a2 then
7:          R ← R ∪ ⟨e1, e2⟩
8:          break
9:    return R
10: end function

11: function FindRedundantRenames(δT1,T2)
12:   I, D, U, M ← the elements inserted, deleted, updated, or moved
      in δT1,T2, respectively.
13:   return MissedNames(D, I, name(x)=name(y)) ∪
14:   MissedNames(D, U, name(x)=newname(y)) ∪
15:   MissedNames(D, M, name(x)=[new]name(y)) ∪
16:   MissedNames(U, I, oldname(x)=name(y)) ∪
17:   MissedNames(M, I, [old]name(x)=name(y)) ∪
18:   MissedNames(U, U, oldname(x)=newname(y)) ∪
19:   MissedNames(U, M, oldname(x)=[new]name(y)) ∪
20:   MissedNames(M, U, oldname(x)=newname(y)) ∪
21:   MissedNames(M, M, oldname(x)=[new]name(y)) ∪
22:   MissedNames(M, M, oldname(x)=newname(y))
23: end function
```

6.2 Redundancy Patterns

A SCCD algorithm X may mismatch a conceptual element t_i=⟨t_{i1}, t_{i2}⟩ according to 10 redundancy patterns:

DI) X partitions t_i in two false elements t_j=⟨t_{i1}, ∅⟩ and t_k=⟨∅, t_{i2}⟩. Effect: X deletes t_{i1} and inserts t_{i2}. For example, `outputDir` in Listing II, case A. The remaining cases must follow more complex transformations, of t_{i1} into t_{k2} (t_k=⟨t_{k1}, t_{k2}⟩) and/or of t_{j1} (t_j=⟨t_{j1}, t_{j2}⟩) into t_{i2}.

DU) X mismatches t_{i2} with a t_j, where t_{j1}'s value ≠ t_{i2}'s value (a spurious match). Effect: X deletes t_{i1} and updates t_{j1} to t_{i2}. For example, `Combine` in Listing II, case A.

Table 1: REDUNDANT PATTERNS (S: Symptoms, F: File Revision Pairs)

Pattern	Population S	Population F	% of TOTAL S	% of TOTAL F
DI	95445	3922	71.01%	**40.48**
DU	6030	1560	04.49%	16.10
DM	2069	547	01.54%	05.65
UI	8819	3769	06.57%	**38.90**
MI	3361	1036	02.50%	10.69
UU	7302	1557	05.44%	16.07
UM	515	326	00.38%	03.36
MU	458	285	00.34%	02.94
MM	3270	232	02.44%	02.39
M	6978	2574	05.20%	**26.57**
TOTAL	134247	9688		

DM) X mismatches t_{i2} with a t_j, where t_{j1}'s position ≠ t_{i2}'s position. Effect: X deletes t_{i1} and moves t_{j1} to t_{i2}'s position.

UI) X mismatches t_{i1} with a t_k, where t_{i1}'s value ≠ t_{k2}'s value. Effect: X updates t_{i1} to t_{k2} and inserts t_{i2}. See `OnInit` in listing I.

MI) X mismatches t_{i1} with a t_k, where t_{i1}'s position ≠ t_{k2}'s position. Effect: X moves t_{i1} to t_{k2}'s position and inserts t_{i2}. For example, the second `(` in Listing II, case A.

UU) X mismatches t_{i1} with a t_k, where t_{i1}'s value ≠ t_{k2}'s value, **and** mismatches t_{i2} with a t_j where t_{j1}'s value ≠ t_{i2}'s value. Effect: X updates t_{i1} to t_{k2} and updates t_{j1} to t_{i2}.

UM) X mismatches t_{i1} with a t_k, where t_{i1}'s value ≠ t_{k2}'s value, **and** mismatches t_{i2} with a t_j where t_{j1}'s position ≠ t_{i2}'s position. Effect: X updates t_{i1} to t_{k2} and moves t_{j1} to t_{i2}'s position.

MU) X mismatches t_{i1} with a t_k, where t_{i1}'s position ≠ t_{k2}'s position, **and** mismatches t_{i2} with a t_j, where t_{j1}'s value ≠ t_{i2}'s value. Effect: X moves t_{i1} to t_{k2}'s position and updates t_{j1} to t_{i2}.

MM) X mismatches t_{i1} with a t_k, where t_{i1}'s position ≠ t_{k2}'s, **and** mismatches t_{i2} with a t_j, where t_{j1}'s position ≠ t_{i2}'s position. Effect: X moves t_{i1} to t_{k2}'s position and moves t_{j1} to t_{i2}'s position.

M) X does not mismatch t_i **but mismatches t_i's parent**, for instance after a **DI** pattern. Effect: X moves t_{i1} to t_{i2}'s position, a side effect (and a ghost change) of a mismatch among the parents. In Listing II, case A, `filePrefix` moves due to the missed match of `Path.Combine`.

6.3 Results and Diagnostic

Overall and pattern-level analysis. Table 1 summarizes the redundant names found for each redundancy patterns. The heuristics find matches (S) in 9,688 file revision pairs (F), or 6.75% of the total. A large majority (71.01%) of the issues come from the first and simplest pattern, **DI**. Gumtree simply **fails to match a large number of conceptual elements**. All other patterns contribute in minor proportions. The second most common pattern is **UI**, which affects 6.57% of the total. Note that **the amount file revision pairs affected by UI (38.90%) is almost as significant as the one of DI (40.48%)**. A third pattern of interest is the **M** pattern, affecting 5.20% of the elements but **more than a quarter of files (26.57%)**. Since these are collateral effects of other redundancy patterns, we see that some of the patterns indeed tend to spread.

Precision of the heuristics. We evaluated the precision of the heuristics on a per element type basis to determine whether some element types need particular attention. Thus we divided the results according to each (named) element type affected, and inspected a sample of each (or the whole population if small enough). We classified the cases in true bad behaviors (*Issues*) or not. Table 2 shows the results. Each column pair describes the counts of symptoms (S) and affected file revision pairs (F). Let the accuracy $a = \frac{bad\ cases}{inspected\ cases}$, the confidence interval c, and the total of modified element types (E) across the entire corpus (*Corpus*). We estimate imprecisions in the corpus (*%Issues*) as $\frac{\lfloor (a-c)/100 \rfloor}{\text{Corpus}} * 100$ both for S and F.

The accuracy is very high overall: it hovers above 95% for most categories, with some at 100% (since the whole population was inspected for those). Two categories have comparatively subpar accuracy: formal arguments (73.51%) and variables (86.94%), but both remain high. This increases our confidence that the heuristics are not overstating the issue (they may understate them, as we have no way to account for false negatives).

Affected elements. Table 2 shows that the most frequent problems of redundant names in GumTree are in variables, fields, and methods. In particular, variable mismatches are widespread at the file level. Methods seem to have less potential mismatches, but these may be actually more worrisome, since a mismatch at that level may trigger further issues below it, see Listing I.

In general, the size of an element influences the matching behavior. The larger it is, the more information the algorithm has to discover a good match. This largely explains the high accuracy in most of the type definitions (classes, structs, interfaces). As the textual extension decreases, the number of issues increases. For instance, properties are more challenging than the type definitions, but less challenging than variables or formal arguments. Another factor is how many potential matches to choose from. There are usually few classes, constructors, or destructors in a file. Methods, fields, and variables on the other hand are numerous.

Detailed observations. From our notes, we list some particular situations that foster imprecisions on different element types:

Split or merged declarations: Listing I shows a case where the original method splits its implementation between its modified counterpart and a new method. Alternatively, two method's implementations can be merged in one modified version, while the other is deleted. The most similar subtrees are not always the same conceptual element, leading to mismatches.

Stub first, implementation later: Some elements start as stubs to be implemented later, e.g., a method that throws an exception or returns a constant to show it has still not been implemented. The concrete implementation may drastically differ from the stub, leading to a mismatch between the original and the modified versions. This mismatch may even propagate up to the ancestors, so that their container type definitions do not match either.

Members moved up across a hierarchy: Due to refactorings, methods, properties and fields can move from a subclass to a superclass. Occasionally, the modified subclass diverges from the original; the original subclass may even look textually closer to the modified superclass. As a result, the subclass match is missed, and it may even by spuriously matched with the superclass.

Textually dissimilar, but conceptually similar: Some changes do not affect the conceptual identity of the statement. For example, a variable declaration `List<string> _parameters = new List<string>();` may be modified by adding the namespace `System.Collections.Generic` to the type, or changing it to an implicit type (`var` keyword).

7 SPURIOUS CHANGES

Spurious changes are changes between two conceptually different source code elements. They happen when the algorithm spuriously matches them, and as a consequences "forces" the conversion of one source code element into the other.

To detect potential spurious changes, we measure the amount of changes affecting a given entity, and pinpoint entities that change "too much". The heuristic computes a *transformation coefficient* for each entity, and for entities that are usually stable (have few changes), reports outliers that have much more changes.

Transformation Coefficient. Let the *in-actions* be the insertions, updates or moves affecting one original conceptual version, and the *out-actions* be the deletions or moves from it. We defined a transformation coefficient as the ratio between in-actions and out-actions, filtering out the elements that were entirely inserted or deleted (i.e., $in{-}actions \geq 0$ and $out{-}actions \geq 0$ is required).

Finding Outliers. Intuitively, in the element types where the transformation coefficient is stable, coefficients higher than a certain threshold may highlight imprecisions. We computed the transformation coefficient $C_{ij}(t_j, \langle T_{1i}, T_{2i} \rangle)$ for each element type t_j in each file revision pair $\langle T_{1i}, T_{2i} \rangle$. Subsequently, we averaged these coefficients to inhibit the influence of large file revision pairs. To pick the transformation coefficient threshold $C(t)$, we used the method of Oliveira et al. [21] to compute thresholds of the form "80% of the systems in the corpus should have $C(t) \leq M$".

Results. The results are shown in Table 3. We apply the heuristic only to elements where the threshold is over 80%. We find that 23.78% of file revision pairs may have spurious change according to the heuristic. However, a preliminary manual inspection shows that the accuracy of the heuristic is lower than for redundant changes. Out of 243 file revision pairs inspected, 164 were indeed rated as spurious (67%). Assuming a representative sample, we estimate that 16% of file revision pairs could be subjected to spurious changes, still an important portion. While GumTree economizes transformations by moving subexpressions from deleted statements (e.g., from method calls), the conceptual expressions moved are not always good matches, but rather arbitrary updates.

8 ARBITRARY CHANGES

Arbitrary changes are particularly spurious matches that foster transformations that are particularly hard to believe. While conducting the exploratory study, we kept notes of particular transitions between specific elements or element types, and systematically searched for them. We defined 9 such heuristics. During our manual inspection, we posed two questions to evaluate the credibility of a change. First, *Does the change express an action a developer would do?* e.g., an operator or a parenthesis moving from one method to a conceptually different method is rarely credible according to this criterion. Converting a boolean to another type is also unlikely. Then, *Does the change have a believable semantic?* e.g., a rename

Table 2: REDUNDANT NAMES PER ELEMENT TYPES (S: Symptoms, F: File Revision Pairs)

Types	Population		Sampling		Issues		Accuracy ± Confidence (95%)		Corpus		%Issues	
	S	F↓	S	F	S	F	S	F	E	F	E	F
destructors	5	5	5	5	5	5	100%	100%	163	103	03.07%	04.85%
struct	151	13	151	13	151	13	100%	100%	2015	1602	07.49%	00.81%
enum values	278	58	96	36	96	36	100%	100%	4965	1000	05.60%	05.80%
interface	32	26	32	26	32	26	100%	100%	1741	1437	01.84%	01.81%
enum	94	92	94	92	94	92	100%	100%	5146	4794	01.83%	01.92%
class	955	352	133	83	128	81	96.24% ± 03.00	97.59% ± 02.89	120230	100796	00.74%	00.33%
constructors	1100	510	137	85	132	83	96.35% ± 02.94	97.65% ± 02.94	19167	12767	05.34%	03.78%
property	2809	539	961	42	951	40	98.96% ± 00.52	95.24% ± 06.19	37645	13785	07.34%	03.47%
field	7737	684	502	73	475	70	94.62% ± 01.91	95.89% ± 04.31	13147	6090	54.55%	10.28%
formal args.	8829	866	268	75	197	53	73.51% ± 05.20	70.67% ± 09.55	145268	40825	04.15%	01.30%
methods	43740	3952	306	115	294	115	96.08% ± 02.17	100%	256648	91631	16.00%	04.31%
variable	25994	4789	245	107	213	104	86.94% ± 04.20	97.20% ± 03.09	43550	14797	49.38%	**30.45%**
TOTAL	134247	9688	2897	680	2644	654	91.27%	96.18%				

Table 3: SPURIOUS CHANGES IN STABLE ELEMENT TYPES (S: Symptoms, F: File Revision Pairs)

Types	C(t)	Population		Sampling		Issues		Accuracy	
	p% ≤M	S	F↓	S	F	S	F	S	F
condition	80 - 0.50	3622	2400	40	22	28	18	70.00%	81.82%
init	84 - 0.50	6333	3924	32	28	17	14	53.13%	50.00%
expr_stmt	86 - 0.50	13237	6874	34	23	18	13	52.94%	56.52%
decl	84 - 0.46	12992	7292	45	34	32	26	71.11%	76.47%
name	80 - 0.09	21631	8421	38	24	26	17	68.42%	70.83%
argument	84 - 0.34	24179	10057	34	22	26	17	76.47%	77.22%
method	85 - 0.61	15915	11085	71	45	38	27	53.52%	60.00%
call	85 - 0.34	30514	12255	130	75	71	47	54.61%	62.67%
TOTAL		355441	34110	424	243	256	164	60.38%	67.49%

between two elements should involve elements with semantically consistent names and capture the intention of developers.

Results. The heuristics are shown in Table 4. Our manual analysis shows an accuracy of 82% at the file level, allowing us to estimate that 8.5% of file revision pairs may have arbitrary changes pinpointed by our heuristisc. Most of the times, these are textually optimal but semantically imprecise or controversial.

Renames between instance expressions ("this" and "base") and simple names (e.g., of variables or methods): The instance expressions are represented in SrcML's trees as simple names and they can match indiscriminately with simple names, for example of variables, fields, properties, or methods.

Arbitrary updates, for example between boolean and non-boolean literals, or null literal and arbitrary names. These literals may even move among unrelated scopes (e.g., different methods). A surprising number of built-in types are updated from, or to user types. In practice, many of them are spurious and arbitrary changes. Classifying packages might help, but requires C# knowledge.

Arbitrary renames among incompatible element types: The renames must respect the semantic relationship between the old name and the new name. Sometimes, there are updates among type names that are conceptually incompatible. These violate a general rule: two elements should match if at least they are of a same type. For instance, GumTree may rename a method call with the name of an accessed property. There are exceptions to this general rule: some types are compatible, such as a variable switching to a field, but handling these cases however requires knowledge of the programming language.

9 DISCUSSION

Space for improvement. The estimated extent of our heuristics show that there is space for improvement over GumTree: more than 6% of file revision pairs were highlighted by our redundant change heuristic. Spurious changes could affect 16% of file revision pairs, and 8.5% of file revision pairs could be affected by arbitrary changes. These values do not consider false negatives, and a pair may exhibit several mismatches. Furthermore, methods appear to be affected by redundant changes. These changes to high-level source code entities could have a larger impact, as shown in Listing I.

Edit scripts are not enough. Spurious and arbitrary changes are caused by the SCCD algorithm finding a match between elements that should not exist. A possible reason for this is that a common evaluation metric for SCCD algorithm is the size of the edit script that it produces. In some cases, a spurious match may actually produce a smaller edit script than a correct absence of match (issuing updates rather than a set of deletions followed by insertions). Barabucci [2, 3] argues that the edit script is not necessarily the best metric, and proposes additional metrics.

Table 4: ARBITRARY UPDATES (S: Symptoms, F: File Revision Pairs)

Between	Population		Sampling		Issues		Accuracy	
	S	F↓	S	F	S	F	S	F
base and simple names	370	259	40	33	35	29	87.50%	87.88%
false and non-boolean literals	886	540	27	25	25	23	91.67%	92.59%
true and non-boolean literals	865	588	28	21	27	20	96.43%	95.24%
non-boolean but different literals	2833	1238	36	22	36	22	100%	100%
different simple names	2431	1441	33	23	29	21	87.88%	91.30%
this and simple names	3310	1479	65	42	59	39	90.77%	92.86%
null and simple names	2399	1567	61	45	45	36	73.77%	80.00%
system and user type names	17166	7259	161	94	113	70	70.19%	74.47%
operators	17160	7488	169	96	121	73	71.60%	76.04%
TOTAL	47422	14766	620	352	490	289	79.03%	82.10%

Tree-based vs Language-based. Two generic conceptual versions should match if they share enough information. In tree differencing this holds if their similarity overcomes pre-determined metric thresholds (in the case of GumTree, obtained through hashing and Dice comparisons). However, this favors the most similar subtrees even in contexts apparently better resolved by language rules and development conventions, such as Listing I. This makes GumTree fail in several patterns of refactoring, such as extracting methods.

More specific information could improve matching. Embedding in the algorithm knowledge of specific language rules (e.g., *event handlers* are more similar than other methods) could prevent spurious matches. Other examples include: increasing the weight of the name of properties (Listing II, case B); recognizing system packages and compatible types; recognizing `this` and `base`; reducing the weight of access modifiers. This suggests a new approach in SCCD: be more *language-aware*, and not only tree-aware.

Per-element matching. Following the previous recommendation to the fullest, a way to reduce the number of mismatches could be to make the matching polymorphic according to the element type. The elements vary not only in the type, but also in the average size of their contents (with an impact on performance, see Table 2), the naming conventions, their syntactical structure, or their semantic role. Different elements are modified with distinct frequencies and in distinct ways. State of the art SCCD algorithms such as GumTree and Change Distiller [10] are limited in this regard: They propose SCCD strategies based on whether the element to match is a tree leaf or an internal node, or the size of the subtree below it.

Tuning the matching procedure to the type of elements would allow to select the best matching techniques for each one, varying, for instance: the similarity techniques (e.g., monograms [4] or bigrams [10, 29]; hashing [9, 13], Dice [9], or longest common subsequence [4]); the testing thresholds (static [4, 9] or dynamic [8, 10]); and discriminant parts (e.g., [29]).

10 LIMITATIONS

Preliminary study. Our preliminary study was conducted on a single system to gain and preserve knowledge about the system. Other systems may have different issues. On the other hand, our corpus for the follow-up study is much larger than the one used in related work (3 systems for [10], 16 systems for [9]), which increases our generalization in another dimension. During the manual investigation of a random sample on those 107 systems, we did not encounter cases of issues that seemed drastically different than what we observed during the preliminary study.

GumTree specific. Our diagnostic is specific to the current behavior of GumTree and not of any other algorithm nor configuration. Our heuristics will detect the same issues on other algorithms; comparable results would indicate similar issues. A preliminary exploratory study as we did in Section 4 would be needed to ensure additional issues do not affect other algorithms.

GumTree's support for C#. The support of GumTree for C# has some limitations stemming from the ASTs produced by SrcML. We detected two of those that we were able to work around. Comments were not associated with the entities that they described, but with their parents, resulting in a large number of ghost moves; we filtered out the comments and re-processed the data. Some `enum` declarations contained nested classes, which caused redundant changes in enum values. These cases were filtered out from the heuristics, although they may affect some elements that SrcML nested under the enums. While we mitigated the impact of these two issues, there might be additional issues we did not detect. We think this is unlikely, as we have done an extensive amount of manual analysis and have not detected anything else, but we cannot exclude it outright. We note that this outlines an additional limitation of SCCD algorithms: their sensibility to the topology of the AST, topology that is rarely documented in the papers.

False positives. Our analysis may suffer from false positives returned by heuristics. This could lead us to overstate the importance of the issues. To mitigate this, we manually evaluated the output of the heuristics, finding that redundant changes were generally very accurate, with spurious and arbitrary changes somewhat less so. An additional issue is that the rating was done by the first author only—however, the first author is the only author to possess the required C# experience to reliably do this rating.

False negatives. Our heuristics may not return all the imprecisions, leading us to understate the importance of the issues. Some spurious changes may satisfy the relative thresholds of our spurious change heuristic. Moreover, redundant changes are not exclusive to elements with identical names. Algorithm 1 can be customized, for example to support more flexible forms of matching to reflect this. The redundant change heuristic also likely suffers from false

negatives due to the filtering out of elements nested under enums by SrcML. Thus, it is likely that the "space for improvement" over GumTree that we identified is greater than our estimate.

Imprecise symptoms. The symptoms of redundancies can be imprecise. Ideally, each should describe a single mismatch. However, in some cases, several mismatches combine together (e.g., an **UM** pattern and an **UU** pattern). In other cases, several conceptual elements can be simultaneously redundant if they have equal names but belong to different scopes. The most commonly affected elements are variables and fields. For example, n variables may exist with the same name in different methods. The heuristic does not discriminate and may report in the worst case n^2 symptoms instead of n (if all variables changed at once). Since variables may be moved among different methods, these situations cannot be automatically resolved. We mark an issue as a true bad behavior if at least one of the versions is truly involved in a bad behavior. This may lead to overestimating some symptoms in Tables 1 and 2, particularly for variables and fields. We thus refrain from making strong conclusions about the number of symptoms. The number of files is not affected by this issue.

11 RELATED WORK

11.1 Change Detection

Change detection in trees. Zhang and Shasha [30] introduced an early algorithm to compute the edit distance between two trees. Chawathe et al. [4] introduced the preliminary notions and algorithms of change detection in hierarchically structured information (e.g., Latex documents), splitting the general problem in the matching phase, and the minimum conforming edit script subproblems. Pawlik and Augsten introduced **RTED** [22], one of the best algorithms to detect the minimum-cost edit script on ordered, labeled trees. RTED is agnostic to the syntax and the kind of the nodes; it can detect incompatible changes, such as updates between two different kinds of nodes. This is why algorithms extending it, such as GumTree, must filter its output.

Source code change detection. Beyond GumTree [9] that we covered extensively, other algorithms have been developed. UmlDiff by Xing and Stroulia [29] reports design changes for attributes or dependencies between packages, classes, interfaces, methods and fields. They combine name-based with structure-based measuring and identify a taxonomy of changes. Neamtiu et al. [19] report changes at the level of the AST, comparing types and variable names. The approach fails when there are a lot of changes and the ASTs being compared have very different shapes. Fluri et al. [10] show the limitations of Chawathe et al.'s approach when applied to source code. They then adapt the algorithm to source code by changing its matching criterion and similarity computations. Hashimoto and Mori [13] present Diff/TS, a tool supporting fine-grained analysis; it extends the work of Zhang and Shashas [30] with control based on heuristics related to a given programming language. A recent addition to the state of the art is the work of Dotzler and Philippsen [7], which introduces generic optimizations to tree differencing algorithms to more precisely detect moves.

Barabucci et al. [3] argue that the size of the edit script is only one aspect with which to measure the quality of a change detection algorithm. They introduce additional metrics that measure other aspects, that may be more adequate for different usage scenarios.

11.2 Beyond change detection

Several approaches address some of the shortcomings of change detection, many via post-processing edit scripts. These could likely be improved by higher quality edit scripts. Origin analysis approaches focus on detecting entities that were split or merged [11]. Several approaches (too many to list) detect refactorings in software repositories—an early example is the approach by Weissgerber and Diehl [28]; a more recent one is RefDiff [25]. Hora et al. studied how often refactorings threatened to break the continuity of entity histories, finding it very common [15]. Missed matches have the same effect, while spurious matches may lengthen the entity's history. Kim and Notkin detect systematic changes [18] that can be described more succinctly by rules (e.g., add a method to an entire class hierarchy). Kawrykow and Robillard detect non-essential changes, changes in edit scripts beyond refactorings that do not affect functionality [17]. Herzig and Zeller separate the tangled changes in logical units of functionality [14].

Others proposed radical departures from the status quo. Nguyen et al. proposed a versioning system that stores the structure of object-oriented programs, rather than merely its source code as text [20]. A follow-up version also explicitly stored refactoring operations [6]. Apel et al. proposed an intermediate solution, the semi-structured merge, where the higher level of the structure is used for merging, with text below it [1]. Others advocated recording changes as they were made to a system, instead of storing versions [24]. The systematic mapping study of approaches using first-class changes by Soetens et al. [26] provides a comprehensive overview.

12 CONCLUSIONS

Source Code Change Detection algorithms are an essential building block for many MSR approaches. Any imprecision that they have may be amplified in subsequent steps. As such, improving SCCD is an important goal in MSR. We presented an empirical study characterizing the limitations of GumTree, a state of the art SCCD algorithm. We first applied GumTree to one C# software system. An exploratory study in which we manually analyzed the issues present in GumTree edit scripts found that 27% the 86 file version pairs we analyzed were not optimal.

We then classified the imprecisions in four categories: redundant, spurious, arbitrary, and ghost changes. For each category (except ghost changes), we developed detection heuristics. We applied the heuristics on a corpus of 107 C# systems, manually rated their precision, and estimated the extent of imprecisions in the corpus.

The imprecisions pointed out by the heuristics were common enough that there are opportunities to improve GumTree. In particular, our results indicate that GumTree has issues in the matching phase; a matching phase tailored to the specific element types to match may yield promising results. GumTree, and other SCCD algorithms, mostly treat source code as "just an AST", and do not consider many language features. We are currently actively working on the development of a SCCD approach that is tailored to the specificities of the C# programming language, to improve on the state of the art in SCCD.

REFERENCES

[1] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured merge: rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 190–200.

[2] Gioele Barabucci. 2013. Introduction to the universal delta model. In *Proceedings of the 2013 ACM symposium on Document engineering - DocEng '13*. ACM Press, New York, New York, USA, 47. https://doi.org/10.1145/2494266.2494284

[3] Gioele Barabucci, Paolo Ciancarini, Angelo Di Iorio, and Fabio Vitali. 2016. Measuring the quality of diff algorithms: a formalization. *Computer Standards & Interfaces* 46 (2016), 52–65. https://doi.org/10.1016/j.csi.2015.12.005

[4] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change detection in hierarchically structured information. *ACM SIGMOD Record* 25, 2 (jun 1996), 493–504. https://doi.org/10.1145/235968.233366

[5] Michael L Collard, Huzefa H Kagdi, and Jonathan I Maletic. 2003. An XML-based lightweight C++ fact extractor. In *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE, 134–143.

[6] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N Nguyen. 2007. Refactoring-aware configuration management for object-oriented programs. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 427–436.

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

[8] Adam Duley, Chris Spandikow, and Miryung Kim. 2012. Vdiff: a program differencing algorithm for Verilog hardware description language. *Automated Software Engineering* 19, 4 (may 2012), 459–490. https://doi.org/10.1007/s10515-012-0107-6

[9] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. ACM Press, New York, New York, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[10] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (nov 2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

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

[12] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 358–368.

[13] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *2008 15th Working Conference on Reverse Engineering*. IEEE, 279–288. https://doi.org/10.1109/WCRE.2008.44

[14] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016), 303–336.

[15] Andre Hora, Danilo Silva, Marco Tulio Valente, and Romain Robbes. 2018. Assessing the Threat of Untracked Changes in Software Evolution. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, in press.

[16] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software: Evolution and Process* 19, 2 (2007), 77–131.

[17] David Kawrykow and Martin P. Robillard. 2011. Non-essential changes in version histories. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. ACM Press, New York, New York, USA, 351. https://doi.org/10.1145/1985793.1985842

[18] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 309–319. https://doi.org/10.1109/ICSE.2009.5070531

[19] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories - MSR '05*, Vol. 30. ACM Press, New York, New York, USA, 1–5. https://doi.org/10.1145/1083142.1083143

[20] Tien N Nguyen, Ethan V Munson, and John T Boyland. 2004. Object-oriented, structural software configuration management. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM, 35–36.

[21] Paloma Oliveira, Marco Tulio Valente, and Fernando Paim Lima. 2014. Extracting relative thresholds for source code metrics. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 254–263. https://doi.org/10.1109/CSMR-WCRE.2014.6747177

[22] Mateusz Pawlik and Nikolaus Augsten. 2011. Rted. *Proceedings of the VLDB Endowment* 5, 4 (dec 2011), 334–345. https://doi.org/10.14778/2095686.2095692

[23] Romain Robbes and Michele Lanza. 2005. Versioning systems for evolution research. In *Principles of Software Evolution, Eighth International Workshop on*. IEEE, 155–164.

[24] Romain Robbes and Michele Lanza. 2007. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science* 166 (2007), 93–109.

[25] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: detecting refactorings in version histories. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 269–279.

[26] Quinten David Soetens, Romain Robbes, and Serge Demeyer. 2017. Changes as First-Class Citizens: A Research Perspective on Modern Software Tooling. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 18.

[27] Mario F Triola. 2006. *Elementary statistics*. Pearson/Addison-Wesley Reading, MA.

[28] Peter Weissgerber and Stephan Diehl. 2006. Identifying refactorings from source-code changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 231–240.

[29] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05*. ACM Press, New York, New York, USA, 54. https://doi.org/10.1145/1101908.1101919

[30] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (dec 1989), 1245–1262. https://doi.org/10.1137/0218082