This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

Expanding Fix Patterns to Enable Automatic Program Repair

Vesna Nowack
Queen Mary University of London
v.nowack@qmul.ac.uk

David Bowes
Lancaster University
d.h.bowes@lancaster.ac.uk

Steve Counsell
Brunel University
steve.counsell@brunel.ac.uk

Tracy Hall
Lancaster University
tracy.hall@lancaster.ac.uk

Saemundur Haraldsson
Stirling University
saemundur.haraldsson@stir.ac.uk

Emily Winter
Lancaster University
e.winter@lancaster.ac.uk

John Woodward
Queen Mary University of London
j.woodward@qmul.ac.uk

***Abstract*—Automatic Program Repair (APR) has been proposed to help developers and reduce the time spent repairing programs. Recent APR tools have applied learned templates (fix patterns) to fix code using knowledge from fixes successfully applied in the past. However, there is still no general agreement on the representation of fix patterns, making their application and comparison with a baseline difficult. As a consequence, it is also difficult to expand fix patterns and further enable APR.**

We automatically generate fix patterns from similar fixes and compare the generated fix patterns against a state-of-the-art taxonomy. Our automated approach splits fixes into smaller, method-level chunks and calculates their similarity. A threshold-based clustering algorithm groups similar chunks and finds matches with state-of-the-art fix patterns. In our evaluation, we present 33 clusters whose fix patterns were generated from the fixes of 835 Defects4J bugs. Of those 33 clusters, 22 matched a state-of-the-art taxonomy with good agreement. The remaining 11 clusters were thematically analysed and generated new fix patterns that expanded the taxonomy. Our new fix patterns should enable APR researchers and practitioners to expand their tools to fix a greater range of bugs in the future.

***Index Terms*—automatic program repair, similarity metric, clustering, fix pattern**

I. Introduction

Automatic Program Repair (APR) has been an increasingly popular research topic for more than a decade [1] and APR tools are gradually being adopted by industry [2]–[4]. APR tools aim to find a repaired variant of a program in the infinite search space of variants. To reduce the size of the search space and improve repair quality, many researchers have reported benefit in using learned templates (*fix patterns*) out of known (human-written or automatically generated) bug fixes.

The fix patterns analysed and applied by APR tools have been either collected from the literature [5]–[8] or generated from bug fixes (manually [9], [10] or automatically [3], [11]). Some researchers [12]–[14] have mined data platforms and repositories with human fixes to generate fix patterns and collect information about bugs and their human-written fixes. To evaluate fix patterns, template-based APR tools [3], [5]–[9], [11]–[13], [15]–[18] apply templates to buggy code and leverage knowledge about previous coding repairs. Expanding state-of-the-art taxonomies by proposing new fix patterns and applying them in current APR tools would help APR tools to fix a greater range of bugs and improve their currently limited performance. However, comparison against a state-of-the-art taxonomy and its expansion is difficult due to the lack of a common representation of fix patterns in the APR community. Koyuncu et al. [11] manually compared their fix patterns against the fix patterns found in the literature. Our work is the first (to the best of our knowledge) that does this comparison automatically.

Our approach automatically clusters bug fixes and generates fix patterns. It is based on splitting human-written fixes into smaller, method-level chunks; then clustering similar chunks and generating their fix patterns, which are compared against state-of-the-art fix patterns [6].

Some bug fixes are difficult to cluster due to their complexity. In an analysis of human-written bug fixes, Sobreira et al. [10] showed that 27% of bug fixes are complex and change more than one method. In general, the majority of APR tools are still not mature enough to repair multiple-location bugs. In Listing 1, we show an example of a multiple-location bug Chart-15 and its human-written fix from Defects4J [19].

```
1377a1378,1380
+     if ( this . dataset == null ) {
+         return 0.0;
+     }
      double result = 0.0;
---
2050a2054
+ if ( this . dataset != null ) {
      state . setTotal ( DatasetUtilities
          . calculatePieDatasetTotal ( plot . getDataset ()));
---
2052a2057
+ }
```

Listing 1: A human-written fix for Chart-15 adds the lines of code marked with '+'.

The buggy code accesses a null variable and the fix consists of inserting `if` statements (as null checks) into two methods. APR tools [6], [11], [15], [18], [20]–[24], are unable to generate fixes similar to the human fix in Listing 1, one of the

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

reasons being that such multiple-location bugs require two fix patterns to be applied simultaneously. To tackle this problem, we apply method-level splitting of bug fixes. Splitting fixes was also suggested by Kreutzer et al. [25] with the limitation that their implementation ignores the parts of fixes outside of methods. By contrast, we divide a bug fix into code modifications from each method as well as from the class declaration. This allows us to recognise smaller parts of fixes that are more likely to be clustered with parts of other fixes.

To represent code modifications (potential bug fixes), we use a linear sequence of simplified Abstract Syntax Tree (AST) nodes. We also divide each sequence into separated parts, where neighbour nodes form sub-sequences of length N called *n-grams*. We compare n-grams of bug fixes to calculate their similarity and cluster the fixes. Using n-grams ensures that a similarity formula includes the similarities in both the AST nodes and the order of neighbour-nodes that form the n-grams. This ensures finding similar fixes even when they consist of modifications that are not in the same exact order.

We apply a similarity metric based on mutual information [26], proposed by Baxter et al. [27] for clone detection. We adapted the metric formula to be able to apply it to the code before and after a bug fix, which is necessary for finding similarities between fixes.

For clustering similar fixes, we use threshold-based clustering [28]. Instead of pre-defining the number of clusters as in the traditional *k*-means algorithm [29], [30], the threshold-based algorithm generates as many clusters as necessary to ensure that items from a dataset are in the same cluster only if their similarity is above the threshold. Although the highest threshold means that fixes from the same cluster are equivalent, lowering the threshold value could be useful for ranking fix patterns in APR tools and choosing a similar, but not equivalent fix pattern for a repair.

The contributions of this paper are as follows:

- We introduce a tool that automatically generates fix patterns from human-written fixes. The tool splits fixes that modify multiple methods into smaller chunks, which are simpler code modifications more likely to be clustered with other chunks.
- We represent chunks as linear sequences of simplified AST nodes forming n-grams. N-grams are straightforward to compare and to calculate similarity from.
- We present a metric (adapted from [27]) for calculating the similarity of chunks. Our threshold-based clustering algorithm (based on [28]) uses similarity to automatically group similar chunks and generate fix patterns.
- Our tool automatically compares fix patterns against a state-of-the-art taxonomy [6] (regardless of the representation of fix patterns) by comparing the clusters that can be described by the same fix pattern.
- We present a methodology for manually confirming new fix patterns and our proposed new fix patterns expand the state-of-the-art taxonomy from Liu et al. [6].

We use a well defined dataset for our evaluation – Defects4J [19] with 17 open-source projects containing 835 bugs in total and their human-written fixes. We automatically cluster these fixes and generate fix patterns, which show good agreement with a manual state-of-the-art taxonomy of fix patterns [6]. We thematically analyse the clusters that do not match the taxonomy and propose an additional eleven fix patterns that expand the taxonomy and can guide APR tools in fixing a greater range of bugs. Our replication package is available online: https://github.com/scc-fixie/fix-patterns/.

II. Background

Using code similarity in APR helps to find fixes and apply them to buggy code [15], [18], [21], [22], [31]. Ji et al. [31], Xin et al. [22], Wen et al. [21] and Jiang et al. [18] presented tools that calculated the similarity between fixing ingredients and buggy code. To rank candidate patches, Saha et al. [15] calculated the contextual similarity between a fix and the context of a buggy code snippet. One component of the reusability metric [31] was based on the similarity metric proposed by Baxter et al. [27]. Baxter's formula calculates the code similarity for clone detection Type 2 [32], which is a copy of code where a variable name, a type or a method name can differ from the original. In our work, we modify and apply Baxter's similarity formula on human-written fixes. Finding similarity at the level of code clones Type 2 allows us to find fixes across various methods, classes and projects that belong to the same fix pattern. Previous studies have used similarity metrics [15], [18], [21], [22], [31] for applying existing fix patterns, rather than for discovering new fix patterns.

To find similarities, APR tools usually represent code as tokens or AST nodes. When a method is converted into a linear sequence of AST nodes or tokens, the linear sequence can then be chopped up into sub-sequences using a sliding window of fixed length to generate n-grams. Using n-grams is a simpler model than an AST, and it preserves the order of neighbour nodes/tokens that form an n-gram. In defect prediction, Shippey et al. [33] used n-grams to identify common faulty AST nodes. Wang et al. [34] used n-grams in bug detection by identifying uncommon n-grams as bugs. Our use of n-grams differs from Shippey's and Wang's since we use n-grams to find similar fixes and generate their fix patterns.

Although APR has been a popular topic for more than a decade, there is still no well-known and established taxonomy of fix patterns accepted by APR researchers [35]. The first taxonomy was proposed by Pan et al. [36], who manually analyzed five open-source projects and extracted 27 fix patterns. Following this work, Kim et al. [9] presented ten fix patterns generated from open-source projects and evaluated them by their template-based APR tool, which outperformed an APR tool based on random edits. Sobreira et al. [10] manually generated 26 patterns from Defects4J human-written bug fixes. Koyuncu et al. [8] collected 13 fix patterns from the literature, implemented them an APR tool and evaluated them on Defects4J. To generate fix patterns from human-written fixes, researchers have also proposed mining data platforms and software repositories [12]–[14]. Liu and Zhong [12] presented 13 fix patterns generated from Stack Overflow. Long et al. [13]

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

focused on three types of bugs (null pointer, out of bounds, and class cast) from GitHub repositories and generated related fix patterns. Rolim et al. [14] presented nine fix patterns, also from human fixes from GitHub repositories. All the work mentioned above created a new set of fix patterns, instead of comparing against and improving an already published and known taxonomy, making the current state-of-knowledge in APR fix patterns disparate and difficult to build on.

Liu et al. [6] presented a systematic and detailed fix pattern taxonomy. The taxonomy contains 35 fix patterns collected from the literature and was presented together with the bugs from the Defects4J dataset automatically repaired by their tool TBar using these patterns. TBar outperforms other state-of-the-art APR tools in terms of the number of fixes. Since the TBar taxonomy is a systematic selection of fix patterns from the literature, we use it as a baseline in our work.

Another tool for mining software repositories, FixMiner, proposed by Koyuncu et al. [11], generated 31 fix patterns. Due to the lack of common representation of fix patterns in the APR literature, the authors manually performed a comparison of the FixMiner fix patterns against twelve state-of-the-art fix pattern taxonomies. By contrast, our tool automatically compares fix patterns (regardless of their representation) by comparing the clusters of fixes that can be described by the same fix pattern. This allows us to find the clusters that match the baseline clusters, as well as to find non-matching clusters that are used to expand the taxonomy of fix patterns and enable APR tools to fix a greater range of bugs in the future.

III. Methodology

In this section, we explain the workflow of our tool (see Fig. 1): how we represent code (III-A), generate code chunks of Java-specific AST tokens (*Kinds*) (III-B), post-process Kinds (III-C), generate n-grams (III-D) and calculate similarity (III-E). In addition, we present the clustering algorithm (III-F) that uses the calculated similarities to group similar chunks.

A. Code Representation

To represent code we use a linear simplified version of an AST representation that contains only types (*Kinds*) of AST nodes as a sequence. Kinds are Java-specific tokens that are implemented as an `enum` in the package `com.sun.source.tree`. To extract the Kind sequences, we extended the Java 8 `PrettyPrinter` from `com.sun.tools` to parse ASTs from class declarations and class methods.

We annotate standard enumerated Java Kinds with meta data. Additional information is extracted from an AST node (e.g. a variable name, a constant value, etc.) to be able to perform the differencing action between the buggy and fixed code. To illustrate original and annotated Kinds, we use the bug Closure-123 from Defects4J. The bug is an incorrect assignment and the fix replaces an enumerated constant `Context.OTHER` with the method invocation `getContextForNoInOperator(context)` (see Listing 2).

```
285c285
− Context rhsContext = Context.OTHER;
−−−
+ Context rhsContext = getContextForNoInOperator(context);
```

Listing 2: A human-written fix for Closure-123 deletes the line marked with '-' and adds the line marked with '+'.

For this example, our tool generates sequences of Kinds for the buggy and fixed method. The difference in Kinds is in the middle of the methods: two Kinds from the buggy method are replaced by three Kinds in the fixed method. The Kinds are then annotated (see Fig. 2 for the example in Listing 2).

In annotating, a Kind for a variable `IDENTIFIER` is extended by the variable name. A Kind for a field access `MEMBER_SELECT` is extended by the name of the variable and the name of the field. `METHOD_INVOCATION` and following `IDENTIFIER` are extended by the method name. Additionally, a Kind for a primitive type `PRIMITIVE_TYPE` would be extended by its type. A Kind for a literal (i.e. a constant value), e.g. `INT_LITERAL`, `FLOAT_LITERAL`, and `BOOLEAN_LITERAL`, would be extended by its value. Note that a Kind for a method declaration `METHOD` remains unchanged because its signature (the return value, the method name and the method parameters) are already described in the Kinds following the method declarations. Annotations of Kinds are used by a differencing tool to recognise and find differences between two pieces of code (before and after a fix) e.g., when a name of a variable was changed.

B. Representation of code chunks

When fixing a bug, developers modify source code by deleting, inserting or modifying parts of code. The difference between the original and revised versions of code is called a *fix*. In our implementation, a fix consists of one or multiple *chunks*. Declarations of class attributes are extracted into new methods during the parsing process so that chunks can also contain code modifications in class attribute declarations.

We use a differencing tool from the DiffUtils library [37] to compute a fine-grained difference between two sequences of Kinds. The resulting chunk consists of *deltas*, i.e., lists of consecutive modified Kinds. A delta is represented as an original and revised list of AST Kinds. Original and revised lists are empty when the modification action is "insert" and "delete", respectively. For our example of Closure-123 (Listing 2), the generated chunk consists of a delta that changes the original to the revised sequence of Kinds (see Fig. 3).

C. Post-processing of Kinds

Simplifying a chunk of a fix and preparing it for comparison against other chunks is performed in *post-processing*. Post-processing changes Kinds in a chunk by 1) removing the annotation added earlier because it is irrelevant for our similarity calculations, 2) detecting when the code was moved from one place to another, but not modified, and 3) making the modifications more general and using more generic Kinds to find similarities between chunks.

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

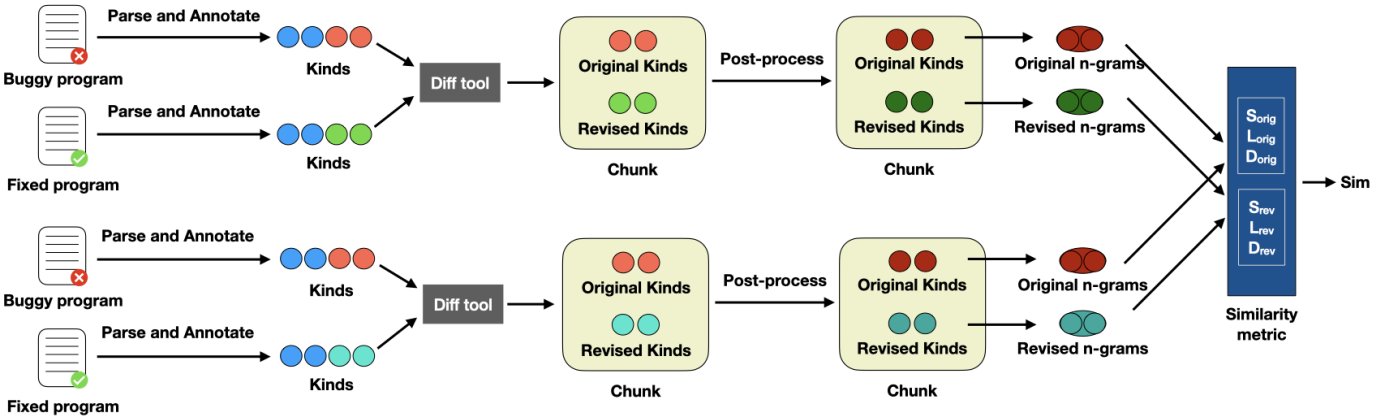

Fig. 1: The workflow of generating Kinds, chunks, n-grams and calculating their similarity. Blue Kinds are the Kinds that are the same in a buggy and fixed program. Red Kinds and n-grams are buggy and green and turquoise are fixed.

Buggy method	**Fixed method**
METHOD PRIMITIVE_TYPE$_{VOID}$ ...	METHOD PRIMITIVE_TYPE$_{VOID}$ ...
MEMBER_SELECT$_{Context.OTHER}$ IDENTIFIER$_{Context}$	METHOD_INVOCATION$_{getContextForNoInOperator}$ IDENTIFIER$_{getContextForNoInOperator}$ IDENTIFIER$_{context}$
... IDENTIFIER$_{cc}$ IDENTIFIER$_{n}$	... IDENTIFIER$_{cc}$ IDENTIFIER$_{n}$

Difference

Fig. 2: Kinds in the buggy and fixed method of Closure-123.

Original Kinds: MEMBER_SELECT$_{Context.OTHER}$
IDENTIFIER$_{Context}$

Revised Kinds: METHOD_INVOCATION$_{getContextForNoInOperator}$
IDENTIFIER$_{getContextForNoInOperator}$
IDENTIFIER$_{context}$

Fig. 3: The chunk of the fix for Closure-123.

The level of similarity we achieve with this processing is similar to the Type 2 clone [32]. Type 2 clones have the same sequence of Kinds and the same semantics, although a variable name, a type or a method name can differ. With the Type 2 clones, we can find the similarities between chunks across different methods, classes and projects.

To detect Type 2 clones, post-processing checks if "deleted" and "inserted" deltas have the same Kinds. This happens when a fix moves code from one place to another. Since Kinds in this type of fix are irrelevant, they are replaced with a new generic Kind `MOVE_BLOCK`. Similarly, deleting code is detected by having only a "deleted" delta, of which the content is irrelevant and replaced with a new generic Kind `DELETE_BLOCK`.

In addition, the Java 8 parser generates different Kinds for operators. Post-processing replaces operators and constants with a new generic Kind `OPERATOR` and `LITERAL`, respectively. A pair of `MEMBER_SELECT` and `IDENTIFIER` is simplified by replacing it with `IDENTIFIER` - a more generic representation of a variable (see Closure-123 Kinds in Fig. 4).

Original Kinds: IDENTIFIER

Revised Kinds: METHOD_INVOCATION
IDENTIFIER
IDENTIFIER

Fig. 4: The Closure-123 chunk after post-processing.

Original n-grams: [IDENTIFIER]

Revised n-grams: [METHOD_INVOCATION]
[METHOD_INVOCATION, IDENTIFIER]
[IDENTIFIER, IDENTIFIER]
[IDENTIFIER]

Fig. 5: The bigrams of the Closure-123 chunk.

D. Generating n-grams

Our final modification of chunks is to break a sequence of Kinds into *n-grams*. When $N = 1$, each n-gram consists of 1 Kind. When $N = max$ (max is the length of the whole chunk), the sequence is transformed into only one n-gram. When N is between 1 and max, the neighbour Kinds form n-grams. N-grams length 1, 2 and 3 are also called unigrams, bigrams and trigrams, respectively. In this work, n-grams are generated from original and revised Kinds separately. For our example of Closure-123, the bigrams are shown in Fig. 5. Note that the beginning and the ending of the sequence of n-grams contain n-grams lengths from 1 to N-1 and n-grams lengths from N-1 to 1, respectively. This way, we ensure that each Kind from a chunk appears N times in the sequence and has an equal impact on the calculation of similarity.

E. Similarity metric

Calculating the similarity between two chunks is performed by determining the proportion of n-grams common to both chunks regardless of the order of their n-grams. When comparing n-grams length $N = max$, two chunks can be similar if they have similar Kinds and a similar order of Kinds. With the lower value N, the order of n-grams has less impact on

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

the similarity. This allows chunks with the same changes in a slightly different order to be declared as similar.

We modified Baxter's similarity formula [27] for calculating the similarity between two sub-trees for clone detection:

$$Sim = \frac{2 * S}{2 * S + L + D}$$

where S is the number of shared nodes and L and R are the numbers of different nodes in one (left) sub-tree and the other (right) sub-tree, respectively. S can have a value between 0 (when there is no matching in sub-trees) to 1 (when sub-trees are equivalent).

Our similarity formula is:

$$Sim = \frac{2 * S_{orig} + 2 * S_{rev}}{2 * S_{orig} + L_{orig} + D_{orig} + 2 * S_{rev} + L_{rev} + D_{rev}}$$

Our code modifications are represented as n-grams, so the similarity formula compares 1) the original n-grams of the first chunk ("left") and the original n-grams of the second chunk ("right"), and 2) the revised n-grams of the first chunk ("left") and the revised n-grams of the second chunk ("right"). Our implementation ignores duplicated n-grams to avoid considering the same code modification more than once.

As explained earlier (Section III-C), when we compare two chunks, we do not require that there is an exact match between them. Similar to Type 2 code clones, we ignore variable names, method names and primitive types. Additionally, we ignore the type of operators and constant values. This enables us to find similar chunks across different methods, classes and projects.

F. Clustering

We use a threshold-based clustering algorithm [28] to automatically cluster chunks. Similarly to the *k*-means algorithm [29], [30], K items (i.e. generated chunks) are chosen from the dataset and each becomes the centre (*centroid*) of a cluster. Every other item is then assigned to a cluster with the smallest distance to the centroid. The distance is calculated as *1 - Sim*. If the distance is greater than a pre-defined threshold, the threshold-based algorithm creates a new cluster with the item and dynamically changes the number of clusters K. After assigning all items, for each cluster the item that has the minimum total distance to other items in the cluster becomes the centroid. The threshold-based algorithm eliminates one of the limitations of k-means because the number of clusters K does not have to be user-defined [38].

IV. Evaluation

In this section, we evaluate the similarity metric and the clustering algorithm using the Defects4J dataset [19]. Our tool automatically generates fix patterns out of the clusters with similar chunks and compares them with the clusters extracted from Liu et al. [6].

A. Baseline

Liu et al. presented a collection of fix patterns from the literature resulting in a taxonomy of 35 patterns used by their APR tool TBar. The taxonomy is well defined, straightforward to both use and expand when new fix patterns are discovered.

TBar outperforms other state-of-the-art APR tools (in terms of the number of fixed Defects4J bugs) by applying the fix patterns from their taxonomy to the Defects4J bugs (a previous version with 395 bugs). TBar generates partial, plausible and correct fixes from the 30 fix patterns. Five fix patterns did not generate fixes for any Defects4J bug, e.g., due to addressing different bug types than those found in Defects4J.

For our evaluation, we generated the baseline clusters out of information extracted from the TBar publication [6]. For each fix pattern in the TBar taxonomy, we made a cluster of the fixed Defects4j bugs (see details at https://github.com/scc-fixie/fix-patterns/blob/main/TBar.md). When multiple fixes were generated for the same bug, we chose the one most similar to the human-written fix and discarded the rest. Additionally, we decided to exclude three fixes (Lang-24, Math-4, Math-58) because they were not available online and we did not have access to their code modifications. As a result, the number of baseline clusters was 23 and the total number of TBar fixes 47.

To compare calculated and baseline clusters, we used the adjusted Rand index (ARI) [39], a preferable method for calculating the agreement between two sets of clusters without labeling [40]. The value of the ARI can be in the range from -1 to 1 (close to 0 for random clustering, 1 for total agreement and -1 for complete disagreement). We use the ARI implementation provided in the JSAT tool [41].

B. Characteristics of the Defects4J projects

The latest version of Defects4J (2.0.0) contains 835 bugs and their human-written fixes from 17 open-source projects: Chart, Cli, Closure, Codec, Collections, Compress, Csv, Gson, JacksonCore, JacksonDatabind, JacksonXml, Jsoup, JxPath, Lang, Math, Mockito and Time ("cha" , "cli", "clo", "cod", "col", "com", "csv", "gso", "jaC, "jaD", "jaX", "jso", "jxP", "lan", "mat", "moc", and "tim", respectively for shorter). The bugs are reproducible and covered by at least one failing test case, which passes when a fix is applied. The fixes contain only the code modifications applied to fix the bugs and are not related to code refactoring or implementing new features. As a result, Defects4J is a well defined dataset suitable for understanding software bugs and fixes, as well as for the evaluation of APR tools.

As explained in Section III, our tool transforms buggy and fixed code into sequences of Kinds and applies the DiffUtils tool to generate method-level chunks. We show the output related to chunks in Table I. The table contains the basic characteristics of Defects4J projects: the number of bugs, the number of generated chunks, the maximum number of chunks for a fix and the number of bugs fixed by TBar. Note that TBar was evaluated only on one of the previous versions of the Defects4J dataset that included six projects: Chart, Closure, Lang, Math, Mockito and Time.

Chart is a project with many bugs that require small local code changes. Many fixes (17 out of 26) modify only one

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

TABLE I: Characteristics of the Defects4J projects collected from Defects4J (1), our tool (2 and 3) and TBar (4).

	cha	clo	lan	mat	moc	tim	cli	cod	col	com	csv	gso	jaC	jaD	jaX	jso	jxP
1. Number of bugs	26	174	64	106	38	26	39	18	4	47	16	18	26	112	6	93	22
2. Number of generated chunks	39	444	98	158	94	54	64	24	6	80	21	33	57	227	6	169	53
3. Max number of chunks for a fix	4	46	9	11	20	10	8	4	3	10	4	11	7	22	2	16	9
4. Number of bugs fixed by TBar	14 (54%)	23 (13%)	14 (22%)	23 (22%)	3 (8%)	4 (15%)	-	-	-	-	-	-	-	-	-	-	-

method and, for the remaining fixes, the number of modified methods (and as a consequence, the number of generated chunks) is never greater than 4. TBar is more effective at repairing bugs from Chart (54%) than other projects.

Projects with fixes that modify a large number of methods (i.e., with the highest maximum numbers of chunks) are Closure and Mockito having fixes that modify 46 and 20 methods, respectively. These projects are also harder to fix automatically using TBar.

Since APR tools are able to fix only one portion of bugs in a dataset, clustering numerous human-written fixes gives us more opportunity to understand bugs and cluster their fixes. Even further, splitting complex fixes into chunks, which are simpler code modifications at the method level more likely to cluster with other chunks, allows us to have larger clusters that match an existing state-of-the-art fix pattern taxonomy, as well as to discover new fix patterns.

C. Clustering projects separately

After generating chunks, we apply our clustering algorithm to each project from Defects4J separately and compare our clusters with the TBar clusters for the same project by calculating the ARI values (see Fig. 6). Note that Mockito and Time are not shown due to a small number of fixes (three and four, respectively).

We vary the length of n-grams (*N*) from 1 (when each n-gram is one Kind) to the maximum (when one n-gram was generated out of all Kinds in a chunk) and calculate ARI for different similarity threshold values[1]. The similarity threshold is a parameter that controls how similar are the chunks that belong to the same cluster.

Although our aim is to cluster human-written fixes, we also apply our clustering algorithm on the fixes automatically generated by TBar (*N = 2 (auto)* in Fig. 6). With *N = 2 (auto)*, the agreement is improved due to eliminating the differences between human-written and automatically generated fixes. The remaining disagreement is a result of implementation differences between our clustering algorithm and the baseline.

For Chart, we get almost perfect agreement ($ARI = 0.93$) with the baseline for $N > 1$ and $threshold > 0.64$ (see Fig. 6) because the human fixes are very similar to the fixes automatically generated by TBar and our clustering algorithm generates almost the same clusters as the baseline.

Closure's fixes are more complex and harder to cluster, so the best agreement (ARI = 0.37) is for $N > 1$ and threshold ≥ 0.68 (see Fig. 6). Some human fixes are very different from the automatically generated fixes, so *N = 2 (auto)* significantly increases the agreement - the best agreement ($ARI = 0.64$) is for $threshold \geq 0.68$.

For Lang, we have a poor agreement generally: $ARI = 0.29$ for trigrams and $threshold = 0.24 - 0.26$ (see Fig. 6). One reason for the poor agreement is the low number of Lang fixes that form clusters with each other (7 out of 14 in the baseline clusters) and matching our clusters with the baseline clusters is seen more as due to chance than due to the good performance of our clustering algorithm. The second reason is a different implementation of human and automatically generated fixes. The comparison of clusters with only automatically generated fixes (*N = 2 (auto)*) shows a good agreement ($ARI = 0.55$) for and $threshold = 0.68 - 0.80$.

Clusters in Math achieve very good agreement with the baseline; there is some performance drop due to the different implementations of human and automatically generated fixes. Maximum $ARI = 0.71$ is achieved for any N value and $threshold \geq 0.9$ and $ARI = 0.85$ for *N = 2 (auto)* for $threshold \geq 0.9$ (see Fig. 6).

Although it is easier to analyse clusters and understand similarities when fixes from different projects are clustered separately, it is important to cluster fixes from various projects together to understand how and if bugs and fixes are similar across projects.

D. Overall project clustering

To measure the overall performance of our tool, we merged the clusters from all the Defects4J projects and compared them with the TBar clusters. In Fig. 7, we show the calculated ARI values with various N values for n-grams and various similarity threshold values (the parameters explained in Section IV-C).

Similarly to the ARI values for clustering individual projects separately, the ARI values for clustering all projects together are increased significantly for $threshold = 0.68$[2]. The maximum $ARI = 0.55$ is reached for $N = 2$ and $threshold = 0.86$. For this threshold value, the calculated clusters contain the same items as for $threshold = 1.0$, and additional ones that are not in baseline clusters. The number of chunks matching the baseline is 565 for $threshold = 0.86$ and 538 for $threshold = 1.0$. This shows that the calculated clusters include more items without deteriorating the performance when $threshold = 0.86$ and that the calculated clusters are

[1]N values between 3 and max are omitted from the charts due to the ARI values with an insignificant difference to N=3 and N=max.

[2]The increment is by 21.02%, 20.98%, 20.95% and 19.86% for N=1, N=2, N=3 and N=max, respectively, in comparison to threshold=0.66.

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

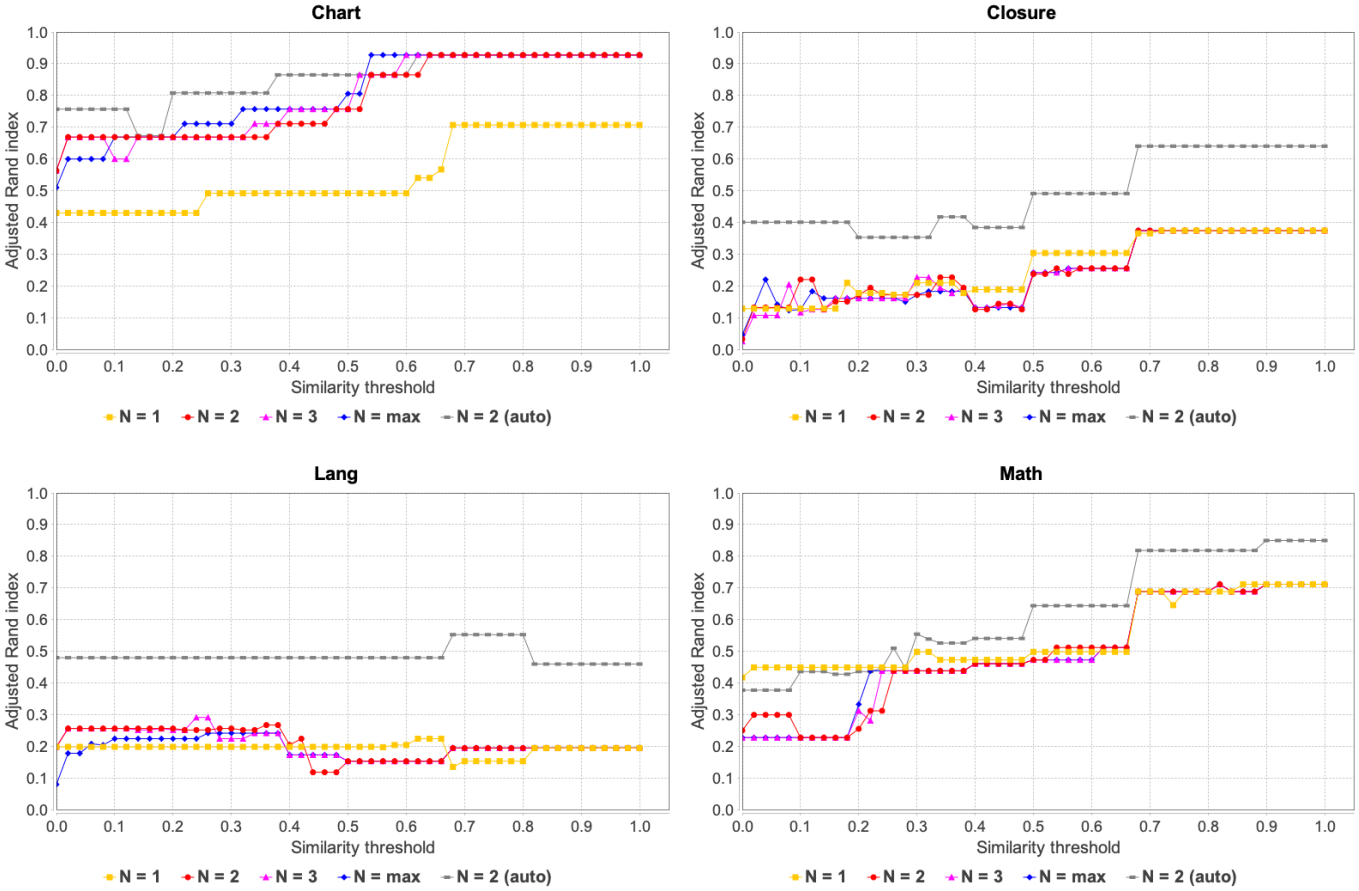

Fig. 6: Adjusted Rand index values of four projects: Chart, Closure, Lang and Math.

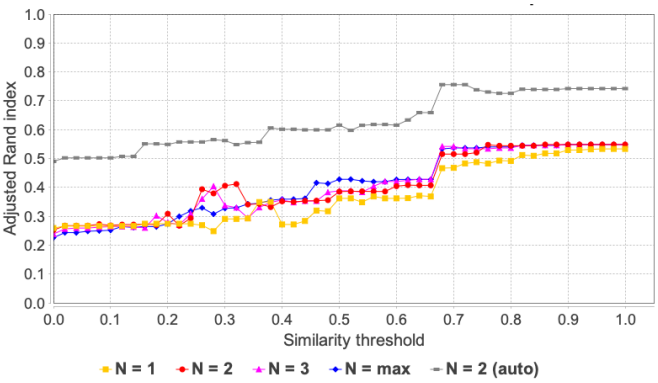

Fig. 7: Adjusted Rand index values of the projects together.

over-fitting when $threshold = 1.0$. With $threshold = 0.86$, more chunks match the state-of-the-art taxonomy. As a consequence, more bugs could be fixed by applying the fix patterns from the taxonomy than when $threshold = 1.0$.

Baxter et al. [27] effectively used unigrams ($N = 1$) when looking at the similarity of methods for identifying code clones. However, for unigrams, the similarity metric ignores the order of Kinds and in Fig. 7 we show that $N > 1$ outperforms unigrams for $threshold \geq 0.4$.

In Table II we show clusters that are automatically generated by our clustering algorithm with bigrams (because it reaches the maximum $ARI = 0.55$) and $threshold = 0.86$ (to avoid over-fitting) and compared with the TBar clusters. The items (chunks) are represented as a combination of the shorter project name and the bug number in the project. The index of an item shows which chunk (i.e. which part of a fix) belongs to the cluster. When an item does not have any index, this means that the item represents a complete fix.

For each cluster, we show a centroid, its fix pattern (as a simplified chunk with Kinds) and the matching TBar cluster. The full list of items in the clusters is available online: https://github.com/scc-fixie/fix-patterns/blob/main/MatchingFPs.pdf.

When we compare our and baseline clusters, a mismatch usually comes as a result of different implementations of human and automatically generated fixes. According to Wang et al. [42] who analysed the differences between human-written and machine-generated fixes, 25.4% of analysed machine-generated fixes are syntactically different from their human counterpart. In our case, eight human fixes are more complex, i.e., have a bigger number of modified lines of code (clo-38, lan-22, tim-7, clo-40, clo-21. clo-22, lan-63, mat-15). For five fixes, the implementation is not identical, but semantically

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

TABLE II: Fix patterns, centroids of the calculated clusters and their matching TBar fix patterns (FP).

	Fix pattern	**Centroid**	**TBar FP**
1	Replace an operator Change: (OPERATOR) to (OPERATOR)	mat-85	FP11.1
2	Replace a primitive type Change: (PRIMITIVE_TYPE) to (PRIMITIVE_TYPE)	mat-57	FP7.1
3	Replace a method invocation Change: (METHOD_INVOCATION, IDENTIFIER) to (METHOD_INVOCATION, IDENTIFIER)	jaD-31_1	FP10.1
4	Replace a variable Change: (IDENTIFIER) to (IDENTIFIER)	lan-21	FP13.1
5	Replace 2 variables Change: (IDENTIFIER, IDENTIFIER) to (IDENTIFIER, IDENTIFIER)	cha-20	FP10.2
6	Replace an assignment with a method invocation Change: (ASSIGNMENT, IDENTIFIER) to (METHOD_INVOCATION, IDENTIFIER)	mat-35_1	FP4.1
7	Replace a constant Change: (LITERAL) to (LITERAL)	cod-3	FP9.1
8	Replace an operand and a block with a constant Change: (OPERATOR) to (OPERATOR), Insert: (INSERT_BLOCK)), Delete: (DELETE_BLOCK)	mat-11	FP8.3
9	Remove a method Delete: (METHOD, DELETE_BLOCK)	clo-163_1	FP15.2
10	Remove a block Delete: (DELETE_BLOCK)	jso-68	FP15.1 | FP6.2
11	Move code Delete: (MOVE_BLOCK), Insert: (MOVE_BLOCK)	tim-7	FP14
12	Insert a parameter Insert: (IDENTIFIER)	lan-26	FP6.3
13	Insert a return statement Insert: (RETURN, LITERAL)	lan-51	FP4.2
14	Insert parentheses to change the order of operations Delete: (BLOCK), Insert: (PARENTHESIZED, OPERATOR)	mat-80	FP11.2
15	Insert a method invocation Insert: (METHOD_INVOCATION, IDENTIFIER)	mat-34	FP13.2
16	Insert a condition Insert: (OPERATOR, PARENTHESIZED)), Insert: (PARENTHESIZED, OPERATOR, IDENTIFIER, IDENTIFIER)	cha-9	FP6.3
17	Insert an if statement Insert: IF, PARENTHESIZED, OPERATOR, IDENTIFIER, LITERAL, BLOCK	tim-3_1	FP4.4
18	Insert an if block Insert: (IF, PARENTHESIZED, OPERATOR, IDENTIFIER, NULL_LITERAL, BLOCK, EXPRESSION_STATEMENT, ASSIGNMENT, IDENTIFIER, LITERAL)	lan-47_1	FP2.3
19	Insert a null pointer checker Insert: (IF, PARENTHESIZED, OPERATOR, IDENTIFIER, NULL_LITERAL, BLOCK)	jaD-80_1	FP2.1
20	Insert a null pointer checker with return Insert: (IF, PARENTHESIZED, OPERATOR, IDENTIFIER, NULL_LITERAL, BLOCK, RETURN, LITERAL)	cha-14_1	FP2.2
21	Insert a null pointer checker with throwing an exception Insert: (IF, PARENTHESIZED, OPERATOR, IDENTIFIER, NULL_LITERAL, BLOCK, THROW, NEW_CLASS, IDENTIFIER, LITERAL)	cha-19_1	FP2.5
22	Insert if-else with throwing an exception Insert: (IF, PARENTHESIZED, INSTANCE_OF, IDENTIFIER, PARAMETERIZED_TYPE, IDENTIFIER, UNBOUNDED_WILDCARD, BLOCK), Insert: (BLOCK, THROW, NEW_CLASS, IDENTIFIER, LITERAL)	mat-89	FP1

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

equivalent (clo-2, lan-33, lan-39, moc-29, moc-38). clo-63 is deprecated in the version of Defects4J that we are using so it is missing in our clusters. The more complex, different or missing (due to deprecation) fixes cause mismatches in clusters and reduce the agreement between our clustering algorithm and the baseline.

Each of our 21 calculated clusters matches a cluster from the baseline. Only the 10th cluster matches two clusters due to differences between our clustering algorithm and the baseline. The reason is that our approach is more general and it ensures that chunks with any code removal belong to one cluster (except a removal of a method, which is the 9th cluster).

Similarly to the results from clustering projects separately (see Section IV-C), when we apply our clustering algorithm on the TBar fixes (*N = 2 (auto)*), the agreement is improved due to eliminating the differences between human-written and automatically generated fixes. In this case, the best agreement ($ARI = 0.76$) is for $threshold = 0.68 - 0.72$.

Since the items in clusters are chunks, sometimes chunks of the same fix form a cluster together, e.g., two chunks of jaC-14 belong to the 1st cluster because its fix consists of a replacement of an operator in two methods. It also happens that chunks of the same fix belong to different clusters, e.g., cha-15 has one chunk in 19th cluster and another in 20th cluster. This means that some multiple-location bugs could be fixed by applying the same fix pattern multiple times or by applying different fix patterns simultaneously.

In Fig. 7, $N > 1$ outperforms unigrams for $threshold \geq 0.4$. Since unigrams are equivalent to Kinds, the results show that $N > 1$ is preferable for achieving a good agreement with the baseline. However, we still see the potential use of unigrams for clustering. E.g. cha-20 belongs to the 4th cluster when using unigrams since its fix replaces two variables in the same statement and it is seen as the same modification applied twice, so one is removed. When using bigrams, both modifications are included in the chunk, so cha-20 has no similarities with other chunks and it forms a cluster on its own. This example shows that if we combine the clusters from unigrams and bigrams, we could have a hierarchy of clusters - from more general (with unigrams) to more specific (with bigrams). The detailed clusters generated by using unigrams is available online: https://github.com/scc-fixie/fix-patterns/blob/main/UnigramFPs.pdf.

E. New fix patterns

Apart from the clusters presented in the previous section, our clustering algorithm also generates clusters that do not match the baseline taxonomy of fix patterns presented by Liu et al. [6]. To analyse, understand and organise the non-matching clusters and their items, and to expand the existing taxonomy with new clusters of chunks, we conducted a thematic analysis [43].

We selected three raters (from the authors of this paper) to perform the thematic analysis of the chunks from non-matching clusters. Firstly, the raters became familiar with Liu's taxonomy. Secondly, they received a random list of 25 chunks (five items from five clusters that did not match the baseline clusters). Thirdly, when possible, the raters matched the chunks with fix patterns from the baseline. Otherwise, they suggested new fix patterns. Initially, the raters agreed on 15 chunks. Cohen's Kappa value was 0.41, which showed a moderate agreement. After discussion (seen as an essential part of inter-rater agreement [44]), the raters agreed on the choice of fix patterns and the decisions needed to identify a pattern. The raters identified three new fix patterns. This round served as a pilot to ensure that the raters understood the baseline taxonomy and the steps of the thematic analysis.

In the second round, the raters received the remaining 151 chunks from the non-matching clusters with three or more chunks. Similar to the first round, the raters matched the chunks with the fix patterns from the baseline when possible or suggested new fix patterns. After discarding complex chunks (i.e., chunks that would be generated by applying more than one fix pattern) and excluding new fix patterns that were initially hard for raters to describe without discussion, Cohen's Kappa value was 0.63, which indicated substantial agreement. After a discussion that achieved agreement, the raters observed in total eleven new fix patterns as an extension of the baseline taxonomy. In Table III we show the fix patterns and the centroids of the new clusters as well as how they expand the TBar taxonomy. The full content of the clusters of the new fix patterns is available online: https://github.com/scc-fixie/fix-patterns/blob/main/NewFPs.pdf.

The 1st proposed fix pattern is a null pointer checker with returning a null value. Although our clustering algorithm makes a difference if a return statement is within parenthesis (jaD-13 and mat-4) or not (jso-22), the raters agreed that this is irrelevant and that this is the same fix pattern.

The raters observed the 2nd and the 3rd fix patterns from the cluster with the centroid com-29: an insertion of an assignment and an insertion of a field initialisation.

The 4th and the 5th proposed fix patterns are an insertion of a declaration with and without the initialisation, respectively. Note that our clustering algorithm generated two clusters with the same fix pattern "Insert Declaration with Initialisation" and they both match the 4th cluster. This is the only example we are aware of where the similarity of two centroids is greater than the similarity threshold and where two clusters should have been merged into one. Currently we are not able to explain this behaviour.

The 6th proposed fix pattern is an insertion of a case in a switch statement, and the 7th is an insertion of a variable initialisation. The 8th and 9th proposed fix patterns are a mutation of a literal and a method, respectively.

Finally, the 10th and 11th proposed fix patterns are different mutations of a variable: a variable is replaced with a new instance of a class, and a variable is replaced with a method invocation having the same variable as an argument.

These eleven fix patterns expand the baseline taxonomy and can enable APR tools to fix a greater range of bugs.

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication. Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

TABLE III: Fix patterns, centroids of the new clusters and proposed extension of the TBar fix pattern (FP) taxonomy.

	Fix pattern	Centroid	Expanded TBar FP
1	Insert a null pointer checker with return null Insert: (IF, PARENTHESIZED, OPERATOR, IDENTIFIER, NULL_LITERAL, RETURN, NULL_LITERAL)	jso-22_1	Insert Null Pointer Checker (FP2.6): + if (exp == null) return null;
2 | 3	Insert variable initialisation Insert: (IDENTIFIER, IDENTIFIER, EXPRESSION_STATEMENT, ASSIGNMENT)	com-29_1	Insert Assignment (FP4.5): + var1 = var2 | Insert Field Initialisation (FP4.6): + obj.field = var
4	Insert Declaration with Initialisation Insert: (PRIMITIVE_TYPE, LITERAL, VARIABLE)	com-3	Insert Declaration with Initialisation (FP4.7): + T var = literal
	Insert Declaration with Initialisation Insert: (PRIMITIVE_TYPE, LITERAL, VARIABLE)	lan-23	
5	Insert Declaration Insert: (IDENTIFIER, VARIABLE)	com-29_7	Insert Declaration without Initialisation (FP4.8): + T var
6	Insert Case Insert: (CASE, IDENTIFIER)	clo-103	Insert Case (FP4.9): + case exp
7	Insert an assignment Insert: (VARIABLE, PRIMITIVE_TYPE, LITERAL)	clo-30_2	Insert Initialisation (FP4.10): - var + var = literal
8	Replace null with a variable Change: (NULL_LITERAL) to (IDENTIFIER)	jaD-95_2	Mutate Literal (FP9.3): - literal + var
9	Replace a method invocation with a variable Change: (METHOD_INVOCATION, IDENTIFIER) to (IDENTIFIER)	jaC-12_3	Mutate Method (FP10.6): - method + var
10	Replace a variable with a new instance Change: (IDENTIFIER) to (NEW_CLASS, IDENTIFIER, IDENTIFIER)	moc-6_1	Mutate Variable (FP13.3): - var + new instance
11	Insert a method invocation Insert: (METHOD_INVOCATION, IDENTIFIER)	jso-29	Mutate Variable (FP13.4): - var + method(var, ...)

V. Threats to validity

Threats to *external validity* are related to our choice of the fix pattern taxonomy and the dataset. We chose Liu's fix pattern taxonomy [6] as our baseline because it is a selection of the various taxonomies found in the literature. The taxonomy is well presented and easy to adopt. In our evaluation, we use Defects4J which contains 835 bugs and fixes that are reproducible and covered by at least one test case. The fixes are isolated from any other code modifications, e.g., refactoring or implementing new features. These characteristics make Defects4J suitable for understanding bugs and fixes, and Defects4J has become an established dataset against which the performance of APR techniques can be benchmarked.

Threats to *internal validity* are related to the correctness of our thematic analysis. The novel fix patterns were reviewed and merged into the baseline taxonomy by three raters, all of whom had experience in developing software as well as in conducting research in automated predicting and repairing of software bugs. Every conflict among the raters was resolved by discussion, leading to agreement on either an existing fix pattern or a new fix pattern that expands the existing taxonomy.

VI. Conclusions and future work

We presented our approach to automatically generate fix patterns and compare them against the state-of-the-art taxonomy of fix patterns. Our tool splits fixes into smaller chunks (more likely to be clustered with other chunks) and calculates their similarity. The threshold-based clustering algorithm groups similar chunks together. Each cluster representing a fix pattern was compared against the taxonomy of fix patterns regardless of their representation. We evaluated our tool with human-written fixes from Defects4J. The tool generated 22 clusters that matched the baseline taxonomy with good agreement and 11 clusters that were thematically analysed and observed as new fix patterns that expand the baseline taxonomy.

We plan to expand the taxonomy further by generating fix patterns from other datasets and adding fix patterns from other taxonomies. Future application of the new fix patterns could improve the limited performance of current APR tools by guiding them to fix a larger set of bugs.

Acknowledgement

This work is funded by an Engineering and Physical Sciences Research Council grant EP/S005730/1.

This article has been accepted for publication in a future proceedings of this conference, but has not been fully edited. Content may change prior to final publication.
Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

REFERENCES

[1] Martin Monperrus. The living review on automated program repair. Technical Report hal-01956501, HAL/archives-ouvertes.fr, 2018.

[2] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, Tracy Hall, Saemundur Haraldsson, and John Woodward. On the introduction of automatic program repair in bloomberg. *IEEE Software*, 38(4):43–51, 2021.

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[4] M. Nayrolles and A. Hamou-Lhadj. CLEVER: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 153–164, 2018.

[5] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 349–358, 2017.

[6] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 31–42, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyande. AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, February 2019.

[8] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. Ifixr: Bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 314–325, New York, NY, USA, 2019. Association for Computing Machinery.

[9] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811, 2013.

[10] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140, 2018.

[11] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024, Mar 2020.

[12] X. Liu and H. Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129, 2018.

[13] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 727–739, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. Learning quick fixes from code repositories, 2018.

[15] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659, 2017.

[16] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 12–23, New York, NY, USA, 2018. Association for Computing Machinery.

[17] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 47(1):165–188, 2021.

[18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 298–309, New York, NY, USA, 2018. Association for Computing Machinery.

[19] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[20] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 302–313, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 660–670. IEEE Press, 2017.

[23] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224, 2016.

[24] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, Oct 2016.

[25] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M. Eskofier, and Michael Philippsen. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 61–72, New York, NY, USA, 2016. Association for Computing Machinery.

[26] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[27] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, 1998.

[28] Sanjiv K. Bhatia. Adaptive k-means clustering. In *International Florida Artificial Intelligence Research Society Conference*, 2004.

[29] E. Forgy. Cluster analysis of multivariate data : efficiency versus interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[30] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[31] T. Ji, L. Chen, X. Mao, and X. Yi. Automated program repair by using similar code containing fix ingredients. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 197–202, 2016.

[32] Rainer Koschke. Survey of research on software clones. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[33] Thomas Shippey, David Bowes, and Tracy Hall. Automatically identifying code features for software defect prediction: Using ast n-grams. *Information and Software Technology*, 106:142 – 160, 2019.

[34] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 708–719, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Eduardo Cunha Campos and Marcelo de Almeida Maia. Common bug-fix patterns: A large-scale observational study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 404–413, 2017.

[36] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[37] https://github.com/java-diff-utils/java-diff-utils.

Citation information: DOI: 10.1109/ISSRE52982.2021.00015, 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)

[38] S. E. Y. Nayini, S. Geravand, and A. Maroosi. A novel threshold-based clustering method to solve k-means weaknesses. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 47–52, 2017.

[39] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 1985.

[40] Douglas Steinley. Properties of the hubert-arabie adjusted rand index. *Psychological Methods*, 9(3):386–396, 2004.

[41] Edward Raff. Jsat: Java statistical analysis tool, a library for machine learning. *Journal of Machine Learning Research*, 18(23):1–5, 2017.

[42] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao. How different is it between machine-generated and developer-provided patches? : An empirical study on the correct patches generated by automated program repair techniques. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019.

[43] Daniela S. Cruzes and Tore Dybå. Research synthesis in software engineering. *Inf. Softw. Technol.*, 53(5):440–455, May 2011.

[44] Barbara Kitchenham, Dag I. K. Sjøberg, O. Pearl Brereton, David Budgen, Tore Dybå, Martin Höst, Dietmar Pfahl, and Per Runeson. Can we evaluate the quality of software engineering experiments? In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, New York, NY, USA, 2010. Association for Computing Machinery.