Enhancing Lexical Representation of Test Coverage for Failure Clustering

Juyeon Yoon
School of Computing
KAIST
Daejeon, Republic of Korea
juyeon.yoon@kaist.ac.kr

Shin Yoo
School of Computing
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

***Abstract*—Failure clustering aims to group multiple test failures based on shared root causes, helping developers to comprehend and debug each root cause (i.e., the underlying fault) in isolation. Clustering of failing test executions requires distances between those executions, for which distance measures between coverage vectors are widely used. Lexical representation of coverage has been suggested as an alternative, representing each structural element covered by a failing execution with the lexical tokens in the element. This paper investigates whether the granularity of the lexical representation affects the effectiveness of the failure clustering. We evaluate varying levels of tokenisation granularity by using them for clustering coexisting real-world test failures in Defects4J benchmark. Our results show that the traditionally adopted subtokenisation can actually deconstruct larger meaningful semantic token units, resulting in suboptimal clustering.**

***Index Terms*—Failure Clustering, Test Clustering, Coverage Representation**

I. Introduction

As the scale and the complexity of software increase, multiple test failures can occur simultaneously. When faced with multiple test failures, knowing which failure is due to which root cause is the critical first step for the subsequent debugging process. Failure clustering aims to group failing test executions that may have been caused by the same bug [1], [2], [3], [4].

Most failure clustering work assumes that test cases that fail due to the same bug would be *similar* to each other. As an example, tests sharing a considerable amount of covered elements are more likely to fail from the same cause than others that do not, given that the coverage of a test case can approximate its behaviour. Test coverage is generally represented as a binary vector, indicating whether a structural element (e.g. line, method) has been executed by the test or not. Jones et al. [2] have measured Jaccard distance between coverage vectors of the failing tests and performed hierarchical clustering. Golagha et al. [5] also utilised hierarchical clustering for grouping failing test coverage, and compares distance metrics such as Euclidean, Cosine, and Jaccard.

While being readily available, structural coverage is limited in its capability to capture the semantic behaviour of test executions. For example, some faults can exist across multiple locations because the developer made the same mistake in multiple locations. Any coverage based measure of test similarity will fail to capture this, as the different locations of the same fault will be captured as different coverage. To cater for such scenarios, DiGiuseppe et al. [4] introduced lexical representation of test coverage. Exploiting the fact that code lines with similar semantics are likely to share similar sets of identifiers, DiGiuseppe et al. tokenised covered source code lines into subword units (by splitting identifiers based on their naming conventions), and represented each test execution coverage as a bag of these subwords. Distances between test executions can be measured as distances between these bag of subwords. Their preliminary evaluation showed that the lexical representation could lead to better failure clustering.

The motivation behind subword tokenisation in existing work is based on the fact the source code identifiers are often composite words, i.e., consist of multiple words, each of which has its own semantic meaning: by breaking down identifiers to semantic units, it is expected that the resulting representation will capture their semantics more precisely.

However, we argue that breaking identifiers to their atomic semantic units may actually harm our application goal, which is failure clustering. By breaking down identifiers into subwords, we are essentially making each term in our bag of words representation *more common*, as the same subwords can appear in different identifiers that have different semantic meanings. Consequently, these unintended overlaps in subword occurrences may introduce noise into failure clustering when we use bags of subwords to represent failing test executions.

To avoid this problem, we hypothesise that tokenisation with coarser granularity may in fact be more beneficial for failure clustering, as it preserves the semantics associated with the source code better. We test our hypothesis by evaluating various granularities in the bag of words representation, with their resulting failure clustering effectiveness. We evaluate tokens, lines, and groups of similar lines as units in our bag, and apply failure clustering to real-world faults in the widely studied Defects4J benchmark. Our results suggest that source code lines consistently outperform subwords as the representation granularity, contrary to the belief that subword tokenisation is necessary. We also show that even coarser granularity, i.e., groups of similar lines, can sometimes outperform lines:

however, their performance is less consistent.

The technical contributions of this paper are as follows:

- We show that, for failure clustering, the ideal tokenisation granularity for lexical representation of test coverage is coarser than subwords.
- We suggest a refined embedding strategy for constructing the lexical representation based on similar line groups.
- We verify the effectiveness of lexical coverage representation for failure clustering on the multi-fault Defects4J dataset.

The remainder of this paper is organised as follows. Section II introduces the concept of lexical representation of test coverage, and motivates why we are after tokenisation with coarser granularity. Section III describes the configuration of our empirical evaluation, the results of which are reported in Section IV. Section V presents the related work, and Section VI concludes.

II. Lexical Coverage Representation for Failure Clustering

In this section, we briefly introduce the concept of lexical coverage representation that incorporates covered source code text. Next, we provide motivating examples that show the necessity of appropriate tokenisation granularity to better capture the test intents in vector representations.

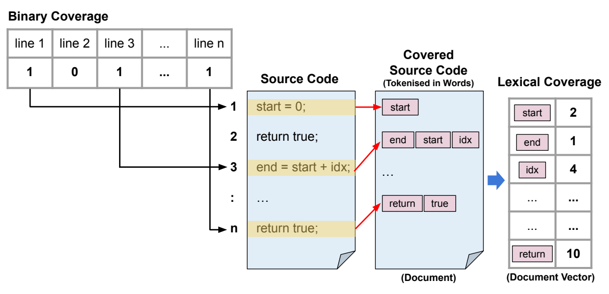


Fig. 1. Building lexical coverage representation

A. Building Lexical Coverage Representation

Coverage profiling tools typically produce coverage vectors that contain hit counts of program elements (e.g. statement). Each element of a coverage vector indicates whether a program element is covered by the test or not. Lexical representation of coverage, on the other hand, is based on the lexical tokens that form the covered program elements: by extracting tokens from the covered elements, we can build a *document* of covered source code, which we can vectorise, as described in Fig. 1.

```
    @@ -282,16 +282,14 @@ public FastDateFormat
         getDateInstance(int style, TimeZone ti..
285  -  if (locale != null) {
     +  if (locale == null) {
286  -    key = new Pair(key, locale);
     +    locale = Locale.getDefault();
287     }
     +  key = new Pair(key, locale);
    @@ -462,15 +460,13 @@ public FastDateFormat
         getDateTimeInstance(int dateStyle, int..
465  -  if (locale != null) {
     +  if (locale == null) {
466  -    key = new Pair(key, locale);
     +    locale = Locale.getDefault();
467     }
     +  key = new Pair(key, locale);
```

Listing 1. Root Cause of the Multi-location Fault

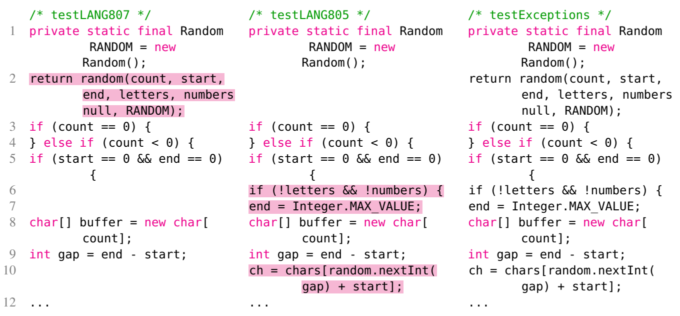
We use the source code in Listing 1 to illustrate how the lexical representation of coverage can complement the traditional coverage vector. The code snippet shows a multi-location fault from Defects4J. Defects4J contains two separate test cases that fail due to the same reason, which is the lack of a null check for the variable `locale`. However, each test case covers only one of the two faulty locations (Lines 285-285, and 465-466). In this case, traditional coverage vectors fail to capture the similarity between two failing tests, as their coverage patterns over the faulty locations are mutually exclusive.

However, note that Line 285 and Line 465 have identical source code content. A coverage representation that reflects the lexical contents of the covered program elements will be able to capture the affinity between these two failing test executions. We can borrow the Vector Space Model (VSM) [6] from Information Retrieval (IR) to achieve this. One of the most widely used VSM representations is the weighted term frequency (TF-IDF) [7], which represents each document using how frequently as well as uniquely each term appears in it. What remains is to determine the granularity with which we define a *term*.

1) Representing Coverage as Bag of Subwords: DiGiuseppe et al. [4] adopt subword-level tokenisation to vectorise coverage using TF-IDF. Here, subword tokenisation is adopted to better capture the domain concepts. For example, DiGiuseppe et al. argue that, by splitting `isTreeFull` into `is`, `Tree`, and `Full` following the camel case convention, it is possible to capture the association between the original identifier and the concept of a tree.

However, we posit that subword tokenisation may also be detrimental, as it may fail to preserve the surrounding contexts of each tokenised subword. For example, consider the following code snippets: `synchronizedMap(new HashMap<String, CharSet>())` and `private boolean useIdentityHashCode = true`. While both involve the subword `hash` in common, it is likely that these two snippets are not semantically close to each other. Upon closer inspection, the occurrence of `hash` in `synchronizedMap(new HashMap<String, CharSet>())` is simply due to the name of the generic datastructure, whereas the second occurrence is more application-specific. Bag of subwords representation fails to capture the difference in the surrounding contexts, as it does not preserve the ordering of subword tokens.

Faults in Defects4J actually contain such occurrences of common subwords. Consider the coverages of `testLANG805` and `testLANG807` in Fig. 2, which are from Apache `commons-lang` project. Note that `testLANG807` fails due to the fault in Line 6 and 7, whereas the root cause of `testLANG805` lies elsewhere. With subword tokenisation, Line 2, covered



```
    /* testLANG807 */            /* testLANG805 */            /* testExceptions */
 1  private static final Random  private static final Random  private static final Random
          RANDOM = new                 RANDOM = new                 RANDOM = new
         Random();                    Random();                    Random();
 2  return random(count, start,                               return random(count, start,
         end, letters, numbers                                     end, letters, numbers
         null, RANDOM);                                            null, RANDOM);
 3  if (count == 0) {            if (count == 0) {            if (count == 0) {
 4  } else if (count < 0) {      } else if (count < 0) {      } else if (count < 0) {
 5  if (start == 0 && end == 0)  if (start == 0 && end == 0)  if (start == 0 && end == 0)
          {                            {                            {
 6                               if (!letters && !numbers) {  if (!letters && !numbers) {
 7                               end = Integer.MAX_VALUE;     end = Integer.MAX_VALUE;
 8  char[] buffer = new char[    char[] buffer = new char[    char[] buffer = new char[
         count];                      count];                      count];
 9  int gap = end - start;       int gap = end - start;       int gap = end - start;
10                               ch = chars[random.nextInt(   ch = chars[random.nextInt(
                                      gap) + start];               gap) + start];
12  ...                          ...                          ...
```

Fig. 2. Covered lines of failing tests with 2 separate root causes: `testLANG805` and `testExceptions` share the faulty lines 6, 7.

by `testLANG807`, is tokenised into `return`, `random`, `count`, `start`, `end`, `letters`, `numbers`, and `null`. Line 6, 7, and 10, covered by `testLANG805`, after tokenisation, produce a large overlap with subwords in Line 2 with `letters`, `numbers`, `end`, `random`, and `start`. This overlap may undesirably reduce the distance between `testLANG805` and `testLANG807`.

Line 10 contains another example in which subword tokenisation can introduce ambiguity. The token `nextInt` will be split into `next` and `int`, both of which can result in overlaps with other lines as they are relatively common tokens, leading to unexpected affinity between test executions.

These examples support our claim that subword tokenisation can obscure the test intent due to the common usage of the fine-grained subwords in different contexts. Rows 'Subword' and 'Word' in Table I show the cosine distance computed between the entire coverages of the three test cases shown in Fig. 2 using subword and word tokenisation granularity, respectively. In both granularity levels, the cosine distance between `testLANG807` and `testLANG805` is shorter than that of `testExceptions` and `testLang805`, which actually share the same root cause.

TABLE I
COSINE DISTANCES AMONG FAILING TESTS WITH VARYING TOKENISATION GRANULARITIES

Token Unit	(testLang807, testExceptions)	**(testExceptions, testLang805)**	(testLang805, testLang807)
Subword	0.04	0.06	**0.02**
Word	0.12	0.19	**0.04**
Line	0.33	**0.20**	0.28

2) Representing Coverage as Bag of Lines: To overcome the issues in subword tokenisation, we propose a line level granularity to better preserve the contextual information on coverage. With line level granularity, we treat each individual source code *line* as a *term* in our VSM representation. Suppose two separate yet identical lines exist in different locations. If a test case covers these two identical lines, the frequency for the corresponding term would be two when computing TF-IDF. If all lines are unique in a given program, line level granularity would be identical to binary coverage based clustering. However, separate yet identical lines are relatively common: among the lines covered by an arbitrary test case from the 218 multiple fault versions of real-world open-source Java projects that we study, 19.65% are duplicates (Table III shows the breakdown by studied projects).

Consider the source code in Fig. 2 and the cosine distance in row 'Line' in Table I. The lexical coverage with line level tokenisation can correctly identify the two failing tests with the same root cause, `testExceptions` and `testLang805`, as being more similar to each other. This is because the overlap in term occurrences between coverages of `testLANG807` and `testLANG805` is significantly reduced, as Line 2 is identical to neither Line 6, 7, nor 10.

3) Representing Coverage as Bag of Line Groups: While the line level granularity can sometimes capture similarities between test coverages better than word or subword granularity, there are cases that require more flexibility when identifying similarities between coverage. Consider the example of another multi-location fault in Listing 2. The faulty version from Defects4J contains two failing test cases: `BigMatrixImplTest.testMath209` and `RealMatrixImplTest.testMath209`. These two test cases have mutually exclusive coverage, but they both fail from the same root cause, which is the misuse of `v.length` instead of `nRows`.

```
      @@ -988,7 +988,7 @@ public BigDecimal getTrace()
          throws IllegalArgumentException {
990     final int nCols = this.getColumnDimension();
991   - final BigDecimal[] out = new BigDecimal[v.length];
      + final BigDecimal[] out = new BigDecimal[nRows];
992     for (int row = 0; row < nRows; row++) {
      @@ -776,7 +776,7 @@ public double getTrace() throws
          IllegalArgumentException {
778     }
779   - final double[] out = new double[v.length];
      + final double[] out = new double[nRows];
780     for (int row = 0; row < nRows; row++) {
```

Listing 2. Multi-location Fault with Non-identical Hunks

Line level granularity partially captures the relevance of the two test executions that share some identical lines, such as Line 992 and Line 780 in Listing 2. However, the actual faulty lines (Line 991 and Line 779) are not identical, despite sharing the same semantics. Consequently, line level granularity fails to capture the similarity from this semantic overlap.

To overcome this, we propose grouping lexically similar lines, and use these groups as terms in our VSM representation. This is partially inspired by approaches in NLP research [8], [9] to treat *synonyms* on TF-IDF embeddings. Consider two natural language sentences, *"a student of proven ability"* and *"a graduate with proper skill"*, as well as their TF-IDF representations. We would get two distinct vectors if we tokenise at the word level. However, if we identify and group synonyms ($\{student, graduate\}$ and $\{ability, skill\}$) and use synonym groups as our terms, these two sentences will result in similar vectors. Similarly, we would like to consider the two faulty lines in Listing 2 (Line 991 and 779) as synonyms by grouping them together. The remainder of this section describes how we produce these line groupings. Listing 3 shows examples of lines clustered together when the distance threshold is set to 0.2. All four clusters show that it is possible to identify synonyms, i.e., the lines that exhibit similar behaviour based either on lexical patterns or variable types.

```
{'final double[] bpRow = bp[row];', 'final BigDecimal
     [] bpRow = bp[row];'}
{'final double[] bpCol = bp[col];', 'final BigDecimal
     [] bpCol = bp[col];'}
{'double normProduct = v1.getNorm() * v2.getNorm();',
     'double normProduct = u.getNorm() * v.getNorm()
     ;'},
{'return Double.NEGATIVE_INFINITY;', 'ret = Double.
     NEGATIVE_INFINITY;'}
```

Listing 3. Example of grouped lines using TF-IDF vector distance (threshold=0.2)

Finally, the constructed line groups can be used as terms to generate TF-IDF vectors. We modify the original TF-IDF formula to count the term frequency (tf) and inverse document frequency based on the line groups, where each line group g consists of lines $\{l_1, ...l_N\}$, d_t is a document of covered source code by a test t, and D is a set of documents, of which length is the number of all executed test cases.

$$tfidf(g, d_t, D) = \left(\sum_{l_i \in g}^{N} tf(l_i, d_t)\right) \cdot \left(\log \frac{|D|}{|\bigcup_{l_i \in g}\{d \in D : l_i \in d\}|}\right)$$

Note that this modified TF-IDF formula is basically identical to the original TF-IDF formula: it simply counts by treating the lines in the same group as identical.

B. Failure Clustering

To produce the final failure clustering result, we adopt agglomerative clustering, a type of hierarchical clustering, which is widely used by existing work on failure clustering [2], [5], [4]. Agglomerative clustering has the benefit of not having to know the number of clusters in advance. Instead, we simply need to set an adequate stopping criterion on intercluster distance: the merging of clusters stops when the intercluster distance becomes higher than the given threshold.

Prior work [4] set the stopping criterion based on sampled training sets. We adopt an unsupervised method that finds the *knee* point in intercluster distance, which is suggested as a way to find the ideal number of clusters without any training [10]. We performed a preliminary evaluation of stopping criteria using the line level tokenisation. First, we set the threshold value following Disiuseppe et al., using 20% of all data as the training set, and clustered the remaining 80% of the studied multiple fault versions. In addition, we clustered the same 80% of the studied multiple fault versions using the knee-point method. Table II shows that the knee-point method can outperform the threshold method in terms of our clustering evaluation metrics, AMI and ARI (see Section III-B for details).

III. Experimental Setup

This section presents our experimental setup.

TABLE II
Comparison of Stopping Criteria (Bag of Lines Representation)

Stopping Criterion	AMI	ARI
Distance Threshold	0.613	0.701
Knee Method	**0.643**	**0.758**

A. Multi-fault Dataset on Defects4J

To evaluate different granularity levels in lexical coverage representation in terms of the failure clustering effectiveness, we use the recently introduced multi-fault extension of Defects4J [11], which has identified coexisting faults in the original Defects4J [12] benchmark by transplanting fault revealing test cases across versions. The original Defects4J [12] benchmark is one of the most widely studied Java fault benchmark and contains isolated single faults.

From the multi-fault database, we sampled ten multi-fault versions per number of faults that ranges from one to five, from each of five Java projects: `jfreechart`, `closure-compiler`, `joda-time`, `commons-lang`, and `commons-math`. In total, we study 218 multi-fault snapshots of these Java projects, and not 250 (10 versions $\times$ 5 different number of faults (from one to five) $\times$ 5 projects), because `jfreechart` does not have enough multi-fault versions. In addition, 13 versions in `joda-time` and `closure-compiler` have overlapping failing tests among the contained faults, which are omitted because we use non-overlapping clustering. Table III shows the number of studied multiple fault versions per each project, as well as the average percentage of covered source code lines that are duplicates of other covered lines (see Section II-A2).

TABLE III
Multi-fault Dataset for Clustering Evaluation

Project	Lang	Chart	Time	Math	Closure	total
# of versions	50	31	38	50	49	218
% of dup. lines	11.6	18.5	23.9	14.2	20.9	19.6

B. Evaluation Metrics

Using the ground truth root causes for the failing tests in our dataset, we can compute external criteria of clustering quality. There are several suggested criteria used in the literature. Among them, we chose 2 criteria, Adjusted Mutual Information (AMI) and Adjusted Random Index (ARI). Both are adjusted metrics, meaning that they are not affected by the number of ground truth cluster labels.

1) Adjusted Mutual Information (AMI): Mutual Information (MI) is used to measure the information shared by two random variables. When applied to clustering, Mutual Information between the predicted clusters U and the true classes V is given as $MI(U,V) = \sum_{i=1}^{R}\sum_{j=1}^{C} P_{UV}(i,j) \log \frac{P_{UV}(i,j)}{P_U(i)P_V(j)}$, where $P_U(i)$ denotes the probability that an object falls into cluster U_i. This is further adjusted to account for chance. The AMI score between the two partitions U and V is given as:

$$AMI = \frac{MI(U,V) - E[MI(U,V)]}{avg(H(U), H(V)) - E[MI(U,V)]}$$

The calculated AMI score is 1.0 at perfect clustering and 0 at chance. Note that the value of AMI could be negative.

2) Adjusted Rand Index (ARI): Random Index (RI) in clustering evaluation is a measure of the percentage of correctly assigned element pairs. RI is calculated through the following equation: $RI = \frac{TP+TN}{TP+FP+FN+TN}$. For example, if two failing tests are assigned to the same cluster but not in the ground truth, we regard this pair as a false positive. Based on RI, the adjusted RI is obtained as follows:

$$ARI = \frac{RI - E[RI]}{max(RI) - E[RI])}$$

Random clusters have ARI close to 0, while a perfect cluster that is identical to the ground truth will have the value of 1.0.

C. Research Questions

We aim to answer the following research questions.

RQ1. Comparison of Tokenisation Granularities: How do the varying tokenisation granularities on constructing the lexical representation of coverage affect failure clustering? We answer RQ1 by comparing the clustering evaluation metrics, AMI and ARI, for failure clusters generated with different tokenisation granularities. We consider subword, word, and line level tokenisations. In addition, we also evaluate 'phrases', which are results of white-space tokenisation only (instead of tokenising also based on lexical delimiters such as the dot operator in Java). See Table IV for examples of all tokenisation levels.

TABLE IV
TOKENISATION GRANULARITIES FOR COMPARISON

line	{'`end = Integer.MAX_VALUE;`'}
phrase	{'`end`', '`Integer.MAX_VALUE`'}
raw word	{'`end`', '`Integer`', '`MAX_VALUE`'}
subword	{'`end`', '`integer`', '`max`', '`value`'}

RQ2. Impact of Line Distance on Line Group Granularity: What is the impact of the distance threshold used for line grouping? We answer RQ2 by computing failure clusters using line groups obtained with varying thresholds of line distance, and subsequently comparing the effectiveness using AMI and ARI.

RQ3. Effectiveness of Lexical Coverage Representation: Is the lexical coverage representation itself effective for failure clustering compared to the traditional binary coverage representation? We answer RQ3 by comparing the multiple failure clustering results: one obtained using Jaccard distance based on binary coverage representation, and others obtained using lexical representations of coverage at subword and line group level granularity.

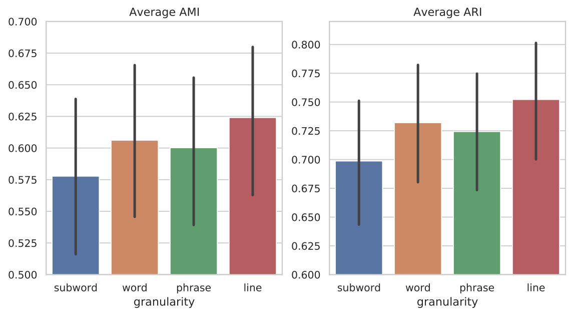


Fig. 3. Clustering performance (in AMI/ARI metrics) according to different tokenisation granularities

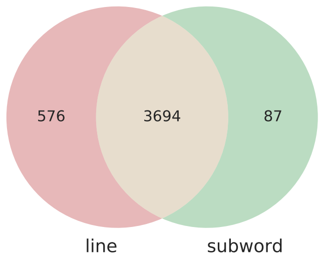


Fig. 4. Venn diagram drawn from the number of correctly clustered test pairs by bag of lines/subwords representations

IV. Result and Analysis

This section presents the results of our empirical evaluation and answers to the research questions.

A. Tokenisation Granularity (RQ1)

Fig. 3 shows the average AMI and ARI metrics of different tokenisation levels across all multiple fault versions studied. The vertical black bar shows the range of $\pm\sigma$. Bag of lines representation shows the best performance, with an AMI score of 0.624, which is about 8% higher than that of the subword representation, which is 0.578. This result supports our point that the subword source code unit may lose finer details in test coverage similarities, as discussed in Section II-A1.

Despite higher AMI and ARI, line level granularity does not completely dominate subword level granularity. The results suggest that similarities between certain pairs of tests are captured better at the coarser granularity, while for others the finer granularity performs better. In Fig. 4, we present a Venn diagram of pairs of failing test cases that are correctly clustered together by different granularity levels. While there is a large intersection (meaning that many pairs can be correctly clustered together at either granularity level), each granularity level can exclusively cluster different subsets of pairs (576 for line level granularity, 87 for subword granularity level). The result suggests that, for further improvement, we may have

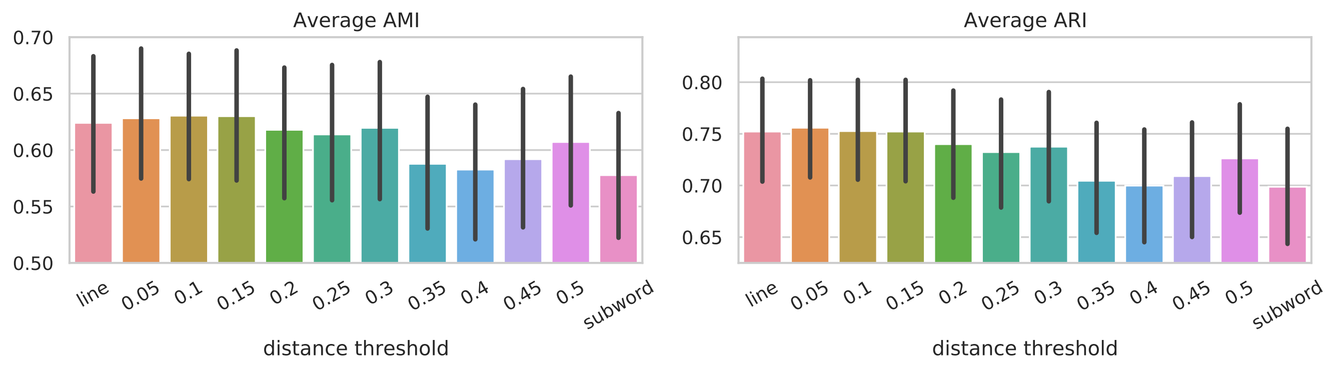


Fig. 5. Clustering performance (in AMI/ARI metrics) on line group based lexical coverage representations (different thresholds)

to consider multiple granularity levels simultaneously, using more advanced clustering algorithms.

Based on these results, we answer RQ1 that finer granularity tokenisation such as the subword tokenisation can at times actually harm the performance of failure clustering because of the unintended overlap in tokens. The line level granularity that we propose achieves the best overall clustering performance in terms of AMI and ARI metrics.

B. Line Group Granularity (RQ2)

Fig. 5 shows how average AMI and ARI change at the line grouping granularity level, as we vary the distance threshold for the grouping. At the left end, we show the result obtained with the line level granularity (which would be identical to line group granularity with a distance threshold of 0.0). We then vary the grouping distance threshold from 0.05 to 0.50 with the step size of 0.05, and report AMI and ARI. Finally, for reference, we also include the result from the subword level granularity.

Both the AMI/ARI metrics show slightly improved scores on some thresholds: 0.05, 0.1, 0.15 for AMI, and 0.05 for ARI. Compared to the subword unit, the line-grouping based representation with the best-performing threshold of 0.1 showed 8.6% improvement with an AMI score of 0.63. However, we also observe a trend that, above a certain distance threshold, line grouping becomes too lax, and fails to improve the clustering. Based on these results, we answer RQ2 that line groups can be an even better unit of lexical representation of coverage with appropriate distance threshold. However, the magnitude of improvement is smaller when compared to the improvement achieved by line granularity level over the subword level granularity.

C. Lexical Coverage Representation (RQ3)

Fig. 6 compares failure clustering results from subword and line group granularity levels to that of binary coverage representation: we use Jaccard distance to perform the clustering. It clearly shows that both lexical representations can produce better failure clustering results. To the best of our knowledge, this is the first empirical evidence that shows the

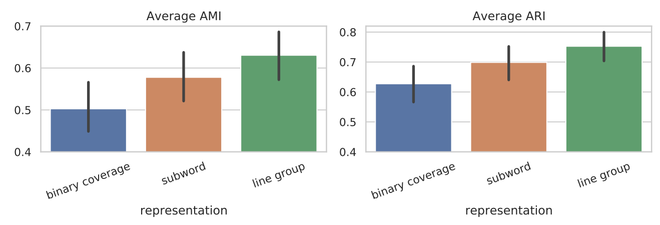


Fig. 6. Comparison of clustering performance (in AMI/ARI metrics) on binary coverage with Jaccard distance, subword-level lexical coverage, and line group based lexical coverage (threshold 0.1)

benefits of lexical representation of coverage for real-world failure clustering. While DiGiuseppe et al. [4] also presented experimental results, the preliminary evaluation included only a single C program, `sed` from SIR [13], which contains manually injected faults instead of real ones. Based on these results, we answer RQ3 that utilising lexical source code tokens for representing coverage helps produce better failure clustering than conventional binary coverage representation.

V. Related Work

This section presents related work, which includes existing work on failure clustering and representations of source code.

A. Failure Clustering

Podgurski et al. [1] first proposed to use execution traces for clustering failing executions. Much early failure clustering work [3], [2] uses execution traces to distinguish execution paths of each failure, but does not consider the lexical tokens in the covered source code. Jones et al. [2] defined the concept of parallel debugging by assigning different *fault-focused* clusters of failing tests cases to the developers, using the suspiciousness score of each line.

DiGiuseppe et al. [4] suggested using lexical source code tokens for failure clustering by presenting the notion of Concept-based failure clustering. We expand this work by considering alternative granularity levels for lexical representation of coverage, and provide an empirical evaluation of failure clustering using larger and more realistic programs and faults.

Chen et al. [14] use execution traces as well as the tokens in program output to filter out undesirable test cases, but do not use the tokens in the covered program source code.

B. Source Code Representation

Recently, several techniques to generate source code representation at various granularities such as identifier [15], [16], expressions [17], method [18], and code changes [19] were suggested, but none of them attempts to embed all covered source code to measure the distance between test coverage. For embedding identifiers, subtokenisation was generally regarded to be necessary [20], [21], [22], but the drawbacks of using subtokenisation have rarely been addressed.

VI. Conclusion

In this paper, we study the importance of tokenisation granularity on lexical coverage representation and, in turn, on the effectiveness of failure clustering based on the representation of coverage. Contrary to the use of subword tokenisation in existing work, we propose coarser-grained units of coverage representation and show that they are capable of producing better failure clustering. As future work, we will consider more advanced representations (such as one that is capable of incorporating multiple granularity levels) and clustering algorithms (such as one that can consider multiple distance metrics) for more precise failure clustering.

Acknowledgement

This work is supported by National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629), as well as Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001).

References

[1] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 465–475.

[2] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[3] W. Dickinson, D. Leon, and A. Podgurski, "Pursuing failure: the distribution of program failures in a profile space," in *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of Software Engineering*, 2001, pp. 246–255.

[4] N. DiGiuseppe and J. A. Jones, "Concept-based failure clustering," in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, 2012, pp. 1–4.

[5] M. Golagha, A. Pretschner, D. Fisch, and R. Nagy, "Reducing failure analysis time: An industrial evaluation," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).* IEEE, 2017, pp. 293–302.

[6] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, Nov. 1975.

[7] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets.* New York, NY, USA: Cambridge University Press, 2011.

[8] Z. Yun-tao, G. Ling, and W. Yong-cheng, "An improved tf-idf approach for text classification," *Journal of Zhejiang University-Science A*, vol. 6, no. 1, pp. 49–55, 2005.

[9] M. Kumari, A. Jain, and A. Bhatia, "Synonyms based term weighting scheme: An extension to tf. idf," *Procedia Computer Science*, vol. 89, pp. 555–561, 2016.

[10] S. Salvador and P. Chan, "Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms," 12 2004, pp. 576–584.

[11] G. An, J. Yoon, and S. Yoo, "Searching for multi-fault programs in defects4j," *arXiv preprint arXiv:2108.04455*, 2021.

[12] R. Just, D. Jalali, and M. 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*, 2014, pp. 437–440.

[13] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[14] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 197–208.

[15] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[16] Y. Wainakh, M. Rauf, and M. Pradel, "Idbench: Evaluating semantic representations of identifier names in source code," *arXiv preprint arXiv:1910.05177*, 2019.

[17] M. Allamanis, P. Chanthirasegaran, P. Kohli, and C. Sutton, "Learning continuous semantic representations of symbolic expressions," in *International Conference on Machine Learning.* PMLR, 2017, pp. 80–88.

[18] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[19] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[20] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).* IEEE, 2020, pp. 1073–1085.

[21] M. Hucka, "Spiral: splitters for identifiers in source code files," *Journal of Open Source Software*, vol. 3, no. 24, p. 653, 2018.

[22] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *European Conference on Object-Oriented Programming.* Springer, 2011, pp. 130–154.