Quality Assurance of Bioinformatics Software: A Case Study of Testing a Biomedical Text Processing Tool Using Metamorphic Testing[*]

Madhusudan Srinivasan
Montana State University
Bozeman, Montana
madhusuda.srinivasan@montana.edu

Morteza Pourreza Shahri[†]
Montana State University
Bozeman, Montana
mpourrezashahri@montana.edu

Indika Kahanda
Montana State University
Bozeman, Montana
indika.kahanda@montana.edu

Upulee Kanewala[‡]
Montana State University
Bozeman, Montana
upulee.kanewala@montana.edu

ABSTRACT

Bioinformatics software plays a very important role in making critical decisions within many areas including medicine and health care. However, most of the research is directed towards developing tools, and little time and effort is spent on testing the software to assure its quality. In testing, a test oracle is used to determine whether a test is passed or failed during testing, and unfortunately, for much of bioinformatics software, the exact expected outcomes are not well defined. Thus, the main challenge associated with conducting systematic testing on bioinformatics software is the oracle problem.

Metamorphic testing (MT) is a technique used to test programs that face the oracle problem. MT uses metamorphic relations (MRs) to determine whether a test has passed or failed and specifies how the output should change according to a specific change made to the input. In this work, we use MT to test LingPipe, a tool for processing text using computational linguistics, often used in bioinformatics for bio-entity recognition from biomedical literature.

First, we identify a set of MRs for testing any bio-entity recognition program. Then we develop a set of test cases that can be used to test LingPipe's bio-entity recognition functionality using these MRs. To evaluate the effectiveness of this testing process, we automatically generate a set of faulty versions of LingPipe. According to our analysis of the experimental results, we observe that our MRs can detect the majority of these faulty versions, which shows the utility of this testing technique for quality assurance of bioinformatics software.

KEYWORDS

Software testing, Metamorphic testing, Bioinformatics, Biomedical Natural Language Processing

ACM Reference Format:
Madhusudan Srinivasan, Morteza Pourreza Shahri, Indika Kahanda, and Upulee Kanewala. 2018. Quality Assurance of Bioinformatics Software: A Case Study of Testing a Biomedical Text Processing Tool Using Metamorphic Testing. In *MET'18: MET'18:IEEE/ACM International Workshop on Metamorphic Testing , May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3193977.3193981

1 INTRODUCTION

Researchers studying living organisms often rely on bioinformatics tools to answer research questions and develop conclusions [3]. These bioinformatics tools help to create considerable biomedical research breakthroughs involving genomes [16]. For example, bioinformatics tools are used in predicting the functions and modeling the structure of proteins [3] as well as drug discovery [16]. At the same time, laboratory experiments and clinical investigations are highly time consuming and may require up to 15 years and millions to billions of dollars for introducing a compound to the market [16]. Hence bioinformatics tools can help in analyzing the data faster and eventually reduce the time for discovery. This as a result can reduce the cost for researchers as well as the industry.

The use of software systems have grown exponentially over the years [18]. But, the number of problems due to software faults have also increased in recent times [18]. Thus, software testing plays a major role in detecting as many software faults as possible [8]. Bioinformatics programs are complex and require novel techniques to test them [9]. Typically, these tools are developed by researchers for the analysis of biological and biomedical data but they are not tested adequately with varied and all possible levels of test data either due to time or budget constraints [12]. Previous studies have shown that bioinformatics tools are not rigorously tested before being delivered to the end users since it is hard to validate the correctness of the output [2]. Furthermore, testing of bioinformatics

[*]Produces the permission block, and copyright information.
[†]The authors wish it to be known that, in their opinion, the first two authors should be regarded as joint First Authors.
[‡]Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MET'18, May 27, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5729-6/18/05…$15.00
https://doi.org/10.1145/3193977.3193981

tools may require extensive interactions between the testers and the domain experts. The lack of communication can lead to the failure of using unit testing for important modules in the tool which reduces the reliability of the bioinformatics tools [10].

Test oracle is an essential part of software testing and it is a mechanism for verifying the correctness of test output for a given set of test inputs. Many bioinformatics programs fall into the category of non-testable programs, where either a test oracle is unavailable or it is practically difficult to develop test oracles [17]. Metamorphic testing (MT) is a testing technique that can alleviate the test oracle problem [7]. Metamorphic testing involves identifying a set of properties from the program under test known as metamorphic relations (MRs). Multiple test cases are executed and results of the test cases are used to check whether they satisfy or violate the identified MR. Violation of MRs indicate faults in the program. We hypothesized that MT can be an effective approach for testing bioinformatics tools.

In this study, we explore the feasibility and effectiveness of using MT for bioinformatics software. In particular, we identify metamorphic relations for the task of biomedical natural language processing. We use MT to test LingPipe [1], a tool for processing text using computational linguistics and often used in bioinformatics for bio-entity recognition [4, 5]. Bio-entity recognition is the process of extracting biomedical terms from text, and assigning them to appropriate categories [1]. Bio-entity recognition tools such as LingPipe use named-entity detection to extract bio-entities such as genes, organisms, and diseases from very large corpora of biomedical literature [1]. The typical outputs from a bio-entity recognition process are hard for developers to validate and require the support from domain experts. The effectiveness of our proposed metamorphic relations are evaluated using mutants [8]. Our results show that proposed MRs identify a majority of the mutants used in the evaluation and indicates that MT is an effective method for testing biomedical natural language processing software.

The rest of this paper is organized as follows. Section 2 describes the background related to test oracle, MT and bio-entity recognition. Section 3 provides details of each of the MRs we identify for testing bio-entity recognition software. Section 4 provides the methodology for validating the LingPipe tool using the proposed metamorphic relations. Sections 5 and 6 provides the experimental setup and the results, respectively. Section 7 discusses the key observations from experimental results while Section 8 provides related works. Section 9 concludes the paper and discusses promising future directions.

2 BACKGROUND

In this section, we provide some background on test oracles, metamorphic testing, and bio-entity recognition. The test oracles section focuses on the definition of an oracle, test oracle problem and test oracle problem related to our case study tool. The metamorphic testing section describes the step by step approach for conducting metamorphic testing. The Bio-entity recognition section provides a brief description of Named Entity Recognition and its utilization in the biomedical area.

2.1 Test Oracles

Test oracle is a mechanism used to verify the correctness of the output produced by a program [17]. The *Oracle problem* occurs when there is not an existing oracle or when it is practically difficult to determine the correct output [17]. The oracle problem is a common problem in testing scientific software [11]. Programs which face the oracle problem are called *non-testable programs*. Scientific softwares such as LingPipe used in our case study fall into the category of non-testable softwares. The LingPipe outputs are nondeterministic, long and complex which makes it difficult to clearly define a test oracle, as described in detail below (section 2.3).

2.2 Metamorphic Testing

Suppose that a function f is implemented by a program p. Steps for conducting metamorphic testing on p is as follows:

(1) Identify MRs for function f.
(2) Create source test cases. Source test cases I_1 is derived using the traditional test case selection methodology. Most common method for test case selection is random testing.
(3) Follow-up test case I_2 is derived from I_1 by applying MR to the source test case I_1.
(4) Execute test case I_1 and I_2 and get output O_1 and O_2.
(5) If O_1 and O_2 results does not satisfy the MR, then a fault is assumed to be present in the program.

Consider a function that calculates the sum of integer elements in an array. An MR for this summation function can be defined using the property that shuffling the array elements should not change the actual sum of those elements. In order to test this function using this MR, an array containing some integers is used as a source test case. The follow up test case is a shuffled version of that same array. The outputs (in this case sums) obtained from the execution of source and follow up test cases is compared. MR violation occurs when a difference between the outputs are observed.

2.3 Bio-entity Recognition

Bio-entity recognition is the task of identifying and/or extracting bio-entities such as gene and protein names from text. LingPipe [1] is a natural language processing tool often used for the bio-entity recognition by the bioinformatics community [4, 5]. LingPipe employs conditional probability estimates for the *n-best* mentions of entities by utilizing a first-order Hidden Markov Models (HMM) for the task of tagging [5]. Since this is not a deterministic model and not as trivial as simple dictionary-mapping, it is not possible to predict LingPipe outputs with a hundred percent accuracy.

One of the main modules of LingPipe is the "Named Entity Recognition (NER)" module. The NER module employs a pre-trained model to extract the entity terms mentioned in the input text [1]. LingPipe is able to extract bio-entities by using a model pre-trained on biomedical data. There are three trained NER models on LingPipe's website, two out of which (GENIA and GENETAG) are trained on biomedical text which emphasizes the usage of LingPipe in the fiend of bioinformatics. In this work we use the GENETAG model. The following example shows the protein name extracted from the given sample text using LingPipe.

Sample Sentence (PMID: 29320757):
"Neuritin plays an important role in the development and regeneration of the nervous system, and shows good prospects in the treatment and protection of the nervous system".

LingPipe output:
Term = Neuritin, Position = (0, 8)

As shown in this example, LingPipe returns the occurrences of bio-entities within the input text. The returning data contains a list of terms and corresponding positions. The term can be a single word or a combination of multiple words. The position indicates the start and end indices of the corresponding terms occurring within the input text. When applied to a large corpus of text, LingPipe returns a very large number of bio-entities and introduces difficulty in validating the results. Since it is impossible predict the complete set of genes extracted by the Linpipe's probabilistic model with a hundred percent accuracy, an oracle for the task of bio-entity recognition does not exist. Therefore, the oracle problem persists with bio-entity recognition tools such as LingPipe.

3 METAMORPHIC RELATIONS FOR BIO-ENTITY RECOGNITION

We propose three categories of MRs for testing bio-entity recognition software. These categories are Addition, Deletion, and Shuffling relations. In the Addition relations, we extend a span of text by adding it to another span. For example, a sentence can be appended to another sentence, a sentence to a paragraph, a paragraph to an article, or a list of random words to another list of random words. In the Deletion relation, a span of text is truncated by removing a part of it. For instance, removing a consecutive list of words from a sentence, removing a sentence from a paragraph, removing a paragraph from an article, or removing a part from a list of random words. In the Shuffling relations, a span of text gets a new form. Rearranging the paragraphs of an article, or all the words of a span of text are examples of Shuffling relations. According to a preliminary analysis, shuffling the sentences in a paragraph did not satisfy the metamorphic relations; hence not used in this study. All of the above metamorphic relations were validated by bioinformatics domain experts.

In the following definitions, each extracted biomedical entity (BE) contains a term and the position of the extracted BE in the original input text, which are referred to as BE_t and BE_p, respectively. Further, $length(T)$ is the length of the text T in number of charters.

3.1 Addition Relations

3.1.1 MR1: Adding a sentence to another sentence. Given two sentences S_1 and S_2, we append S_2 to S_1. Referring to the new text as $S\prime$,

$$BE_t(S\prime) = BE_t(S_1) \cup BE_t(S_2)$$

$$BE_p(S\prime) = \{x \mid x \in BE_p(S_1)\} \cup \{x + length(S_1) \mid x \in BE_p(S_2)\}$$

3.1.2 MR2: Adding a sentence to a paragraph. Given a sentence S and an index i, we add it to the position i of a paragraph P. We refer to the resultant text of this addition as $P\prime$. As a result,

$$BE_t(P\prime) = BE_t(P) \cup BE_t(S)$$

If the sentence is added before the start of the paragraph (i.e. $i = 0$),

$$BE_p(P\prime) = \{x \mid x \in BE_p(S)\} \cup \{x + length(S) \mid x \in BE_p(P)\}$$

If the sentence is added to the end of the paragraph,

$$BE_p(P\prime) = \{x \mid x \in BE_p(P)\} \cup \{x + length(P) \mid x \in BE_p(S)\}$$

If the sentence is added to the middle of the paragraph (i is neither the start nor the end of the paragraph),

$$\begin{aligned} BE_p(P\prime) = \{x \mid x \in BE_p(P) \wedge x < i)\} \cup \\ \{x + length(S) \mid x \in BE_p(P) \wedge x >= i)\} \cup \\ \{x + i \mid x \in BE_p(S))\} \end{aligned}$$

3.1.3 MR3: Adding a paragraph to an article. Given a paragraph P and an index i, we add this paragraph to the position i of an article A. We refer to the resultant text of this addition as $A\prime$. As a result, the following equations should be true.

$$BE_t(A\prime) = BE_t(A) \cup BE_t(P)$$

If the paragraph is added before the start of the article (i.e. $i = 0$),

$$BE_p(A\prime) = \{x \mid x \in BE_p(P)\} \cup \{x + length(P) \mid x \in BE_p(A)\}$$

If the paragraph is added to the end of the article,

$$BE_p(A\prime) = \{x \mid x \in BE_p(A)\} \cup \{x + length(A) \mid x \in BE_p(P)\}$$

If the paragraph is added to the middle of the article (i.e. i is neither the start nor the end of the article),

$$\begin{aligned} BE_p(A\prime) = \{x \mid x \in BE_p(A) \wedge x < i)\} \cup \\ \{x + length(P) \mid x \in BE_p(A) \wedge x >= i)\} \cup \\ \{x + i \mid x \in BE_p(P))\} \end{aligned}$$

A high-level depiction of MR3 is given in Figure 1.

3.1.4 MR4: Adding a list of random words to another list of random words. Given two list of random words L_1 and L_2, we append L_2 to L_1 and refer to the resultant list as $L\prime$. In this MR, we make sure that L_2 is added to L_1 with a newline in-between to avoid combining the words at the interconnecting position. We should have the following relations:

$$BE_t(L\prime) = BE_t(L_1) \cup BE_t(L_2)$$

$$BE_p(L\prime) = \{x \mid x \in BE_p(L_1)\} \cup \{x + length(L_1) \mid x \in BE_p(L_2)\}$$

3.2 Deletion Relations

3.2.1 MR5: Removing a list of words from a sentence. Given a sentence S, we remove a list of words, L, from S. Referring to the resultant text of this deletion as $S\prime$,

$$BE_t(S\prime) = BE_t(S) - BE_t(L)$$

$$BE_p(S\prime) = \{x - c \mid x \in BE_p(S) \wedge c \in \mathbb{Z}_{\geq 0}\}$$

$\mathbb{Z}_{\geq 0}$ is the non-negative integer space.

3.2.2 MR6: Removing a sentence from a paragraph. Given a paragraph P and an index i, we remove a sentence S, starting from position i, from P. We refer to the resultant text of this deletion as $P\prime$. Therefore,

$$BE_t(P\prime) = BE_t(P) - BE_t(S)$$

If the first sentence of the paragraph is removed (i.e. $i = 0$),

$$BE_p(P\prime) = \{x - length(S) \mid x \in BE_p(P) \wedge x \notin BE_P(S)\}$$

If the last sentence of the paragraph is removed,

$$BE_p(P\prime) = \{x \mid x \in BE_p(P) \wedge x \notin BE_P(S)\}$$

If a sentence is removed from the middle of the paragraph (i.e. i is neither the start nor the end of the paragraph),

$$BE_p(P\prime) = \{x \mid x \in BE_p(P) \wedge x < i\} \cup \{x - length(S) \mid x \in BE_p(P) \wedge x \notin BE_p(S) \wedge x > i\}$$

3.2.3 MR7: Removing a paragraph from an article. Given an article A, we remove a random paragraph P from A. Referring to the resultant text of this deletion as $A\prime$,

$$BE_t(A\prime) = BE_t(A) - BE_t(P)$$

If the first paragraph of the article is removed (i.e. $i = 0$),

$$BE_p(A\prime) = \{x - length(P) \mid x \in BE_p(A) \wedge x \notin BE_P(P)\}$$

If the last paragraph of the article is removed,

$$BE_p(A\prime) = \{x \mid x \in BE_p(A) \wedge x \notin BE_P(P)\}$$

If a paragraph is removed from the middle of the article (i.e. i is neither the start nor the end of the article),

$$BE_p(A\prime) = \{x \mid x \in BE_p(A) \wedge x < i\} \cup \{x - length(P) \mid x \in BE_p(A) \wedge x \notin BE_p(P) \wedge x > i\}$$

3.2.4 MR8: Removing some words from a list of random words. Given a list of random words L_1, we remove half of these words from the end of file, and refer to the removed part and the remaining part as L_2 and $L\prime$, respectively. We should have the following relations.

$$BE_t(L\prime) = BE_t(L_1) - BE_t(L_2)$$

$$BE_p(L\prime) = \{x \mid x \in BE_p(L_1) \wedge x \notin BE_P(L_2)\}$$

The subtract (-) operator only removes the corresponding bio-entity from the source set.

3.3 Shuffling Relations

3.3.1 MR9: Shuffling paragraphs of an article. Given an article A, we shuffle all the paragraphs of A and refer to the new resultant text as $A\prime$. As a result, following equation must be true.

$$BE_t(A\prime) = BE_t(A)$$

With respect to this MR, the positions of bio-entities (BE_p's) can vary and there is no predefined relation between them.

3.3.2 MR10: Shuffling a list of random words. Given a list of random words L, we shuffle all words and create a new list of words $L\prime$. Therefore,

$$BE_t(L\prime) = BE_t(L)$$

MR10 does not satisfy a predefined relation on BE_ps. Note that we feed these words to LingPipe as separate words.

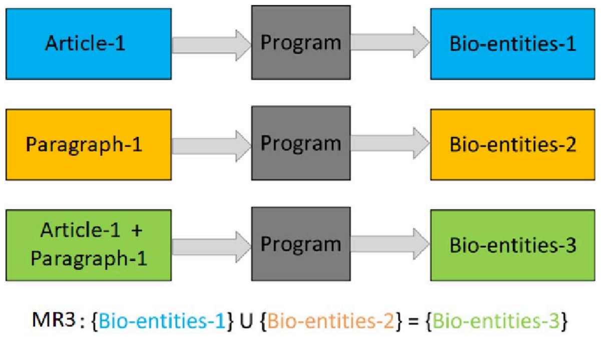

Figure 1: A toy example of MR3 Relation

4 METHODOLOGY

In this section, we describe our methodology for testing LingPipe 4.1.2 using MT. Figure 2 depicts the sequence of steps required to perform MT. Below is the sequence of steps involved.

(1) Identify a set of MRs for testing the bio-entity recognition task in LingPipe. The MRs that we developed are described in section 3.
(2) Develop source test cases and follow-up test cases for each of the MRs. Below, we describe how we created them. Note that all the articles, paragraphs, sentences, and words are extracted from a collection of biomedical articles available through the PubMed [1] website.
    I. MR1 (Adding a sentence to another sentence)
        - Source test case: Two sentences extracted from a biomedical article (PMCID: 100320).
        - Follow-up test case: The sentence formed by appending one of them to the other sentence.
    II. MR2 (Adding a sentence to a paragraph)
        - Source test case: A paragraph and a sentence obtained from biomedical text (PMCID: 100320).
        - Follow-up test case: The sentence is added to the end of the paragraph forming an extended paragraph.
    III. MR3 (Adding a paragraph to an article)
        - Source test case: A full-text biomedical article (PMCID: 100320) and a paragraph from another article (PMID: 28881451).
        - Follow-up test case: The paragraph is appended to the end of the article.
    IV. MR4 (Adding a list of random words to another list)
        - Source test case: Two lists of five hundred random words extracted from 15 randomly selected biomedical articles.
        - Follow-up test case: Append the first list to the second list to form a new list of thousand random words.
    V. MR5 (Removing a list of random words from a sentence)
        - Source test case: A sentence extracted from a biomedical article (PMCID: 100320) and a consecutive list of words from this sentence.

[1] https://www.ncbi.nlm.nih.gov/pubmed

  - Follow-up test case: Remove the consecutive list of words from the sentence to create a truncated sentence.

VI. MR6 (Removing a sentence from a paragraph)
  - Source test case: A paragraph obtained from a biomedical article (PMCID: 100320) and a randomly selected sentence from that paragraph.
  - Follow-up test case: A new truncated paragraph created by removing the randomly selected sentence.

VII. MR7 (Removing a paragraph from an article)
  - Source test case: A biomedical article and a randomly selected paragraph from that article (PMCID: 100320).
  - Follow-up test case: Remove the randomly selected paragraph from the article.

VIII. MR8 (Removing some words from a list of random words)
  - Source test case: A list of one thousand random words extracted from 15 randomly selected biomedical articles and the second half of this list that contains five hundred words. Five hundred words were removed from the end of the list in order to have a relation between the extracted bio-entity positions.
  - Follow-up test case: First five hundred words of this list.

IX. MR9 (Shuffling paragraphs of an article)
  - Source test case: A biomedical article (PMCID: 100320).
  - Follow-up test case: A new article created by shuffling the paragraphs.

X. MR10 (Shuffling a list of random words)
  - Source test case: A list of a thousand random words extracted from 15 randomly selected biomedical articles.
  - Follow-up test case: a new list formed by shuffling all the above words.

(3) Identify set of classes related to the bio-entity recognition task. LingPipe 4.1.2 is written using Java and is a large software system consisting of 25 Java packages. Each of the packages contains a number of Java classes varying from three to 32 classes. We selected the classes related to bio-entity recognition task based on the class hierarchy structure. We also verified that these selected classes contain functionality related to bio-entity recognition task by examining execution traces.

(4) The source and follow-up test cases for each MR are executed on LingPipe. The MT process is put on hold if any of the MRs are violated.

(5) The errors in the program which caused the fault are identified and fixed. The execution of test cases is conducted on the rectified program. This process continues until all the MRs are satisfied (note: we did not find any violations of MRs on the LingPipe).

(6) Mutants are generated for each of the classes identified in Step 3. Mutants are the faulty versions of the program under test and are created by introducing a single syntactic change in the source code. More information about the mutation generation process can be found in Section 5.4.

(7) The source and follow-up test cases for each MR is executed on the set of mutants.

(8) The results of the source and follow-up test cases are checked to identify if the corresponding MR is violated. Violation of a MR is recognized if the corresponding mutant is *killed*.

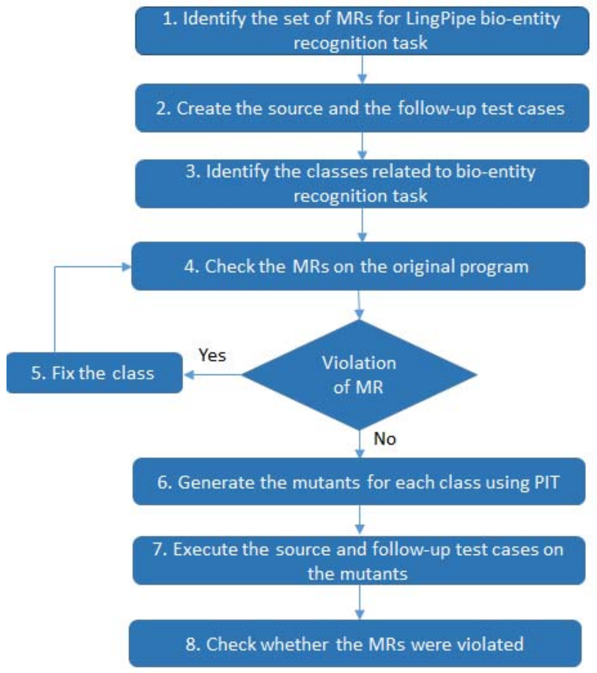
1. Identify the set of MRs for LingPipe bio-entity recognition task
2. Create the source and the follow-up test cases
3. Identify the classes related to bio-entity recognition task
4. Check the MRs on the original program
Violation of MR
Yes
5. Fix the class
No
6. Generate the mutants for each class using PIT
7. Execute the source and follow-up test cases on the mutants
8. Check whether the MRs were violated

Figure 2: MT process for testing LingPipe

5 EXPERIMENTAL SETUP

In this section we provide the details of the experimental setup, especially the research questions to be answered, the system under test (SUT), description of the Java classes that were tested and information regarding mutants generated for each class.

5.1 Research Question

RQ1: How effectively does MT identify faults in the system under test?

RQ2: Which MRs perform better in identifying faults in the system under test?

5.2 LingPipe-4.1.2

LingPipe is a tool for processing text using computational linguistics [1]. LingPipe can help in performing tasks such as named entity recognition, sentiment analysis on news feeds and suggesting spelling corrections. We selected LingPipe as our case study since LingPipe is one of the most popular biomedical bio-entity recognition programs in the bioinformatics community. The tool also performs quick extraction of bio-entities from large paragraphs and articles which provides an ideal platform to consider LingPipe in our case study.

Java Class	# LOC
ChunkTagHandler	268
IndoEuropeanTokenizer	310

Table 1: LOC (lines of code) for LingPipe classes

5.3 Java classes under test

LingPipe provides a total of about 350 Java classes and each Java class performs functionality related to text mining. We determined that 15 classes are related to the bio-entity recognition task in particular. These classes were identified based on the dynamic sequence diagrams generated using the Jive tool [2]. Two classes out of these 15 classes were selected for this study by examining the class hierarchy and execution trace. We conducted the mutation analysis using these two classes. Functionality of each of the two selected classes are explained below:

(1) **ChunkTagHandler**
The ChunkTagHandler class sets the chunk handler to the specified bio-entity. This class deals with the arrays of tokens, tags, and white spaces. ChunkTagHandler converts the three arrays to a chunking and then passes the chunking to the chunk handler. The main functionality includes identifying bio-entities and dividing into chunks. The chunks are associated with a tag handler for extraction and representation [1].

(2) **IndoEuropeanTokenizer**
IndoEuropeanTokenizer returns a tokenized version of the required string. An array of tokens is constructed using the words, characters, and white spaces from the paragraphs. The generated tokens are grouped into chunks by the above mentioned ChunkTagHandler class [1].

5.4 Mutant Generation

Mutation testing is used in our experiments to determine the quality and effectiveness of the MRs to identify faults in the program. Mutants are generated for the two classes using the PIT tool [3]. The faulty versions of the program are created using *mutation operators*. Mutation operators apply changes to a statement in the program which creates fault in the program. Below, we describe the mutation operators used by PIT to generate the mutants:

(1) **Conditional Operator:** The conditional boundary mutator replaces the relational operators <, ≤, >, ≥ with the alternate operator.
(2) **Increment Operator:** The increment operator ++, − will replace increments with decrements and vice versa.
(3) **Math Operator:** The math operators such as = -,*,%, &,≪,≫ are replaced by alternate operator in the list.
(4) **Negate Conditional Operator:** The conditional operators such as ==, !=, ≤, ≥, <, > will be replaced by alternate operator from the list.
(5) **Return value mutator Operator:** The return types such as boolean, long, float double will be replaced by *null*, *zero*, *1*, *true* or *false*.

[2]https://www.cse.buffalo.edu/jive/latest.html
[3]http://pitest.org/

Table 2 shows the total number of mutants generated, *equivalent* mutants detected, mutants that generated exceptions for each class and the final set of mutants used for MT. The second column provides details of the total mutants generated for the two classes. The third column indicates the total mutants generating exceptions during the execution. Mutants causing exceptions such as the program crashing abruptly or leading to an infinite loop in execution of the program are removed from further consideration, since these are trivial faults that could be detected without MT. Further, the mutants that produce the same outputs as the original program were manually inspected to filter out any mutants that were outside the scope of the bio-entity recognition task. The last column in Table 2 presents the total number of mutants used in the experiment after filtering the mutants using the above criteria.

6 RESULTS

In thi section, we discuss our findings for each of the research questions described in section 5.

6.1 RQ1: Effectiveness of MT

The MT process, described in Section 4, killed 65% (24 out of 37) of the total mutants for the two classes used for the evaluation. Based on the killing percentage, we can conclude that MT effectively detects faults in LingPipe.

Table 3 shows the individual mutant killing rates for the ChunkTagHandler class and the IndoEuropeanTokenizer class. Neither of them has any unit test cases provided by the developers of Lingpipe. Thus, we could not compare the fault detection effectiveness of MT with the existing unit tests for those two classes.

6.2 RQ2: Effectiveness of individual metamorphic relations for fault detection

Figure 3 and Table 4 show the mutant killing rates of the individual MRs for the the two classes used in the experiment. For the ChunkTagHandler class, MR9 (shuffling the paragraphs of an article) and MR7 (removing a paragraph from an article) has killed the most mutants (67% each). The least performing MR is MR10 (shuffling the words) within a list of random words and it only killed 17% of the mutants.

In the IndoEuropean class, MR9 (shuffling the paragraphs) of an article performed best by killing 55% mutants, followed by the second best performing MR7 (removing a paragraph) from an article killing 48% mutants. Interestingly, the least performing MR10 (shuffling a set of random words) did not kill any mutant.

When comparing the overall mutant killing rates of the MRs for both the classes, MR9 (shuffling the paragraphs) of an article performed best by providing 56% overall killing rate. The second best performing MR7 (removing a paragraph) from an article provides an overall mutant killing rate of 51%. The least performing MR10 (shuffling the words) within a list of random words provided an overall 2% mutant killing rate.

Java Class	Total # of Mutants generated	# of Mutants giving exceptions	# of Mutants with equal outputs	# Mutants tested with MT
ChunkTagHandler	28	6	16	6
IndoEuropeanTokenizer	64	18	15	31

Table 2: # Mutants for each Java class

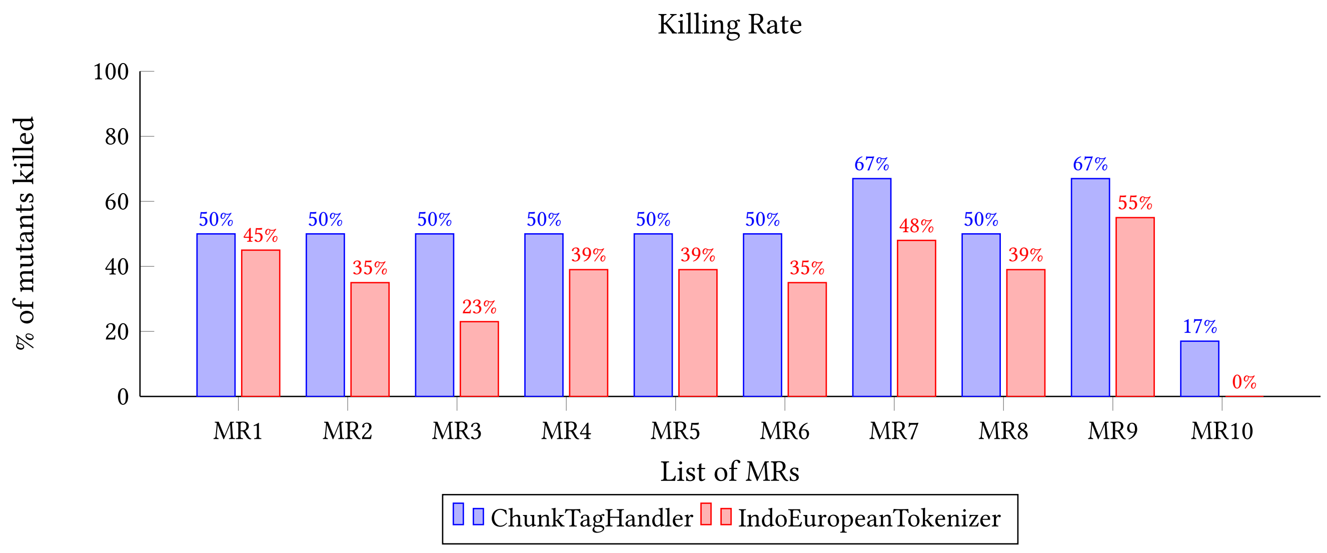

Figure 3: Killing rate of MRs for Java classes

Java Class	Mutants killed by unit testing	Mutants killed by MT approach
ChunkTagHandler	not available	5 (83%)
IndoEuropeanTokenizer	not available	19 (62%)

Table 3: # Mutants killed by unit testing vs Metamorphic Testing approach

MR#	ChunkTagHandler (out of 6)	IndoEuropeanTokenizer (out of 31)
MR1	3	14
MR2	3	11
MR3	3	7
MR4	3	12
MR5	3	12
MR6	3	11
MR7	4	15
MR8	3	12
MR9	4	17
MR10	1	0

Table 4: # Mutants killed for each class by the individual MRs.

7 DISCUSSION

In this study, we developed 10 novel MRs for bio-entity recognition and evaluated their effectiveness for conducting MT using a widely used bio-entity recognition tool, LingPipe.

Feasibility of MT approach for bio-entity recognition: Our experiment results show that MT killed 83% of the mutants for the ChunkTagHandler class and 61% of the mutants for the IndoEuropeanTokenizer class. Thus, we could detect most of the mutants using the developed MRs showing that MT is an effective method for testing bio-entity recognition tools.

Both the classes we used in the experiments provides important functionality for bio-entity recognition task in LingPipe. The chunk and tokenizer package contains unit test cases, but no unit test cases are available for both ChunkTagHandler and IndoEuropeanTokenizer classes. This could be due to the oracle problem associated with the bio-entity recognition task. Therefore, MT provides an effective approach for quality assurance of these important classes especially in the absence of unit tests.

Performance of individual MRs in fault detection: In the IndoEuropeanTokenizer class, the best performing MRs, MR7 (removing a paragraph from an article) and MR9 (shuffling the paragraphs of an article) covered more statements compared to the other MRs. Also these two MRs showed different execution paths in the IndoEuropeanTokenizer class for the source and the follow-up test cases. In the ChunkTagHandler class, these two MRs showed different execution paths for the source and follow-up test cases but the number of statements covered by these MRs were not different from the other MRs.

The experiment results also show that MR9 and MR7 killed the same set of mutants except one mutant that was only killed by MR7. This indicates the possibility that the MR9 is redundant.

Overall, MR10 (shuffling a set of random words) had the lowest mutant killing rate indicating that MR10 might be a less effective MR compared to the rest of the MRs. But, interestingly, MR10 killed a mutant of the ChunkTagHandler class which could not be killed by any of the other MRs. Our analysis showed that only the source and follow-up test cases created for MR10 could execute the mutated statement, which eventually led to killing that mutant. The rest of MRs could not kill that mutant due to exceptions being generated even before the mutated statement could be reached. Future work would verify if the particular mutant could be killed by the other MRs by additional test cases.

For the mutants that were not killed by any of the MRs, we observed that even though the source and follow-up cases provided different execution paths in both Chunktaghandler and Indoeuropean classes, the change in the execution path did not cause a change in the output. Hence they could not be killed using MT.

8 RELATED WORK

Over the past years, there has been an enormous growth in the quantity and variety of biological data. With this growth, researchers have developed a lot of programs in bioinformatics. But, most of time and effort is spent on developing complex statistical methods, while there is a lack of effort in systematically testing their implementations that is essential for the quality assurance [6].

Chen et al. [6] employed MT on two open-source bioinformatics programs. The first program GNLab [4] is a tool for large-scale analysis and simulation of gene regulatory networks. The second program SeqMap [5] deals with mapping a short sequence that reads with a reference genome.

They introduced new MRs for these programs and showed that MT is beneficial in detecting faults as well as its relative ease of use. Pullum and Ozmen [14] employed MT for testing epidemiological models. They used the model parameters based on ordinary differential equation and agent based model and the expected results which are obtained from executing the model with MR-transformed parameter values. Ramanathan et al. [15] also utilized MT to build a work flow of compartments of susceptible, infectious, and recovered epidemiological models. They showed that MT can be useful where the mathematical models may fail. Lundgren and Kanewala [13] examined the effectiveness of a pseudo-oracle and MT to detect subtle faults in the genome alignment tool BBMap[6]. The results showed that MT performed better than pesudo-oracles for detecting subtle faults. To the best of our knowledge, our study is the first time that MT is applied for the quality assurance of bio-entity recognition[7].

[4]http://en.bio-soft.net/other/gnlab.html
[5]http://www-personal.umich.edu/ jianghui/seqmap/
[6]https://sourceforge.net/projects/bbmap/
[7]https://jgi.doe.gov/data-and-tools/bbtools/bb-tools-user-guide/bbmap-guide/

9 CONCLUSIONS AND FUTURE WORK

MT is a technique to test programs which do not have a test oracles. Bio-entity recognition tools face the oracle problem since it produces a very large number of bio-entities for a given text corpus making it hard to validate. In this study, we examined the effectiveness of MT for the quality assurance of bio-entity recognition. First, we proposed 10 novel MRs for validating bio-entity recognition tools in general. Then we applied these MRs for testing a popular bio-entity recognition tool, LingPipe. Our results show that MRs that we developed are effective in identifying faults in the tool.

In our future work, we plan to increase the mutant killing percentages by creating source test cases based on an effective test case generation mechanism. We also plan to combine MRs so that we can increase their effectiveness while reducing the cost associated with executing multiple MRs. We will also extend our evaluation to include other classes associated with bio-entity recognition in LingPipe. We also plan to develop new MRs and investigate further into the effectiveness of different categories of mutants.

REFERENCES

[1] Alias-i. 2017 (accessed September 30, 2017). *LingPipe 4.1.2.* http://alias-i.com/lingpipe.

[2] Stephen Altschul, Barry Demchak, Richard Durbin, Robert Gentleman, Martin Krzywinski, Heng Li, Anton Nekrutenko, James Robinson, Wayne Rasband, James Taylor, et al. 2010. The Anatomy of Successful Computational Biology Software. *Nature Biotechnology*, 894–897.

[3] Ardeshir Bayat. 2002. Bioinformatics. In *BMJâĂř: British Medical Journal*. BMJ, 1018–1022.

[4] Bob Carpenter. 2004. Phrasal queries with LingPipe and Lucene: ad hoc genomics text retrieval.. In *TREC*, Vol. 1. 1–10.

[5] Bob Carpenter. 2007. LingPipe for 99.99% recall of gene mentions. In *Proceedings of the Second BioCreative Challenge Evaluation Workshop*, Vol. 23. 307–309.

[6] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. 2009. An Innovative Approach for Testing Bioinformatics Programs using Metamorphic Testing. *BMC bioinformatics* 10, 1 (2009), 24.

[7] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2017. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Computing Surveys (in press)* (2017).

[8] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37 (2011), 649–678.

[9] Amir Hossein Kamali, Eleni Giannoulatou, Tsong Yueh Chen, Michael A Charleston, Alistair L McEwan, and Joshua WK Ho. 2015. How to test bioinformatics software? *Biophysical reviews* 7, 3 (2015), 343–352.

[10] David W Kane, Moses M Hohman, Ethan G Cerami, Michael W McCormick, Karl F Kuhlmman, and Jeff A Byrd. 2006. Agile Methods in Biomedical Software Development: a Multi-site Experience Report. *BMC Bioinformatics*, 7–273.

[11] Upulee Kanewala and James M Bieman. 2014. Testing scientific software: A systematic literature review. *Information and software technology* 56, 10 (2014), 1219–1232.

[12] Diane Kelly and Rebecca Sanders. 2008. The Challenge of Testing Scientific Software. In *In Proc. of the Conference for the Association for Software Testing (CAST)*. IEEE, 30–36.

[13] Anders Lundgren and Upulee Kanewala. 2016. Experiences of testing bioinformatics programs for detecting subtle faults. In *Software Engineering for Science (SE4Science), IEEE/ACM International Workshop on*. IEEE, 16–22.

[14] Laura L Pullum and Ozgur Ozmen. 2012. Early Results from Metamorphic Testing of Epidemiological Models. In *BioMedical Computing (BioMedCom), 2012 ASE/IEEE International Conference On*. IEEE, 62–67.

[15] Arvind Ramanathan, Chad A Steed, and Laura L Pullum. 2012. Verification of Compartmental Epidemiological Models using Metamorphic Testing, Model Checking and Visual Analytics. In *BioMedical Computing (BioMedCom), 2012 ASE/IEEE International Conference on*. IEEE, 68–73.

[16] V. Gupta S.K. Gill, A. F. Christopher and P. Bansal. 2016. Emerging Role of Bioinformatics Tools and Software in Evolution of Clinical Research. *Perspectives in Clinical Research*, 115–122.

[17] Elaine J. Weyuker. 1982. On Testing Non-Testable Programs. *Comput. J.* 25, 4 (1982), 465–470.

[18] Craig S Wright and Tanveer A Zia. 2011. Quantitative Analysis into the Economics of Correcting Software Bug. In *Int. Conf. Comput.Intell. Security Inf. Syst.* IEEE, 198–205.