Clone Detection and Evaluation Within the TLA+ Formal Specifications

Department of Computer Science

School of Engineering

Final Year Project 2024

Course Code: COMP30030

Student's Name : Shun Le Yi Mon

Student's ID : 10830930

Supervisor Name: Dr. Marie Farrell

Word Count: 10020

Abstract

In this report, we present an empirical study on specification clones within the formal specification language Temporal Logic of Actions (TLA+). TLA+ is a formal specification language used in concurrent systems and distributed systems for designing, modelling, verification and documentation of programs. Our research introduces new definitions of clone types and algorithms to detect code clones within TLA+ specifications. By studying a substantial corpus of TLA+ specifications, we aim to find insights into the frequency and attributes of specification clones, potentially advancing research in specification clones, software engineering, TLA+, code clones, and code modularization. Our findings indicate that less than 45% of clones across files exhibit a similarity exceeding 81%. Notably, high-frequency code clones predominantly encompass fundamental TLA+ syntax elements, offering valuable insights into their intended purposes and usage patterns. Additionally, the detection of clones within files revealed no significant correlation between clone pairs and line count.

Acknowledgement

I would like to express my sincere gratitude to Dr. Marie Farrell for her invaluable guidance, support, and mentorship throughout the whole process of this research journey. Her expertise, encouragement, and insightful feedback have been significant in shaping the direction and methodology of this study. I am also deeply thankful to my friends for their support and constructive feedback during the drafting process. Additionally, I wish to express my gratitude to the University of Manchester for providing the opportunity to turn this research into a reality. Lastly, I am grateful to all those who, directly or indirectly, contributed to this project. Your assistance, encouragement, and inspiration have been invaluable.

Contents

1 Introduction and Motivation **7**

2 Background and Related Work **9**

2.1 TLA+ 9

2.1.1 An Overview of TLA+: Theory and Applications 9

2.1.2 TLA+: Specification Example 10

2.1.3 TLA+: A Review of Recent Research 13

2.2 Clones in Code and Specifications 13

2.3 Related Work 15

3 Detecting Specification Clones (Approach) **17**

3.1 Code Clone Definition 17

3.2 Architecture Diagram & Functional Requirements 21

3.3 Specification Corpus & Program Algorithms 23

3.3.1 A Corpus of TLA+ Specifications 24

3.3.2 Program Implementation 24

4 Analysis and Evaluation **36**

4.1 Analysis 37

4.2 Evaluation: TLA+ Specification Clones 45

4.3 Evaluation: Project Overview 47

5 Threats to Validity **50**

6 Summary and Future Work **52**

6.1 Summary 52
6.2 Future Work 53

7 Reflection **55**

A Appendix **60**

A.1 Specification sources 60
A.2 Graphs & Code 61

List of Figures

1 System Architecture Design *[rounded rectangle = component, cylinder = database, arrow = process flow]* 21

2 Distribution of clones (the small dataset) 38

3 Distribution of clones (the large dataset) 38

4 Distribution of clones (the whole dataset) 39

5 A part of the table displaying statistics about code clones 41

6 A part of the table displaying statistics about code clones 42

7 First few rows of table displaying statistics of each file 43

8 graph of line count and clone pair 43

9 Minimum and Maximum Percentage Range of Each Clone Type . . 44

10 Validation for parser component requirements 48

11 Distribution of clones with tokenized Type-5 clone (whole dataset) . 61

12 Distribution of clones with tokenized Type-5 clone (small dataset) . 62

13 Distribution of clones with tokenized Type-5 clone (large dataset) . 62

14 Clone types in each file 67

15 Clone types in each file without Type-5 clones 67

List of Tables

1 Functional requirements ('Tokenizer' component) 22

2 V&V for Tokenizer Functional Requirements 23

3 Validation table for T4, T5 and T6. Note that the notation has been simplified for presentation purposes. 24

4 Label table for the following requirement tables 63

5 Complete list of functional requirements 64
6 Requirement ID and description 65
7 Requirement and Verification 66
8 Validation table of functional requirements part 1 68
9 Validation table of functional requirements part 2 69
10 Validation table of functional requirements part 3 70
11 Validation table of functional requirements part 4 71

Listings

1 TLA+ code example from [1] 11
2 SequenceMatcher code examples [2] 18
3 SequenceMatcher class constructor [2] 18
4 A Few Examples From The Renaming Log File 25
5 ‘Preprocessor’ pseudocode 26
6 ‘Duplication Removal’ Pseudocode 27
7 ‘Tokenizer’ Pseudocode 29
8 ‘Code Analysis’ Pseudocode 30
9 ‘Clone Detector’ Pseudocode part 1 32
10 ‘Clone Detector’ Pseudocode 33
11 Data Visualization: Clone Type Pie Chart Pseudocode 34
12 Code in file 1 40
13 Code in file 2 40
14 Tokenized code in file 1 41
15 Tokenized code in file 2 41
16 ‘Preprocessor’ Code 72
17 ‘Tokenizer’ Code 73
18 Code Analysis: ‘count_line’ Code 74
19 ‘Code Detector’ Code part 1 75
20 ‘Code Detector’ Code part 2 76

21 ‘Code Detector’ Code part 3 77
22 ‘Code Detector’ Code part 4 78
23 ‘Code Detector’ Code part 5 79
24 ‘Clone Type Pie’ Code 80

1 Introduction and Motivation

Temporal Logic of Actions (TLA+), invented by Leslie Lamport in the late 1980s, is a formal specification language designed to model and analyze concurrent and distributed systems [3]. It is applied onto system design, verification, and validation in both academia and industry [4]. In [5], the authors explore TLA+'s syntax, industrial significance, and ensuring correctness and reliability in applications.

In this research, we analyze TLA+ specifications as software artifacts and expand software engineering methodologies to identify specification clones written in the TLA+ language by analyzing a corpus of specifications.

Drawing from insights obtained through empirically examining Event-B specifications [6], our study represents the first exploration within the domain of TLA+. By examining a diverse range of TLA+ specifications, we introduce a new quantifiable classification of clone types and develop an algorithm for identifying specification clones within TLA+ specifications. Furthermore, we analyze a substantial collection of TLA+ specifications to detect trends related to the frequency and attributes of specification clones. This research has the potential to motivate investigations into specification clones and modularization written in the TLA+ formal specification.

In summary, we contributed:

1. a corpus of TLA+ specifications that were collected from various sources
2. new definitions of specification clones for TLA+
3. a suite of Python programs to parse, tokenize TLA+ specifications, and to detect and analyse TLA+ specification clones
4. analysis and evaluation of TLA+ specification clones

Research in software engineering is increasing focused on identifying, analyzing, organizing, and assessing tools related with code clones. TLA+ holds significance in system design, validation, and verification within industry such as major companies like Amazon, Intel, and Microsoft. Given the increasing number of usage of code clones in programming, coupled with the previous factors, I see the need to uncover any specification clones in TLA+. This research aims to raise awareness in this domain and potentially encourage further research with the help of insights gained from this research.

This report is structured as follows. In Section 2, we describe TLA+ theory and applications supported with a specification example and recent research done related to the domain of TLA+. In Section 3, we provide details on code clone definition, both from research and our own definitions illustrated with examples. Next detailed explanations on architecture diagram and functional requirements of the software of this project. Then, we delve into how the specification corpus was collected and it is being processed in each of the components of the architecutre diagram. This is illustrated with algorithm of the componenets along with comprehensive explanation of the algorithm. In Section 4, we summarize our exploratory analysis of the assembled TLA+ project corpus, insights into the data results obtained from our search and evaluation, with illustrations. Next, we conduct evaluation from these analysis. Then, we conduct a critical analysis and evaluation of the project's overall execution. In Section 5, we identify potential threats to the validity of our work, while Section 6 summarizes the report with our contributions and potential for future research. Finally, in Section 7, a detailed reflection and evaluation on the project's advancement, successful accomplishments, challenges, and proposed strategies for improvement in future work.

2 Background and Related Work

In this chapter, we summarise background information on TLA+, code clones, and specifications along with the discussion of found related work.

2.1 TLA+

In this section, we will delve into a comprehensive explanation of TLA+ with three subsections. The first subsection will detail an overview of TLA+ theory and applications. The second subsection will describe an illustrative example of a TLA+ specification. The third subsection will focus on research findings relevant to TLA+.

2.1.1 An Overview of TLA+: Theory and Applications

TLA+ is a formal specification language designed for creating detailed models of software systems and hardware components at a level above traditional code implementation. It enables programmers to conceptualize and analyze system behavior and mechanisms without getting bogged down in specific coding details. By focusing on high-level models, TLA+ promotes rigorous thinking, simplifies system design, and facilitates the discovery of optimal algorithms. This approach leads to more efficient and reliable software and hardware systems by catching design errors early and reducing complexity in the implementation process.

By applying TLA+ and its tools, developers can identify fundamental design flaws, which are hard to track down and costly to address within actual code. Engineers have access to the TLC model checker, an Integrated Development Environment (IDE) for drafting models and running various tools for verification, as well as a proof checker [7]. To understand and learn about its syntax, the book "Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers" by Leslie Lamport, Learning TLA+ webpage [8], and Learn TLA+ website [9] are great resources to investigate. Learning to use TLA+ can be quite a learning curve, therefore engineers often find starting with PlusCal to be the easiest way to start learning TLA+ [10].

PlusCal is a language for writing algorithms, where it aims to replace pseudocode with precise, testable code and offers constructs for describing concurrency and nondeterminacy. The language is highly expressive, allowing any mathematical formula to be used as an expression. PlusCal algorithms are translated into TLA+ models for checking with TLA+ tools. Possible limitations of using PlusCal is its incapability to express for complex models [10].

The initial application of TLA+ in industry was to model hardware, where engineers aimed to eliminate bugs with the need to understand and critical thinking before building and implementing it in silicon [11]. Over time, its application has expanded into enhancing the design and verification processes of software and hardware systems. Another key characteristic is its syntactic simplicity based on set theory and logic, enabling users to express complex system behaviours clearly.

Companies like Amazon, Microsoft, Intel and OpenComRTOS use TLA+ to develop complex and reliable systems. Amazon uses TLA+ to model DynamoDB and S3 to identify and avoid complex bugs that are often not detected using traditional testing methods [12]. OpenCoomRTOS, a real-time operating system, uses TLA+ to create more efficient and manageable systems. Other uses include eSpark Learning's infrastructure team utilizing TLA+ to refactor a large system and catch critical bugs [1] and more [12]. With the help of TLA+, companies can work efficiently, and prevent significant revenue loss caused by uncaught bugs.

2.1.2 TLA+: Specification Example

Listing 1: TLA+ code example from [1]

```
EXTENDS Integers, Sequences, TLC
CONSTANTS Business, Credits

set ++ x == set \union {x}
set -- x == set \ {x}

VARIABLES owner, offers
vars == <<owner, offers>>

Init ==
  /\ owner \in [Credits -> Business]
  /\ offers = {}

Propose(from, to, credit) ==
  /\ owner[credit] = from
  /\ offers' = offers ++ <<from, to, credit>>
  /\ UNCHANGED owner

Accept(from, to, credit) ==
  /\ <<from, to, credit>> \in offers
  /\ offers' = offers -- <<from, to, credit>>
  /\ owner' = [owner EXCEPT ![credit] = to]

Reject(from, to, credit) ==
  /\ <<from, to, credit>> \in offers
  /\ offers' = offers -- <<from, to, credit>>
  /\ UNCHANGED owner

Next ==
  \E from, to \in Business, credit \in Credits:
    /\ from /= to
    /\ \/ Propose(from, to, credit)
       \/ Accept(from, to, credit)
       \/ Reject(from, to, credit)

Spec == Init /\ [][Next]_vars

\* If ownership changes from A to B
\* It's because B accepted an offer from A
ValidChange(credit) ==
  LET co == owner[credit]
  IN co /= co' =>
    Accept(co, co', credit)

\* All changes in the system are valid changes
ChangeInvariant ==
  [][\A c \in Credits: ValidChange(c)]_owner
```

The TLA+ code shown in Listing 1 aims to model a system involving ownership and offers. It initializes the system, defines actions for proposing, accepting, and rejecting offers, specifies the system's transition behavior, and ensures that changes in ownership are valid. Overall, it provides a formal description of a system where entities can make offers to transfer ownership, and it verifies properties related to these offers and ownership changes.

Line 1 indicates that the TLA+ module extends its capabilities by including predefined modules for handling integers, sequences, and TLC (the TLA+ Toolbox model checker).

Line 2 declares two constant values, 'Business' and 'Credits'.

Line 4 and 5 shows set operators defined to add or remove an element 'x' from the set 'set'.

Line 7 declares two variables, 'owner' and 'offers'.

Line 8 defines a tuple 'vars' containing the variables 'owner' and 'offers'.

Line 10 is the initialization condition for the system. It specifies that the 'owner' variable should map each 'credit' to a 'Business' value, and the 'offers' set should be empty initially shown in Line 11 and 12.

Line 14, 19 and 24 are actions representing proposing, accepting, and rejecting offers respectively. They manipulate the 'owner' and 'offers' variables according to certain conditions.

Line 29 describes the system's transition behavior. It specifies that the next state of the system can be any of the actions Propose, Accept, or Reject, provided certain conditions are met.

Line 36 defines the overall specification of the system, which combines the initialization condition 'Init' with the transition behavior specified by 'Next'.

Line 40 and 46 define additional properties of the system. 'ValidChange' specifies conditions under which changes in ownership are valid, while 'ChangeInvariant' ensures that all changes in the system are valid according to the 'ValidChange' definition.

2.1.3 TLA+: A Review of Recent Research

Work on extracting symbolic transitions from TLA+ specifications [13] introduces an unique approach for system specification, where logical formulas are used to constrain system behavior, contrasting with imperative languages. This logic-based methodology lacks assignments and imperative statements commonly applied by model checkers. Model checkers compute successor states either explicitly or symbolically. To enhance efficiency, TLA's model checker, TLC, introduces side effects like interpreting equality as assignment. Inspired by TLC, the paper proposes an automatic technique for identifying expressions within TLA+ formulas suitable for assignments. Unlike TLC, this method doesn't evaluate expressions directly but reduces the assignment problem to the satisfiability of an SMT formula. This enables slicing TLA+ formulas into symbolic transitions, enabling their use as input for symbolic model checkers. The paper's prototype successfully deduce symbolic transitions from various TLA+ benchmarks.

The authors in [14] explores the application of formal methods, specifically TLA+ and PlusCal, in developing fault-tolerant and safety-critical modules for the TAS Control Platform used in railway control applications. It discusses about the challenges of creating fault-tolerant distributed algorithms in safety-critical industries and highlights the benefits of using formal methods in improving algorithm correctness and development efficiency. The paper discusses how formal methods help bridge the gap between model and implementation by translating formal models to C code. Additionally, it describes a design process called property-driven design that enhances trust in the formal model and tools and implicitly addresses software quality metrics such as code coverage.

2.2 Clones in Code and Specifications

Code clones are similar or identical fragments of code in a software system. Research in software engineering is increasingly focused on the detection, analysis, management, and evaluation of tools related to code clones [15]. The practice of code cloning often proves advantageous in software development by promoting the reuse of reliable code, saving time and effort. However, it potentially could mean there's limitations in the modularization mechanisms of programming paradigms, indicating the need for improvements.

If you look back to the TLA+ code example 1, you could see that lines 19 to 21 and lines 24 to 26 are exactly the same exact for the variable 'Accept' and 'Reject'. This potentially is a code clone. If we try to identify all of code clones in the code example, we would end up with different kinds of similarity of context which may be hard to differentiate and describe each of them. That is why Roy et al. has came up with a way to identify these different code clones.

Roy et al. have identified four distinct types of code clones, each categorized based on the nature of the match between different code fragments [15]:

Type-1: Code fragments that are identical but may differ only in variations of white space and comments.

Type-2: Structurally or syntactically identical code fragments that differ only in identifiers, literals, types, layout, and comments.

Type-3: A less strict version of Type-2 clones, allowing differences such as additions, deletions, or modifications of statements.

Type-4: Code fragments show the same functional behaviour but implemented through vastly different syntactic structures.

Specifications are vital in software development, especially for complex applications like those in large business systems, healthcare, and transportation control. They are used to identify and document functional and non-functional requirements, bridging the gap between documented and actual requirements to prevent software errors. Specifications also help manage complexity by detailing system behavior, integrity constraints, and interaction descriptions, ensuring system reliability and effectiveness. They provide a foundation for formal methods, allowing clear reasoning in software behavior and complementing traditional development methodologies [16]. Compared to software specifications, formal specifications describe a software system's behavior and properties mathematically. They define system requirements, constraints and behavior in formal languages. They are highly precise and unambiguous, leaving no room for interpretation. They are primarily used in cases where system correctness and verification are critical, such as safety-critical systems, security protocols, or highly complex systems where errors potentially brings severe consequences. They serve as a basis for formal verification methods to ensure the system behaves as desired [17].

2.3 Related Work

There are several similar research studies related to clone detection and specification clones. A specification clone analysis and extraction tool called Puzzle is developed to detect specification clones for DSLs (domain-specific languages) [18]. This tool uses static analysis techniques for identifying specification clones within DSLs constructed using the executable metamodeling paradigm. It allows the extraction of specification clones as reusable language modules, which can subsequently be used in the construction of new DSLs. Other papers discuss the issue of copy and paste resulting in cloning and the effectiveness of clone detection on requirement specifications [19], and a collection of refinement laws and practical guidelines for program development [20].

Cloning in source code is a recognized quality issue that has negative effects on software maintenance. These negative effects include increased maintenance costs and a higher possibility of introducing defects into the software. Code cloning can lead to inconsistencies in changes made to duplicated code segments, resulting in unexpected behavior within the program. This inconsistency poses a risk to program correctness and can produce faults, potentially threatening the dependability and effectiveness of the software [21]. Finding these clones is one of our aim in this project, hoping to extract valuable insights during and at the end of the process.

Research indicates a significant presence of cloning, although there is variability among specifications, indicating that certain authors can mitigate cloning [19]. The use of clone detection aids in assessing the quality of requirements specifications by identifying a widely acknowledged quality issue: redundancy. Redundancy is viewed as a barrier to requirement modifiability and is highlighted as a significant challenge in automotive requirement engineering, among other domains [22]. The formal specifications we have gathered come from various domain, rather than a single domain like automotive requirement engineering. Therefore, insights gained in our research generalizes over the various domain.

The authors discuss the need for dependable robotics control software to manage complex behaviors called missions [23]. It notes the limitations of domain-specific languages being tied to specific robot models and the difficulty for non-experts in using logical languages like LTL. This research introduces a catalogue of 22 mission specification patterns for mobile robots, along with tools for creating mission specifications. These patterns help resolve common specification problems and provide a template mission specification in temporal logic. These patterns described in this domain can be considered as clones in our work. These patterns are identified for the purpose of creating a library of reusable specifications.

The authors in [20] addresses the challenge of transforming abstract specifications into executable programs, revealing new techniques and simplifications in program development. Originating from extensions to Dijkstra's guarded command language, it offers increased expressive power and procedural meaning. It integrates Z specifications into programming languages, leading to the formulation of refinement laws and practical guidelines.

The empirical study conducted on Event-B code clones [6] introduces the idea of code clones in Event-B, a formal specification language, and the research methodology which is also used in our research. The objective of the paper shares with ours, but focusing on different formal languages. The analysis and evaluation of our research, together with this paper's, could potentially provide interesting insights on code clones within formal specification languages.

No known work has been found on the detection of TLA+ specification clones or metrics for TLA+ specifications. Therefore, this report fills an obvious research gap in this domain.

3 Detecting Specification Clones (Approach)

In this chapter, we outline the code clone definition used for detection, system architecture, functional requirements and project code of the project.

3.1 Code Clone Definition

In this report, we are taking the idea of code clone definitions by Roy et al.[15] to create new definitions of clones and integrating it to detect clones in TLA+ specifications. We will be using an existing python library to detect clones by comparing pairs of sequences of code.

The difflib.SequenceMatcher class is designed to be flexible, allowing for the comparison of pairs of sequences of any type, provided that the elements within the sequences are hashable. It aims to find the longest contiguous matching subsequence while disregarding "junk" elements, such as blank lines or whitespace. This approach is applied recursively to both sides of the matching subsequence, resulting in matches that may not be minimal edit sequences but are more human-readable. This is calculated by the formula 2.0 * M / T, where T represents the total number of elements in both sequences, and M denotes the number of matches. It gives a value of 1.0 when the sequences are identical and 0.0 when they have no common elements. It is important to note that it is order-sensitive, meaning different order of characters in the method will give different values. With the ratio() method, it returns a measure of the sequences' similarity in the range [0,1] as float [2]. For better understanding, check the code example in Listing 2.

Listing 2: SequenceMatcher code examples [2]

```
1  >>> a = SequenceMatcher(None, "tide", "diet")
2  >>> b = SequenceMatcher(None, "diet", "tide")
3
4  >>> matches1 = sum(triple[-1] for triple in a.get_matching_blocks())
5  >>> matches1
6  1
7  >>> matches2 = sum(triple[-1] for triple in b.get_matching_blocks())
8  >>> matches2
9  2
10
11 >>> length = len("tide") + len("diet")
12 >>> cal_matches1 = 2.0 * matches1 / length
13 >>> cal_matches2 = 2.0 * matches2 / length
14
15 >>> cal_matches1 == a.ratio()
16 True
17 >>> cal_matches2 == b.ratio()
18 True
19
20 >>> a.ratio()
21 0.25
22 >>> b.ratio()
23 0.5
```

Listing 3: SequenceeMatcher class constructor [2]

```
SequenceMatcher(isjunk=None, a='', b='', autojunk=True)
```

The first argument of the call to SequenceMatcher shown in Listing 3 (line 1 in Listing 2 for example) is an optional argument. It takes in ‘None’, which is the default behavior, or a one-argument function that takes a sequence element. In Listing 3, the optional argument ‘isjunk=None’ is given, meaning no elements are ignored. This function should return True if the element is considered “junk” and should be ignored.A few examples of ‘junk’ include blanks and hard tabs.

Listing 2 shows how the order of two strings in the method could potentially result in different output values. SequenceMatcher.ratio uses SequenceMatcher.get_matching_blocks internally to compute the similarity ratio. Given two string variables, a and b, the SequenceMatcher.get_matching_blocks returns a list of triples, (i, j, n), where i and j are starting indices and n is the length of matching subsequences between two sequences. The triples are ordered increasingly by i and j. The last triple, (len(a), len(b), 0), denotes the end of matching subsequences. Adjacent triples don't represent directly adjacent equal blocks. The SequenceMatcher.ratio then uses the results obtained from SequenceMatcher.get_matching_blocks method to calculate the similarity ratio by finding the sum the sizes of all matched sequences shown in lines 4 and 7. Then using the formula 2.0 * M / T, it is multiplied times by 2 and divide by the total length of both strings shown in lines 12 and 13. Lines 15 to 18 shows the results of using SequenceMatcher.get_matching_blocks and formula are the same with SequenceMatcher.ratio, giving the final results shown in lines 20 to 23.

Using the ratio from 'SequenceMatcher' class, we have defined the different code clone types as such:

Type-1 code fragments that are identical with the ratio of 1.

Type-2 code fragments that are syntactically similar with the ratio, r, where $0.9 < r < 1$.

Type-3 code fragments that are syntactically similar with the ratio, r, where $0.8 < p \leq 0.9$.

Type-4 code fragments that are syntactically similar with the ratio, r, where $0.7 < p \leq 0.8$.

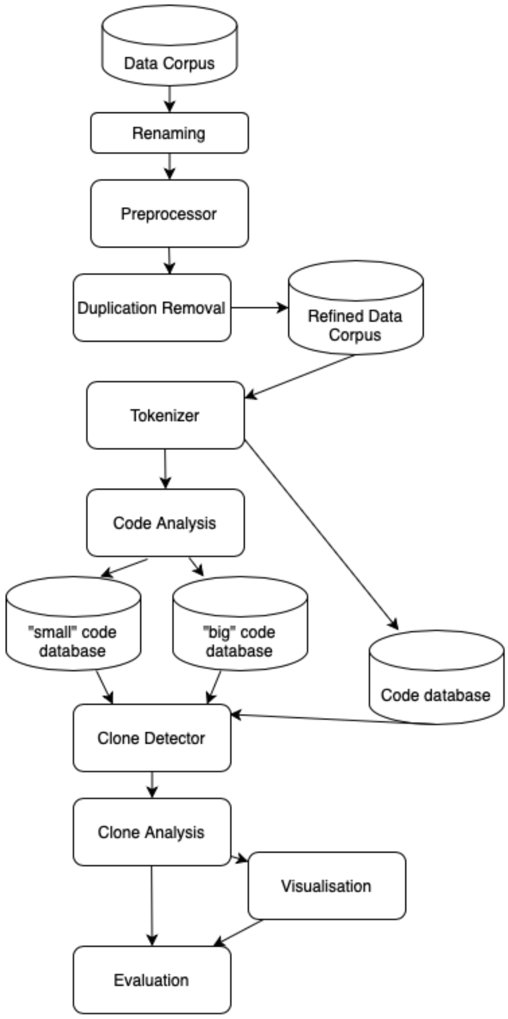
Type-5 code fragments that are syntactically similar with the ratio, r, where $0.7 < p \leq 0.2$.

We have chose to create new code clone definitions in order to find code clones in a quantifiable way, rather than using Roy et al.[15] which can be misclassified due to human error. In addition, Roy et al. focused on the functional behaviour of the program in their definition of Type-4 clones, whereas we focus on the syntax of the core logic of the program.

The type clone ranges are chosen to categorize code fragments based on their degree of similarity, allowing for more nuanced classification.

3.2 Architecture Diagram & Functional Requirements

Figure 1: System Architecture Design *[rounded rectangle = component, cylinder = database, arrow = process flow]*



The architecture diagram shown in Figure 1 provides an overview of the software used in this project. The rounded rectange indicates the components, the cylinder indicates the database and arrows indicate the flow of the system. We first gather

Requirement ID	Description
T1	Should only read files with the extension of tla
T2	Should tokenize files contained in the input directory (output directory of preprocessor)
T3	Should display the type of each token
T4	Should remove whitespace
T5	Should anonymize variable names
T6	Should rename different variable with the different label
T7	Should write the tokenized code into a new tla file
T8	Should store the new file in the output directory

Table 1: Functional requirements ('Tokenizer' component)

TLA+ specifications into a corpus, then we process and tokenize the files. These files are then used to detect clones producing analysis and statistics which will be later used for graph creation and in-depth clone analysis.

Functional requirements describe the functionality of the component or system. They are then checked using verification and validation to see if the requirements are met. Table 1 presents the functional requirements for the 'Tokenizer' component. The "T" in the requirement ID signifies 'Tokenizer'. The Description column outlines the specific functional requirement associated with each ID.

In Table 2, the Verification Method column specifies how each requirement will be verified. If a verification method is listed as 'by design', it indicates that the requirement is inherently fulfilled by the design of the code. If a verification method is listed as 'testing', it indicates that the requirement will be validated later using test cases. You can see that T1, T2, T3, T7, and T8 are requirements are fulfilled by design, and supported with explanation in Verification Explanation. T4, T5, and T6 requirements are verified separately through testing with different test cases. These are validated in Table 3. For each requirement and test case, an input is given as shown in the second column 'Input', and expected output after execution is shown in third column 'Expected Output'. After we run our code, the results are then shown in the last column 'Actual Output', where we compare if the actual output is the same with the expected output. This is to validate if our

Requirement ID	Verification Method	Verification Explanation
T1	by design	not require to test as it is set to only read the specific extension in code
T2	by design	not require to test as it is designed in the code
T3	by design	not require to test as it is designed in the code
T4	testing	to check if all whitespaces are being removed
T5	testing	to check if all variable names are correctly anonymized and the same variable name should be given the same anaoymized name
T6	testing	to check that different lables are given to different variables
T7	by design	not require to test as it is designed in the code
T8	by design	not require to test as it is designed in the code

Table 2: V&V for Tokenizer Functional Requirements

code is working as desired. This validation plays a major role as it checks if the functional requirements are fulfilled. Requirements with more complex functionality are validated with more than one test case, such as requirements on 'Clone Detector' component.

Further explanations regarding validation on these requirements can be found in 'Evaluation: Project Overview' (Section 4.3). The extensive list of functional requirements of the whole system and their validations can be found in the Appendix 5.

3.3 Specification Corpus & Program Algorithms

In this section, we discuss how we have gathered TLA+ specifications to create a specification corpus, and then discuss some algorithms of the components of the system.

Req. ID	Input	Expected Output	Actual Output
T4 & T5	Range(f) = {f[x] : x ∈ DOMAIN f}	\$n1(\$n2) = { \$n2[\$n3]:\$n3 ∈ DOMAIN\$n2}	\$n1(\$n2) = {\$n2[\$n3]:\$n3 ∈ DOMAIN\$n2}
T6	e = 4 r = 2 q = e + r t = e - r	\$n1 = # \$n2 = # \$n3 = \$n1+\$n2 \$n4 = \$n1-\$n2	\$n1 = # \$n2 = # \$n3 = \$n1+\$n2 \$n4 = \$n1-\$n2

Table 3: Validation table for T4, T5 and T6. Note that the notation has been simplified for presentation purposes.

3.3.1 A Corpus of TLA+ Specifications

Before we begin to detect specification clones to gather useful results and insights, we first collect specifications to process. The specifications are collected into a single directory named 'Data Corpus'. These specifications are sourced from various resources consisting of different domains, listed here.

3.3.2 Program Implementation

Here, we will present and interpret some of the algorithms of the components in our architecture diagram (Figure 1). Full original code of these algorithms can be accessed in 16.

3.3.2.1 'Renaming' Component

The 'Renaming' component iterates through all the files within the 'Data Corpus' are renamed in ID format for simplicity and to eliminate filenames such as 'specification.tla copy'. Since all components further down the pipelines exclusively handle TLA+ files, filenames such as 'specification.tla copy' will be disregarded, resulting in missed opportunities to identify valuable clones from that specific file. Thus, leading to lower number of processed files, detected code clones and inaccurate results. The original and new filenames are then logged in a separate text file, such as Basic.tla is renamed as 0001.tla. Here's a few examples of renaming from the log file as shown in Listing 4.

Listing 4: A Few Examples From The Renaming Log File

```
Renamed Level\_test.tla to 0001.tla
Renamed Basics.tla to 0002.tla
Renamed Paxos2.tla to 0003.tla
Renamed foo1.tla copy to 0004.tla
Renamed TestBug140131B.tla to 0005.tla
Renamed test47a.tla to 0006.tla
Renamed AllocatorRefinement.tla to 0007.tla
Renamed Test2a.tla to 0008.tla
Renamed Consensus.tla copy 2 to 0009.tla
Renamed April25MC.tla to 0010.tla
Renamed Euclid2.tla to 0011.tla
Renamed Quicksort05.tla to 0012.tla
Renamed test27.tla to 0013.tla
Renamed function17_test.tla to 0014.tla
Renamed false_proves_false.tla to 0015.tla
Renamed ExpandOnlyENABLED_test.tla to 0016.tla
Renamed function16_test.tla to 0017.tla
Renamed test33.tla to 0018.tla
Renamed SequenceTheorems.tla copy to 0019.tla
Renamed smt_false_test.tla to 0020.tla
```

3.3.2.2 'Preprocessor' Component

Afterward, the files are processed by the 'Preprocessor' component. Certain TLA+ files might include translations to PlusCal, which can be generated from the TLA+ Toolbox IDE and appear after the "BEGIN TRANSLATION" text. Therefore, this component considers both these translations and any non-code lines. The code is written in python.

In Listing 5, the preprocess_and_parse function from lines 1 to 16 reads through the content of the file and preprocesses lines according to specified rules and then returns the result. It first opens a file specified by the file_path. It reads the content of the file line by line and processes each line according to specified rules. If a line contains the phrase "BEGIN TRANSLATION", it sets a flag called ignore_lines to True, indicating that subsequent lines should be ignored because these are PlusCal, and not the standard TLA+ that we focus our analysis on. If not, lines are further examined. Lines starting with at least four spaces are appended to the last element of the list preprocessed_lines, and lines containing

Listing 5: ‘Preprocessor’ Pseudocode

```
1  function preprocess_and_parse(file_path)
2      open and read lines from file
3      create empty list preprocessed_lines
4      set ignore_lines to False
5
6      for each line in lines
7          if line contains "BEGIN TRANSLATION"
8              set ignore_lines to True
9          else if not ignore_lines
10             if line starts with at least four spaces
11                 if preprocessed_lines is not empty
12                     append trimmed line to the last element of preprocessed_lines
13             else if "==" is in line and line starts with a letter, digit, or space
14                 append trimmed line to preprocessed_lines
15
16     return preprocessed_lines joined with newline characters
17
18 function process_tla_files(input_dir, output_dir)
19     iterate through files in input directory
20         if file ends with ".tla"
21             input_file_path = input_dir concatenated with file
22             output_file_path = output_dir concatenated with file
23             parsed_code = preprocess_and_parse(input_file_path)
24             open output_file_path and write parsed_code
```

“==” and starting with a letter, digit, or space are appended directly to preprocessed_lines. Finally, the function returns the preprocessed lines as a single string joined by newline characters.

The process_tla_files function from lines 18 to 24 is responsible for managing the preprocessing of all TLA+ files within the input directory. First, it checks if the output directory specified by output_dir exists and creates it if it does not. For each file ending with “.tla” in the input directory (input_dir), it constructs the input and output file paths, and writes the preprocessed content from preprocess_and_parse function to the corresponding output file.

In the main script (not shown in the listing), the input directory and the output directory are set. The process_tla_files function is called with these directories as arguments, initiating the preprocessing of TLA+ files.

Listing 6: 'Duplication Removal' Pseudocode

```
function find_duplicate_files(directory):
    files_by_content = {}  // Map content to file paths
    removed_files_log = []  // Store removed file paths

    for each file in directory:
        if file is not directory:
            content = read_file_content(file)
            if content in files_by_content:
                add file to files_by_content[content]
            else:
                create new entry in files_by_content with content mapped to file

    for each content, file_paths in files_by_content:
        if length of file_paths > 1:
            for i from 1 to length of file_paths:
                remove file_paths[i]
                add file_paths[i] to removed_files_log

    write removed_files_log to log file outside directory

function read_file_content(file):
    open and read file content
    return content

set directory_path
find_duplicate_files(directory_path)
```

3.3.2.3 'Duplication Removal' Component

The preprocessed files added into the new directory (output_directory from Listing 5) are then processed through the 'Duplication Removal' component. It looks through each file in the folder. When it finds files with the same content, it keeps only one copy and removes the duplicates. The names of the removed files are saved in a file called removed_files_log.txt. We are doing this to avoid false positives, such as Type-1 clones are detected from two exact same files.

In Listing 6, the code defines a function find_duplicate_files(directory) designed to identify and remove duplicate files within a specified directory. It begins by initializing an empty dictionary named files_by_content to store file contents as keys and their corresponding paths as values in lines.

Using os.walk (directory), it traverses through the files in the directory. 'os.walk()' is a function in Python's built-in os module used for traversing a directory tree, that is, visiting all directories and files recursively starting from a given directory. It generates the file names in a directory tree by walking either top-down or bottom-up. In this case, for each file encountered, it checks if the file is not a directory, reads its content, and stores it in the files_by_content dictionary. If the content already exists as a key in the dictionary, it appends the file path to the list of paths associated with that content; otherwise, it creates a new entry with the content as the key and the file path as the value as shown in lines 2 to 11.

After collecting file paths based on their content, the code proceeds to filter out files with identical content. It iterates through the dictionary and for each content with more than one associated file path, it removes all but one of the files. The removed file paths are stored in the removed_files_log list as sown in lines 13 to 17.

Finally, in line 19, the code writes the names of removed files stored in removed_files_log to a log file named 'removed_files_log.txt' located outside the directory being processed.

3.3.2.4 'Tokenizer' Component

The results are placed in the 'Refined Data Corpus' dataset, which then undergoes the 'Tokenizer' component as shown in Listing 7.

Listing 7 outlines the tokenization process of TLA+ code files. It begins by defining the input and output directories, where TLA+ files are located and where the tokenized files will be saved, respectively. It then checks if the output directory exists and creates it if it doesn't.

Next, from lines 3 to 17, the anonymize_variable_names(tla_code) function is defined. This function takes TLA+ code as input and tokenizes it by anonymizing variable names, replacing integers with a placeholder symbol '#', and preserving other symbols. From lines 6 to 16, the tokenization process involves iterating through each line of the TLA+ code. Comments are removed from each line, and integers are replaced with '#'. Variable names are then anonymized, with each unique variable being replaced by a corresponding anonymized name ('$name1', '$name2', and so on) .

Listing 7: 'Tokenizer' Pseudocode

```
1  Define input and output directories
2
3  Function anonymize_variable_names(tla_code):
4      Split TLA+ code into lines
5      Initialize variable count and mapping
6      Iterate over each line:
7          Remove comments
8          Replace integers with #
9          Tokenize variable names:
10             Extract words from line
11             For each word:
12                 If it's a valid variable name:
13                     If not encountered before, map it to an anonymized name
14                 Append the anonymized word to the list
15         Concatenate anonymized words into a line
16         Append the line to the list of anonymized lines
17     Return the joined anonymized lines
18
19 For each file in input directory:
20     If it's a TLA+ file:
21         Read TLA+ code from the file
22         Tokenize the code using anonymize_variable_names function
23         Write the tokenized code to a new file in the output directory
```

After defining the tokenization function, the code then processes each TLA+ file in the input directory. For each file ending with ".tla", it reads the content of the file and passes through the anonymize_variable_names function. The tokenized code is then written to a new file in the output directory with the original filename appended with '_tokenized.tla' as shown in lines 19 to 23.

Finally, upon completing the tokenization process for all TLA+ files, the code prints a message indicating that the TLA+ code tokenization is complete in line 25.

3.3.2.5 'Code Analysis' Component

In the 'Code Analysis' component, the files are analyzed to gather code lines, which are later used as an indicator to categorize "small" and "large" datasets. The average total code lines from all files are used as a threshold for categorization.

Listing 8 defines a function named count_lines_in_files, which takes in several parameters: the directory containing the TLA+ files (directory), the filename for

Listing 8: ‘Code Analysis’ Pseudocode

```
1  Define function count_lines_in_files(directory,csv_output,small_file,large_file):
2      List all files with ".tla" extension in the specified directory
3      Initialize an empty list to store line count data
4      Initialize a variable for total line count
5
6      For each file in the directory:
7          Open and count the lines in the file
8          Append file name and line count to line count data list
9          Update total line count
10
11     Calculate mean line count
12
13     Write line count data to a CSV file
14
15     Create directories for small files and large files
16     Iterate through line count data:
17         Determine source and destination paths based on line count
18         Copy files to corresponding directories
```

CSV output (csv_output), and the names of directories for storing small and large files (small_file and large_file) respectively. In line 2, it begins by listing all files in the specified directory that have the ‘.tla’ extension. In lines 6 to 9, it then iterates over each file, calculating the sum of number of lines in each file. These line counts, along with the corresponding filenames, are stored in a list of dictionaries named line_count_data in line 8.

After collecting line count data for all files, the code calculates the mean line count across all files as shown in line 11. This mean is used to categorize the files into “small” and “large” categories based on whether their line count is below or above the mean, respectively.

In line 13, the code writes the line count data to a CSV file specified by csv_output, organizing the data with columns for file names and their respective line counts.

Then in lines 14 to 18, the code creates directories for “small” and “large” files outside of the specified directory path. It then copies the TLA+ files into these directories based on their line counts. Files with line counts below the mean are copied to the “small” directory, while those with counts above the mean are copied to the “large” directory.

The count_lines_in_files function is called twice, each time with different input directories ("tokenized_files" and "parsed_files"). It is done to analyzes and categorizes TLA+ files from both directories, generating corresponding CSV files and organizing the files into "small" and "large" categories.

3.3.2.6 'Clone Detector' Component

The main dataset, dataset before categorization , "small" and "large" datasets are then passed through the 'Clone Detector' to identify any code clones within individual files, between two files from parsed directory and between two files from tokenized directory.

Listing 9: ‘Clone Detector’ Pseudocode

```
Function detect_clones_individual(file_path)
    code = Read lines from file_path
    clones = Empty list to store clone information
    For each line1 in code with index i
        For each line2 in code starting from i+1 with index j
            similarity = Calculate similarity between line1 and line2
            If similarity > 0.2
                If similarity is 1
                    clone_type = "Type-1 Clone"
                Else If similarity > 0.9
                    clone_type = "Type-2 Clone"
                Else If similarity > 0.8
                    clone_type = "Type-3 Clone"
                Else If similarity > 0.7
                    clone_type = "Type-4 Clone"
                Else
                    clone_type = "Type-5 Clone"
                Append (i + 1, j + 1, similarity, clone_type) to clones
    Return clones
Function individual_process_directory(directory_path, clone_csv, statistics_csv)
    Initialize start_time with current time
    Open clone_csv and statistics_csv for writing
    Write header rows to both CSV files
    For each file in directory_path
        If file ends with ".tla"
            clones = Call detect_clones_individual with file path
            Calculate clone statistics
            Write clone information to clone_csv
            Write clone statistics to statistics_csv
Function detect_clones_files(file_path1, file_path2)
    Read lines from file_path1 as code1
    Read lines from file_path2 as code2
    clones = Empty list to store clone information
    For i from 0 to length of code1
        For j from 0 to length of code2
            similarity = Calculate similarity between code1[i] and code2[j]
            If similarity is 1
                clone_type = "Type-1 Clone"
            Else If similarity > 0.9
                clone_type = "Type-2 Clone"
            Else If similarity > 0.8
                clone_type = "Type-3 Clone"
            Else If similarity > 0.7
                clone_type = "Type-4 Clone"
            Else
                clone_type = "Type-5 Clone"
            If similarity > 0.2
                Append (file_path1 basename, file_path2 basename, i + 1, j + 1,
                similarity, clone_type) to clones
    Return clones
```

Listing 10: ‘Clone Detector’ Pseudocode part 2

```
Function files_process_directory(directory_path, output_csv, output_txt)
    Initialize start_time with current time
    Open output_csv and output_txt for writing
    Write header row to output_csv
    Initialize clone statistics counters
    For each file1 in directory_path
        For each file2 in directory_path starting from file1+1
            clones_tokenized = Call detect_clones_files with file paths
            If clones_tokenized is not empty
                Increment total_clone_pairs counter by length of clones_tokenized
                For each clone in clones_tokenized
                    Extract clone information
                    Increment respective clone type counter
                    Write clone information to output_csv
    Write clone statistics to output_txt
```

In Listing 9, the code's purpose is to detect code clones within and between files. The code first defines several parameters such as input directories for both parsed and tokenized files, and output file paths for storing clone data and statistics. It then initializes directories for storing small and large files.

In lines 1 to 19, it implements functions to detect clones for individual files. The ‘detect_clones_individual’ function scans individual files to identify code clones within each file. It uses the SequenceMatcher class from the difflib module to calculate similarity ratios between lines, categorizing clones based on predefined thresholds. Explanation on SequenceMatcher class from the difflib module can be found in Section 3.

In lines 20 to 29, the ‘individual_process_directory’ function processes a directory of tokenized files, detecting clones within each file using the function ‘detect_clones_individual’ and writing the results to CSV files. It also computes statistics on clone types and their frequencies, providing insights into the distribution of clones within files.

The ‘detect_clones_files_directory’ function, on lines 31 to 50, analyzes files within a directory, detecting clones between files and recording the results in CSV files. It then creates statistics on clone types and their frequencies, providing a comprehensive understanding of clone distribution. In Listing 10, the ‘files_process_directory‘ function reads files within a directory, detecting clones be-

Listing 11: One of Many Code for Data Visualization (Clone Type Pie Chart Pseudocode)

```
1  function clone_type_pie(parsed_stats, tokenized_stats, png_image):
2      read content from parsed_stats file into lines_file1
3      read content from tokenized_stats file into lines_file2
4
5      type1_file1 = extract Type-1 clone count from lines_file1
6      type1_file2 = extract Type-1 clone count from lines_file2
7      type2_file2 = extract Type-2 clone count from lines_file2
8      type3_file2 = extract Type-3 clone count from lines_file2
9      type4_file2 = extract Type-4 clone count from lines_file2
10     type5_file2 = extract Type-5 clone count from lines_file2
11
12     calculate total number of clones across all types
13     calculate percentage of each clone type relative to total
14     create labels, sizes, percentages for pie chart segments
15     plot pie chart with specified parameters
16     save the plot as png_image
```

tween files and recording the results along with statistics on clone types.

Lastly, in the main section, the script executes the individual clone detection process for the whole dataset using 'detect_clones_individual' and 'individual_process_directory' functions. Then, the parsed and tokenized files clone detection process for all small, large and whole datasets, using 'detect_clones_files_directory' and 'files_process_directory' functions.

3.3.2.7 'Visualization' Component

The results and statistics from 'Clone Detector' are used to create more useful statistics and visualization in 'Clone Analysis' and 'Visualization' components respectively, followed by chapter reference to evaluation.

In Listing 11, the code defines a function 'clone_type_pie' that generates pie charts representing the distribution of different clone types across two sets of statistics files. These statistics files contain information about the types of clones detected in parsed and tokenized datasets. The function takes three parameters: 'parsed_stats', 'tokenized_stats', and 'png_image'.

In lines 2 and 3, the function reads the contents of two statistics files, which are typically generated by the code analyzing cloned code fragments. These files contain counts of different types of clones, such as Type-1, Type-2, Type-3, etc., along with their percentages. In lines 5 to 10, the function then extracts relevant data from these files, specifically focusing on the counts of Type-1 clones in parsed files and the counts of all clone types in tokenized files.

After extracting the necessary data, the function calculates the percentage of each clone type out of the total number of clones detected in lines 12 and 13. This percentage information is then used when representing the data in the pie chart.

Using matplotlib in lines 14 and 15, the function plots a pie chart with labeled segments representing each clone type. The size of each segment corresponds to the proportion of that clone type relative to the total number of clones. The legend of the pie chart presents labels of the clone type, the percentage of that clone type, and the count of clones belonging to that type.

Finally, in lines 16, the function saves the generated pie chart as an image file in PNG format using the specified filename provided in the 'png_image' parameter. The function is called three times, each time with different pairs of statistics files representing different datasets: small files, large files, and the entire dataset.

We evaluate useful insights in the 'Evaluation' component, which will be discussed in Section 4.

4 Analysis and Evaluation

In this chapter, we will analyze and evaluate the results obtained from clone detection from three datasets (main, small, and large), and discuss what insights and conclusions can be drawn from this information. In the following paragraphs, we will use abbreviations when naming different clone types for readability. The abbreviations used are the same definition used in Section 3.

Our clone types are defined as follows:

parsed-type1 indicates type-1 clone from **parsed** file (untokenized), meaning the clones are identical in original context (Type-1 as described in Section 3)

tokenized-type1 indicates type-1 clone from **tokenized** file, meaning the clones are identical in tokenized form (Type-1 as described in Section 3)

tokenized-type2 indicates type-2 clone from **tokenized** file, meaning the clones have the similarity percentage, p where $90 < p < 100$ (Type-2 as described in Section 3)

tokenized-type3 indicates type-3 clone from **tokenized** file, meaning the clones have the similarity percentage, p where $80 < p \leq 90$ (Type-3 as described in Section 3)

tokenized-type4 indicates type-4 clone from **tokenized** file, meaning the clones have the similarity percentage, p where $70 < p \leq 80$ (Type-4 as described in Section 3)

tokenized-type5 indicates type-5 clone from **tokenized** file, meaning the clones have the similarity percentage, p where $70 < p \leq 20$ (Type-5 as described in Section 3)

We have split our corpus of TLA+ specifications into three datasets for analysis: main, small and large. The main dataset is the original tokenized dataset, whereas the rest is categorized by the number of code lines in the file. The average number of code lines is calculated from the files from the main dataset, which in this case is 11. Using it as a threshold, files are placed into small dataset if the number of code lines in the focused file is less than the average number of code lines. If the number of code line is larger or equal to the average, then it is placed into the large dataset. After categorizing, we get 776 files in small dataset, and 362 files in large dataset.

4.1 Analysis

The pie chart analysis of small datasets (Figure 2) reveals interesting insights into clone distribution, excluding tokenized-type5 clones. Given that tokenized-type5 clones comprise a substantial number of pairs (3458567), far exceeding other types of clones, we chose to omit them for clarity and to concentrate on the narrower categories.

Parsed-type1 clones comprise a modest 2.01%, indicating relatively infrequent exact matches in parsed files. Conversely, tokenized-type1 clones show a slight increase at 3.92%, suggesting a higher occurrence of identical segments in tokenized files. Tokenized-type2 clones represent 4.39%, indicating a significant frequency of closely resembling segments, although not exact duplicates. Additionally, tokenized-type3 clones account for a considerable 32.87%, indicating a notable occurrence of moderately similar code segments among tokenized files. However, the most striking finding is tokenized-type4 clones, dominating at 56.82%.

Figure 2: Distribution of clones (the small dataset)

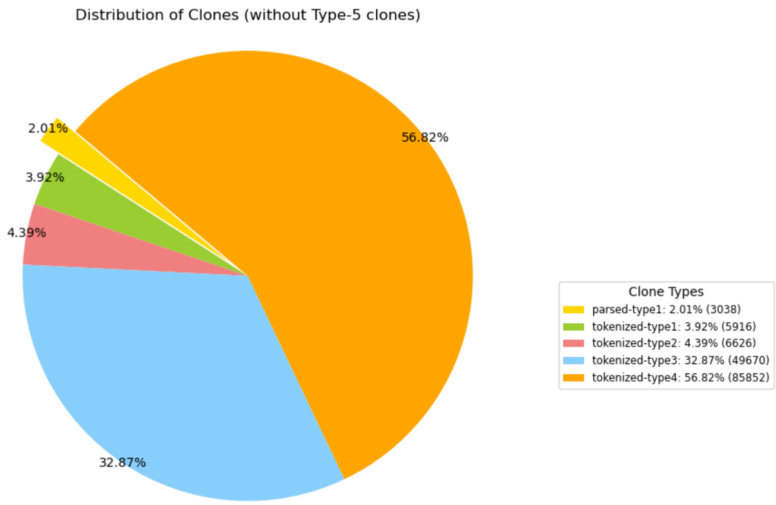
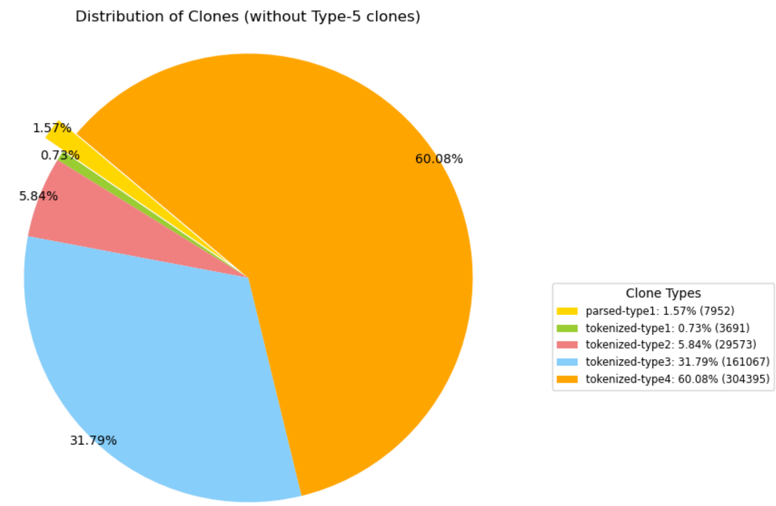
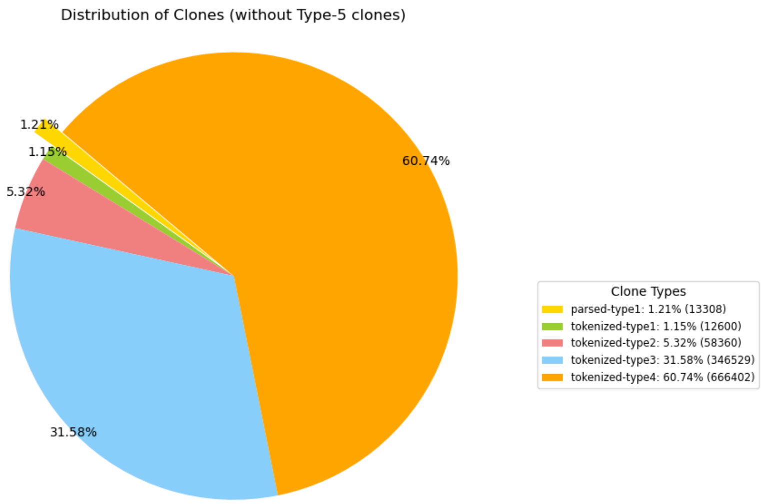


Figure 3: Distribution of clones (the large dataset)



In the pie chart shown in Figure 3, it is evident that different types of clones, excluding tokenized-type5, contribute to varying degrees within the dataset. Specifically, parsed-type1 clones account for 1.57%, indicating a relatively lower

Figure 4: Distribution of clones (the whole dataset)



occurrence of exact matches between parsed files. On the other hand, tokenized-type1 clones make up 0.73%, suggesting a similarly low occurrence of exact matches among tokenized files. However, the prevalence of tokenized-type4 clones is notably, taking up 60.08%. Moreover, tokenized-type3 clones contribute to a considerable extent, constituting 31.79% of the total clones. This suggests a considerable occurrence of highly similar code segments with minor differences among tokenized files. Additionally, tokenized-type2 clones, though relatively lesser in comparison, still make up a notable portion at 5.84%, indicating a moderate occurrence of similar code segments with relatively more significant variations among tokenized files.

When comparing between the large dataset (Figure 3) and the small dataset (Figure 2), the trend of increasing tokenized-type1 to tokenized-type4 remains consistent. The only notable difference is the larger number of parsed-type1 clones compared to tokenized-type1 clones in the large datasets, whereas the small dataset shows a higher count of tokenized-type1 clones over parsed-type1 clones.

In the pie chart shown in Figure 4, it is evident that different types of clones, excluding tokenized-type5, contribute to varying degrees within the dataset. Among these, parsed-type1 clones occupy a relatively smaller proportion, consti-

Listing 12: Code in file 1

```
1 x + y = 4
2 c - e = 10
```

Listing 13: Code in file 2

```
1 a + b = 4
2 x + y = 4
```

tuting only 1.21% of the total. In contrast, tokenized-type1 clones, while slightly lower at 1.15%, still represent a significant presence. However, the data indicates a substantial prevalence of tokenized-type3 clones, comprising a significant portion at 31.58%. Moreover, tokenized-type4 clones dominate the distribution, representing a majority share at 60.74%.

The distribution of clone types among files in the 'small', 'large', and 'main' datasets is represented in the three pie charts above. For a comprehensive view of the distribution of all clone types, including tokenized-type5, please refer to the appendix (Figure 11).

It's evident that tokenized-type4 composes over 55% of the distribution, followed by tokenized-type3, tokenized-type2, and tokenized-type1 in descending order across all charts. Additionally, there is a noteworthy observation in the 'large' and 'main' datasets, where the percentage of parsed-type1 exceeds that of tokenized-type1. This suggests a higher occurrence of clones that are identical in their original context compared to their tokenized form. This is due to the naming of tokenized variables in different files. In parsed files, the code detector will compare the syntax as it is. But in tokenized files, the naming of variable when tokenizing can result in different clone type identification. Consider the following case:

If you look at line 1 of file 1 [Listing 12] and line 2 of file 2 [Listing 13], you can see they are exactly the same. This would give us similarity ratio of 1. But if you look at the same files but tokenized, line 1 in file 1 [Listing 14] and line 2 in file 2 [Listing 15], you can see that 'var1 + var2 = #' and 'var3 + var4 = #' are not exactly the same. Thus, the similarity ratio is less than 1, even though its

Listing 14: Tokenized code in file 1

```
var1 + var2 = #
var3 - var4 = #
```

Listing 15: Tokenized code in file 2

```
var1 + var2 = #
var3 + var4 = #
```

Figure 5: A part of the table displaying statistics about code clones

1	Line	Count	Files
2	Init ==	70	0862.tla, 0917.tla, 1189.tla, 0685.tla,
3	Spec == Init /\ [][Next]_vars	47	1003.tla, 0917.tla, 0240.tla, 0452.tla,
4	Next ==	34	0294.tla, 0492.tla, 0874.tla, 0858.tla,
5	TypeOK ==	33	0862.tla, 0917.tla, 1189.tla, 0492.tla,
6	Next == UNCHANGED x	33	1639.tla, 0902.tla, 0526.tla, 1599.tla,
7	Init == x = 0	32	0123.tla, 0042.tla, 0849.tla, 1617.tla,
8	TRUSTED_HEIGHT == 1	21	0916.tla, 0657.tla, 0508.tla, 0286.tla,
9	TypeInvariant ==	20	0137.tla, 0333.tla, 0124.tla, 0544.tla,
10	Spec == Init /\ [][Next]_vars /\ WF_vars(Next)	20	0888.tla, 0525.tla, 0327.tla, 0441.tla,
11	Spec == Init /\ [][Next]_x	19	0123.tla, 0082.tla, 0253.tla, 1429.tla,

original syntax ratio is 1. This is due to the order of renaming the variables during tokenization.

In the ‘small’ datasets, parsed-type1, tokenized-type1, and tokenized-type2 collectively account for approximately 10% of the chart, while in the ‘large’ and ‘main’ datasets, they make up around 8%. This variance indicates a lower frequency of clones with higher similarity in larger files compared to smaller ones.

Figure 5 presents code clones extracted from parsed files before syntax anonymization. Each code line is accompanied by its frequency and the filenames containing that specific code line.

Particularly, the first code line has a frequency of 70, which is expected since many specifications require initialization before building or testing. A similar concept is observed in rows 3, 4, and 9.

Figure 6: A part of the table displaying statistics about code clones

1	Line	Count	Files
31	vars == <<active, color, tpos, tcolor>>	10	0862.tla, 0492.tla, 0682.tla, 0790.tla,
32	TerminationDetection ==	10	0862.tla, 0492.tla, 0682.tla, 0790.tla,
33	LET Automorphisms(S) == { f \in [S -> S] : \A y \in S : \E x \in S : f[x] = y } f ** g == [x \in DOMAIN g |-> f[g[x]]]	10	0327.tla, 0441.tla, 0118.tla, 1577.tla,
34	vars == <<x>>	10	1502.tla, 0218.tla, 0754.tla, 1113.tla,
35	Spec == Init /\ [][Next]_vars /\ WF_vars(System)	9	0862.tla, 1060.tla, 0790.tla, 0394.tla,
36	AllNodesTerminateIfNoMessages ==	9	0862.tla, 0492.tla, 0682.tla, 0790.tla,
37	ASSUME ConstantAssump == /\ N \in Nat \ {0} /\ A0 \in [1..N -> Int]	9	0327.tla, 0441.tla, 0118.tla, 1577.tla,
38	vars == <<A, U>>	9	0327.tla, 0441.tla, 0118.tla, 1577.tla,
39	Init == /\ A = A0 /\ U = { <<1, N>> }	9	0327.tla, 0441.tla, 0118.tla, 1577.tla,
40	Spec == Init /\ [][Next]_<<x,y>>	9	1013.tla, 1316.tla, 0619.tla, 0591.tla,
41	InitiateProbe ==	8	0862.tla, 0682.tla, 0790.tla, 0394.tla,
42	SendMsg(i) == /\ active' = [active EXCEPT ![j] = TRUE] /\ color' = [color EXCEPT ![i] = IF j>i THEN "black" ELSE @]	8	0862.tla, 0492.tla, 0682.tla, 0790.tla,
43	Send(m) == msgs' = msgs \cup {m}	8	0081.tla, 1365.tla, 0356.tla, 0750.tla,
44	Spec == Init /\ [][Next]_vars /\ Fairness	8	0492.tla, 0682.tla, 1115.tla, 0803.tla,

Upon closer examination of more complex code lines, rows 31 and 33 as shown in Figure 6 stand out with frequencies of 10 each, indicating that this section of the code has been duplicated across different specifications. By setting a threshold of 5 for frequency, a total of 183 code clones are identified out of 2011. Total frequencies of 1466 for those with more than the frequency of 4, and total frequencies of 4413 otherwise. That's 9% of the code clones consisting 33.2% of the whole frequency of code clones. We would like to emphasize that these code clones do not come from files that are duplicated which could give us higher number of clone pairs and inaccurate results. All duplicated files are removed in 'Duplication Removal' component, and code clone collection takes place in 'Clone Analysis' component.

Figure 7: First few rows of table displaying statistics of each file

File Name	Line Count	Clone Pairs	Type-1 Clones	Type-2 Clones	Type-3 Clones	Type-4 Clones	Type-5 Clones
1124_tokenized.tla	117	5033	0	232	366	68	4367
0140_tokenized.tla	115	3546	0	151	196	33	3166
1027_tokenized.tla	114	4747	0	241	356	61	4089
1612_tokenized.tla	109	2967	0	18	13	55	2881
0724_tokenized.tla	106	3367	0	36	59	100	3172
1191_tokenized.tla	105	3296	0	38	60	102	3096
1321_tokenized.tla	99	2980	0	69	216	258	2437
0994_tokenized.tla	99	2988	0	73	196	287	2432
1303_tokenized.tla	83	1487	0	8	5	24	1450
0146_tokenized.tla	81	1208	0	5	5	5	1193
0912_tokenized.tla	80	1570	0	20	53	31	1466
0543_tokenized.tla	79	3002	0	1336	876	5	785

Figure 8: graph of line count and clone pair

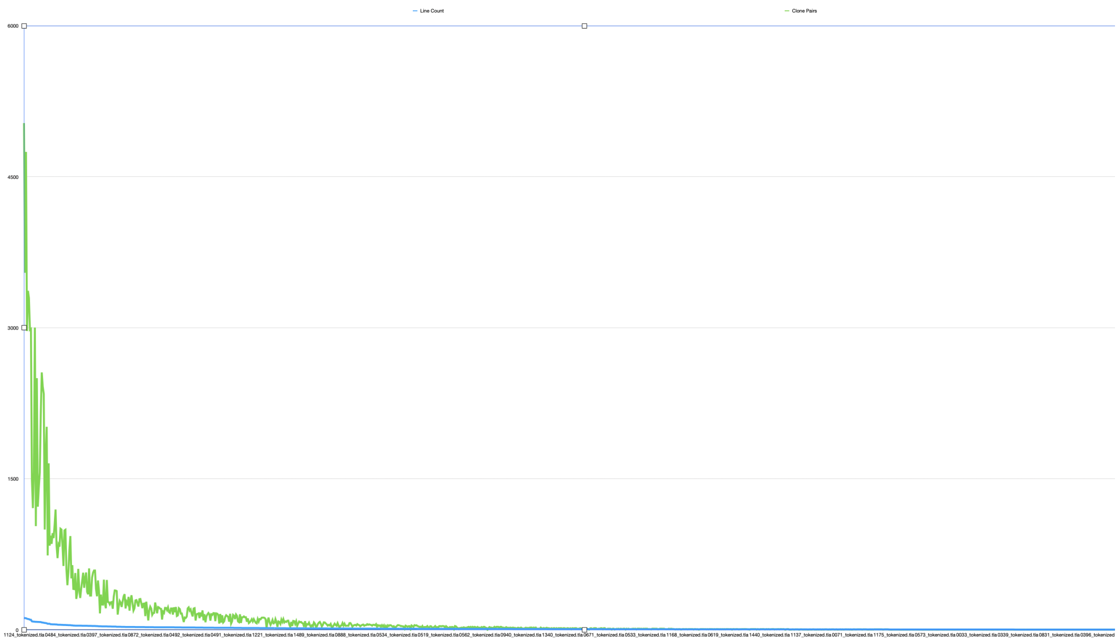
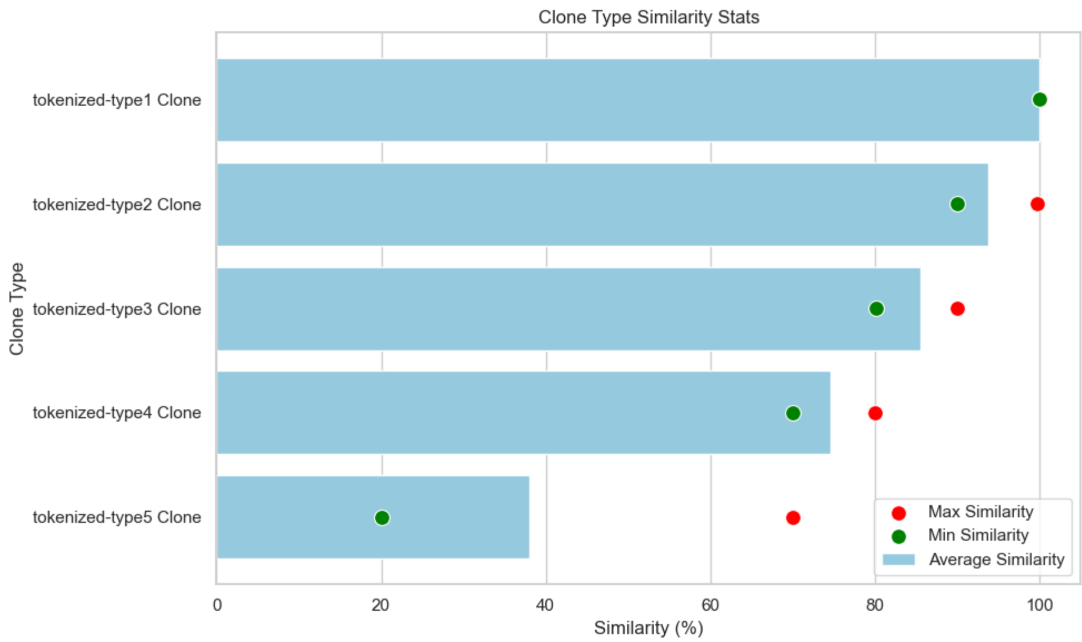


Table 7 displays the counts of code lines, clone pairs, and different types

Figure 9: Minimum and Maximum Percentage Range of Each Clone Type



of clones in each file, and Figure 8 displays line graph of line count and clone pair in each files. By looking at the table and graph, it's evident that there is no consistent correlation between the number of code lines and the presence of clone pairs. In Figure 8, you can see that on the left side of the curve, there are rapid fluctuations of the green line (clone pairs), compared to smooth curve of the blue line (line count). The tokenized-type5 clones are the most common across the majority of files, while the occurrence of other clone types varies. tokenized-type1 clones are the least common, appearing in only 12 files, with a maximum count of 7 and a minimum count of 0.

Figure 9 displays the average similarity percentage of each clone type along with the minimum and maximum percentage values of each clone type (tokenized-type1 clones to tokenized-type5 clones) as green and red dots respectively. If there's only one green dot and no red dot present on a single row, this indicates equal minimum and maximum value of percentage.

Looking at the first bar (tokenized-type1 clones), the average, minimum and maximum similarity percentages are the same at 100%. If you look back to the code clone definition used in this research in Section 3. The criteria for a clone to be tokenized-type1 clones is to have a similarity ratio of 1. Therefore, the average, minimum and maximum similarity percentages are the same.

If you look at the rest of the bar (tokenized-type2 clones, tokenized-type3 clones, tokenized-type4 clones, tokenized-type5 clones), there are average, minimum and maximum similarity percentage values. This is the case as if we refer back to the code clone definition used in this research (described in Section 3), the criteria for a clone to be categorized are in range values instead of a single value like the first bar (tokenized-type1 clones). The minimum and maximum similarity values of each bar are the range value of each code clone. You will see that the last bar (tokenized-type5 clones as a short bar and average similarity value, as this is because the range for a clone to be tokenized-type5 clones is to have the ratio of less than 0.7 and greater than or equal to 0.2. In contrast, tokenized-type2 clones, tokenized-type3 clones, and tokenized-type4 clones have the range value of $0.9 < r < 1$, $0.8 < p \leq 0.9$, $0.7 < p \leq 0.8$ respectively, and where r is the similarity ratio.

4.2 Evaluation: TLA+ Specification Clones

The detailed analysis reveals that less than 45% of the distribution of clone types across files comprises tokenized-type3 clones, tokenized-type2 clones, tokenized-type1 clones, and parsed-type1 clones, where the similarity exceeds 81%. It is notable that in larger files, there is a higher probability of identifying parsed-type1 clones compared to tokenized-type1 clones, particularly when examining smaller files.

Code clones with a very high frequency often represent basic syntax elements of TLA+, such as initialization. Furthermore, other code clones presenting sufficiently high frequencies offer valuable insights into their intended purpose and usage patterns. This represents the importance of investigating the potential causes and implications of code clones for TLA+ programmers. Moreover, it hints at the potential need of implementing effective modularization techniques within the TLA+ formal language.

Upon the detection of clones within individual files, the findings indicate no significant correlation between the presence of a greater number of clone pairs and a high count of code lines. Additionally, the distribution of clone types may vary significantly.

An argument can be made that the occurrences of parsed-type1 and tokenized-type1 should align. However, due to differences in naming conventions during tokenization and the sequential comparison approach, the statistics for code detection differ between parsed-type1 clones and tokenized-type1 clones, as illustrated and explained in Listing 12.

The discovery of a relatively low number of parsed-type1 clones within TLA+ specifications does not provide sufficient grounds for drawing specific conclusions. This could possibly be due to a deficiency in the specifications themselves.

Upon analyzing Table 8 carefully, it suggests that there is no consistent correlation between the number of code lines and the presence of clone pairs. This observation is supported by the fluctuating nature of the green line (representing clone pairs) compared to the relatively smooth curve of the blue line (representing line count).

The analysis done by reading the statistics and graphs of different datasets identifies tokenized-type5 clones as the most common across the majority of files, while other clone types exhibit varying occurrences. The tokenized-type1 clones are noted to be the least common, appearing in only 12 files. The graph is provided in Appendix 11.

In Figure 9, the graph explains the consistency in the average, minimum, and maximum similarity percentages for tokenized-type1 clones due to their definition requiring a similarity ratio of 1. Conversely, the range values for tokenized-type2 clones to tokenized-type5 clones are highlighted, indicating different criteria for each clone type. The similarity percentage values for tokenized-type2 clones to tokenized-type5 clones are likely to vary if the set range for these types changes. The value range for tokenized-type5 is much greater and the lower bound is much more lower compared to the rest. This is done so to (hopefully) capture any less noticeable clones within the specifications that we may have missed.

In contrast to the code clones identified in the research paper [6], our focus will be on action clones, which we believed to be most closely related to the core logic of the TLA+ language and the definition of code clones employed in this research. We found a few notable findings. Despite variations in the ratio between the numbers of different clone types, we observed a consistent increasing trend in the number of clones from type-1 to type-3 (from [6] paper), just like tokenized-type1 to tokenized-type3 (from this research).

4.3 Evaluation: Project Overview

Throughout the research process, thorough validation and testing procedures were carefully conducted. These procedures aimed to verify the accuracy of both the code and methodology applied to each functional requirement across all system components as shown in the architecture diagram 1. The concept of 'Preprocessor', 'Tokenizer', and 'Clone Detector' components is inspired by the Event-B specification paper [6]. Additional components such as 'Renaming', 'Duplication Removal', 'Code Analysis', 'Clone Analysis', 'Visualization', and 'Evaluation' have been included to adapt to our file storage approach and other project requirements. For instance, the 'Visualization' component handles the rendering of results for evaluation. Some components compromise more than one code file from the code base.

As mentioned in Section 3.2, functional requirements with verification method of 'testing' are validated separately. Table 10outlines the validation process for two functional requirements of the system. 'Requirement ID' and 'Requirement Description' columns specify the identification and description of each requirement, respectively. 'Test case number' and 'Test case description' columns provide details of individual test cases. 'Input' and 'Expected Output' columns describe the input provided and the expected output for each test case. 'Actual Output' column records the output generated by the system during testing. Finally, the 'Result' column indicates whether each test case passed or failed based on a comparison between the 'Actual Output' and the 'Expected Output'. A total of 26

Figure 10: Validation for parser component requirements

Requirement ID	Requirement Description	Test case number	Test case description	Input	Expected Output	Actual Output	Result: Pass/ Fail
R1	Should rename all filenames in the directory into ID number	1	files with different file formats	test_files	all the files in test_files are renamed in the form of ID	all the files are renamed in ID	Pass
R2	Should log the ID number with its original filenames in a text file	1	logs all changes made on file names	test_files	new text file is produced containing a list of orginal filenames corresponding to the updated ID name	new text file containing a list of orginal filenames corresponding to the updated ID name is created	Pass

requirements are validated, with some functional requirements may require more than one test case, which will be tested with different inputs and expected outputs. Full validation table can be read in Table 8.

The time taken to process a total of 1670 files is estimated around 330000 seconds (3 days, 19 hours and 40 minutes). This involves all the components in the architecture diagram (Figure 1), where 'Clone Detector' took the majority of the total time. The component consists of clone detection of individual tokenized files, between two parsed (untokenzied) files, and between two tokenized files. Thus, it requires iterating through six large directories. The extended execution time isn't attributed to the program's complexity but rather to the number of steps involved, particularly the iterations in this scenario.

Several strategies have been considered to decrease the execution time, one of which involves condensing two comparisons into a single comparison. Single comparison would reduce the number of comparison. However, due to the method we have applied using the difflib.SequenceMatcher class as explained in Section 3, single comparison is not ideal. This is because it would only yield one of the results from the comparison between "tide" and "diet", and the comparison between "diet" and "tide", but not both. It's important to note that the outcomes of these two comparisons differ (also explained in Section 3), potentially leading to the loss of valuable results and insights.

Another strategy involves the use of quick_ratio() and real_quick_ratio(), which returns the upper bound of ratio(). The function quick_ratio() is relatively fast approximation of the ratio(), and real_quick_ratio() which is even faster. However, it's important to note that these methods may compromise the accuracy of clone detection, which is a trade-off we aim to avoid.

An alternative approach to reduce execution time would require adopting a completely new methodology for clone detection. Rather than sequential detection, we could try context-based detection method. This approach would allow for the implementation of the single comparison idea, where two comparisons yield the same result. Another approach involves adopting methodologies utilized in related research papers such as those in [6] and [18].

5 Threats to Validity

Our research introduces a unique perspective by establishing a corpus of TLA+ specifications, categorized into "small" and "large" from the main dataset. However, the segmentation of the dataset into "small" and "large" poses a potential challenge to conclusion validity due to inherent heterogeneity within the groups. Hence, we also conducted our analysis on the main dataset to assess if the results were comparable.

The use of difflib.SequenceMatcher class might not be the best approach as we later discovered the issue of different variable naming could lead to unequal number of parsed-type1 and tokenized-type1 as shown in Listing 12. Thus, potentially poses doubt in statistic validity.

Decisions regarding which syntax to employ as the core logic of the code may vary, potentially yielding different results and posing a threat to conclusion validity. In adapting and refining the definition of code clones for TLA+, decisions on measurement criteria and matching levels introduce variability; adjusting these decisions could lead to disparate outcomes. For example, our identification of type-5 clones, based on a similarity percentage of 20%, may result in fewer clone-pairs and uncover more useful information when using a higher similarity threshold of 40%.

With a total of 1670 TLA+ files in our corpus, concerns arise about the adequacy of the sample size for study, posing a threat to external validity in terms of result generalizability. Moreover, the limited availability of projects from diverse fields raises concerns about the adequacy of sample variety for study. Nonetheless, we contend that compiling and maintaining a well-defined corpus of TLA+ programs stay a valuable endeavour for future research.

The TLA+ specifications we gathered consist of multiple domains, such as software testing, distributed systems and concurrency control. This might possibly decrease the number of code clones and raise questions about validity. There's a chance that specifications from the same domain could yield more code clones than those from different domains. This could serve as a potential idea for future work, where we categorize specifications based on domain rather than size to explore this further.

6 Summary and Future Work

In this chapter, we gathered and summarised the key contents on our research, and explore potential future work that can be done using the resources and insights gained from this research.

6.1 Summary

In this report, we have discussed topics in TLA+ formal language, code clones and specifications, along with related work. We then introduced new definition of code clones that was used with explanation as to why we chose to create one. We then outlined the system architecture design and functional requirements, which were supported with detailed explanation. The algorithm for each components in the architecture design are interpreted. With the program and specification corpus we have gathered, we run the program to create useful statistics and graphs for analysis and evaluation. Then, we have critically evaluate on the project nature and potential threats to validity.

Our analysis shows that less than 45% of clone types across files consist of tokenized-type3, tokenized-type2, tokenized-type1, and parsed-type1 clones, with a similarity exceeding 81%. High-frequency code clones often represent basic TLA+ syntax elements, providing insights into their purpose and usage patterns. Detecting clones within files reveals no significant correlation between clone pairs and line count. Differences in naming conventions during tokenization and sequential comparison approaches leads to varying statistics for parsed-type1 and tokenized-type1 clones. The low occurrence of parsed-type1 clones in TLA+ specifications may suggest deficiencies in the specifications themselves. Additionally, there's no consistent correlation between code lines and clone pair presence. Tokenized-type5 clones are the most common across datasets, while tokenized-type1 clones are the least common.

In conclusion, our research has introduced novel perspectives and methodologies to the realms of TLA+, software engineering, and clone specifications. By proposing new definitions of clone types and devising an algorithm for identifying code clones within TLA+ specifications, we have contributed to the advancement of the field. The analysis of the TLA+ specification corpus has yielded valuable insights into the frequency and characteristics of specification clones, guiding the way for future research and developments in areas such as specification clones, software engineering, TLA+, code clones, and code modularization.

6.2 Future Work

Many future projects can be undertaken using the ideas and insights gained from this research. Firstly, there is an opportunity for optimization in the code for clone detection, with a focus on reducing processing time and complexity. Improvements in this area would not only reduce processing time but also allow code reusability in software engineering. An additional feature can be implemented where the code accepts two large datasets to find any code clones between the datasets, instead of just comparing files within a single directory.

An alternative approach to clone detection, instead of using the SequenceMatcher class which is a sequential-based method, one could try a context-based detection. Additionally, adapting existing methodologies, such as those from research papers like [6] and [18], is another viable option. Moreover, categorizing specifications based on domain rather than size may provide different insights into the distribution and patterns of code clones.

The use of Roy et al.'s [15] definition of clones could reveal new insights on TLA+ specification clones, or altering the range of each clone types definition used here could potentially give new interesting insights. Expanding the core logic in clone detection could lead to new discoveries and provide a deeper understanding of code similarities.

Additionally, exploring clone detection along with PlusCal presents a promising area for future research. Understanding clone detection in PlusCal could give us insights into how formal specifications become actual code, improving our understanding of software development. In summary, our current work lays the foundation for future studies aimed at improving and broadening clone detection in formal methods.

7 Reflection

In this chapter, we will discuss our reflection and evaluation on the process of working on this project. We will outline what went well, what went wrong, and any improvements that can be made.

Throughout the course of my project on “Clone Detection and Evaluation of Specification Clones within the TLA+ Formal Language,” I experience a significant learning journey and encountered various challenges along the way. I had the opportunity to delve into diverse research papers, including topics like formal methods, specification clones, TLA+, Event-B, and modularization, which expanded my understanding of the field of software engineering field and broadened my perspective.

From this research, the completion of this project would be one of the many accomplishments I have achieved. Other achievements include the completion of code for all components as described on system architecture, fulfilling the functional requirements of the components, and conducting a comprehensive analysis and evaluation of the result of clone detection as well as on the overall project evaluation. Personal achievements I made include effective planning, meeting short-term and long-term deadlines, and maintaining clear communication with my supervisor. The valuable advice, support and feedback from my supervisor played a major role in all the accomplishments I achieved in this project.

During the research process, several challenges I have encountered were not major but still valuable. One includes having to learn and understand the TLA+ language, which differs considerably from other programming languages. Additionally, I faced some obstacles during the project, including information overload from the abundance of research papers, which sometimes led to confusion. Moreover, not being able to consistently update the functional requirements after component revisions resulted in additional work and time spent verifying compliance for each component.

If I were to undertake this project again or engage in a similar project, these are things that I would do differently. First, I would create a document to systematically list and summarize all relevant resources for the project. This would help me remember what each research paper is about, rather than reading the paper again later in the future. Additionally, I would organize files effectively and provide clearer file names to avoid potential confusion in the future. I would also leverage GitHub for version control while writing code. I would ensure that requirements are updated promptly after every revision made on the components. Keeping a well-written journal documenting project progress and completed tasks would also be essential. Lastly, creating a list of tasks to resume after every break from the project would help maintain momentum and productivity.

References

[1] H. Wayne, "The business case for formal methods," 2020. `https://www.hillelwayne.com/post/business-case-formal-methods`.

[2] P. S. Foundation, "difflib — helpers for computing deltas," 2024. `https://docs.python.org/3/library/difflib.html#difflib.SequenceMatcher.ratio`.

[3] L. Lamport, "The tla+ home page," 2022. `https://lamport.azurewebsites.net/tla/tla.html`.

[4] X. Gu, W. Cao, Y. Zhu, X. Song, Y. Huang, and X. Ma, "Compositional model checking of consensus protocols specified in tla+ via interaction-preserving abstraction," 2022.

[5] C. Denis, D. Damien, L. Leslie, M. Stephan, R. Daniel, and V. Hernan, "Tla+ proofs." `https://lamport.azurewebsites.net/pubs/tlaps.pdf`.

[6] M. Farrell, R. Monahan, and J. F. Power, "Specification Clones: An Empirical Study of the Structure of Event-B Specifications," in *Software Engineering and Formal Methods*, pp. 152–167, Springer, 2017.

[7] L. Lamport, "Tla+ tools," 2022. `https://lamport.azurewebsites.net/tla/tools.html`.

[8] L. Lamport, "Learning tla+," 2022. `https://lamport.azurewebsites.net/tla/learning.html`.

[9] H. Wayne, "Learn tla+," 2022. `https://learntla.com/#learn-tla`.

[10] L. Lamport, "A high-level view of tla+," 2021. `https://lamport.azurewebsites.net/tla/high-level-view.html`.

[11] L. Lamport, "Industrial use of tla+," 2019. `https://lamport.azurewebsites.net/tla/industrial-use.html`.

[12] C. Newcombe, "Why amazon chose tla + ," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pp. 25–39, 2014.

[13] J. Kukovec, T.-H. Tran, and I. Konnov, "Extracting symbolic transitions from tla+ specifications," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pp. 89–104, Springer International Publishing, 2018.

[14] S. Resch and M. Paulitsch, "Using tla+ in the development of a safety-critical fault-tolerant middleware," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 146–152, 2017.

[15] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, IEEE, Feb. 2014.

[16] V. S. Alagar and K. Periyasamy, *The Role of Specification*, pp. 3–22. Springer London, 2011. `https://doi.org/10.1007/978-0-85729-277-3_1`.

[17] N. Nissanke, *Introduction*. Springer London, 1999. `https://doi.org/10.1007/978-1-4471-0791-0_1`.

[18] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, and B. Baudry, "Puzzle: A tool for analyzing and extracting specification clones in dsls," in *Software Reuse: Bridging with Social-Awareness*, pp. 393–396, 2016.

[19] C. Domann, E. Juergens, and J. Streit, "The curse of copy&paste — cloning in requirements specifications," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 443–446, 2009.

[20] C. Morgan, K. Robinson, and P. Gardiner, "On the refinement calculus," Tech. Rep. PRG70, OUCL, October 1988.

[21] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *2009 IEEE 31st International Conference on Software Engineering*, 2009.

[22] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit, "Can clone detection support quality assessments of requirements specifications?," p. 79–88, 2010. https://doi.org/10.1145/1810295.1810308, booktitle = Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2.

[23] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification patterns for robotic missions," 2019.

A Appendix

A.1 Specification sources

TLA+ specification resource links:

- tlaplus
- gterzian/ArchetypeConcurrencyFix.tla
- hwayne/learntla-v2
- dfinity/formal-models
- tlaplus/DrTLAPlus
- tlaplus/vscode-tlaplus
- tlaplus/CommunityModules
- tlaplus/tlapm
- tlaplus/azure-cosmos-tla
- Alexander-N/tla-specs
- tlaplus/tlaplus
- kuujo/just-in-time-paxos
- atomix/atomix-tlaplus
- tendermint/spec
- pingcap/tla-plus
- tlaplus/tlaplus-standard
- tlaplus/foundation

- tlaplus/conf
- tlaplus/ConcurrentSCC
- tlaplus/lecture
- tlaplus/www
- tlaplus/tlapm_alternative_parser_experiment

A.2 Graphs & Code

Figure 11: Distribution of clones with tokenized Type-5 clone (whole dataset)

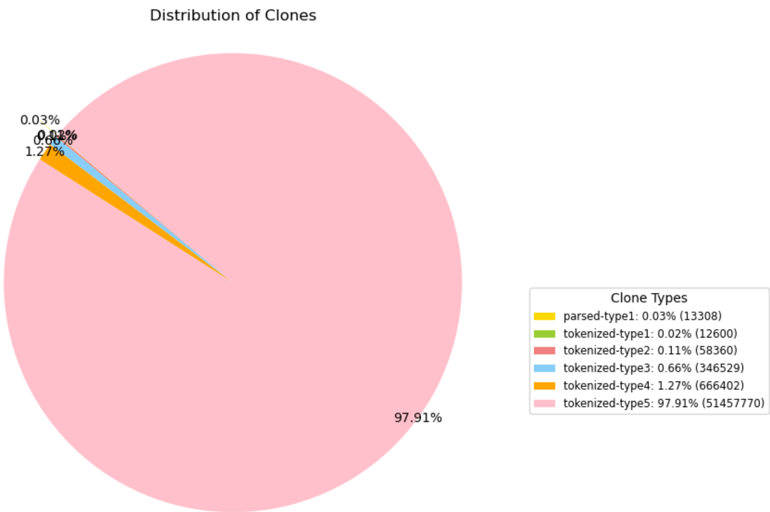


Figure 12: Distribution of clones with tokenized Type-5 clone (small dataset)

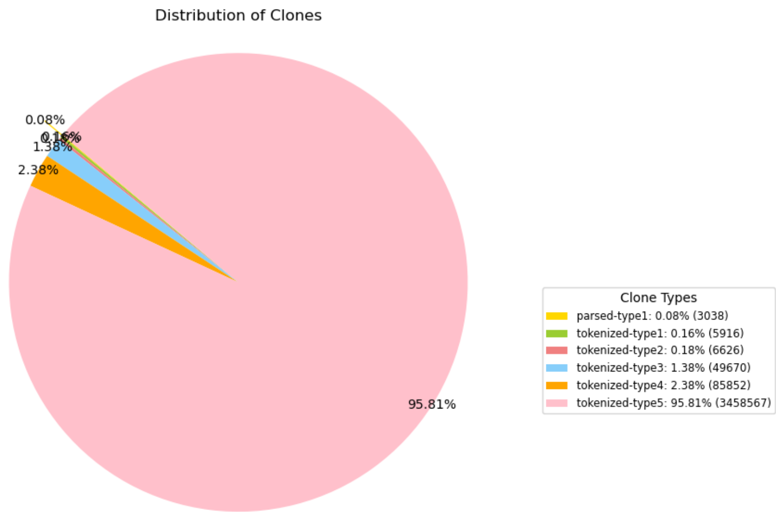
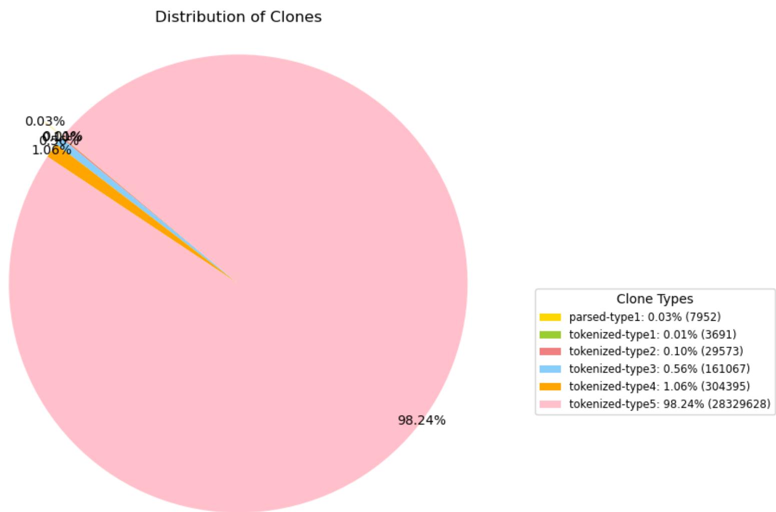


Figure 13: Distribution of clones with tokenized Type-5 clone (large dataset)



Componenet	ID Label
Renaming	R
Parser	P
Duplication Removal	DR
Tokenizer	T
Visualisation	V

Table 4: Label table for the following requirement tables

Requirement ID	Description	Verification Method	Verification Explanation	File Name
R1	Should rename all filenames in the directory into ID number	testing	"required to check if the file names have changed in ID format for all files in the directory; rename files that may be copies of an existing file or files that may have the same file names"	renaming.py
R2	Should log the ID number with its original filenames in a text file	testing	to allow backtracking	
P1	Should only read files with the extension of tla	by design	not require to test as it is set to only read the specific extension in code	parser.py
P2	Should parse files contained in the input directory	by design	not require to test as it is set to read files in a directory instructed in the code	
P3	Should keep lines containing "=="	testing	to make sure all lines containing "==" are kept and all remaining lines are removed	
P4	Should combine the next line with the current line if next line starts with (4 or more empty spaces)	testing	to make indented code pieces into a single line	
P5	Should remove lines that do not start with a character, integer or whitespace	testing	lines starting with * are used to write comments, and —- are used for code seperation and do not contribute to code processing	
P6	Should remove the last lines containing "===="	testing	it tells the program that it is the end of the code and doesn't contribute to the main code function	
P7	Should ignore lines after the line containing "BEGIN TRANSLATION"	testing	this is to prevent processing the plusCal part of code	
P8	Should write the processed code into a new tla file	testing	to check if the desired preprocessed code is outputed corretly into a new tla file	
P9	Should store the new file in the output directory	by design	not require to test as it is designed in the code	
DR1	Should only read files with the extension of tla	by design	not require to test as it is set to only read the specific extension in code	remove_dup.py
DR2	Should compare the content of all the files in the given directory	by design	not require to test as it is designed in the code	
DR3	Should find duplicated files and remove one of the files if found	testing	to check if it works as desired when there's no/ one/ more than one duplicated file	
DR4	Should log the files removed in a new text file	testing	to check if it logs the correct filenames that are being removed	
T1	Should only read files with the extension of tla	by design	not require to test as it is set to only read the specific extension in code	tokenizer.py
T2	Should tokenize files contained in the input directory (output directory of preprocessor)	by design	not require to test as it is designed in the code	
T3	Should display the type of each token	by design	not require to test as it is designed in the code	
T4	Should remove whitespace	testing	to check if all whitespaces are being removed	
T5	Should anonymize variable names	testing	to check if all variable names are correctly anonymized and the same variable name should be given the same anaoymized name	
T6	Should rename different variable with the different label	testing	to check that different lables are given to different variables	
T7	Should write the tokenized code into a new tla file	by design	not require to test as it is designed in the code	
T8	Should store the new file in the output directory	by design	not require to test as it is designed in the code	
CD1	Should only read files with the extension of tla	by design	not require to test as it is set to only read the specific extension in code	clone_detector.py
CD2	Should process files in the correct directory	by design	not require to test as it is designed in the code	
CD3	Should find difference in code using difflib.SequenceMatcher and give ratio	testing	to check the library is working as desired	
CD4	Should identify code types using ratio (Type-1 when ratio = 1 ; Type-2 when 0.9 <ratio <1 ; Type-3 when 0.8 <ratio <= 0.9 ; Type-4 when 0.7 <ratio <= 0.8 ; Type-5 when 0.2 <ratio <= 0.7)	testing	to check the clones are correctly labelled	
CD5	Should find clones, line number of the clone pairs, similarity and clone type in a single file	testing	to check the clones are correctly detected from a single file	
CD6	Should create csv file that outputs the file name, line number of the clone pairs, similarity, and clone type within each individual file [categories: file name, line number, similarity, clone type	by design	not require to test as it is set to only read the specific extension in code	
CD7	Should create csv file that outputs the file name, total number of clone pairs, number of each clone type in that file of all the files in the directory [categories: file name, number of clone pairs, number of Type-1 clone, number of Type-2 clone, number of Type-3 clone, number of Type-4, number of Type-5]	by design	not require to test as it is set to only read the specific extension in code	
CD8	Should find clones, line number of the clone pairs, similarity, clone type and the clone line code between two files in the given directory (parsed, "small_files", "large_files" directories)	testing	to check the clones are correctly detected between two files from correct directory	
CD9	Should find clones, line number of the clone pairs, similarity and clone type between two files in the given directory (tokenized, "small_files", "large_files" directory)	testing	to check the clones are correctly detected between two files from correct directory	
CD10	Should create csv file that output the name of the first file (file1), name of the second file being compared to (file2), line number of clone found in file1, line number of clone found in file2, similarity and clone type [categories: file name (1), file name (2), line number (1), line number (2), similarity, clone type]	by design	not require to test as it is set to only read the specific extension in code	
CD11	Should create a .txt file that output the number of clone pairs, number of Type-1 clone, number of Type-2 clone, number of Type-3 clone exist in the given directory [categories: number of clone pairs, number of Type-1 clone, number of Type-2 clone, number of Type-3 clone, Type-4 clone, Type-5 clone]	by design	not require to test as it is set to only read the specific extension in code	
CD12	Should display the similarity in percentage	by design	not require to test as it is set to only read the specific extension in code	
V1	Should read data from the correct files and produce a pie chart using the value of Type-1 clone produced from clone detection of output_file (files that are parsed), and values of all the clone types produced from clone detection of tokenized_files (files that are tokenized)	testing	to check the code reads the correct file and data and the graph is outputted as desired	clone_type_pie.py
V2	Should read data from the correct file and produce a bar graph showing the average, maximum and minimum of each clone type, output png file of the graph	testing	to check the code reads the correct file and data and the graph is outputted as desired	clone%_graph.py
V3	Should read data from the correct file, find and count the maximum number of clone type of each file in the directory and plot it on a bar graph, output png file of the graph (Should ignore the files where there are two or more clone types of the same highest number in a single file)	testing	to check the code reads the correct file and data and the graph is outputted as desired	indi_most_clone_types.py
V4	Should read data from the correct files; produce a pie chart using the values of all the clone types (excluding type-5 clone) produced from clone detection of tokenized directory of individual files; display legend (clone type, percentage, number of clones); output png file of the graph	testing	to check the code reads the correct file and data and the graph is outputted as desired	indi_most_clone_type_pie_no_ty5.py
V5	Should create a CSV file that counts the number of code lines for all files in the specified directory, calculating the mean number of code lines as an integer, using this mean as a threshold to categorize files into 'small' if their code lines are below the mean and 'large' if equal to or exceeding the mean, and subsequently copying the files into new directories named 'Small_files' and 'Large_files', respectively.	testing	to check the code reads the correct file and data and the graph is outputted as desired	count_line.py
V6	Should read data from the correct files; produce a pie chart using the value of Type-1 clone produced from clone detection from parsed directory, and values of clone types 1, 2, 3, and 4 produced from clone detection of tokenized directory; display legend (clone type, percentage, number of clones); output png file of the graph	testing	to check the code reads the correct file and data and the graph is outputted as desired	clone_type_pie_no_ty5.py
V7	Should find Type-1 clones from parsed file, count them; output result in csv file along with the file names containing the clone	testing	to check the code reads the correct file and data and the graph is outputted as desired	clone_line_count.py
V8	Should read data from the correct CSV files (tokenized_line_count and indi_file_statistics), sort the line count from tokenized_line_count in descending order, merge two tables together in the correct order of title (File Name, Line Count, Clone Pairs, Type-1 clones, Type-2 clones, Type-3 clones, Type-4 clones, Type-5 clones), and fill in the value of 0 for any missing data/cells.	testing	to check the code reads the correct file and data and the graph is outputted as desired	csv_merging.py

Table 5: Complete list of functional requirements

Requirement ID	Description
R1	Should rename all filenames in the directory into ID number
R2	Should log the ID number with its original filenames in a text file
P1	Should only read files with the extension of tla
P2	Should parse files contained in the input directory
P3	Should keep lines containing “==”
P4	Should combine the next line with the current line if next line starts with (4 or more empty spaces)
P5	Should remove lines that do not start with a character, integer or whitespace
P6	Should remove the last lines containing “====”
P7	Should ignore lines after the line containing “BEGIN TRANSLATION”
P8	Should write the processed code into a new tla file
P9	Should store the new file in the output directory
DR1	Should only read files with the extension of tla
DR2	Should compare the content of all the files in the given directory
DR3	Should find duplicated files and remove one of the files if found
DR4	Should log the files removed in a new text file
T1	Should only read files with the extension of tla
T2	Should tokenize files contained in the input directory (output directory of preprocessor)
T3	Should display the type of each token
T4	Should remove whitespace
T5	Should anonymize variable names
T6	Should rename different variable with the different label
T7	Should write the tokenized code into a new tla file
T8	Should store the new file in the output directory
CD1	Should only read files with the extension of tla
CD2	Should process files in the correct directory
CD3	Should find difference in code using difflib.SequenceMatcher and give ratio
CD4	Should identify code types using ratio (Type-1 when ratio = 1 ; Type-2 when 0.9 <ratio <1 ; Type-3 when 0.8 <ratio <= 0.9 ; Type-4 when 0.7 <ratio <= 0.8 ; Type-5 when 0.2 <ratio <= 0.7)
CD5	Should find clones, line number of the clone pairs, similarity and clone type in a single file
CD6	Should create csv file that outputs the file name, line number of the clone pairs, similarity, and clone type within each individual file [categories: file name, line number, similarity, clone type
CD7	Should create csv file that outputs the file name, total number of clone pairs, number of each clone type in that file of all the files in the directory [categories: file name, number of clone pairs, number of Type-1 clone, number of Type-2 clone, number of Type-3 clone, number of Type-4, number of Type-5]
CD8	Should find clones, line number of the clone pairs, similarity, clone type and the clone line code between two files in the given directory (parsed, ”small_files”, ”large_files” directories)
CD9	Should find clones, line number of the clone pairs, similarity and clone type between two files in the given directory (tokenized, ”small_files”, ”large_files” directory)
CD10	Should create csv file that output the name of the first file (file1), name of the second file being compared to (file2), line number of clone found in file1, line number of clone found in file2, similarity and clone type [categories: file name (1), file name (2), line number (1), line number (2), similarity, clone type]
CD11	Should create a .txt file that output the number of clone pairs, number of Type-1 clone, number of Type-2 clone, number of Type-3 clone exist in the given directory [categories: number of clone pairs, number of Type-1 clone, number of Type-2 clone, number of Type-3 clone, Type-4 clone, Type-5 clone]
CD12	Should display the similarity in percentage
V1	Should read data from the correct files and produce a pie chart using the value of Type-1 clone produced from clone detection of output_file (files that are parsed), and values of all the clone types produced from clone detection of tokenized_files (files that are tokenized)
V2	Should read data from the correct file and produce a bar graph showing the average, maximum and minimum of each clone type, output png file of the graph
V3	Should read data from the correct file, find and count the maximum number of clone type of each file in the directory and plot it on a bar graph, output png file of the graph (Should ignore the files where there are two or more clone types of the same highest number in a single file)
V4	Should read data from the correct files; produce a pie chart using the values of all the clone types (excluding type-5 clone) produced from clone detection of tokenized directory of individual files; display legend (clone type, percentage, number of clones); output png file of the graph
V5	Should create a CSV file that counts the number of code lines for all files in the specified directory, calculating the mean number of code lines as an integer, using this mean as a threshold to categorize files into ’small’ if their code lines are below the mean and ’large’ if equal to or exceeding the mean, and subsequently copying the files into new directories named ’Small_files’ and ’Large_files’, respectively.
V6	Should read data from the correct files; produce a pie chart using the value of Type-1 clone produced from clone detection from parsed directory, and values of clone types 1, 2, 3, and 4 produced from clone detection of tokenized directory; display legend (clone type, percentage, number of clones); output png file of the graph
V7	Should find Type-1 clones from parsed file, count them; output result in csv file along with the file names containing the clone
V8	Should read data from the correct CSV files (tokenized_line_count and indi_file_statistics), sort the line count from tokenized_line_count in descending order, merge two tables together in the correct order of title (File Name, Line Count, Clone Pairs, Type-1 clones, Type-2 clones, Type-3 clones, Type-4 clones, Type-5 clones), and fill in the value of 0 for any missing data/cells.

Table 6: Requirement ID and description

Requirement ID	Verification Method	Verification Explanation	File Name
R1	testing	"required to check if the file names have changed in ID format for all files in the directory; rename files that may be copies of an existing file or files that may have the same file names"	renaming.py
R2	testing	to allow backtracking	
P1	by design	not require to test as it is set to only read the specific extension in code	parser.py
P2	by design	not require to test as it is set to read files in a directory instructed in the code	
P3	testing	to make sure all lines containing "==" are kept and all remaining lines are removed	
P4	testing	to make indented code pieces into a single line	
P5	testing	lines starting with * are used to write comments, and —- are used for code seperation and do not contribute to code processing	
P6	testing	it tells the program that it is the end of the code and doesn't contribute to the main code function	
P7	testing	this is to prevent processing the plusCal part of code	
P8	testing	to check if the desired preprocessed code is outputed corretly into a new tla file	
P9	by design	not require to test as it is designed in the code	
DR1	by design	not require to test as it is set to only read the specific extension in code	remove_dup.py
DR2	by design	not require to test as it is designed in the code	
DR3	testing	to check if it works as desired when there's no/ one/ more than one duplicated file	
DR4	testing	to check if it logs the correct filenames that are being removed	
T1	by design	not require to test as it is set to only read the specific extension in code	tokenizer.py
T2	by design	not require to test as it is designed in the code	
T3	by design	not require to test as it is designed in the code	
T4	testing	to check if all whitespaces are being removed	
T5	testing	to check if all variable names are correctly anonymized and the same variable name should be given the same anaoymized name	
T6	testing	to check that different lables are given to different variables	
T7	by design	not require to test as it is designed in the code	
T8	by design	not require to test as it is designed in the code	
CD1	by design	not require to test as it is set to only read the specific extension in code	clone_detector.py
CD2	by design	not require to test as it is designed in the code	
CD3	testing	to check the library is working as desired	
CD4	testing	to check the clones are correctly labelled	
CD5	testing	to check the clones are correctly detected from a single file	
CD6	by design	not require to test as it is set to only read the specific extension in code	
CD7	by design	not require to test as it is set to only read the specific extension in code	
CD8	testing	to check the clones are correctly detected between two files from correct directory	
CD9	testing	to check the clones are correctly detected between two files from correct directory	
CD10	by design	not require to test as it is set to only read the specific extension in code	
CD11	by design	not require to test as it is set to only read the specific extension in code	
CD12	by design	not require to test as it is set to only read the specific extension in code	
V1	testing	to check the code reads the correct file and data and the graph is outputted as desired	clone_type_pie.py
V2	testing	to check the code reads the correct file and data and the graph is outputted as desired	clone%_graph.py
V3	testing	to check the code reads the correct file and data and the graph is outputted as desired	indi_most_clone_types.py
V4	testing	to check the code reads the correct file and data and the graph is outputted as desired	indi_most_clone_type_pie_no_ty5.py
V5	testing	to check the code reads the correct file and data and the graph is outputted as desired	count_line.py
V6	testing	to check the code reads the correct file and data and the graph is outputted as desired	clone_type_pie_no_ty5.py
V7	testing	to check the code reads the correct file and data and the graph is outputted as desired	clone_line_count.py
V8	testing	to check the code reads the correct file and data and the graph is outputted as desired	csv_merging.py

Table 7: Requirement and Verification

Figure 14: Clone types in each file

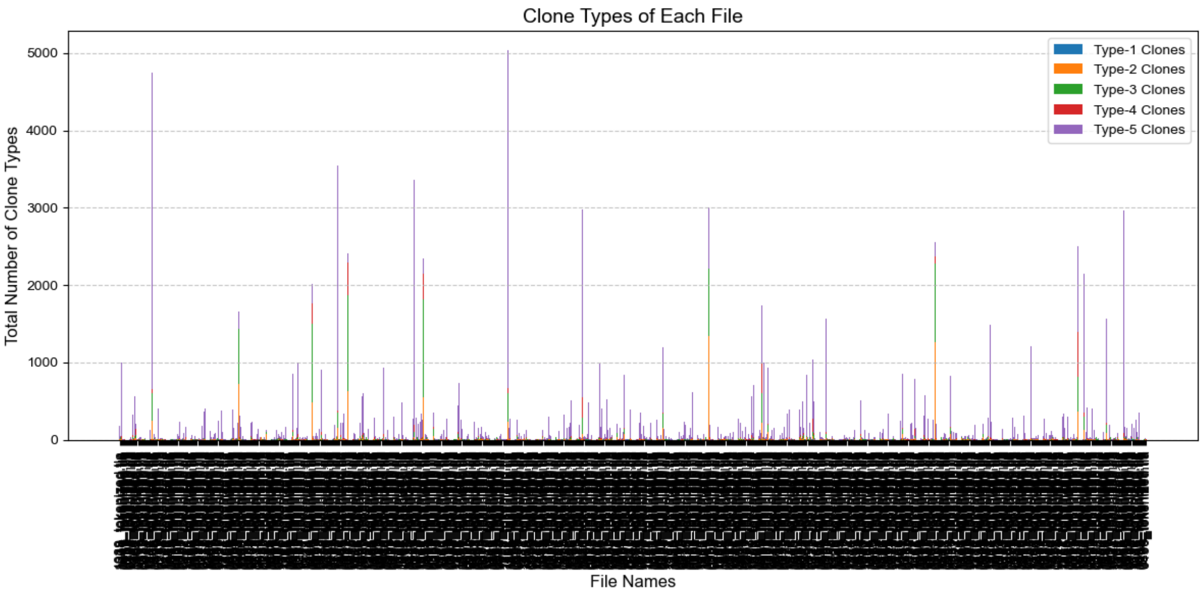
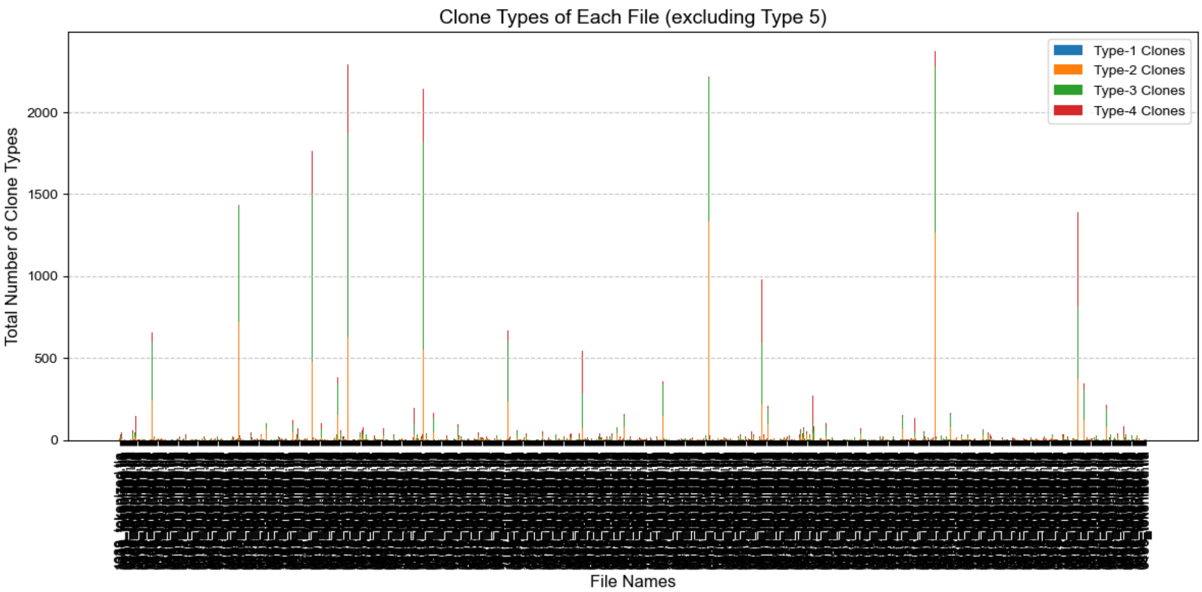


Figure 15: Clone types in each file without Type-5 clones



Requirement ID	Requirement Description	Test case number	Test case description	Input	Expected Output	Actual Output	Result: Pass/ Fail
R1	Should rename all filenames in the directory into ID number	1	files with different file formats	test_files	all the files in test_files are renamed in the form of ID	all the files are renamed in ID	Pass
R2	Should log the ID number with its original filenames in a text file	1	logs all changes made on file names	test_files	new text file is produced containing a list of orginal filenames corresponding to the updated ID name	new text file containing a list of orginal filenames corresponding to the updated ID name is created	Pass
P3	Should keep lines containing “==”	1	no ”==” in file	”ASSUME(ChiSquare(A, A, ””0.001””)) ASSUME(ChiSquare(A, A, ””0.01””))”	(empty)	(empty)	Pass
		2	contains one ”==” in file	”EXTENDS Statistics ASSUME LET T == INSTANCE TLC IN T!PrintT(””StatisticsTests””)”	ASSUME LET T == INSTANCE TLC IN T!PrintT(”StatisticsTests”)	ASSUME LET T == INSTANCE TLC IN T!PrintT(”StatisticsTests”)	Pass
		3	contains more than one ”==” in file	”EXTENDS Statistics ASSUME LET T == INSTANCE TLC IN T!PrintT(””StatisticsTests””) A == [i \in 1..6 —->i]”	”ASSUME LET T == INSTANCE TLC IN T!PrintT(””StatisticsTests””) A == [i \in 1..6 —->i]”	”ASSUME LET T == INSTANCE TLC IN T!PrintT(””StatisticsTests””) A == [i \in 1..6 —->i]”	Pass
P4	Should combine the next line with the current line if next line starts with (4 or more empty spaces)	1	no ” ” in file	”ASSUME(ToSet(<<>>) = {}) ASSUME(ToSet(<<1>>) = {1})”	”ASSUME(ToSet(<<>>) = {}) ASSUME(ToSet(<<1>>) = {1})”	”ASSUME(ToSet(<<>>) = {}) ASSUME(ToSet(<<1>>) = {1})”	Pass
		2	next line starts with 4 or more empty spaces	”ASSUME(LET s == {””t””,””l””,””a””,””p””,””l””,””u””,””s””} seq == SetToSeq(s) IN Len(seq) = Cardinality(s) /\ToSet(seq) = s)”	ASSUME(LET s == {”t”,”l”,”a”,”p”,”l”,”u”,”s”} seq == SetToSeq(s) IN Len(seq) = Cardinality(s) /\ToSet(seq) = s)	ASSUME(LET s == {”t”,”l”,”a”,”p”,”l”,”u”,”s”} seq == SetToSeq(s) IN Len(seq) = Cardinality(s) /\ToSet(seq) = s)	Pass
P5	Should remove lines that do not start with a character, integer or whitespace	1	does not contain any symbols other than character, integer or whitespace	”ASSUME(ToSet(<<>>) = {}) ASSUME(ToSet(<<1>>) = {1})”	”ASSUME(ToSet(<<>>) = {}) ASSUME(ToSet(<<1>>) = {1})”	”ASSUME(ToSet(<<>>) = {}) ASSUME(ToSet(<<1>>) = {1})”	Pass
		2	contains any symbols other than character, integer or whitespace	”ASSUME(Reverse(<<””a””,””a””,””b””>>) = <<””b””,””a””,””a””>>) ————————————————”	”ASSUME(Reverse(<<””a””,””a””,””b””>>) = <<””b””,””a””,””a””>>) ”	”ASSUME(Reverse(<<””a””,””a””,””b””>>) = <<””b””,””a””,””a””>>) ”	Pass
P6	Should remove the last lines containing “====”	1	does not contain ”====”	ASSUME AssertEq(RemoveFirstMatch(<<1,2,1,2>>, LAMBDA e: e = 2), <<1,1,2>>)	ASSUME AssertEq(RemoveFirstMatch(<<1,2,1,2>>, LAMBDA e: e = 2), <<1,1,2>>)	ASSUME AssertEq(RemoveFirstMatch(<<1,2,1,2>>, LAMBDA e: e = 2), <<1,1,2>>)	Pass
		2	contains ”====”	”ASSUME AssertEq(RemoveFirstMatch(<<1,2,1,2>>, LAMBDA e: e = 2), <<1,1,2>>) ==”	”ASSUME AssertEq(RemoveFirstMatch(<<1,2,1,2>>, LAMBDA e: e = 2), <<1,1,2>>) ”	”ASSUME AssertEq(RemoveFirstMatch(<<1,2,1,2>>, LAMBDA e: e = 2), <<1,1,2>>) ”	Pass
P7	Should ignore lines after the line containing “BEGIN TRANSLATION”	1	does not cotain line containing “BEGIN TRANSLATION”	”–algorithm MultiAssignment process Proc \in 1..3 variables A = [i \in 1..5 —->i] ; x = 0 ; begin a : A[1] := A[3] —— x := 7 —— A[3] := A[1] ; assert <<3 , 1>>= <<A[1], A[3]>>; b : assert <<3 , 1>>= <<A[1], A[3]>>; end process end algorithm”	”–algorithm MultiAssignment process Proc \in 1..3 variables A = [i \in 1..5 —->i] ; x = 0 ; begin a : A[1] := A[3] —— x := 7 —— A[3] := A[1] ; assert <<3 , 1>>= <<A[1], A[3]>>; b : assert <<3 , 1>>= <<A[1], A[3]>>; end process end algorithm”	”–algorithm MultiAssignment process Proc \in 1..3 variables A = [i \in 1..5 —->i] ; x = 0 ; begin a : A[1] := A[3] —— x := 7 —— A[3] := A[1] ; assert <<3 , 1>>= <<A[1], A[3]>>; b : assert <<3 , 1>>= <<A[1], A[3]>>; end process end algorithm”	Pass

Table 8: Validation table of functional requirements part 1

Requirement ID	Requirement Description	Test case number	Test case description	Input	Expected Output	Actual Output	Result: Pass/ Fail
		2	cotain line containing “BEGIN TRANSLATION”	”–algorithm MultiAssignment process Proc \in 1..3 variables A = [i \in 1..5 —->i] ; x = 0 ; begin a : A[1] := A[3] —— x := 7 —— A[3] := A[1] ; assert <<3 , 1>>= <<A[1], A[3]>>; b : assert <<3 , 1>>= <<A[1], A[3]>>; end process end algorithm ****) * BEGIN TRANSLATION - the hash of the PCal code: PCal-922bd40bbb66854d6ea0cf6dbef935cb VARIABLES pc, A, x vars == <<pc, A, x >>”	”–algorithm MultiAssignment process Proc \in 1..3 variables A = [i \in 1..5 —->i] ; x = 0 ; begin a : A[1] := A[3] —— x := 7 —— A[3] := A[1] ; assert <<3 , 1>>= <<A[1], A[3]>>; b : assert <<3 , 1>>= <<A[1], A[3]>>; end process end algorithm ****)”	”–algorithm MultiAssignment process Proc \in 1..3 variables A = [i \in 1..5 —->i] ; x = 0 ; begin a : A[1] := A[3] —— x := 7 —— A[3] := A[1] ; assert <<3 , 1>>= <<A[1], A[3]>>; b : assert <<3 , 1>>= <<A[1], A[3]>>; end process end algorithm ****)”	Pass
P8	Should write the processed code into a new tla file	1		”——————— MODULE scratch ——————— — EXTENDS Integers, TLC, Sequences e == 4 r == 2 q == e + r === * Modification History * Last modified Thu Oct 19 21:24:32 BST 2023 by sherylshunlin * Created Tue Oct 10 11:23:25 BST 2023 by sherylshunlin”	”e == 4 r == 2 q == e + r ”	”e == 4 r == 2 q == e + r ”	Pass
DR3	Should find duplicated files and remove one of the files if found	1	no duplicate files	file_1, file_2	file_1, file_2	file_1, file_2	Pass
		2	one duplicate file	file_1, file_2, file_3 (where file_1 and file_2 are exactly the same)	{file_1, file_3} or {file_2, file_3}	file_2, file_3	Pass
		3	more than one duplicate files	file_1, file_2, file_3(where all files are exactly the same)	{file_1} or {file_2} or {file_3}	file_1	Pass
DR4	Should log the files removed in a new text file	1		test_files	expect new text file created containing the list of filename that are removed	expect new text file containing the list of filename that are removed is created	Pass
T4	Should remove whitespace	1		Range(f) == { f[x] : x \in DOMAIN f }	$name1($name2)=={$name2[$name3]:$name3\inDOMAIN$name2}	$name1($name2)=={$name2[$name3]:$name3\inDOMAIN$name2}	Pass
T5	Should anonymize variable names	1		Range(f) == { f[x] : x \in DOMAIN f }	$name1($name2)=={$name2[$name3]:$name3\inDOMAIN$name2}	$name1($name2)=={$name2[$name3]:$name3\inDOMAIN$name2}	Pass
T6	Should rename different variable with the different label	1		”e == 4 r == 2 q == e + r t == e - r”	”$name1==# $name2==# $name3==$name1+$name2 $name4==$name1-$name2 ”	”$name1==# $name2==# $name3==$name1+$name2 $name4==$name1-$name2 ”	Pass
CD3	Should find difference in code using difflib.SequenceMatcher and give ratio	1	exactly the same code line (A from file_1, B from file_2)	”A: $name1==# B: $name1==#”	1	1	Pass
		2	slightly different code line (A from file_1, B from file_2)	”A: $name1==# B: $name2==#”	0.9	0.9	Pass
		3	very different code line (A from file_1, B from file_2)	”A: $name1==# B: $name3==$name1+$name2”	0.4375	0.4375	Pass
CD4	Should identify code types using ratio (Type-1 when ratio = 1 ; Type-2 when 0.9 <ratio <1 ; Type-3 when 0.8 <ratio <= 0.9 ; Type-4 when 0.7 <ratio <= 0.8 ; Type-5 when 0.2 <ratio <= 0.7)	1	code with the ratio of 1	”A: $name1==# B: $name1==# (ratio = 1)”	Type-1 Clone	Type-1 Clone	Pass
		2	code with the ratio of 0.91	”$name3==$name1+$name2 $name4==$name1-$name2 (ratio = 0.91)”	Type-2 Clone	Type-2 Clone	Pass
		3	code with the ratio of 0.90	”$name2==# $name1==# (ratio = 0.9)”	Type-3 Clone	Type-3 Clone	Pass
		4	code with the ratio of 0.75	”*name*3 ==name1+*name*2name5==<<*name*6,name7>>(ratio = 0.75)”	Type-4 Clone	Type-4 Clone	Pass
		5	code with the ratio of 0.437	”$name1==# $name3==$name1+$name2 (ratio = 0.437)”	Type-5 Clone	Type-5 Clone	Pass
CD5	Should find clones, line number of the clone pairs, similarity and clone type in a single file	1		”$name1==# $name2==# $name3==$name1+$name2”	File Name,Line Numbers,Similarity (%),Clone Type	” tokenized_test_files/scratch_same_tokenized.tla,1 - 2,90.00%,Type-3 Clone tokenized_test_files/scratch_same_tokenized.tla,1 - 3,43.75%,Type-5 Clone tokenized_test_files/scratch_same_tokenized.tla,2 - 3,43.75%,Type-5 Clone”	Pass
CD8	Should find clones, line number of the clone pairs, similarity, clone type and the clone line code between two files in the given directory (parsed, ”small_files”, ”large_files” directories)	1	for parsed directory	parsed_test_files	File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type,Code line	”File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type,Code line scratch_d2.tla,scratch_d.tla,1,1,100.00%,Type-1 Clone,””e == 4 ”” scratch_d2.tla,scratch_d.tla,2,2,100.00%,Type-1 Clone,””r == 2 ”” scratch_d2.tla,scratch_d.tla,3,3,100.00%,Type-1 Clone,””q == e + r ”” operator.tla,empty.tla,1,1,100.00%,Type-1 Clone,”” ”” scratch_same.tla,scratch.tla,1,1,100.00%,Type-1 Clone,””a == 4 ”” scratch_same.tla,scratch.tla,2,2,100.00%,Type-1 Clone,””b == 2 ”” scratch_same.tla,scratch.tla,3,3,100.00%,Type-1 Clone,””c == a + b ”””	Pass
		2	for ”small_files”	parsed_small_file	File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type,Code line	File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type,Code line	Pass

Table 9: Validation table of functional requirements part 2

Requirement ID	Requirement Description	Test case number	Test case description	Input	Expected Output	Actual Output	Result: Pass/ Fail
		3	for "large_files"	parsed_large_file	File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type,Code line	"File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type,Code line 0016.tla,0003.tla,14,51,100.00%,Type-1 Clone,""Spec == Init /\[][Next]_vars "" 0010.tla,0048.tla,5,1,100.00%,Type-1 Clone,""Range(f) == { f[x] : x \in DOMAIN f } """	Pass
CD9	Should find clones, line number of the clone pairs, similarity and clone type between two files in the given directory (tokenized, "small_files", "large_files" directory)	1	for tokenized directory	tokenized_test_files	File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type	"operator_tokenized.tla,empty_tokenized.tla,1,1,100.00%,Type-1 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,1,1,100.00%,Type-1 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,1,2,90.00%,Type-3 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,1,3,43.75%,Type-5 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,2,1,90.00%,Type-3 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,2,2,100.00%,Type-1 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,2,3,43.75%,Type-5 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,3,1,43.75%,Type-5 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,3,2,43.75%,Type-5 Clone scratch_same_tokenized.tla,scratch_d_tokenized.tla,3,3,100.00%,Type-1 Clone scratch_same_tokenized.tla,not_same_tokenized.tla,1,1,41.86%,Type-5 Clone scratch_same_tokenized.tla,not_same_tokenized.tla,1,2,44.44%,Type-5 Clone ... "	Pass
		2	for "small_files"	tokenized_small_file	File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type	"File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type 0013_tokenized.tla,0017_tokenized.tla,1,1,28.07%,Type-5 Clone 0013_tokenized.tla,0017_tokenized.tla,2,1,28.07%,Type-5 Clone 0013_tokenized.tla,0017_tokenized.tla,3,1,38.66%,Type-5 Clone 0013_tokenized.tla,0039_tokenized.tla,1,1,42.70%,Type-5 Clone 0013_tokenized.tla,0039_tokenized.tla,1,2,42.35%,Type-5 Clone 0013_tokenized.tla,0039_tokenized.tla,2,1,42.70%,Type-5 Clone ..."	Pass
		3	for "large_files"	tokenized_large_file	File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type	"File Name 1,File Name 2,Line Numbers 1,Line Numbers 2,Similarity (%),Clone Type 0049_tokenized.tla,0010_tokenized.tla,1,1,56.76%,Type-5 Clone 0049_tokenized.tla,0010_tokenized.tla,1,2,35.85%,Type-5 Clone 0049_tokenized.tla,0010_tokenized.tla,1,3,67.92%,Type-5 Clone 0049_tokenized.tla,0010_tokenized.tla,1,4,45.57%,Type-5 Clone 0049_tokenized.tla,0010_tokenized.tla,1,5,44.44%,Type-5 Clone ..."	Pass
V1	Should read data from the correct files and produce a pie chart using the value of Type-1 clone produced from clone detection of output_file (files that are parsed), and values of all the clone types produced from clone detection of tokenized_files (files that are tokenized)	1		raw_statistics.txt, files_statistics.txt	expected to correctly produce a pie chart displaying values of clone types from given files	a pie chart with correct values of clone types is produced	Pass
V2	Should read data from the correct file and produce a bar graph showing the average, maximum and minimum of each clone type, output png file of the graph	1		files_clones.csv	expected to correctly produce a bar graph displaying the average, max and min value of each clone type	bar chart with correct average, max, min values of clone types is produced	Pass
V3	Should read data from the correct file, find and count the maximum number of clone type of each file in the directory and plot it on a bar graph, output png file of the graph (Should ignore the files where there are two or more clone types of the same highest number in a single file)	1		tokenized_file_statistics.csv	expected to correctly produce bar graph that displays the maximum number of clone type of each file	bar chart with the correct maximum number of clone type of each file is produced	Pass

Table 10: Validation table of functional requirements part 3

Requirement ID	Requirement Description	Test case number	Test case description	Input	Expected Output	Actual Output	Result: Pass/ Fail
V4	Should read data from the correct files; produce a pie chart using the values of all the clone types (excluding type-5 clone) produced from clone detection of tokenized directory of individual files; display legend (clone type, percentage, number of clones); output png file of the graph	1		tokenized_file_statistics.csv	expected to correctly produce a pie chart displaying values of clone types from given files	a pie chart with correct values of clone types is produced	Pass
V5	Should create a CSV file that counts the number of code lines for all files in the specified directory, calculating the mean number of code lines as an integer, using this mean as a threshold to categorize files into 'small' if their code lines are below the mean and 'large' if equal to or exceeding the mean, and subsequently copying the files into new directories named 'Small_files' and 'Large_files', respectively.	1		parsed_test_files, tokenized_test_files	expected to correctly produce a csv file containing the number of code line and two new folders are created with correct files in each	a csv file and two new folders are produced with correct data in each	Pass
V6	Should read data from the correct files; produce a pie chart using the value of Type-1 clone produced from clone detection from parsed directory, and values of clone types 1, 2, 3, and 4 produced from clone detection of tokenized directory; display legend (clone type, percentage, number of clones); output png file of the graph	1		raw_statistics.txt, files_statistics.txt	expected to correctly produce a pie chart displaying values of clone types from given files	a pie chart with correct values of clone types is produced	Pass
V7	Should find Type-1 clones from parsed file, count them; output result in csv file along with the file names containing the clone	1		parsed_test_files	expected to correctly produce a csv file containing the clone code line, number of count and filename of where the clone exist	a csv file containing the clone code line, number of count and filename of where the clone exist is produced	Pass
V8	Should read data from the correct CSV files (tokenized_line_count and indi_file_statistics), sort the line count from tokenized_line_count in descending order, merge two tables together in the correct order of title (File Name, Line Count, Clone Pairs, Type-1 clones, Type-2 clones, Type-3 clones, Type-4 clones, Type-5 clones), and fill in the value of 0 for any missing data/cells.	1		tokenized_line_count.csv, indi_file_statistics.csv	expected to correctly produce a new csv file containing the filename, line count, clone pairs, and number of each clone types	a new csv file containing the filename, line count, clone pairs, and number of each clone types is produced	Pass

Table 11: Validation table of functional requirements part 4

Listing 16: ‘Preprocessor’ Code

```
import os
import re
def preprocess_and_parse(file_path):
    with open(file_path, ‘r’, encoding=‘utf-8’, errors=‘ignore’) as file:
        lines = file.readlines()

    preprocessed_lines = []
    ignore_lines = False

    for line in lines:
        if re.search(r"\bBEGIN TRANSLATION\b", line):
            ignore_lines = True
        elif not ignore_lines:
            if re.match(r"\s{4,}", line):
                if preprocessed_lines:
                    preprocessed_lines[-1] += " " + line.strip()
            elif "==" in line and re.match(r"^[A-Za-z0-9 ]", line):
                preprocessed_lines.append(line.strip())

    return ‘\n’.join(preprocessed_lines)

def process_tla_files(input_dir, output_dir):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    for filename in os.listdir(input_dir):
        if filename.endswith(".tla"):
            input_file_path = os.path.join(input_dir, filename)
            output_file_path = os.path.join(output_dir, filename)

            parsed_code = preprocess_and_parse(input_file_path)

            with open(output_file_path,‘w’) as output_file:
                output_file.write(parsed_code + ‘\n’)

if __name__ == "__main__":
    input_dir = ‘files’
    output_dir = ‘parsed_files’
    process_tla_files(input_dir, output_dir)
    print("TLA+ code processing complete.")
```

Listing 17: ‘Tokenizer’ Code

```python
import os
import re

# Define the input and output directories
input_dir = ‘parsed_files’
output_dir = ‘tokenized_files’

# Create the output directory if it doesn’t exist
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

def anonymize_variable_names(tla_code):
    # Split the TLA+ code into lines
    lines = tla_code.split(‘\n’)

    variable_count = 1
    variable_mapping = {}
    anonymized_lines = []

    for line in lines:
        # Remove comments
        line = re.sub(r‘--.*’, ‘’, line)

        # Replace integers with #
        line = re.sub(r‘\b\d+\b’, ‘#’, line)

        # Anonymize variable names
        words = re.findall(r‘[A-Za-z_][\w.\\]*|\\\w+|"[A-Z_ ]+"|\S’, line)
        anonymized_words = []

        for word in words:
            if re.match(r‘^[A-Za-z_][\w.]*$’, word) and not
            re.match(r‘^[A-Z_]+$’, word) and not re.match(r‘^\\’, word):
                if word not in variable_mapping:
                    variable_mapping[word] = f‘$name{variable_count}’
                    variable_count += 1
                anonymized_words.append(variable_mapping[word])
            else:
                anonymized_words.append(word)

        anonymized_line = ‘’.join(anonymized_words)  # Remove whitespaces
        anonymized_lines.append(anonymized_line)

    return ‘\n’.join(anonymized_lines)

# Process all TLA+ files in the input directory
for filename in os.listdir(input_dir):
    if filename.endswith(".tla"):
        input_file_path = os.path.join(input_dir, filename)
        output_file_path = os.path.join(output_dir,
        f"{os.path.splitext(filename)[0]}_tokenized.tla")

        with open(input_file_path, ‘r’) as input_file:
            tla_code = input_file.read()

        # Anonymize, replace integers, and preserve symbols
        tokenized_code = anonymize_variable_names(tla_code)

        # Write the tokenized code to the output file
        with open(output_file_path, ‘w’) as output_file:
            output_file.write(tokenized_code)

print("TLA+ code tokenization complete.")
```

Listing 18: Code Analysis: ‘count_line’ Code

```
import os
import csv
import shutil
import matplotlib.pyplot as plt

def count_lines_in_files(directory, csv_output, small_file, large_file):
    files = [file for file in os.listdir(directory) if file.endswith(‘.tla’)]
    line_count_data = []

    total_line_count = 0

    for file_name in files:
        file_path = os.path.join(directory, file_name)
        with open(file_path, ‘r’) as file:
            line_count = sum(1 for _ in file)  # Count the lines in the file
            line_count_data.append({‘File Name’: file_name,
            ‘Line Count’: line_count})

            # Update total line count
            total_line_count += line_count

    # Calculate mean line count
    mean_line_count = int(total_line_count / len(files))

    # Writing to CSV
    with open(csv_output, ‘w’, newline=‘’) as csvfile:
        fieldnames = [‘File Name’, ‘Line Count’]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(line_count_data)

    # Create directories for Small_files and Large_files outside
    of directory_path
    small_files_dir = os.path.join(os.path.dirname(directory), small_file)
    large_files_dir = os.path.join(os.path.dirname(directory), large_file)

    os.makedirs(small_files_dir, exist_ok=True)
    os.makedirs(large_files_dir, exist_ok=True)

    # Copy files to Small_files or Large_files based on line count
    for data in line_count_data:
        file_name = data[‘File Name’]
        line_count = data[‘Line Count’]
        source_path = os.path.join(directory, file_name)

        if line_count < mean_line_count:
            destination_path = os.path.join(small_files_dir, file_name)
        else:
            destination_path = os.path.join(large_files_dir, file_name)

        shutil.copyfile(source_path, destination_path)

count_lines_in_files("tokenized_files", "tokenized_line_count.csv",
"tokenzied_small_file", "tokenized_large_file")
count_lines_in_files("parsed_files", "parsed_line_count.csv",
"parsed_small_file", "parsed_large_file")
```

Listing 19: ‘Code Detector’ Code part 1

```
import difflib
import os
import csv
import time
from collections import defaultdict

# find clones within individual files
def detect_clones_individual(file_path):
    with open(file_path, ‘r’) as file:
        code = file.readlines()
    clones = []
    for i, line1 in enumerate(code):
        for j, line2 in enumerate(code[i + 1:], start=i + 1):
            similarity = difflib.SequenceMatcher(None, line1, line2).ratio()
            if similarity > 0.2:
                if similarity == 1:
                    clone_type = "Type-1 Clone"
                elif similarity > 0.9:
                    clone_type = "Type-2 Clone"
                elif similarity > 0.8:
                    clone_type = "Type-3 Clone"
                elif similarity > 0.7:
                    clone_type = "Type-4 Clone"
                else:
                    clone_type = "Type-5 Clone"
                clones.append((i + 1, j + 1, similarity, clone_type))
    return clones

def individual_process_directory(directory_path, clone_csv, statistics_csv):
    start_time = time.time()
    with open(clone_csv, ‘w’, newline=’’) as csv_file, open(statistics_csv,
    ‘w’, newline=’’) as stats_file:
        csv_writer = csv.writer(csv_file)
        stats_writer = csv.writer(stats_file)

        csv_writer.writerow(["File Name", "Line Numbers", "Similarity (%)",
        "Clone Type"])
        stats_writer.writerow(["File Name", "Clone Pairs", "Type-1 Clones",
        "Type-2 Clones", "Type-3 Clones", "Type-4 Clones", "Type-5 Clones"])

        for root, _, files in os.walk(directory_path):
            for file in files:
                if file.endswith(".tla"):  # Adjust the file extension as needed
                    file_path = os.path.join(root, file)
                    clones = detect_clones_individual(file_path)

                    clone_pairs = len(clones)
                    clone_types_count = defaultdict(int)
                    for _, _, _, clone_type in clones:
                        clone_types_count[clone_type] += 1

                    if clone_pairs > 0:
                        for line1, line2, similarity, clone_type in clones:
                            line_numbers = f"{line1} - {line2}"
                            similarity_percentage = f"{similarity * 100:.2f}%"
                            csv_writer.writerow([file_path,
                            line_numbers, similarity_percentage, clone_type])

                        stats_writer.writerow([file_path, clone_pairs] +
                        [clone_types_count[type]
                        for type in ["Type-1 Clone", "Type-2 Clone",
                        "Type-3 Clone", "Type-4 Clone", "Type-5 Clone"]])
```

Listing 20: ‘Code Detector’ Code part 2

```
# find clones between two different files
# from parsed directory
def parsed_detect_clones_files(file_path1, file_path2):
    clones = []
    with open(file_path1, ‘r’, encoding=‘utf-8’, errors=‘ignore’) as file1,
    open(file_path2, ‘r’, encoding=‘utf-8’, errors=‘ignore’) as file2:
        code1 = file1.readlines()
        code2 = file2.readlines()

        for i, line1 in enumerate(code1, start=1):
            for j, line2 in enumerate(code2, start=1):
                similarity = difflib.SequenceMatcher(None, line1,
                line2).ratio()
                clone_type = "Type-1 Clone" if similarity == 1 else None
                clones.append((os.path.basename(file_path1),
                os.path.basename(file_path2), i, j, similarity, clone_type,
                line1))
    return clones

def parsed_files_process_directory(directory_path, output_csv, output_txt):
    start_time = time.time()
    with open(output_csv, ‘w’, newline=‘’, encoding=‘utf-8’) as csv_file,
    open(output_txt, ‘w’,
    encoding=‘utf-8’) as txt_file:
        csv_writer = csv.writer(csv_file)
        csv_writer.writerow(["File Name 1", "File Name 2", "Line Numbers 1",
        "Line Numbers 2", "Similarity (%)", "Clone Type", "Code line"])
        type_1_clones = 0

        files = os.listdir(directory_path)
        for i, file1 in enumerate(files):
            for file2 in files[i + 1:]:
                file_path1 = os.path.join(directory_path, file1)
                file_path2 = os.path.join(directory_path, file2)
                clones_parsed = parsed_detect_clones_files(file_path1,
                file_path2)

                for clone in clones_parsed:
                    _, _, _, _, _, clone_type, _ = clone
                    if clone_type == "Type-1 Clone":
                        file_name1, file_name2, line_numbers1,
                        line_numbers2, similarity, _, code_line = clone
                        similarity_percentage = f"{similarity * 100:.2f}%"
                        csv_writer.writerow([file_name1, file_name2,
                        line_numbers1, line_numbers2, similarity_percentage,
                        clone_type, code_line])
                        type_1_clones += 1
        # Write clone statistics to the text file
        txt_file.write(f"Type-1 Clones: {type_1_clones}\n")

    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Files process execution time (parsed): {execution_time} seconds")
```

Listing 21: ‘Code Detector’ Code part 3

```
# find clones between two different files
# from tokenized directory
def detect_clones_files(file_path1, file_path2):
    with open(file_path1, ‘r’, encoding=‘utf-8’, errors=‘ignore’) as file1,
    open(file_path2, ‘r’,
    encoding=‘utf-8’, errors=‘ignore’) as file2:
        code1 = file1.readlines()
        code2 = file2.readlines()
    clones = []
    for i in range(len(code1)):
        for j in range(len(code2)):
            similarity = difflib.SequenceMatcher(None, code1[i],
            code2[j]).ratio()
            if similarity == 1:
                clone_type = "Type-1 Clone"
            elif similarity > 0.9:
                clone_type = "Type-2 Clone"
            elif similarity > 0.8:
                clone_type = "Type-3 Clone"
            elif similarity > 0.7:
                clone_type = "Type-4 Clone"
            else:
                clone_type = "Type-5 Clone"
            if similarity > 0.2:  # You can adjust this threshold
            clones.append((os.path.basename(file_path1),
            os.path.basename(file_path2), i + 1, j + 1, similarity,
            clone_type))
    return clones
```

Listing 22: 'Code Detector' Code part 4

```
#create csv file to output clone information between two files
#create txt file to output the total number of clone pairs, and clone
types in the directory
def files_process_directory(directory_path, output_csv, output_txt):
    start_time = time.time()  # Start time measurement
    with open(output_csv, 'w', newline='', encoding='utf-8') as
    csv_file, open(output_txt, 'w', encoding='utf-8') as txt_file:
        csv_writer = csv.writer(csv_file)
        csv_writer.writerow(["File Name 1", "File Name 2",
        "Line Numbers 1", "Line Numbers 2", "Similarity (%)", "Clone Type"])
        total_clone_pairs = 0
        type_1_clones = 0
        type_2_clones = 0
        type_3_clones = 0
        type_4_clones = 0
        type_5_clones = 0
        for root, _, files in os.walk(directory_path):
            for i in range(len(files)):
                for j in range(i + 1, len(files)):
                    file_path1 = os.path.join(root, files[i])
                    file_path2 = os.path.join(root, files[j])
                    clones_tokenized = detect_clones_files(file_path1,
                    file_path2)

                    if clones_tokenized:
                        total_clone_pairs += len(clones_tokenized)

                        for clone in clones_tokenized:
                            file_name1, file_name2, line_numbers1,
                            line_numbers2, similarity,
                            clone_type = clone
                            similarity_percentage = f"{similarity *
                            100:.2f}%"
                            csv_writer.writerow([file_name1, file_name2,
                            line_numbers1,
                            line_numbers2, similarity_percentage, clone_type])

                    # Count the different clone types
                    for clone in clones_tokenized:
                        _, _, _, _, _, clone_type = clone
                        if clone_type == "Type-1 Clone":
                            type_1_clones += 1
                        elif clone_type == "Type-2 Clone":
                            type_2_clones += 1
                        elif clone_type == "Type-3 Clone":
                            type_3_clones += 1
                        elif clone_type == "Type-4 Clone":
                            type_4_clones += 1
                        elif clone_type == "Type-5 Clone":
                            type_5_clones += 1

        # Write clone statistics to the text file
        txt_file.write(f"Total Clone Pairs: {type_1_clones+type_2_clones+
        type_3_clones+type_4_clones+type_5_clones}\n")
        txt_file.write(f"Type-1 Clones: {type_1_clones}\n")
        txt_file.write(f"Type-2 Clones: {type_2_clones}\n")
        txt_file.write(f"Type-3 Clones: {type_3_clones}\n")
        txt_file.write(f"Type-4 Clones: {type_4_clones}\n")
        txt_file.write(f"Type-5 Clones: {type_5_clones}\n")
```

Listing 22: ‘Code Detector’ Code part 5

```
if __name__ == "__main__":
    raw_directory_path = ‘parsed_files’
    tokenized_directory_path = ‘tokenized_files’
    parsed_small_directory_path = ‘parsed_small_file’
    parsed_large_directory_path = ‘parsed_large_file’
    tokenized_small_directory_path = ‘tokenzied_small_file’
    tokenized_large_directory_path = ‘tokenized_large_file’
    indi_clone_csv = ‘indi_file_clones.csv’  # Output CSV file for
    individual clone results
    indi_statistics_csv = ‘indi_file_statistics.csv’
# Output CSV file for clone statistics
    raw_output_csv = ‘raw_clones.csv’  # Output CSV file for clone results
    raw_output_txt = ‘raw_statistics.txt’  # Output TXT file for clone
    statistics
    small_raw_output_csv = ‘small_raw_clones.csv’  # Output CSV file for
    clone results
    small_raw_output_txt = ‘small_raw_statistics.txt’
# Output TXT file for clone statistics
    large_raw_output_csv = ‘large_raw_clones.csv’  # Output CSV file for
    clone results
    large_raw_output_txt = ‘large_raw_statistics.txt’
# Output TXT file for clone statistics
    output_csv =‘files_clones.csv’  # Output CSV file for clone results
    output_txt =‘files_statistics.txt’  # Output TXT file for clone
    statistics
    small_output_csv = ‘small_files_clones.csv’  # Output CSV file for
    clone results
    small_output_txt = ‘small_files_statistics.txt’
# Output TXT file for clone statistics
    large_output_csv = ‘large_files_clones.csv’  # Output CSV file
    for clone results
    large_output_txt = ‘large_files_statistics.txt’
# Output TXT file for clone statistics

    #small files
    print("Detecting clones from the Small_files dataset...")
    parsed_files_process_directory(parsed_small_directory_path,
    small_raw_output_csv, small_raw_output_txt)
    files_process_directory(tokenized_small_directory_path,
    small_output_csv, small_output_txt)
    #large files
    print("Detecting clones from the Large_files dataset...")
    parsed_files_process_directory(parsed_large_directory_path,
    large_raw_output_csv, large_raw_output_txt)
    files_process_directory(tokenized_large_directory_path,
    large_output_csv, large_output_txt)
    #whole dataset
    print("Detecting clones from the whole dataset...")
    parsed_files_process_directory(raw_directory_path,
    raw_output_csv, raw_output_txt)
    individual_process_directory(tokenized_directory_path,
    indi_clone_csv, indi_statistics_csv)
    files_process_directory(tokenized_directory_path, output_csv,
    output_txt)
```

Listing 23: ‘Clone Type Pie’ Code

```
import matplotlib.pyplot as plt

def clone_type_pie(parsed_stats, tokenized_stats, png_image):
        # Read content from file1.txt
        with open(parsed_stats, ‘r’) as file1:
            lines_file1 = file1.readlines()
        # Read content from file2.txt
        with open(tokenized_stats, ‘r’) as file2:
            lines_file2 = file2.readlines()
        # Extracting Type-1 Clones values from both files
        type1_file1 = int(lines_file1[0].split(‘: ’)[1].strip())
        type1_file2 = int(lines_file2[1].split(‘: ’)[1].strip())
        # Extracting Type-2 to Type-5 Clones values from file2
        type2_file2 = int(lines_file2[2].split(‘: ’)[1].strip())
        type3_file2 = int(lines_file2[3].split(‘: ’)[1].strip())
        type4_file2 = int(lines_file2[4].split(‘: ’)[1].strip())
        type5_file2 = int(lines_file2[5].split(‘: ’)[1].strip())
        #calculate percentage for legend
        total= type1_file1 + type1_file2 + type2_file2 + type3_file2 +
 type4_file2 + type5_file2
        t1f1_percentage = type1_file1 / total * 100
        t1f2_percentage = type1_file2 / total * 100
        t2f2_percentage = type2_file2 / total * 100
        t3f2_percentage = type3_file2 / total * 100
        t4f2_percentage = type4_file2 / total * 100
        t5f2_percentage = type5_file2 / total * 100
        # Creating data for the pie chart
        labels = [‘Raw Type-1 Clones’, ‘Tokenized Type-1 Clones’,
 ‘Tokenized Type-2 Clones’, ‘Tokenized Type-3 Clones’,
 ‘Tokenized Type-4 Clones’, ‘Tokenized Type-5 Clones’]
        sizes = [type1_file1, type1_file2, type2_file2, type3_file2,
 type4_file2, type5_file2]
        percentages = [t1f1_percentage, t1f2_percentage,
 t2f2_percentage, t3f2_percentage, t4f2_percentage, t5f2_percentage]
        colors = [‘gold’, ‘yellowgreen’, ‘lightcoral’, ‘lightskyblue’,
 ‘orange’, ‘pink’]
        explode = (0.1, 0, 0, 0, 0, 0)
        # Plotting the pie chart
        plt.figure(figsize=(12, 6))
        patches, texts, _ = plt.pie(sizes, explode=explode, labels=None,
 colors=colors,
 autopct=‘%1.2f%%’, startangle=140, pctdistance=1)
        plt.axis(‘equal’)
        plt.title(‘Distribution of Clones’)
        plt.tight_layout()
        # Create legend with labels and percentages
        legend_labels = [f‘{label}: {percentages:.2f}% ({sizes[i]})’
 for i, label, size, percentages in zip(range(len(labels)), labels,
 sizes, percentages)]
        plt.legend(patches, legend_labels, loc="best",
 bbox_to_anchor=(1, 0.5), title="Clone Types", fontsize='small’)
        # Save the plot as an image file
        plt.savefig(png_image)
clone_type_pie(‘small_raw_statistics.txt’, ‘small_files_statistics.txt’,
‘small_clone_type_pie.png’)
clone_type_pie(‘large_raw_statistics.txt’, ‘large_files_statistics.txt’,
‘large_clone_type_pie.png’)
clone_type_pie(‘raw_statistics.txt’, ‘files_statistics.txt’,
‘whole_clone_type_pie.png’)
```