ABSTRACT

WANG, PEIPEI. Regular Expression Testing, Comprehension, and Repair. (Under the direction of Kathryn T. Stolee.)

Although there are tools to help developers understand the matching behaviors between a regular expression and a string, developers report testing their regular expressions less than the rest of their code, and regular-expression related faults are still common.

In this work, we firstly present exploration on how thoroughly tested regular expressions are by examining open source projects. Using standard metrics of coverage, such as line and branch coverage, gives an incomplete picture of the test coverage of regular expressions. We adopt graph-based coverage metrics for the DFA representation of regular expressions, providing fine-grained test coverage metrics. Using over 15,000 tested regular expressions in 1,225 Java projects on GitHub, we measure node, edge, and edge-pair coverage. Our results show that only 17% of the regular expressions in the repositories are tested at all. For those that are tested, the median number of test inputs is two. For nearly 42% of the tested regular expressions, only one test input is used. Average node and edge coverage levels on the DFAs for tested regular expressions are 59% and 29%, respectively. Due to the lack of testing of regular expressions, we explore whether a string generation tool for regular expressions, Rex, achieves high coverage levels. With some exceptions, we found that tools such as Rex can be used to write test inputs with similar coverage to the developer tests.

Learning developers' behavior through the change history of regular expressions can identify common edit patterns, which can inform the creation of mutation and repair operators to assist with testing and fixing regular expressions. Therefore, we also present our exploration of how regular expressions evolve over time, focusing on the characteristics of regular expression edits, the syntactic and semantic difference of the edits, and the feature changes of edits. Our exploration uses two datasets. First, we look at GitHub projects that have a regular expression in their current version and look back through the commit logs to collect the regular expressions' edit history. Second, we collect regular expressions composed by study participants during problem-solving tasks. Our results show that 1) 95% of the regular expressions from GitHub are not edited, 2) most edited regular expressions have a syntactic distance of 4-6 characters from their predecessors, 3) over 50% of the edits in GitHub tend to expand the scope of regular expression, and 4) the number of features used indicates the regular expression language usage increases over time. This work has implications for supporting regular expression repair and mutation to ensure test suite quality.

We propose two regular expression repair strategies. One is fixing incorrect regular expressions through regular expression mutation, and the other is replacing misused regular expressions by converting them into string operations. Besides, we propose as well a further study on regular expression testing coverage based on related problems discussed in completed research.

CHAPTER

1

OVERVIEW

In Chapter 1 we describe the overview of this work: the introduction of regular expression testing and evolution problems, questions, and the outline.

1.1 Introduction

A regular expression is a language describing the pattern of strings to match. Regular expressions (regexes) are used fundamentally in string searching and substitution tasks, such as data validation, classification, and extraction for various purposes (e.g., [8, 63]). More advanced uses can be seen in search engines [113], database querying [111], and network security [35, 51, 101].

Recent research has suggested that regular expressions are hard to understand, hard to compose, and error prone [24, 96]. This is because understanding and writing regular expressions requires both knowledge and skills. A single character mistake could cause drastically different regular expression matching behaviors. Regular expressions are frequently used in software development and are also responsible for many errors. For example, a simple search for "regex OR regular expression" in GitHub yields 227,474 issues (and growing), with 25% of those still being open. When a regular expression is responsible for a software bug, the impact can be severe, possibly resulting in corrupted data, security vulnerabilities, or denial of service attacks) [29, 56, 93]. To make it worse, some regular

expression bugs can lead to software performance issues, impacting software response time and even triggering system crashes [83, 109]. Despite this, regular expressions remain under-studied in software engineering. A survey of professional developers reveals that they test their regular expressions *less* than the rest of their code [23].

Given their frequent appearances in software source code and the difficulty of working with them, some effort has been put into easing the burden on developers by providing tools that make regexes easier to understand. Some tools provide debugging environments which explain string matching results and highlight the parts of regex patterns which match a certain string [81, 87]. Other tools present graphical representations (e.g., finite automata) of the regular expressions [89, 90]. Still, others either automatically generate strings according to the given regular expressions [53, 54, 73, 104] or automatically generate regular expressions according to the given list of strings [9, 67]. Specifically for regular expression testing purposes, Rex [104] is a tool for analyzing regular expressions through symbolic analysis, which we use in this work to generate strings for testing. Given a regular expression R, Rex uses the Z3 [30] SMT solver to generate members of the language by treating it as a satisfiability problem.

While these tools are helpful, there are some restrictions which hinder their usage. They may need the regex and a matched string at the same time, do not support some commonly used regular expression features, generate non-printable strings, or generate over-fitted regular expressions. Although we do recommend involving regular expression tools into the testing process, those tools can be made smarter by generating both matching input strings and non-matching input strings. The goal of generating matching inputs has been studied extensively [34, 41, 73, 88, 104]. The goal of generating non-matching inputs can be reached through regular expression mutation techniques [6, 7, 12, 59]. Recent research efforts have explored mutating input strings for a regular expression [59] and a fault-based approach to generating regular expression tests through mutation testing [6]; faults are injected into regular expressions and strings are generated that witness the fault, hence providing examples of non-matching behavior for the original regular expression. Although mutation testing is a maturing research area, there are much fewer efforts on mutation testing specifically on regular expressions [6, 7, 12, 13, 67]. However, these efforts have defined the mutation operators in an ad hoc manner, without consideration of how regular expressions evolve in practice. Hence, the faults injected might not be representative of actual edits.

Regular expression evolution is the first step to revealing common regular expression mistakes and drawing empirical regular expression mutation patterns. Beyond shedding light on the types of edits to consider in fault-based test generation, the history of source code development and the information of bug fixes are valuable in guiding program repairs [60]. Given the fault-proneness of regular expressions [83], and that developers under-test their regular expressions [107], understand-

ing how they evolve can help guide testing and repair efforts. For example, if regular expressions typically increase in scope over time (i.e., the language expands), it is valuable to focus testing efforts on strings beyond a regular expression's language. If, however, regular expressions typically decrease in scope, testing efforts should focus on the regular expression's language.

One important analysis in regular expression evolution is their semantic similarity changes. This is a challenging task because in regular expressions as in source code, there are multiple ways to express the same semantic concept. For example, the regex, `aa*` matches an "a" followed by zero or more "a", and is equivalent to `a+` , which matches one or more "a". Comparing all strings accepted by two regular expressions is resource consuming and sometimes not possible. An approximate is to use a certain number of accepted strings generated by regular expression tools. The accuracy of semantic similarities is thus influenced by both the string size threshold and the regex tools. To provide better semantic similarity calculations, other ways can be explored: using a large number of randomly generated strings of various different string length and using the regex AST tree structure.

The next step of regular expression mutation is a comprehensive regular expression bug study to classify the types of bugs and the common pattern in changing regular expressions to fix the bugs they cause. Finally, we apply these mutation patterns to regular expression generating tools for improving the quality of generated strings. Besides regular expression mutation which is used to fix incorrect regular expressions, another common problem is regular expression misuse where the regular expression is not a necessity in the source code. A solution to this problem is to replace regular expression operations with simple source code such as string operations.

In this work, we present the work to use test metrics for graph-based coverage [2] over the DFA representation of regular expressions. We use it on empirically measuring how well tested regular expressions are and further explore the potential for using existing tools, specifically Rex, to improve the test coverage [107]. The results show that over 80% of regular expressions written in GitHub projects are not tested [107], indicating that developers either do not test regular expressions or use external tools rather than test cases. The results also show that with larger test input size, Rex performs nearly as well as developers.

This work also presents a study of regular expression evolution [106] on two datasets collected separately from two different contexts: Github dataset and Video dataset. The GitHub dataset is comprised of Java regular expressions collected from GitHub projects by mining their source code commit history. It represents the use case of regular expressions in a persistent environment (e.g., within source code) and provides a coarse-grained view of changes to the regular expression over time. However, the commit history can mask the actual evolution of regular expressions as a developer is composing them since the commit history represents only what is pushed to the repository. The Video dataset contains Java regular expressions written by developers during problem-solving

tasks. It was conducted in an ephemeral environment (e.g., grepping/searching a document/IDE) and used as a finer-granularity view into regular expression evolution. In combination, we are able to see the types of changes developers made to regular expressions in both contexts.

According to the learned knowledge of regular expressions, this work proposes 1) regular expression mutation and 2) regular expression misuse and replacement regarding regular expression repair. It also proposes 3) further exploration of regular expression testing coverage according to discussions of testing coverage criteria [107].

1.2 Research Questions

The research questions are divided into five aspects and discussed separately: regular expression testing coverage, regular expression evolution, regular expression mutation, regular expression misuse and replacement, and further exploration of regular expression testing coverage.

1.2.1 Regular Expression Testing Coverage

This research was published at FSE 2018 [107]. To explore the potential of using graph coverage metrics for regular expressions, we evaluate the following research questions:

RQ1: *How well are regular expressions tested in GitHub*?

To answer RQ1, we identify 1,225 Java projects that have existing test suites covering the regular expressions. From these, we extract 15,096 regular expressions and 899,804 total test input strings, measuring NC, EC, and EPC for each regular expression. To obtain the regular expressions and their corresponding strings which are covered by test cases, we use the Java bytecode manipulation framework Javassist [25] to record the regular expressions when pattern matching methods are triggered by test cases. This research question is answered in Section 3.2.1.

RQ2: *How well can the regular expression string generation tool Rex improve the test coverage of regular expressions*?

Using the regular expressions from RQ1, we generate test strings using Rex [104] and calculate the regular expression coverage, comparing it to the coverage of the user-defined test suites from RQ1. Using Rex, we generate test suites of three sizes, one to match the size of the user-defined test suites from the GitHub projects, one 5x that size, and one 10x that size. By comparing the coverage statistics we got in RQ2 to those in RQ1, we evaluate the test coverage

possibilities by using an automated tool. The detailed answer to this question is present in Section 3.2.2.

1.2.2 Regular Expression Evolution

We explore regular expression evolution with respect to syntactic and semantic measures for the purpose of exploration. Here, we consider the regular expression string itself, referred to as r_1 or r_2, and the languages described by the regular expressions, $L(r_1)$ and $L(r_2)$. Evolution is explored in the context of two datasets: one from the commit histories of projects on GitHub, and one from screencasts of students solving regular expression tasks. This research is going to appear at SANER 2019 [106]. Our research questions are:

RQ3: *What are the characteristics of regular expression evolution*?

We explore the number of edits as each regular expression evolves, the invalid regular expressions on edit chains, and the phenomenon of regular expression reversions, when the same regular expression re-appears later in an edit chain. Details of this research question can be found in Section 4.2.

RQ4: *How similar is a regular expression to its predecessor syntactically and semantically*?

We explore syntactic similarity with the Levenshtein distance between regular expression strings over time. Regarding semantic similarity, we are interested in the languages described by regular expressions and how those change over time. For a regular expression r_2 that evolves from r_1, we are interested in the overlap between the languages (i.e., $L(r_2)$ and $L(r_1)$). Details of this research question can be found in Section 4.3.

RQ5: *How do the features change in the evolution of a regular expression*?

We explore the features that are most frequently added, removed, or changed over the whole datasets. We also analyze the number of features added, removed or changed per regular expression edit. Detailed analysis for this research question can be found in Section 4.4.

1.2.3 Proposed: Regular Expression Mutation

This research is being proposed to explore regular expression repair by mutation. It is an ongoing work and further details are provided in Chapter 5.1. Our research questions are:

RQ6: *What are the characteristics of regular expression bugs*?

We explore the type of different mistakes in using regular expressions, the impact of these bugs, and the ways to fix them. A regular expression may be changed, replaced, or deleted in the source code. The reason might be the requirement changes, better performance, regex misunderstandings, or incorrect regular expressions. Not all regex-related bugs can be fixed by modifying regular expression themselves. Bugs being fixed by replacing regular expressions into string operations is addressed by the proposed work below in Section 1.2.4.

RQ7: *What are the common mutation patterns in repairing regular expressions*?

We explore the literal regular expression changes in commit histories. By comparing the different parts of changes before and after the commit, we are going to give mutation patterns per regular expression feature or common components.

RQ8: *How effective is regular expression mutation for repairing buggy regular expressions*?

We are going to apply the regular expression mutation patterns into Generate-and-Validate [61] program repair techniques and measure how effective will it repair buggy regexes. Especially, we will build a benchmark in order to test if buggy regexes can be fixed through mutated regular expressions.

1.2.4 Proposed: Regular Expression Misuse and Replacement

This research is being proposed to explore regular expression misuse and replacement through collected regular expressions in the source code. It is an proposed work and further details are provided in Chapter 5.2. Our research questions are:

RQ9: *Are string operations more understandable than regular expressions*?

In this research, We are going to explore the type of different misuse cases in collected regular expression dataset and estimate the percentage of misused regular expressions in practice. Most importantly, an evaluation of the comprehension of regular expressions versus string operations is needed. A human study will be conducted to compare their understandability using misused regexes and their fixes of string operations.

RQ10: *Are string operations automatically transformed from the regular expression more understandable*?

If code understandability can be improved by replacing the regular expression with string operations, a research tool will be constructed in order to detect misused regexes and to convert them into string operations. With this tool we can detect misused regexes in real

projects and file bug reports along with transformed string operations. The feedback from developers are finally analyzed to measure the effectiveness of transformed string operations.

1.2.5 Proposed: Further Exploration of Regular Expression Testing Coverage

This research is proposed to be the extension of regular expression testing coverage [107]. After getting the common mutation patterns, we are interested in applying them to string generations for regular expressions and verifying the quality of generated strings. Further details are provided in Chapter 5.3. Our research questions are:

RQ11: *How many regular expressions containing infeasible DFA Paths*?

In this research, We are going to explore the regular expressions collected in previous work [107] and study how many of them containing infeasible DFA paths.

RQ12: *What is the maximal testing coverage achieved by a DFA*?

If this research, we are going to propose the methodology to estimate the maximum coverage with the DFA testing coverage criteria.

RQ13: *What are the minimal test suite for maximally covering a DFA*?

If this research, we are going to propose the methodology to build the smallest test suite which still achieve the best possible testing coverage of a DFA.

RQ14: *Could regular expression mutation generate better testing strings for regular expressions*?

This research is going to measure whether regular expression mutation generates better testing strings for regular expressions. In order to do that, a comparison between testing strings generated by regular expression mutations and those generated by other tools will be conducted.

1.3 Timeline

The Table 1.1 shows the outline of the research questions. The topic *regular expression testing coverage* and topic *regular expression evolution* are completed and published as shown in green cells in Table 1.1. Topic *regular expression mutation* is the ongoing research and colored in yellow. The topic *regular expression misuse and replacement* is the proposed work shown in pink. Regarding the schedule, the topic of *regular expression mutation* is expected by April 2019, the topic of *regular expression misuse and replacement* is expected by June 2019, and the topic of *further exploration of regular expression testing coverage* is expected by August 2019.

Table 1.1 Timeline of Research Questions. Green means completed research work; yellow means ongoing research work; and pink means proposed research work.

Topic	Research Questions	Status	Publication
Regular Expression Testing Coverage	RQ1: How well are regular expressions tested in GitHub? RQ2: How well can the regular expression string generation tool Rex improve the test coverage of regular expressions?	Completed	FSE 2018
Regular Expression Evolution	RQ3: What are the characteristics of regular expression evolution? RQ4: How similar is a regular expression to its predecessor syntactically and semantically? RQ5: How do the features change in the evolution of a regular expression?	Completed	SANER 2019 (to appear)
Regular Expression Mutation	RQ6: What are the characteristics of regular expression bugs? RQ7: What are the common mutation patterns in repairing regular expressions? RQ8: How effective is regular expression mutation for repairing buggy regular expressions?	Ongoing Jan 2019 Ongoing Jan 2019 Ongoing April 2019	Goal: FSE 2019 or ASE 2019
Regular Expression Misuse and Replacement	RQ9: Are string operations more understandable than regular expressions? RQ10: Are string operations automatically transformed from the regular expression more understandable?	Proposed May 2019 Proposed June 2019	Goal: ASE 2019 or ICSE 2020
Further Exploration of Regular Expression Testing Coverage	RQ11: How many regular expressions containing infeasible DFA Paths? RQ12: What is the maximal testing coverage achieved by a DFA? RQ13: What are the minimal test suite for maximally covering a DFA? RQ14: Could regular expression mutation generate better testing strings for regular expressions?	Proposed August 2019 Proposed August 2019 Proposed August 2019 Proposed August 2019	Goal: TSE

1.4 Contributions

The completed pieces of research are published at FSE 2018 [107] and going to appear at SANER 2019 [106]. The contributions are:

- Application of graph-based metrics for test coverage of regular expressions: node coverage, edge coverage, and edge-pair coverage (Section 2.1.2 and Section 3).
- Test coverage evaluation of 15,096 regular expressions based on nearly 900,000 input strings from 1,225 Java projects from GitHub (RQ1).
- Evaluation of test coverage achieved by the Rex symbolic analysis tool for regular expressions (RQ2).
- Exploration of regular expression evolution from a coarse-grained lens of GitHub commit logs and a fine-grained lens of programmers working in an IDE (Section 2.1.3 and Section 4).
- An empirical study of how regular expressions are edited syntactically and semantically over time, using two metrics: Levenshtein distance and semantic distance (RQ4).
- An empirical study of how regular expression features changed over the process of editing regular expression (RQ5).

1.5 Outline

This work begins with a background and motivation section on introducing terminology, standards, techniques used in this work, and motivating the research conducted and will be conducted in this work in Chapter 2. Next, it presents the study of regular expression testing in Chapter 3 and the study of regular expression evolution in Chapter 4 with their own results and discussions. Then, it presents the proposed research work and future work opportunities in Chapter 5. Finally, it talks about related work of regular expressions in various research areas in Chapter 6 followed by a conclusion in Chapter 7 and a bibliography at the end of this work

CHAPTER

2

TERMINOLOGY AND MOTIVATION

In Chapter 2 we first introduce terminologies and common terms used in this work and then present our examples to motivate the studies in this work.

2.1 Terminology

2.1.1 DFA with Regular Expression Matching

In this section, we will talk about DFA and the different DFAs generated by re2 [28] with different regular expression matching types.

2.1.1.1 DFA and NFA

A regular expression represents a series of strings sharing the same pattern. The regex processor, in order to match the regex against a string, translates the regular expression into a deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA). For a given input, a state in DFA can only transition to only one other state while a state in NFA has multiple states to transition to and requires backtracking so that it can try other states after previous states are tried.

Although theoretically the DFA and the NFA are equivalent representations of the same regex,

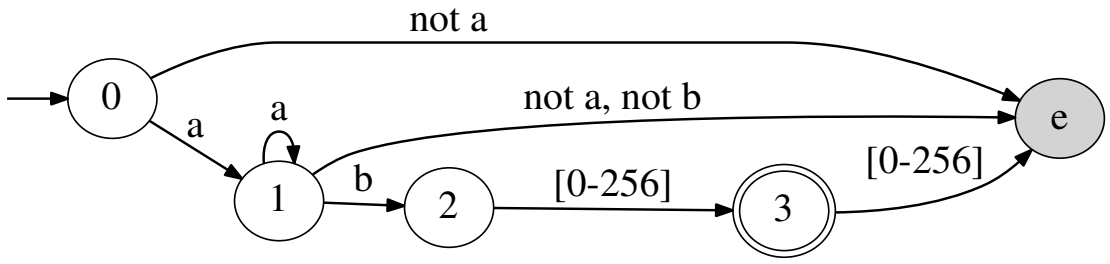


Figure 2.1 Forward FullMatch DFA of regular expression. `'a+b'`.

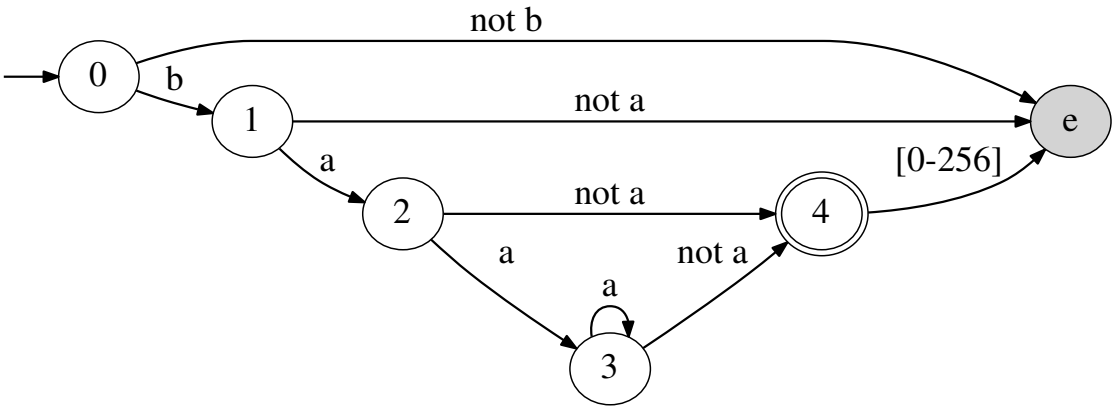


Figure 2.2 Backward FullMatch DFA of regular expression. `'a+b'`.

some regular expressions cannot be converted into a DFA. This is due to that regular expression libraries in current programming languages are more expressive than regular languages. For example, the backreference feature cannot be modeled by a finite-state automaton and requires a push-down automaton. When creating support techniques and tools for regular expression testing, regularity, or how regular are regular expressions, is important since it defines the abstractions we can use for test coverage.

In this work, we chose to use DFA not only because DFA is comparatively simpler than NFA but also because the DFA provides a stable representation at static but NFA graph structure can only be depicted during the matching process.

2.1.1.2 RE2

RE2 [27, 28] is a regular expression processing engine developed by Google with DFA implementations. Figure 2.1 shows one DFA of the regular expression `a+b` built using RE2 [28], and we take this

opportunity to describe the DFA notation used throughout this work.

Node 0 is the start-state, indicated by the incoming arrow. Nodes with double-circles are accepted states, such as Node 3. Node e is the error state, denoting a mismatch. The edges are labeled with transitions, often using syntactic sugar for ease of interpretation. The edge $\overrightarrow{01}$ is traversed when a character 'a' is read. If any other character is read at Node 0, (i.e., `not a`), edge $\overrightarrow{0e}$ is traversed. There is a self-loop on Node 1 for the character `a`. If character 'b' is read from Node 1, then edge $\overrightarrow{12}$ is traversed.

In RE2, when reading an input string, byte `[256]`, is added as a text-end marker. For example, the input string "aab" is transformed to the byte stream `[97 97 98 256]`, as `[97]` is the byte for 'a', `[98]` is for 'b', and `[256]` marks the end of the string. Byte [256] is matched on edges '`[0-256]`', '`not a`', or '`not a, not b`'.

2.1.1.3 DFA Types

There are different types of matching between a regular expression and a string. The Java function *Pattern.matches* requires the regular expression to match a string from its beginning to its end; Python's *re.match* requires the regular expression to match a string only from its beginning, not necessarily match to the end of the string; and the C# function *Regex.Match* requires the regular expression to match only a substring of the input string. These are called *FullMatch*, *FirstMatch*, and *ManyMatch*, respectively.

Given a regular expression and an input string to match, we could build multiple DFAs with different considerations. We can build a Forward DFA and Backward DFA depending on the direction of scanning the regular expression. Figure 2.1 and Figure 2.2 shows the differences between a Forward FullMatch DFA and a Backward FullMatch DFA. For the regular expression "a+b", its Forward DFA (Figure 2.1) and Backward DFA (Figure 2.2) are different in both DFA size and DFA structure if the regex is not a syntactic palindrome. We could build a static DFA with a regular expression alone or build a DFA on-the-fly (dynamic DFA) during a matching between a regular expression and an input string. In this paper, we choose the Forward static DFA for the regular expression and the Forward dynamic DFA for *FullMatch* of the regular expression and the input string.

2.1.2 Regular Expression Testing Coverage Criteria

We explore fine-grained coverage metrics for regular expressions based on a DFA representation. The intuition is that since regular expressions are equivalent to DFAs [95], and 96% of regular expressions in the wild were found to be regular (Section 2.2.1.1), then graph coverage metrics over the DFA can be used to test the behavior within the regular expression. We discuss three levels of

coverage: Node Coverage (NC), Edge Coverage (EC), and Edge-Pair Coverage (EPC). These coverage metrics are adopted from graph coverage metrics proposed by Ammann and Offutt [[]Chapter 7]ammann2016introduction.

2.1.2.1 Graph Notation

For ease of exposition, we expand on the traditional definition of a DFA. In this work, a DFA graph $G = \{N, N_0, N_m, N_e, E\}$ where: N is the set of all nodes, N_0 is the initial node, N_m is the final matching/accept node, N_e is the final failing/error node, and E is a set of all edges. For the DFAs in this work, there is only one initial state, one accept state, and one error state.[1]

The states in a DFA are the nodes $N = \{n_0, n_1, \ldots, n_k\}$. For any two nodes n_1 and n_2 such that $\{n_1, n_2\} \subseteq N$, if there is a transition from n_1 to n_2 in DFA, then the edge $\overrightarrow{n_1 n_2} \in E$; the start and end-state of the path may be the same node, as is the case of self-loops. Edge pairs are defined by paths of length two in the DFA. For example, if $\{\overrightarrow{n_1 n_2}, \overrightarrow{n_2 n_3}\} \in E$, we denote the edge pair as $\overrightarrow{n_1 n_2 n_3}$. In the case of self-loops, $\overrightarrow{n_2 n_2 n_2}$ is also a valid edge-pair.

Given an input string and a regular expression, the initial node N_0 is visited first. Transitions are taken as each character in the string is consumed. The result of the matching process ends in either the accept node N_m or the error node N_e. In standard DFAs, a traversal can end in any node. However, the DFA generation algorithm used in this work is based on the RE2 tool, which always ends processing in an explicit matching/accept (N_m) or error (N_e) state. In this tool, given a regular expression and an input string, the input string is interpreted as a byte stream, with byte `[256]` added to the end to mark the end of the string. Thus, an input string "2" would be interpreted as `[50 256]` and the input string "1001" would be interpreted as `[49 48 48 49 256]`.

As a running example, consider regular expression $R =$ `\d+` and graph G in Figure 2.3. In G, $N = \{0,1,2,3,E\}$, $N_0 = 0$, $N_m = 3$, $N_e = e$, $E = \{\overrightarrow{0e}, \overrightarrow{01}, \overrightarrow{12}, \overrightarrow{13}, \overrightarrow{22}, \overrightarrow{23}, \overrightarrow{3e}\}$, and $EP = \{\overrightarrow{012}, \overrightarrow{013}, \overrightarrow{122}, \overrightarrow{123}, \overrightarrow{13e}, \overrightarrow{222}, \overrightarrow{223}, \overrightarrow{23e}\}$. Edges *0-9* cover bytes `[48-57]`, and edges *not 0-9* cover the byte ranges `[0-47][58-256]`; we use the decimal representation to improve clarity.

At this point, we note that this is not the smallest DFA for the regular expression `\d+`. As the same tool is used for the construction of all the DFAs, any impact of the DFAs not being minimal (e.g., extra nodes or edges compared to the minimal representation) is distributed throughout the whole data set and consistent across all experiments. While we refer to RE2 [28] for full details of the DFA construction, though some intuition is provided in Section 3.1.2.2.

[1] In a FullMatch DFA (see Section 3.1.1), there could be several matching nodes, and only one accept. We simplified to use only one accept state.

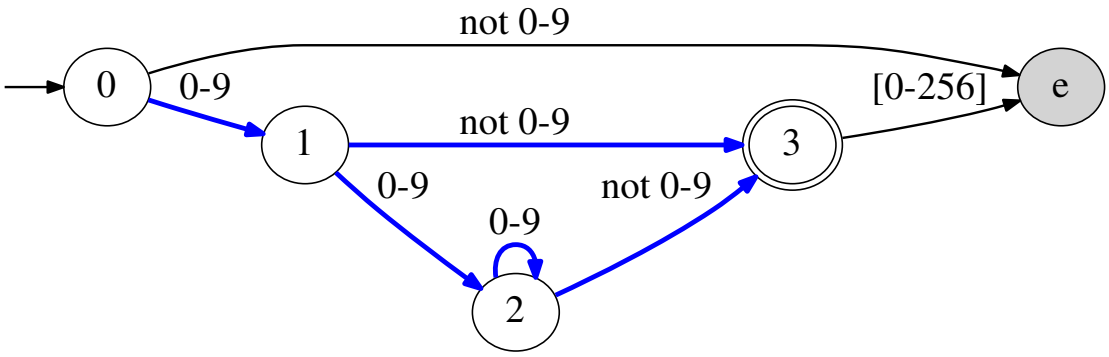


Figure 2.3 Full-match DFA from RE2 [28] for the regular expression `\d+`. RE2 interprets every string as a byte stream; the range of bytes is `[0-256]` where `[256]` is added to mark the end of a string. Thus, the input string "2" would be represented as `[50 256]` and traverse the following path: $0 \rightarrow 1 \rightarrow 3$. The edges marked *0-9* represent the byte range `[48-57]`; edges *not 0-9* represent the byte ranges `[0-47]` `[58-256]`.

2.1.2.2 Coverage Criteria

Given a set of strings S and a DFA G, for all $n \in N$, we mark n as *covered* if n is visited during the processing of some $s \in S$. Similarly, edges $e \in E$ and edge-pairs $ep \in EP$ are marked as *covered* if they are traversed during the processing of some $s \in S$. The sets of covered nodes, edges, and edge-pairs are denoted N_{cov}, E_{cov}, and EP_{cov}, respectively. These sets are aggregated over all $s \in S$.

As defined in prior work [2], we adopt coverage definitions for node coverage (NC), edge coverage (EC), and edge-pair coverage (EPC) as follows:

Definition 1 (Node Coverage %) $NC = 100 \times \frac{|N_{cov}|}{|N|}$

Definition 2 (Edge Coverage %) $EC = 100 \times \frac{|E_{cov}|}{|E|}$

Definition 3 (Edge-Pair Coverage %) $EPC = 100 \times \frac{|EP_{cov}|}{|EP|}$

To illustrate the coverage levels, consider the graph G for the regular expression `\d+` in Figure 2.3 and the string s_0 ="2" with $S = \{s_0\}$. Traversing G visits $0 \rightarrow 1 \rightarrow 3$ (recall that "2" is interpreted as the byte stream [50 256]). Node 3 is the accept node, which denotes that the regular expression matches the input string (i.e., $s \in L(R)$). During the traversal of G, nodes $\{0, 1, 3\}$ are visited, meaning that $N_{cov} = \{0, 1, 3\}$, $E_{cov} = \{\overrightarrow{01}, \overrightarrow{13}\}$, and $EP_{cov} = \{\overrightarrow{013}\}$. The coverage levels for `\d+` by input strings $S = \{s_0\}$ are: $NC = 60\%(3/5)$, $EC = 28.6\%$ (2/7), and $EPC = 12.5\%$ (1/8).

Next, consider adding the string s_1 ="1001", which is interpreted as the byte stream `[49 48 48 49 256]`. Now, $S = \{s_0, s_1\}$. Traversing G on s_1 traverses the following path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 3$, adding node 2 to N_{cov}, edges $\overrightarrow{12}$, $\overrightarrow{22}$, and $\overrightarrow{23}$ to E_{cov}, and edge-pairs $\overrightarrow{012}$, $\overrightarrow{122}$, $\overrightarrow{222}$, and $\overrightarrow{223}$ to EP_{cov}. As a result, the coverage levels for the regular expression `\d+` by input strings $S = \{s_0, s_1\}$ are: $NC =$ 80% (4/5), $EC = 71.4\%$ (5/7), and $EP = 62.5\%$ (5/8).

As an example of a non-matching string, let s_2 = "u", which is interpreted as the byte stream `[117 256]`. The path traversed in G is $0 \rightarrow e$; after reaching e, the processing stops. Node e is added to N_{cov}, edge $\overrightarrow{0e}$ is added to E_{cov}, and there is no change to EP_{cov}. Considering $S = \{s_0, s_1, s_2\}$, the combined coverage levels are: $NC = 100\%$ (5/5), $EC = 85.7\%$ (6/7), and $EPC = 62.5\%$ (5/8).

For another example of a non-matching string, let s_3 = "100u", which is interpreted as the byte stream `[49 48 48 117 256]`. The path traversed in G is $0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow e$. While this input visits all nodes in G, $NC = 100\%$ already, so no nodes are added to N_{cov}. Edge $\overrightarrow{3e}$ is added to E_{cov}, edge-pair $\overrightarrow{23e}$ is added to EP_{cov}. Considering $S = \{s_0, s_1, s_2, s_3\}$, the combined coverage levels are: $NC = 100\%$ (5/5), $EC = 100\%$ (7/7), and $EPC = 75\%$ (6/8).

For each coverage metric, we compute coverage over the entire set of input strings, *total*, and two subsets: *success*, and *failure*. The numbers reported in this section are for the *total* set of input strings, that is, $S = \{s_0, s_1, s_2, s_3\}$. After, we split the input strings into those that terminate in an accept state in N_m, which we call S_{succ}, and those that terminate in the error state N_e, which we call S_{fail}. With this example, $S_{succ} = \{s_0, s_1\}$ and $S_{fail} = \{s_2, s_3\}$.

Table 2.1 presents a summary of the coverage levels for each set of input strings. Achieving 100% for any of the coverage metrics is infeasible for S_{succ} alone because the error state e will never be reached, missing that node and the edges leading to it. In this example, EC for S_{succ} is 71.4% while EC for S is 100%.

Achieving 100% coverage for EPC is the most difficult, but it is possible in this example. The missing edge-pairs are computed by $EP \setminus EP_{cov} = \{\overrightarrow{123}, \overrightarrow{13e}\}$. Two additional input strings can lead to 100% EPC. Input "1u" would be interpreted as the byte stream `[49 117 256]` and traverses the path $0 \rightarrow 1 \rightarrow 3 \rightarrow e$, hence covering $\overrightarrow{13e}$. Input "11u" would lead to byte stream `[49 49 117 256]`, traverse the path $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow e$ and cover $\overrightarrow{123}$.

Note that it is possible to have a DFA which is simply two nodes connected by a single edge. Thus, edge pairs may not exist. For this case, we treat edge-pair coverage as identical to edge coverage. Among the 15,096 regular expressions studied in this work, only two regular expressions have this structure.

Table 2.1 Coverage of `\d+`: $S = \{$"2", "1001", "u", "100u0"$\}$, $S_{succ} = \{$"2", "1001"$\}$, and $S_{fail} = \{$"u", "100u"$\}$.

	S	S_{succ}	S_{fail}
NC	100.0%	80.0%	100.0%
EC	100.0%	71.4%	85.7%
EPC	75.0%	62.5%	50.0%

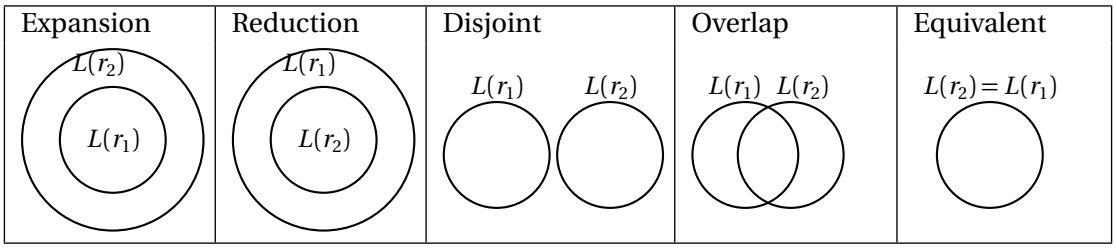


Figure 2.4 Types of semantic evolution from r_1 to r_2

2.1.3 Regular Expression Edit Chain

A series of edited regular expressions over time is called an *edit chain*, which represents the changing history of a regular expression. The *top* of the chain is the most recent version of the regular expression, whereas the *bottom* of the chain is the oldest version. The *length* of the chain is the number of regular expressions in it, and the number of edits in the chain is *length -1*. The bottom of the chain is r_1 and the top is r_k for some length k of the chain. For r_i its *predecessor* is r_{i-1} and its *successor* is r_{i+1}. The similarity is described between *pairs* of adjacent regular expressions in the edit chain, referred to generically as an *edit*. More generally, we compare an edited regular expression r_i that evolved into its successor, r_{i+1}.

For a running example in this section, consider the regular expressions, r_1 and r_2:

r_1 =`caa?a?b`

r_2 =`c{0,2}aa?b`

The languages of r_1 and r_2 are both finite, and enumerated below, with overlapping strings between $L(r_1)$ and $L(r_2)$ bolded for clarity:

$L(r_1) = \{$**"cab"**, **"caab"**, "caaab"$\}$

$L(r_2) = \{$"ab", "aab", **"cab"**, **"caab"**, "ccab", "ccaab"$\}$

2.1.4 Regular Expression Similarity Metrics

Considering the syntax between r_1 and r_2, there are clear similarities and differences. Considering the overlapping semantics between languages, there are also clear similarities and differences. Here, we first describe how these similarities and differences are measured syntactically and semantically and then describe how feature changes are represented in the feature vector. In the end, we provide our implementation details and limitation discussion.

2.1.4.1 Syntactic Similarity

The syntactic similarity is measured by the distance between the literal representation of the strings. We followed the measurement of edit distance used in the work of the automatic generation of regular expressions with genetic programming [12] and thus chose Levenshtein distance.

Levenshtein distance [64] measures the syntactic distance between two strings by counting the number of character insertions, deletions, and substitutions needed to transform one regular expression into the other. In the example with r_1 and r_2, the absolute Levenshtein distance between them is six. This is computed as one replacement (i.e., a → {), four additions (i.e., **0,2}**) and one removals (i.e., ?) as follows: c~~a~~{**0,2}**a~~?~~a?b

2.1.4.2 Semantic Similarity

Semantic similarity measures the amount of overlap between $L(r_1)$ and $L(r_2)$. Adopting the approach from the work of Chapman and Stolee [23], we measure approximate similarity by looking at the overlap in matching strings between two regular expressions. We define the evolution of r_1 to r_2 by measuring three sets of strings, A, B, and C, which are represented in the Venn Diagram in Figure 2.5. Formally:

$A = L(r_1) \setminus L(r_2)$ [2],
$B = L(r_1) \cap L(r_2)$, and
$C = L(r_2) \setminus L(r_1)$.

There are five types of relationships between $L(r_1)$ and $L(r_2)$, which are shown in Figure 2.4[3]. When $L(r_1)$ is a strict subset of $L(r_2)$, then more matching strings are added to the language; this is called *expansion*. When $L(r_1)$ is a strict superset of $L(r_2)$, this means that the language was reduced during evolution; we call this *reduction*. If there is no overlap between $L(r_1)$ and $L(r_2)$, then they are *disjoint*.

[2] $X \setminus Y = \{x \in X : x \notin Y\}$

[3] Prior work used mutation for test case generation and defined similar relationships between regular expressions, omitting disjoint [5].

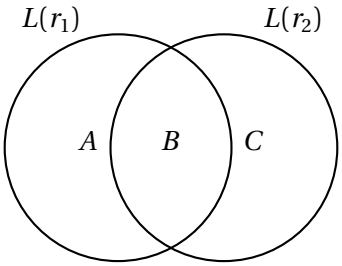


Figure 2.5 Notation for sets A, B and C to compute the semantic evolution from r_1 to r_2.

If $L(r_1)$ and $L(r_2)$ are the same, they are called *equivalent*. The final condition is a *overlap* when $L(r_2)$ and $L(r_1)$ share some strings, but some are removed and some are added during evolution.
Using the example:

$A = \{$"caaab"$\}$,

$B = \{$**"cab"**, **"caab"** $\}$, and

$C = \{$"ab", "aab", "ccab", "ccaab"$\}$.

Migrating from r_1 to r_2 involved an overlap, where all the strings in A are removed from the matching language, and all the strings in C are added. In this way, r_1 and r_2 are partially overlapped. We measure three metrics pertaining to the semantic evolution of regular expressions: *intersection*, *removal*, and *addition*.

2.1.4.2.1 Intersection

The Intersection between $L(r_1)$ and $L(r_2)$ is computed as:

$$intersection(r_1, r_2) = \frac{|B|}{|A|+|B|+|C|}$$

When the languages are *disjoint*, $|B| = 0$, and so the intersection is likewise 0. When the languages are identical, A and C are empty, so the intersection is 1. In the running example with partially intersecting languages, $\frac{2}{1+2+4} = \frac{4}{7} = 57\%$.

2.1.4.2.2 Removal

This metric describes how much of the language of r_1 is removed in the migration to r_2. Generically, the reduction from r_1 to r_2 is:

$$removal(r_1, r_2) = \frac{|A|}{|A|+|B|}$$

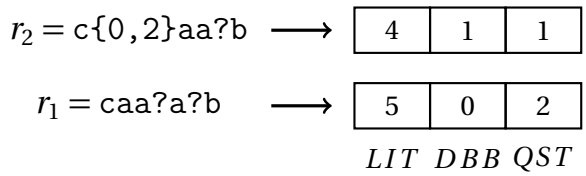


Figure 2.6 Example r_1 and r_2 parsed into feature vectors. LIT = Literal, DBB = Double-Bounded, QST = Questionable

When the r_2 is an *expansion* of r_1, the removal is 0 since $|A| = 0$. When the languages are disjoint, B is empty, so the removal is 1. When the languages are equivalent, A is empty so the removal is 0. In our example of overlap, removal is $\frac{1}{1+2} = \frac{1}{3} = 33\%$ meaning that 33% of the language of r_1 was removed when evolving it to r_2.

We can compute the percentage of the $L(r_1)$ that is retained in $L(r_2)$ by $1 - removal(r_1, r_2)$. In this example, 67% of $L(r_1)$ is carried forward into $L(r_2)$.

2.1.4.2.3 Addition

This metric describes how much of the language $L(r_2)$ is new or added after evolving from r_1. It is computed as:

$$addition(r_1, r_2) = \frac{|C|}{|B| + |C|}$$

When r_2 is disjoint from r_1, this means the addition is 1 since the entire language $L(r_2)$ is new with respect to $L(r_1)$. When r_2 is a subset of r_1, the addition is 0 since nothing is added to $L(r_2)$. In the running example, four strings were added to the language. In terms of the expansion metric, $\frac{4}{2+4} = \frac{4}{6} = 67\%$, meaning that 67% of the strings in r_2 are new.

We can compute the percentage of $L(r_2)$ that is carried forward from $L(r_1)$ by computing $1 - addition(r_1, r_2)$. In this example, 33% of $L(r_2)$ comes from $L(r_1)$.

2.1.4.3 Language Features

The language features used in a regular expression may evolve as the regular expression evolves. In the example from r_1 to r_2, one language feature was added, the double-bounded repetition (i.e., `{0,2}`), or DBB for short. Adopting the abbreviations used in prior work [23], Figure 2.6 shows how the features change using feature vectors. We use the following metrics for feature evolution: F_{remove} is the number of language features appearing in r_1 but not r_2; F_{add} is the number of features in r_2 but not r_1.

In Figure 2.6, the frequencies for LIT, DBB, and QST in r_1 are 4, 1, and 1, respectively. Taking this example, the feature vector of LIT, DBB, and QST is *-1, 1, -1* meaning that from r_1 to r_2 one LIT is removed, one QST is removed, and one DBB is added. Therefore $F_{add} = 1$ and $F_{remove} = 2$.

To measure the most frequently added or moved features, we keep track of each feature in the two metrics above. For DBB, it falls into F_{add} with the value of one. For LIT and QST, they both fall into F_{remove} with the value of one for each of them.

2.1.4.4 Implementation

For the regular expressions, we use an ANTLR-based PCRE parser[4] to extract features used in a regular expression to create a feature vector. We use the class LevenshteinDistance provided in the Java library of Apache Commons Text [5].

For semantic distance, we approximate the language for $L(r)$ by generating strings for r using Rex [104]. The Rex tool analyzes regular expressions using symbolic analysis. When configured as a string generation tool, it aims to generate strings inside $L(r)$. Using Rex, for each regular expression r, we tried to generate $k = 500$ strings which could successfully match r. Considering $L(r)$ could be smaller than k, Rex tried up to five times for each r with different seeds so that the accumulated number of generated strings can get to k. But if $|L(r)| < k$, the total generated strings are equal to $|L(r)|$.

For the semantic comparison between r_1 and r_2, the edit is directional, from r_1 to r_2. To give the total set of all strings in both languages, we compute the union set $L = L(r_1) \bigcup L(r_2)$. This removed duplicates so only unique strings are considered for the analysis (i.e., if the string $s \in L(r_1)$ and $s \in L(r_2)$ for the same string s, we want to count this once). Then, we matched each string in L with r_1 and r_2 separately, to form sets A, B, and C, as described previously.

2.1.4.5 Limitations

For the semantic analysis, we note that a regular expression r could be invalid in Java regular expression syntax, or Rex may not be able to generate strings for r due to feature limitations. If either r_1 or r_2 is invalid in Java or contains features beyond the scope of Rex capabilities, we skipped semantic analysis. For example, an edit chain of length five r_1, r_2, r_3, r_4, r_5 should have four semantic comparisons. If r_3 is invalid in Java syntax, this edit chain reduces to two comparisons, between r_1 and r_2, and between r_4 and r_5.

[4] https://github.com/bkiers/pcre-parser

[5] https://commons.apache.org/proper/commons-text

For infinite languages, and languages larger than the upper bound on strings generated (i.e., $k = 500$), the approach described in Section 2.1.4.2 describes an optimistic approximation of the actual similarity between regular expressions. This is a relatively common scenario due to the common presence of the KLEENE star and ADD operator in a majority of the regular expressions (see Table 4.4). The only sound classification is overlap; all other classifications might be the result from ignoring one or two words in one (or both) of the languages. We depend on Rex for the string generation to determine similarity, and thus inherit any of its biases.

2.2 Motivation

2.2.1 DFA as Representation of Regular Expressions

This section talks about the regularity of regular expressions and the clear visualization of using DFA to describe regular expression equivalence.

2.2.1.1 How Regular Are Regular Expressions?

Regular expressions in source code can contain non-regular features, such as backreferences. An example is the regular expression `([a-z]+\1)`, which matches a repeated word in a string, such as “appleapple". Building a DFA is not possible for this since this regular expression is non-regular. For regular expressions in source code that are indeed regular, we can build DFAs and measure coverage based on a test suite. Here, we are testing how many of the regular expressions in the wild are truly regular.

We explore an existing and publicly available dataset of 13,597 regular expressions scraped from Python projects on GitHub. To test for regularity, we use an empirical approach since the ability to build a DFA from a regular expression implies that it is regular [95]. Of the 13,597 Python regular expressions, 13,029 (95.9%) are regular in that we were successful in building DFAs for each using the RE2 [28] regular expression processing engine. For the remaining 568, we investigated each by hand. One regular expression was removed because its repetition exceeds the RE2 limits. While it may indeed be regular, to be conservative, we mark it as non-regular. An additional 81 contained comments within the regular expressions, which are unsupported in RE2, so these were also assumed to be non-regular; 128 contained unsupported characters. The remaining 368 were non-regular as they contained backreferences.

In the end, with nearly 96% of the regular expressions being regular (as a low estimate), we conclude that most regular expressions found in the wild are regular and thus can be modeled with DFAs.

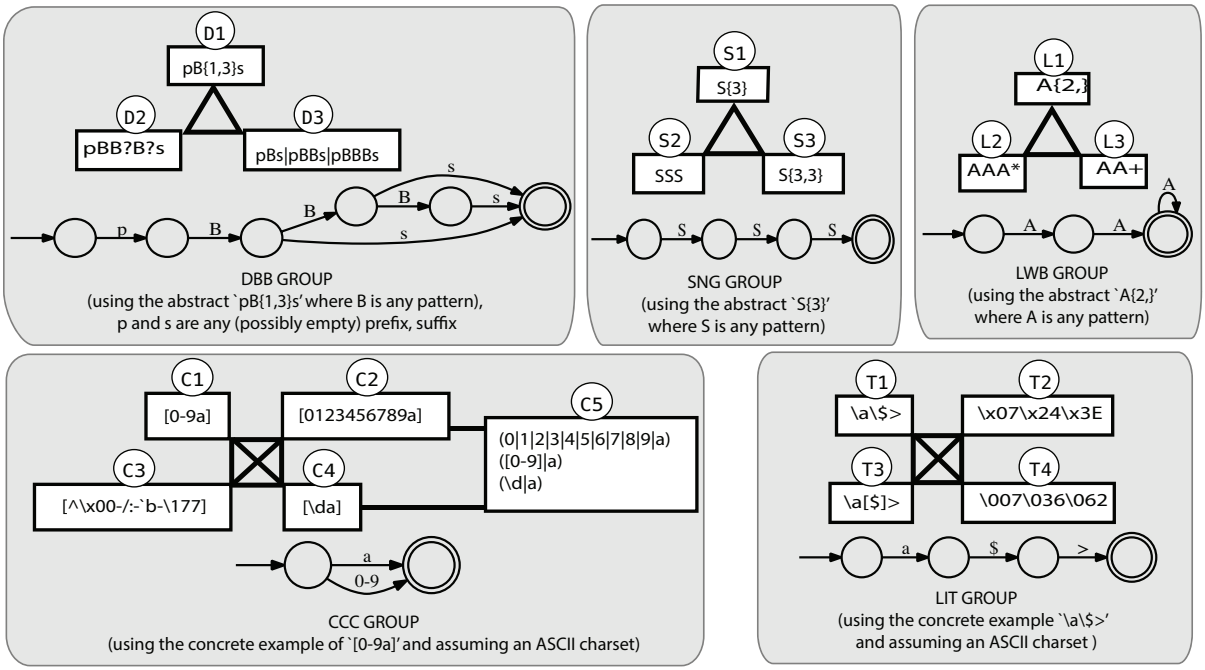


Figure 2.7 Types of equivalence classes based on language features. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal. We use concrete regexes along with their Deterministic Finite Automaton (DFA) in the representations for illustration. However, the A's in the LWB group (or B's in DBB group, S's in SNG group, and so forth) abstractly represent any pattern that could be operated on by a repetition modifier (e.g., literal characters, character classes, or groups). The same is true for the literals used in all the representations.

2.2.1.2 DFA for Regular Expression Equivalence

The minimal DFA can be used to measure the equivalence of two regular expressions. Figure 2.7 shows the five types of equivalence classes in grey boxes and examples of behaviorally equivalent representations in white boxes with identifiers in white circles. Although these regular expressions are different syntactically, they are the same in the terms of the minimal DFA.

2.2.2 Regular Expression Testing

A regular expression is a sequence of characters that defines a search pattern. The set of strings matched by the regular expression is a language. That is, a regular expression R represents a language $L(R)$ over an alphabet Σ, where $L(R)$ is a (possibly infinite) set of strings. For a given language, there are many regular expressions that can describe it. A regular expression can be represented as a string of tokens, a finite state automaton in deterministic (DFA), or a non-deterministic (NFA) form.

In this work, we explore test coverage metrics over the DFA representing a regular expression. This requires three informal explorations to ensure feasibility and assess the potential impact. First, we explore the potential of building DFAs from regular expressions by analyzing regular expressions collected from an existing Python dataset [23] and testing them for regularity [95]. Second, we show intuitively how existing coverage metrics are insufficient. Third, to motivate the structural coverage metrics, we explore whether faults can lie along untested paths in a DFA.

2.2.2.1 Limitations of Code Coverage

Statement coverage requires the regular expression to be invoked at least once. If the regular expression call site appears in a predicate, branch coverage requires that the regular expression is tested with at minimum two strings, one in the language of the regular expression and one not. However, these metrics .

In this work, we posit that code coverage metrics [[]Chapter 2]ammann2016introduction [65, 86, 114] such as statement, branch, and path, are too coarse-grained for regular expressions. Statement coverage requires that the code containing the regular expression is reached, leading to a minimum of one test input for the regular expression. If the regular expression is in a statement where the control flow is dependent on the matching outcome, branch coverage requires that the regular expression have at least two inputs, one that evaluates to true and another that evaluates to false.

Consider the following Java code snippet. The *call site* for method `Pattern.matches` is on line 1. The regular expression is `-d|--data`.

```
if(Pattern.matches("–d|––data",strInput)){
        System.out.println("YES");
        ...
}else{
        System.out.println("NO");
        ...
}
```

Statement coverage of the regular expression requires that line 1 is executed and branch coverage requires two test inputs, one to cover the `true` branch and one to cover the `false` branch. Using coverage metrics based on the DFA representation of the regular expression, on the other hand, would require 1) each branch to be covered, and 2) each *case* in the regular expression, "-d" and "--data", to be covered. Such metrics measure test coverage of the regular expression's control flow (i.e., the DFA) just like branch coverage measures test coverage of source code's control flow graph.

Existing tools and techniques can direct test input generation toward areas of untested paths. One

technique among these is symbolic execution [3, 19, 42, 55, 66], and Rex [104] has been developed for symbolic analysis of regular expressions. However, Rex focuses solely on the matching behavior [104], which limits its ability to cover the false branch in the Java example above. Brics [73] and Hampi [53, 54] similarly only generates passing strings. While useful, there are no guarantees of structural coverage.

2.2.2.2 DFA Coverage Example

Bug reports related to regular expressions abound. A search for "regex OR regular expression" in GitHub yields over 555,000 issues, with 22% of those still being open. One in particular illustrates how coverage metrics on the DFA could have brought a particular bug to the developer's attention sooner. This bug report[6] describes an issue with the regular expression `\d+\.d+` in the NAR plugin for Maven. Figure 2.8 shows the DFA of this regular expression built using RE2 [28], and we take this opportunity to describe the DFA notation used throughout this paper[7].

Node 0 is the start-state, indicated by the incoming arrow. Nodes with double-circles are accept states, such as Node 4. Node e is the error state, denoting a mismatch. The edges are labeled with transitions, often using syntactic sugar for ease of interpretation. The edge $\overrightarrow{01}$ is traversed when a digit from `0-9` is read. If any other character is read at Node 0, (i.e., `not 0-9`), edge $\overrightarrow{0e}$ is traversed. There is a self-loop on Node 1 for digits `0-9`. If the period character is read from Node 1, then edge $\overrightarrow{12}$ is traversed.

In RE2, when reading an input string, byte `[256]`, is added as a text-end marker. For example, the input string "0.0" is transformed to the byte stream `[48 46 48 256]`, as `[48]` is the byte for '0', `[46]` is for '.', and `[256]` marks the end of the string. Byte [256] is matched on edges '`[0-256]`', '`not 0-9`', '`not d`', or '`any except 0-9 and .`'.

The bug report mentions that the regular expression `\d+\.d+` is buggy and the patch adds an escape before the second d, `\d+\.\d+`. The intended behavior is to match input strings with one or more digits, followed by a period, followed by one or more digits.

In this work, the structural metrics could reveal this fault. With the DFA in Figure 2.8, when Node 3 is reached, the fault may be revealed. Input "0.d" traverses $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ and ends in an accept state, when it should fail. However, input "0.d3" traverses $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow e$ and ends in an error state, as expected. Covering edge $\overrightarrow{2e}$ may also reveal the fault; input "2.3" traverses $0 \rightarrow 1 \rightarrow 2 \rightarrow e$ and ends in and error state, when it should be accepted. Requiring coverage of all feasible nodes and edges could have revealed this fault in the regular expression.

[6] `https://github.com/maven-nar/nar-maven-plugin/issues/228`

[7] The regular expression in the bug is triggered by `Matcher.find()` with a ManyMatch DFA. For simplicity, we show the FullMatch DFA, a subgraph of the ManyMatch.

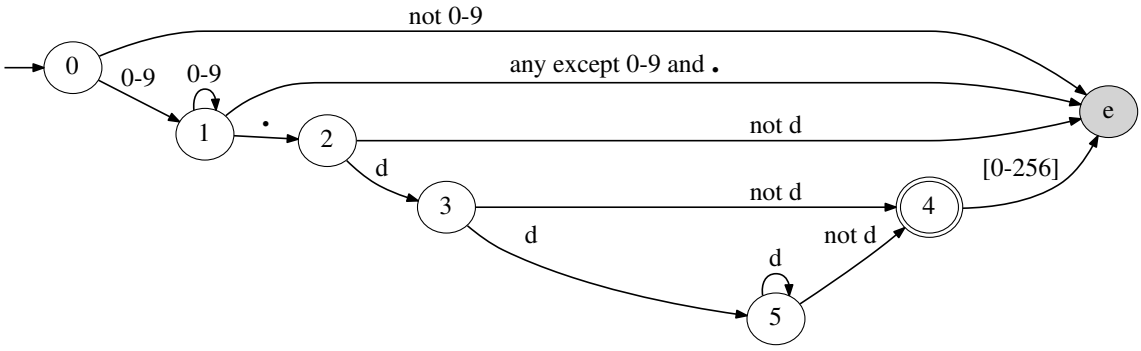


Figure 2.8 Full-match DFA for regular expression: `\d+\.d+`

As with code coverage, uncovered artifacts alert the programmer to untested behavior. Such coverage information can indicate that a regular expression is not well tested and for some inputs it may not behave as intended, as is the case here.

2.2.3 Regular Expression Evolution

While we show later that a majority of regular expressions do not evolve (see Section 4.2), prior work indicates that regular expression bug reports abound [**Spishak:2012:TSR:2318202.2318207**], and that regular expressions are under-tested [107]. In exploring bug reports on regular expressions, we find evidence that regular expressions often evolve through refactoring (i.e., the language is the same), and that edits tend to involve many different mutation operators [6] at once. This provides evidence that understanding evolution is important for test generation and repair.

2.2.3.1 Regular Expression Equivalence

A regular expression is a language to describe a set of strings it can match, and there is usually more than one way to express it. For example, a digit can be described as a character range `[0-9]` and can also be described using shortcut `\d`. A word character expressed in `\w` is equivalent to `[A-Za-z_0-9]`. Sometimes, bug reports require resolution through semantics-preserving transformations that improve performance rather than edits to regular expression behavior.

In one bug report[8], all regular expression capturing groups are changed to non-capturing groups (e.g., `(\r\n|\r|\n|\f)` to `(?:\r\n|\r|\n|\f)`) to avoid back trackingïijŇ so the scope of the regular expression is not changed by the mutation. In another example[9], regular expression

[8]https://github.com/SonarSource/sonar-css/pull/110/files
[9]https://github.com/CaszGamerMD/NootSpeak/pull/14/files

`(\\W|\\d|_)` is changed to `[^A-Za-z]` for better regular expression readability. `\\W` is equivalent to `[^A-Za-z_]`, `\\d` is equivalent to `[0-9]`, the alternation of `(\\W|\\d|_)` is the union of `[^A-Za-z_]`, `[0-9]`, and '_', which is `[^A-Za-z]`.

These provide evidence that not all edits modify the behavior, and thus not all testing efforts should focus on matching behavior, but rather on performance metrics.

2.2.3.2 Regular Expression Feature Changes

The state-of-the-art literature on fault-injection [5] and fixing regular expressions [7] uses simple faults. However, fixing regular expressions in bug reports often involves more than one features and the same fault can be applied multiple times.

In the first bug report mentioned above, another regular expression is changed from

`[+|-]?\\d*\\.?\\d+([a-z]+|%)?`

to

`[+|-]?+(?:\\d++(?:\\.\\d++)?+|\\.\\d++)(?: [a-z]++|%)?+`.

In this example, greedy quantifiers (e.g., `[+|-]?` and `\\d+`) are changed to possessive quantifiers (e.g., `[+|-]?+` and `\\d++`) seven times, the capturing group is changed to a non-capturing group. Understanding the types and frequencies of edits in a regular expression's evolution can shed light on how to guide fault-injection test generation and repair.

CHAPTER

3

REGULAR EXPRESSION TESTING

In Chapter 3, which was published at FSE 2018 [107], we present our study of regular expression coverage by describing our implementation, artifacts, results, and analysis. Finally, we discuss the validity of this piece of work and draw a summary at the end of this chapter.

3.1 Study

Applying the coverage metrics defined in Section 2.1.2.2 to regular expressions from the wild requires (1) instrumentation to capture the regular expressions and strings matched against them (Section 3.1.1), (2) a tool to measure coverage given a regular expression and a set of strings (Section 3.1.2), and (3) a large corpus of projects with regular expressions and test suites that execute the regular expressions (Section 3.1.3). To address RQ2, we use the Rex [104] tool to generate input strings for the regular expressions in our study (Section 3.1.4).

3.1.1 Instrumentation

This section describes our approach to collecting regular expressions from GitHub projects and the strings evaluated against the regular expressions during testing.

3.1.1.1 Instrumented Functions

There are different types of matching between a regular expression and a string. The Java function *Pattern.matches* requires the regular expression to match a string from its beginning to its end; Python's *re.match* requires the regular expression to match a string only from its beginning, not necessarily match to the end of the string; and the C# function *Regex.Match* requires the regular expression to match only a substring of the input string. These are called *FullMatch*, *FirstMatch*, and *ManyMatch*, respectively. In this work, we consider only *FullMatch* matches and related functions in Java projects. The related functions for FullMatch in Java are:

- `java.lang.String.matches(String regex)`
- `java.util.regex.Matcher.matches()`
- `java.util.regex.Pattern.matches(String regex, CharSequence input)`

In these functions the entire string is required to match the regular expression [38]. Thus, a regular expression with end-point anchors (i.e., `^` and `$`) and without are no different.

3.1.1.2 Bytecode Manipulation

Our instrumentation is built on top of the Java bytecode manipulation framework Javassist [25], which can dynamically change the class bytecode in the JVM. All the projects are run in jdk1.7. We intercepted FullMatch function invocations in Java. For each invocation, we collect information about the regular expression itself, its location in the code, and any strings matched against it during test suite execution. These strings matched against the regular expression are referred to as the *input strings* or *test inputs* (i.e., *S* from Section 2.1.2.2).

Since a regular expression may also appear in third-party libraries, we use the Java Reflection API to additionally record the caller function stack of the instrumented methods and extract the file name, class name, and method name of their caller methods. This allows us to identify when the regular expression being executed is from the system under test and when it is from a third-party library. We are dependent on two libraries during the experimentation, `org.junit` and `org.apache.maven`. Because Maven uses regular expressions to automate unit tests, all recorded regular expressions whose test classes are from package `org.junit.runner.*` or from package `org.apache.maven.plugins.*` are treated as regular expressions from third-party libraries and dropped.

3.1.1.3 Recorded Information

We illustrate the recorded information for the regular expression `((:\w+)|\*)` and a string "one-name" from a project used in our study[1]:

- system under test: `mikko-apo/KiRouter.java`
- test file: `SinatraRouteParser.java`
- test class: `kirouter.SinatraRouteParser`
- test method: `compileRoutePattern`
- call site: `line 38`
- regular expression: `((:\w+)|\*)`
- input string:"one-name"

In Section 2.2.2.1, the regular expression in the *call site* on line 1 is hard-coded. However, often the regular expression is passed as a variable, allowing multiple regular expressions to be observed during testing at the same call site (i.e., there is a many-to-one relationship between regular expressions and call sites). When this occurs, the recorded information is the same as above, except *regular expression* and *input string* would be different.

3.1.2 Coverage Analysis

This section details the construction of DFAs for computing coverage. Given a regular expression R and a set of input strings S, we first build a DFA for $L(R)$ and then track the nodes and edges visited in the DFA during pattern matching with each string $s \in S$. We built our infrastructure on top of RE2 [28], a regular expression engine similar to those used in PCRE, Perl, and other languages.[2]

3.1.2.1 DFA Types.

Given a regular expression and an input string to match, we could build multiple DFAs with different considerations. We could build a static DFA with a regular expression alone or build a DFA on-the-fly (dynamic DFA) considering both a regular expression and an input string. For the same regular expression, different input strings will yield different dynamic DFAs. We can also build a Forward

[1] `https://github.com/mikko-apo/KiRouter.java`

[2] Original RE2 at `https://github.com/google/re2` and modified code at `https://github.com/wangpeipei90/re2`

DFA and Backward DFA depending on the direction of scanning the regular expression. These decisions come with various performance tradeoffs during the matching process. For the purpose of our work, we need each DFA to be built consistently regardless of the input string, so we use a static DFA. We chose the forward direction as it seems the most natural for interpretation.

3.1.2.2 DFA Mapping

When matching an input string to a regular expression, RE2 builds a dynamic DFA. However, our coverage is computed over a static DFA. This requires mapping to aggregate coverage of a regular expression given multiple input strings.

For a single regular expression, different input strings often result in different dynamic DFAs. To make matters worse, these DFAs have inconsistent naming of their states. Therefore, to calculate the coverage of a certain regular expression based on the same DFA, these dynamic DFAs have to be mapped to the same static DFA, and then coverage is computed on the static DFA. This is usually straightforward as the dynamic DFA is always an isomorphic subgraph of the static DFA and N_0, N_e and N_m are consistently labeled in the static and dynamic DFAs.

Consider the regular expression `\d+` and $S = \{s_0, s_1, s_2, s_3\}$ from Section 2.1.2.2 where s_0 = "2", s_1 = "1001", s_2 = "u", and s_3 = "100u". Figure 3.1a shows the static forward DFA. The dynamic DFAs corresponding to these four inputs are shown in Figure 3.1b, Figure 3.1c, Figure 3.1d, and Figure 3.1e, respectively. Blue arrows are used to identify the visited edges in the dynamic DFAs when the input string is a match. Red edges are used to identify the visited edges when the input string is not a match. Note that in Figure 3.1, for simplicity, we have already mapped and renamed the nodes in the dynamic DFAs according to the static DFA.

3.1.2.3 RE2 Limitations and Modifications

We enlarged the default memory size of a cached DFA so that it could accommodate large DFA graphs. Due to Linux environment limitations, string length is limited to 131,072 and null type is not allowed. These situations are rare, impacting < 1% of the collected regular expressions (see Section 3.1.3).

3.1.2.4 Coverage Calculation

With the consistent naming between a static DFA and a dynamic DFA, all nodes, edges, and edge pairs in the latter are regarded as visited nodes, edges, and edge pairs of the former. That is, a node only appears in a dynamic DFA when it is visited during matching; these can be thought of as *just-in-time* DFA constructions in the context of a string to match. The coverage metrics from

Section 2.1.2.2 are computed over the static DFAs, aggregating over all input strings observed during testing.

3.1.3 Artifacts for RQ1

RepoReaper [75] provides a curated list of GitHub projects with the ability to sort based on project properties, such as the availability of test suites, which is a pre-requisite for our study. We focused on Java projects due to its popularity on GitHub and the availability of a bytecode analysis framework for instrumentation.

3.1.3.1 Project Selection

In December 2017, we selected the 136,196 Java projects whose unit test ratio reported in RepoReaper is greater than zero. Because the density of regular expressions in projects tends to be low, we automated project builds and test suite execution in order to collect sufficient data. As such, we require all projects we analyze to use *maven* and *junit* to automatically run unit tests. We identified 13,637 Java Maven projects that used Java pattern matchings functions mentioned in Section 3.1.1. From those, we selected the ones that could be successfully compiled and tested in Maven, leaving 5,691 projects on which we attempted to collect coverage information.

3.1.3.2 Regular Expression and Test Input Collection

To collect the input strings for each regular expression, we instrumented each project and executed the test suites. We changed the configurations of the plugin *maven-surefire-plugin* by adding -*javaagent* argument to *argLine* so that when Maven forks a VM to run the unit tests the VM can load the instrumentation library. Each project module that runs tests executes in different VMs and the information is recorded in different files. *testFailureIgnore* is configured to *true* so that one test failure does not affect the other tests, allowing us to record as many regular expressions in the project as possible.

Of the 5,691 projects with Maven, test suites, and pattern matching functions, 1,665 projects contained 24,058 regular expressions executed by test suites. The remaining projects contained regular expressions *not* executed by the test suites, and thus could not be instrumented.

3.1.3.3 Filtering Out Third-Party Regular Expressions

FullMatch invocations from Maven and JUnit have been removed already at this point, but other third-party libraries also use regular expressions. We can detect this by looking for syntactically

Table 3.1 Description of 1,225 Java projects analyzed. All numbers are rounded to nearest integer except the test ratio and KLOC.

Attributes	mean	25%	50%	75%	90%	99%
Tested Regular exp.	12	1	3	7	18	99
Stars	35	0	1	5	30	833
Test ratio	0.238	0.096	0.210	0.346	0.482	0.691
KLOC	55.4	2.0	6.7	25.1	86.7	951.0
Size (KB)	19,062	286	1,079	6,449	33,163	249,915
Call sites	15	2	4	10	31	211
Tested call sites	3	1	2	3	6	20
Reg. exp./tested site	5	1	1	2	5	38

Table 3.2 Description of 15,096 regular expressions analyzed for RQ1. All numbers are rounded to nearest integer

Attributes	mean	25%	50%	75%	90%	99%
Nodes ($\|N\|$)	144	12	28	70	324	939
Edges ($\|E\|$)	565	24	75	212	938	2,813
Edge pairs ($\|EP\|$)	2,115	25	99	414	1,647	16,850
Regular exp. len.	31	13	18	39	67	161
# Input strings ($\|S\|$)	60	1	2	7	27	662
Input string len.	125	9	17	63	318	948

identical regular expressions with invocations on the same file, same class, same method, but in different GitHub projects. If the number of projects is larger than one, then it is regarded as a third-party regular expression, and all records related to the same stack information are dropped. A limitation of this approach is that we miss some third-party invocations that are only present in a single project. Given the large number of projects analyzed, the impact of this is likely to be small.

We identified 8,496 regular expressions as coming from third-party libraries. The resulting dataset contains 1,256 projects and 15,562 regular expressions, 14,040 of which are syntactically unique.

3.1.3.4 RE2 Analysis

Since RE2 only supports the most common regular expression language features, we filtered out the regular expressions containing advanced and non-regular features. RE2 failed to construct DFAs for

457 regular expressions, leaving 15,105 regular expressions spread across 1,225 projects.[3] The RE2 limitations on input string length and the null byte affected 56 regular expressions and 191 input strings, and nine of the 56 regular expressions are removed from coverage analysis because their only input string is dropped.

These 1,225 projects contain 18,426 call sites of the instrumented functions. Only 3,093 call sites are executed by the test suites; the same call site can have many regular expressions in the case of dynamically generated regular expressions.

The final dataset used for analysis contains 1,225 projects, 3,093 call sites, 15,096 regular expressions, of which 13,632 are syntactically unique. As the same regular expression can appear in multiple projects, or multiple places in the same project, all are retained since each is potentially tested differently. These 15,096 regular expressions are executed by 899,804 test inputs.

3.1.3.5 Project Characteristics

Table 3.1 describes the 1,225 projects in terms of *Tested regular exp.* (numbers of tested regular expressions per project), *stars* (a measure of popularity), *KLOC* (lines of code in thousand), *size* (size of the repository in KB), *test ratio* (the ratio of number of lines of code in test files to the total lines of code in repository, as reported by RepoReaper), *Call sites* (the number of FullMatch methods in the source code), *Tested call sites* (the number of FullMatch call sites executed by the tests), and *Reg. exp. / tested site* (the number of regular expressions passed to each tested call site). The *mean* column describes the average value for each attribute. Columns *25%, 50%, 75%, 90%*, and *99%* show the distribution of each attribute at 25 percentile, median, 75 percentile, 90 percentile, and 99 percentile, respectively. The average number of tested regular expressions collected per project was 12 with a range of 1 to 2,004.

3.1.3.6 Regular Expression Characteristics

Table 3.2 shows the DFA information for regular expressions. *Nodes, edges, and edge pairs* are the total number of nodes, edges, edge pairs in the DFA graph of a regular expression. The average regular expression is quite large with 144 nodes, though this is skewed as the median is 28 nodes. *Regular exp. len.* measures the length of the string representing the regular expression itself in characters. *# Input strings* is the number of syntactically unique input strings executed by a project's test suite, per regular expression. The average number of syntactically unique test inputs per regular

[3] Assuming all 457 are non-regular, this means over 97% of the regular expressions sampled are regular, echoing findings from the Python analysis in Section 2.2.1.1.

expression is 60, but the median is 2. *Input string len.* shows the lengths of the input strings (i.e., each $s \in S$) in terms of the number of characters.

3.1.4 Artifacts for RQ2

To explore the coverage of regular expressions using tools, we selected Rex [104] due to its high language feature coverage [23].

3.1.4.1 Artifact Selection

We need a set of regular expressions with the following characteristics: 1) are covered by tests; 2) can be analyzed by RE2 for coverage analysis; and 3) can be analyzed by Rex for test input generation. To satisfy 1) and 2), we begin with the dataset from RQ1 of 1,225 projects and 15,096 regular expressions. To satisfy 3), we select all the regular expressions that Rex supports and for which $|S_{succ}| > 0$, since Rex only generates matching strings, leaving 10,155 regular expressions of which 9,063 are syntactically unique.

3.1.4.2 Rex Setup

Rex defaults to *ManyMatch* as opposed to the *FullMatch* behavior of our dataset. To force Rex to treat each regular expression as a full match, we added endpoint anchors (i.e., `^` and `$`) to each regular expression. Because Rex may get stuck in generating input strings for certain regular expressions, we set a timeout of one hour for Rex to generate strings; regular expressions that exceed the timeout are discarded. Of the 10,155 regular expressions in GitHub whose $S_{succ} > 1$, Rex encountered the timeout for only two.

Another complication comes at the intersection of the Rex and RE2 language support; Rex-generated strings must be processed by RE2 for the coverage analysis. For example, the character class "`\s`" in Rex accepts six whitespace characters and RE2 accepts five. In another example, some generated Unicode strings in Rex could not be processed in RE2 because their Unicode encoding in Rex is UTF-16 while RE2 handles Unicode sequences encoded in UTF-8 or Latin-1. To simplify the experiment, we configured Rex to generated strings in ASCII. We also dropped strings which contain unsupported features or characters in either RE2 or Python 3. We also dropped strings which lead to failed matchings and reported the coverage based on successful matchings.

After filtering out all the unsupported regular expressions, our reported coverages by Rex strings in ASCII encoding are based on 7,926 regular expressions of 985 GitHub projects; 7,007 of them are syntactically unique. Table 3.3 shows the attributes of regular expressions of which Rex could generate regular expressions.

Table 3.3 Description of 7,926 regular expressions for RQ2.

Attributes	mean	25%	50%	75%	90%	99%
Nodes ($\|N\|$)	220	13	31	162	618	970
Edges ($\|E\|$)	773	30	97	663	1,468	3,694
Edge pairs	2,422	36	186	1,021	1,999	21,274
$\|S\|$	70	1	2	8	39	961
$\|S_{succ}\|$	34	1	1	2	8	208
Regular exp. len.	29	12	15	31	71	160

3.1.4.3 Input String Generation

For each regular expression R, we use Rex to generate input string sets relative to the size of the matching strings $|S_{succ}|$. We generate input string sets of three sizes: equal to $|S_{succ}|$; equal to $5 \times |S_{succ}|$; and equal to $10 \times |S_{succ}|$. We refer to these experiments as *Rex1M*, *Rex5M*, and *Rex10M*, respectively. For each experiment, we repeated the string generation using the system time as the random seed to encourage diversity among the generated strings. The averages over five runs (*Rex5M* and *Rex10M*) or ten runs (*Rex1M*) for each metric are reported as Rex's coverage of R.

For example, say a regular expression R from GitHub has five input strings; $|S| = 5$. Three of the input strings are matching; $|S_{succ}| = 3$. For this experiment, Rex would generate three strings ten times, then 15 strings five times, then 30 strings five times, totaling $30 + 75 + 150 = 255$ generated strings. For each set of $\{3, 15, 30\}$ strings, NC, EC, and EPC are computed, averaged over $\{10, 5, 5\}$ runs.

In the case of finite languages, Rex may fail to generate sufficient input strings. For example, the total number of matching input strings in ASCII for a regular expression `\d` is ten (i.e., 0-9). If in the repository there are also three matching input strings, Rex could generate three strings ten times, but would fail to generate $5 \times 3 = 15$ strings. The calculation of NC, EC, and EPC are based on the best-effort: for each run of every regular expression, we calculate coverage with input strings up to $|S_{succ}|$ in *Rex1M*, 5x of $|S_{succ}|$ in *Rex5M*, and 10x of $|S_{succ}|$ in *Rex10M*; and coverage of every regular expression is the averages of its coverages over $\{10, 5, 5\}$ runs in *Rex1M*, *Rex5M*, and *Rex10M*. In other words, if Rex failed to generate required number of input strings, the coverage is calculated based on the input strings Rex can generate.

In the ten runs of generating input string sets equal to $|S_{succ}|$ for *Rex1M*, there are 833 regular expressions which have input strings less than $|S_{succ}|$ in at least one run. In the five runs of generating input string sets 5x of $|S_{succ}|$ for *Rex5M*, there are 2,041 regular expressions which have input strings less than 5x of $|S_{succ}|$ in at least one run. In the five runs of generating input string sets 10x of $|S_{succ}|$

for *Rex10M*, there are 2,336 regular expressions which have input strings less than 10x of $|S_{succ}|$ in at least one run.

3.2 Results

Here, we present the results of RQ1 and RQ2 in turn.

3.2.1 RQ1: Test Coverage of Regular Expressions

We address RQ1 is two ways. First, we look at the number of call sites to FullMatch methods are actually tested. Next, we look at the test coverage for each tested regular expression,

3.2.1.1 Tested Call Sites

In the 1,225 projects, there are 18,426 call sites of the instrumented functions in Section 3.1.1.1. However, only 3,093 call sites are executed by the test suites. This means that 15,333 (83.21%) of the call sites are not covered by the test suites. For those that are, the median of unique regular expressions per tested call site is one, with an average of five (Table 3.1).

Summary: Of the 18,426 call sites for FullMatch methods in 1,225 GitHub projects, only 3,093 (16.8%) are executed by the test suites.

3.2.1.2 Coverage of Tested Regular Expressions

We successfully generated static DFAs for 15,096 regular expressions from 1,225 Java GitHub projects and dynamic DFAs for 899,804 regular expression/input string pairs.[4]Among the regular expressions, 4,941 (32.7%) regular expressions have only failing inputs (i.e., $|S_{succ}| = 0$) and 6,029 (39.9%) have only inputs of successful matching (i.e., $|S_{fail}| = 0$). This means that 10,970 (72.7%) of the regular expressions do not contain test inputs that exercise both the matching and non-matching scenarios. Of these, 6,318 (41.9%) regular expressions contain only one test string (i.e., $|S| = 1$) There are 4,126 (27.3%) regular expressions with both failed and successful matchings.

Table 3.4 describes properties of the test input sets for each regular expression: $|S|$ is the size of the test suite, computed as the number of unique input strings for a regular expression; $|S_{succ}|$ means the number of matching inputs; $|S_{fail}|$ means the number of failing inputs; *succ_ratio* shows the ratio of successful matchings to all matchings for each regular expression; *fail_ratio* shows the ratio of failed matchings to all matchings for each regular expression.

[4]We note that 899,804 is less than $60 \times 15096 = 905760$ because the mean of *# Input strings (|S|)* is 59.60546 and rounded up to 60.

Table 3.4 Description of 15,096 Regular Expressions' test suites. All numbers are rounded to nearest integer except that success ratio rounded to two decimal places.

Attributes	mean	25%	50%	75%	90%	99%
$\|S\|$	60	1	2	7	27	662
$\|S_{succ}\|$	19	0	1	1	4	79
$\|S_{fail}\|$	41	0	1	4	19	383
succ_ratio	49.03	0.00	44.70	100.00	100.00	100.00
fail_ratio	50.97	0.00	55.30	100.00	100.00	100.00

Table 3.5 Coverage values in Figure 3.2.

Coverage	Suite	mean	25%	50%	75%	90%	99%
NC (%)	S	59.05	24.62	63.64	95.65	100.00	100.00
NC (%)	S_{succ}	47.84	0.00	46.15	90.00	99.60	99.89
NC (%)	S_{fail}	18.89	0.00	8.51	25.00	62.26	100.00
EC (%)	S	28.74	6.67	23.90	49.97	53.80	80.00
EC (%)	S_{succ}	23.20	0.00	12.36	49.96	50.00	60.00
EC (%)	S_{fail}	8.55	0.00	2.20	7.80	32.19	65.08
EPC (%)	S	23.77	2.47	12.50	49.96	50.00	66.67
EPC (%)	S_{succ}	20.48	0.00	5.26	49.94	50.00	55.56
EPC (%)	S_{fail}	5.50	0.00	0.00	2.74	22.12	57.14

Table 3.5 describes the distributions of Node Coverage (NC), Edge Coverage (EC), and Edge-Pair Coverage (EPC) over S, S_{succ}, and S_{fail}. Figure 3.2 displays this information graphically; *total* means the total number of input strings for each regular expression, namely the test suite S, *success* means S_{succ}, and *failure* means S_{fail}. Most of the regular expressions are not tested thoroughly since the mean values of coverage are low, especially the edge and edge-pair coverage. Although the coverages on failed matchings are relatively small, they contribute to a high overall test coverage. Failed matching tests are a necessary part of testing regular expressions.

Summary: A majority of regular expressions (10,970, 97.7%) are tested with exclusively passing (6,029, 39.9%) or exclusively failing (4,931, 32.7%) test inputs. Edge and edge-pair coverage are both very low. Full node coverage is infeasible with only passing inputs as N_e would never be covered; the presence of both types of inputs is important for thorough testing.

Table 3.6 Coverage values of the 7,926 regular expressions in GitHub for $Repo_BM$ and $Repo_BS$ in Figure 3.3.

Coverage	Expr	mean	25%	50%	75%	90%	99%
NC (%)	$Repo_BM$	70.41	43.75	80.00	97.67	99.84	99.90
EC (%)	$Repo_BM$	33.79	12.01	45.91	49.97	50.00	66.67
EPC (%)	$Repo_BM$	29.39	4.83	37.50	49.97	50.00	60.00
NC (%)	$Repo_BS$	73.27	46.15	85.71	99.83	100.00	100.00
EC (%)	$Repo_BS$	36.35	12.36	48.39	49.97	60.00	85.71
EPC (%)	$Repo_BS$	30.68	5.13	40.00	49.97	50.00	74.67

Table 3.7 Coverage values of the 7,926 regular expressions using Rex for *Rex1M, Rex5M*, and *Rex10M* in Figure 3.3.

Coverage	Expr	mean	25%	50%	75%	90%	99%
NC (%)	*Rex1M*	69.29	41.67	78.33	97.44	99.84	99.90
EC (%)	*Rex1M*	33.57	11.62	45.00	49.97	50.00	71.43
EPC (%)	*Rex1M*	29.50	4.33	35.00	49.96	50.00	66.67
NC (%)	*Rex5M*	71.69	46.15	83.33	97.67	99.84	99.90
EC (%)	*Rex5M*	36.42	12.77	49.81	50.00	54.55	80.00
EPC (%)	*Rex5M*	33.04	6.63	49.54	50.00	56.67	75.00
NC (%)	*Rex10M*	72.01	46.15	83.33	97.73	99.84	99.90
EC (%)	*Rex10M*	36.87	13.39	49.85	50.00	55.89	80.00
EPC (%)	*Rex10M*	33.77	6.90	49.77	50.00	58.33	75.00

3.2.2 RQ2: Coverage with Rex

Figure 3.3 shows the analysis results given the generated inputs in ASCII encoding, organized by each of five datasets. $Repo_BS$ and $Repo_BM$ show the coverages over S and S_{succ} of 7,926 regular expressions using the developer-defined test suite in GitHub and their details are in Table 3.6. *Rex1M*, *Rex5M*, and *Rex10M* show the coverages of 7,926 regular expressions based on the Rex-generates test inputs with sizes of 1x, 5x, and 10x of the user-defined test suite, respectively. Coverage details are shown in Table 3.7.

Table 3.8 illustrates the differences in coverage between the repository ($Repo_BM$ and $Repo_BS$) and Rex (*Rex1M*, *Rex5M*, and *Rex10M*). Using a paired Wilcoxon signed-rank test, we find that for all three coverage metrics, $Repo_BM$ significantly outperforms *Rex1M* with $\alpha = 0.0001$. However, as test suite size is strongly correlated with coverage [52], as soon as the Rex test set is amplified to 5x

Table 3.8 Differences in coverage based on datasets in Figure 3.3. Hypothesis tests used paired Wilcoxon signed-rank test. Bold text identifies when one of the datasets had significantly higher coverage for all three metrics. If there was a conflict between the metrics (e.g., Set1 > Set2 for NC, and Set1 < Set2 for EPC), there was no winner

		$H_0 : Set1 \stackrel{d}{=} Set2$		
Set1	Set2	NC	EC	EPC
Repo$_B$M	*Rex1M*	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$
Repo$_B$M	***Rex5M***	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$
Repo$_B$M	***Rex10M***	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$
Repo$_B$S	*Rex1M*	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$
Repo$_B$S	*Rex5M*	$p < 0.0001$	$p = 0.0004$	$p < 0.0001$
Repo$_B$S	*Rex10M*	$p < 0.0001$	$p = 0.4147$	$p < 0.0001$
Repo$_B$S	*Repo$_B$M*	$p < 0.0001$	$p < 0.0001$	$p < 0.0001$

and 10x the size, the coverage of Rex outperforms the repository. When considering all test inputs from the repository and not just the successful ones, with test inputs sets of the same size, *Repo$_B$S* outperforms *Rex1M*. However, this comparison is unfair since Rex does not generate non-matching strings. That said, as soon as the Rex dataset is amplified as in *Rex5M* and *Rex10M*, there is no clear winner compared to all test inputs from the repository. While it may appear that Rex can do as well as the repository, the reality is that the error node will never be covered by Rex, a fact which is not apparent by looking at the numbers alone.

Summary: Rex can handle approximately 78.1% of the regular expressions from our dataset. Considering only the matching test inputs and test sets of the same size, Rex does not achieve coverage as high as the developer-written tests. However, the coverage numbers are extremely close. This indicates that tools such as Rex can be used to write test inputs with similar coverage to the developer tests, but will always miss N_e and all edges incident to it.

3.3 Discussion

This section summarizes future work based on our findings and discusses threats to validity.

3.3.1 Implications

We have applied graph coverage metrics to regular expressions and looked at how well existing test suites perform according to the metrics. Coverage provides useful stopping criteria for testing.

However, high coverage does not necessarily imply test suite effectiveness in source code [52], which may also hold true for regular expressions. At the same time, as regular expressions are responsible for many software faults, it is important to explore how to make them less error-prone. Our approach in this work is through test metrics, and there are many areas of future work that follow:

String-generation Tools: Given the low coverage of regular expressions shown in Figure 3.2, a natural next step could be to generate strings to achieve high coverage. In RQ2, Rex achieves high structural coverage. It also ignores the failure case, N_e, and all edges incident to it. Adding a mutation step to the input string may be effective at forcing the Rex-generated strings into the error state to cover the uncovered edges and node. An alternate approach may be to provide the complement of the regular expression to Rex as another way to generate failing inputs.

With automatically-generated strings, one threat is usability. For the developer-written tests, it is likely that the regular expression strings are more meaningful in context than they are for the Rex-generated strings. Future work will look at the overlap in content between the test inputs from the repository and from Rex.

However, it may not always be possible to achieve 100% test coverage, even with a perfect string generation tool. There are untested regular expressions that be untested because they are unreachable. Some regular expressions have hard-coded matching inputs, which makes it impossible to improve the coverage; for example: boolean isMatch = Pattern.matches("ab", "ab"); Future work for improving coverage levels should also consider the potential for improvement based on such factors.

Beyond Structural Coverage: The metrics we explore are structural metrics, which can identify faults that are revealed in the structure of the DFA, such as the example in Figure 2.2.2.2. Alternately, as suggested in prior work [24], refactoring could potentially reveal this particular fault, as the numeric representation `[0-9]` was found to be more understandable than `\d`. Performing the replacement might alert the developer that `d` should be `\d`.

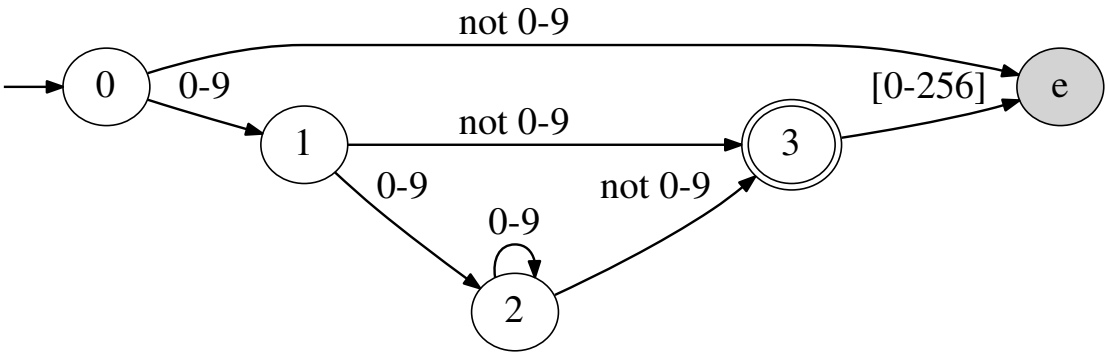
In terms of improving regular expression testing, structural metrics are a first step. Building on the example in Section 2.1.2, achieving 100% coverage requires a minimum number of test inputs that vary in string length and content. In the example of `\d+`, there are strings of length one to length four, though strings could be longer to test multiple iterations on the self-loop. Strings can contain only digits, only non-digits, or both digits and non-digits. Strings can start with digits or start with non-digits. Defining such input space partitions may lead to intuitive test sets with high behavioral coverage.

3.3.2 Threats to Validity

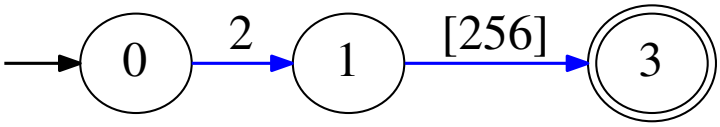
Internal: We measure the test coverage of regular expression used in functions of full matching with FullMatch DFAs in the forward direction. The experimental results may not reflect the test coverage of regular expressions used in other functions, nor the test coverage of regular expressions which could not be converted into a DFA.
External: The Java regular expressions used in this evaluation were collected from RepoReaper Java Maven projects compiled with Java jdk1.7, which is only a small portion of all GitHub Java projects and may not generalize to all Java projects and to other languages. It is possible that there are still regular expressions from third-party libraries in the dataset, which could bias results. Due to limitations of RE2 and Rex, the results of test coverage applies exclusively to the features supported. All our projects had test suites, which may overestimate the test coverage levels for typical regular expressions.

3.4 Summary

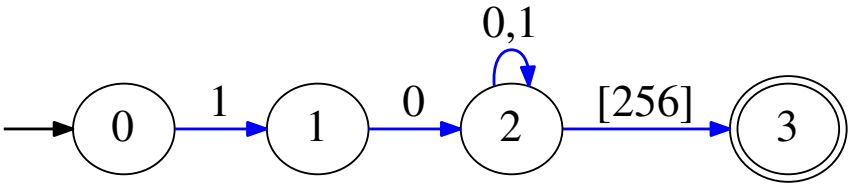
To our knowledge, this is the first research to evaluate fine-grained coverage metrics for regular expressions. We explore coverage over the DFA representation of a regular expression. It is also the first research to explore how often regular expressions are tested in a large set of the software projects. We show the coverage metrics of regular expressions from 1,225 GitHub Java Maven projects and found that over 80% of *FullMatch* functions are not tested and that most of the tested regular expressions have a low edge and edge-pair coverage. We also show that with the help of the regular expression tool Rex it is possible to improve the regular expression testing coverage by adding input strings, but that there is an upper bound for this type of improvement. This work is a first step toward better understanding how regular expressions are tested in practice; future work will explore how various coverage metrics can reduce the bugs associated with regular expressions.



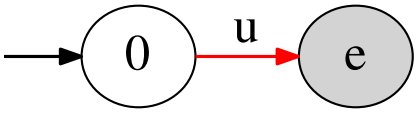
(a) Fully specified static DFA for: `\d+`



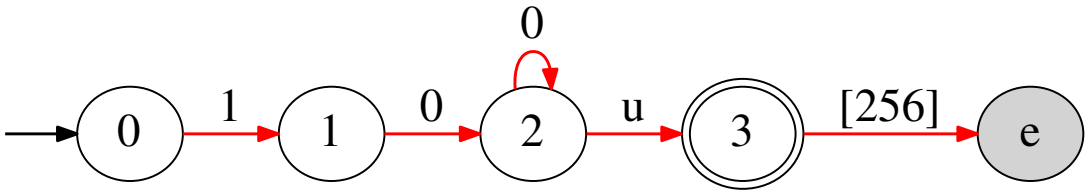
(b) Dynamic DFA for regular expression: `\d+` and input: "2"



(c) Dynamic DFA for regular expression: `\d+` and input: "1001"



(d) Dynamic DFA for regular expression: `\d+` and input: "u"



(e) Dynamic DFA for regular expression: `\d+` and input: "100u"

Figure 3.1 Visited DFA subgraphs for the regular expression `'\d+'`. For each figure, N_0 is the initial node 0, N_m is the accept node 3, N_e is the error node e. The arrows colored blue represent transitions in successful matches. The arrows colored red represent transitions in failed matches. The characters without square brackets are the literal characters in state transitions. For example, 'u' prompts the transition from Node 0 to Node e. `[256]` implies that there are no more bytes from the input string.

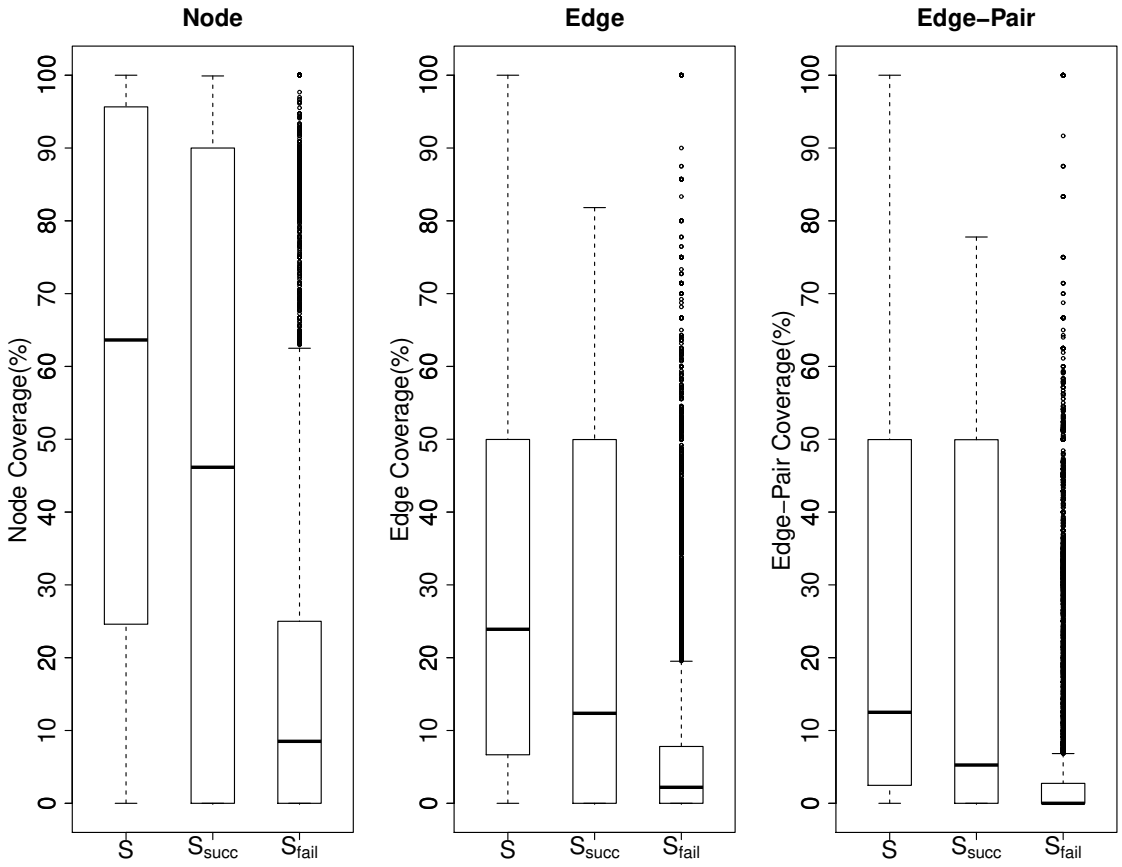


Figure 3.2 Coverage for 15,096 regular expressions.

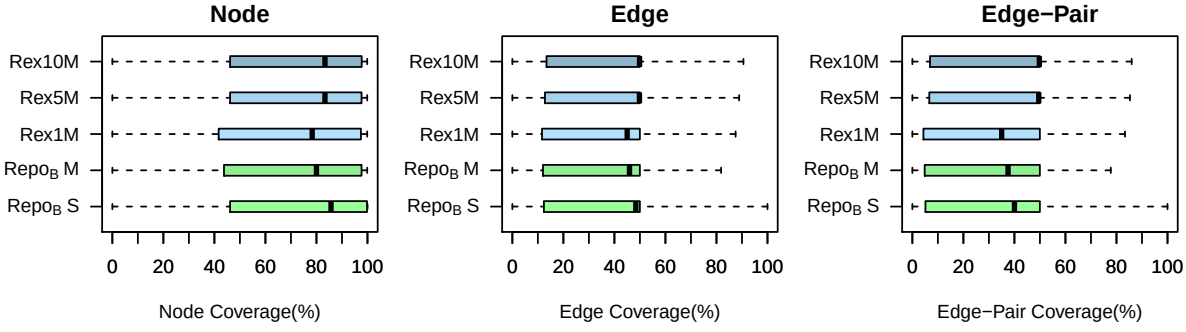


Figure 3.3 Node, edge, edge-pair coverage of 7,926 regular expressions with Rex-generated ASCII inputs ($Rex1M$, $Rex5M$, $Rex10M$) of 7,926 regular expressions in GitHub which are used in Rex ($Repo_BS$, $Repo_BM$).

CHAPTER

4

REGULAR EXPRESSION EVOLUTION

In Chapter 4, which is going to appear at SANER 2019 [106], we present our study of regular expression evolution by describing our artifacts, results for each research questions, and analysis. Finally, we discuss the validity of this piece of work and draw a summary at the end of this chapter.

4.1 Artifacts

We address research questions in this paper using two datasets. One comes from GitHub commit logs, providing a high-level view of regular expression evolution over time. The other comes from screencasts of students solving regular expression tasks in an IDE, providing a low-level view of evolution during problem-solving tasks.

4.1.1 GitHub Dataset

The history of regular expressions in GitHub projects is collected through source code commits. Since only literal regular expressions can be found statically, we are limited to the explicitly written regular expressions.

4.1.1.1 Data Collection

Our data collection starts within the 1,114 Java projects used in a prior work on testing regular expression [107]. We first searched for method invocations of Pattern.compile(String regex) in latest source code version. If the argument is a literal string (as opposed to a variable), we follow the commit history of the literal string to create an edit chain for the regular expression. We filtered out the invocations in which the argument of Pattern.compile contains variables and extracted the files and the line numbers where literal regular expressions appear. For other methods, String.matches(String regex) needs to check if the caller is a String instance, and Pattern.matches(String regex, CharSequence input) contains two arguments that can vary independently; for this first data-driven exploration of regular expression evolution, we focus on the Pattern.compile() method.

There are 9,952 static invocations to Pattern.compile() in the 1,114 projects; 387 (34.74%) projects contain no literal regular expressions in their latest version and were excluded, resulting in 4,156 literal regular expressions in 727 GitHub projects; these are the tops of the edit chains.

4.1.1.2 Building the Edit Chains

Next, in order to retrieve the commit history of each literal regular expression, we used the Git command git log −L <start>,<end>:<file>. We retained information regarding the regular expression edit, commit number, author, and date of each regular expression version. We dropped 194 (4.67%) chains for the following reasons: 1) 123 were dropped because more than one regular expressions are changed in a single commit on the chain; 2) 30 were dropped because non-literal regular expressions exist in their history of commit changes; 3) 24 were dropped because at least one of the invocations to Pattern.compile() are multi-line statements and we failed to parse them; 4) 17 were dropped because their git log −L commands return git activities (e.g., merging files) rather than code edits.

If two adjacent regular expressions are identical to each other in syntax (e.g., something else on the source code line changed, such as a variable name), then this regular expression does not evolve; we squashed these into a single node on the edit chain. There are 144 pairs of such regular expressions. As a result, in the GitHub dataset for study, there are 3,962 edit chains containing 4,224 regular expressions from 708 GitHub projects.

4.1.2 Video Dataset

We ran an exploratory lab study in which participants completed regular expression tasks in Java using the Eclipse IDE. During problem solving, we captured videos of their computer screens. Participants were free to use online resources to help them complete the tasks.

4.1.2.1 Tasks

Participants attempted up to 20 tasks each, with one hour allotted for the study. The order of tasks was randomized per participant to control for learning effects. In each task, the goal was to compose a regular expression that caused an associated JUnit test suite to pass. For example, one task asked participants to compose a regular expression that will *verify that an entire string is composed of one valid email. Extra characters like whitespace before or after, or anything that would invalidate the email are not allowed*. For this task, eight test cases are provided to demonstrate the desired behavior. One test case provides the test input "name@domain.com" with the expected output true, as in the e-mail address is valid. Another test case has the test input "1.2.3.4@crazy.domain.axes" and output true. For invalid examples, "www.website.com" has the expected output false. A repo with all the task data is available: [blinded for review]

4.1.2.2 Participants

There were 29 participants who produced usable data for this analysis (six videos had issues with recording). The participants consist of 25 undergraduate students and four graduate students with on average 4.16 years of programming experience and 3.26 years of Java experience. The survey results found that a majority of participants (76%) considered themselves as having intermediate Java programming knowledge; 20 participants (69%) have little to no experience with regular expressions.

4.1.2.3 Data Extraction

The videos were manually transcribed to logs reflecting the evolution of the regular expression strings during composition on each problem. In the transcription process, we created a log for each task the participant attempted in each video. Each video was transcribed by one of the authors to ensure consistency within a log. An edit chain consists of all the regular expressions written (or copy/pasted) while attempting to solve a single task. The regular expression logged are the ones submitted by participants for testing.

4.1.2.4 Data Description

In total, there are 92 edit chains containing 751 regular expressions from 25 tasks and 29 participants. Twelve pairs of identical adjacent regular expressions are squashed, resulting in 92 edit chains containing 739 regular expressions.

4.2 RQ3: Regular Expression Evolution Characteristics

We describe the characteristics of regular expression evolution through analyzing the edit chains in the datasets.

4.2.1 Edit Frequency of Regular Expressions

In GitHub dataset, of the 3,962 edit chains containing 4,224 regular expressions. Among the edit chains, 3,775 (95.28%) have a length of one, indicating those regular expressions are not edited at all. Regarding the remaining 187 edit chains of 449 regular expressions, 137 contain one edit, 35 contain two edits, five contain three edits, and ten contain four edits; in total, this created 262 edits.

In Video dataset, of the 92 edit chains containing 739 regular expressions, there are 16 (17.39%) chains of length one. Regarding the remaining 76 regular expression edit chains, 11 contain one edit, eight contain two edits, seven contain three edits, four contain four edits, 12 contain five edits, and the others contain edits from six to 48. The regular expression created by participants needs on average seven changes before task completion or abandonment; this creates 647 edits.

Summary: During problem solving, developers tend to modify their regular expressions quite frequently in order to successively complete a task. However, once the related source code committed to the GitHub repositories, edits are rare; 95% of the literal regular expressions we explored were not edited.

4.2.2 Runtime Errors

The grammar errors in a normal program source code can be highlighted in the integrated development environment (IDE) and checked during program compilation. However, the grammar errors of regular expressions in source code can only be found during program runtime.

Of the 4,224 regular expressions in the GitHub Dataset, there are nine (0.21%) that produce runtime errors on nine (0.23%) edit chains. These impact three of the edits. Of the 739 regular expressions in the Video Dataset, there are 65 (8.80%) regular expressions that produce runtime errors in 26 (28.26%) edit chains. These impact 85 of the edits.

Summary: Invalid regular expressions are rarely observed in GitHub; for the video dataset, invalid regular expressions typically usually appear in the early stage of regular expression evolution.

4.2.3 Regular Expression Reversions

Regular expression reversion describes the re-occurrence of some regular expressions after they have been modified in an earlier stage. Suppose three regular expressions r_i, r_j, and r_k $(i < j < k)$

Table 4.1 The distribution of Levenshtein distances in both GitHub and Video edits (where distance > 0).

Dataset	Mean	Min	10%	25%	50%	75%	90%	Max
GitHub	9.32	1	1.00	2.00	5.50	12.75	23.00	52
Video	6.87	1	1.00	2.00	4.00	8.50	15.40	88

on the same edit chain, then the case when r_k is same as r_i but different from r_j is called a regular expression reversion.

In the GitHub dataset, there are 12 regular expression reversions on 12 edit chains; 11 reversions happened in two edits and the other one in three edits.

There are 85 cases of regular expression reversions in Video dataset. Those reversions happened on 25 edit chains; 36 (42.35%) reversions were made in two edits and 11 reversions in three edits. The number of edits in the other 38 reversions varies from four to 28.

Summary: Regular expression reversions imply that developers may repeat the same regular expression even if they have previously modified it. This is especially true for inexperienced developers since reversions are more common in the Video dataset; the high frequency possibly reflects developers' *undo* behavior.

4.3 RQ4: Syntactic and Semantic Evolution

We explore RQ4 regarding syntactic and semantic evolution.

4.3.1 Syntactic Similarity

We report on Levenshtein distance for the GitHub and Video datasets considering individual *edits*. Starting with an example, one regular expression extracted from GitHub was committed by user *ginere* in one version[1] and was changed by the same person in a newer version[2]. The original and modified regular expressions are, respectively,

```
\\|DATE\\[([a-zA-Z0-9_]*)\\]\\|
\\|DATE\\[([a-zA-Z0-9:\\- /]*)\\]\\|
```

The only syntactic edit is the change from `_` to `:\\- /`, resulting in a Levenshtein distance of five ("\\" is considered as one character because backslashes are escaped in Java).

[1] https://github.com/ginere/ginere-site-generator/commit/7c819359
[2] https://github.com/ginere/ginere-site-generator/commit/248d3a25

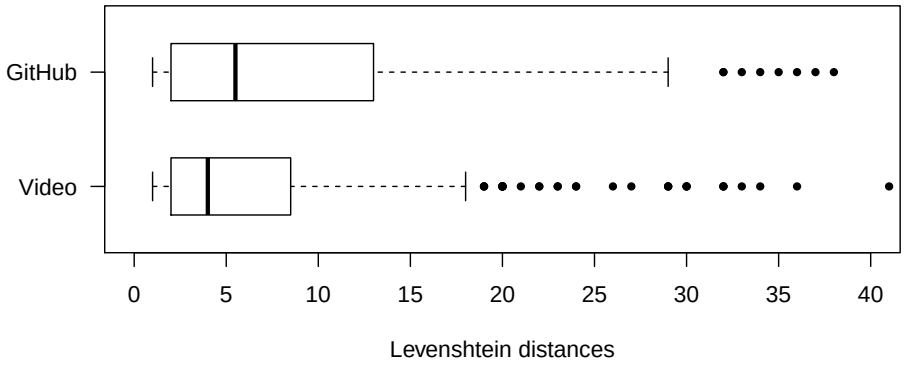


Figure 4.1 The distribution of Levenshtein distances for 262 GitHub edits (with distance > 0) and 647 Video edits (with distance > 0).

In the study of GitHub dataset on Levenshtein distance, we calculated 262 regular expression edits on 187 edit chains. In Video dataset, we calculated 647 regular expression edits among 76 edit chains.

The edit distance for the GitHub edits is generally larger than the edits from the Videos. Figure 4.1 shows the distribution of Levenshtein distances among the edits in both dataset; Table 4.1 details the averages and distributions of the Levenshtein distances for the 264 edits. On average, there is a distance of 9.32 for a GitHub edit with a median of 5.50, and a distance of 6.88 for a Video edit with the median of 4.00.

Summary: The average and median Levenshtein distances in GitHub are larger than in the Video dataset. This reflects our intuition that developers try many small edits while composing and debugging a regular expression. Accordingly, the regular expressions which are committed to version control software reflect larger edits from their predecessors.

From the perspective of regular expression changes, both of the datasets have over 50% regular expression edits in which at most six characters change. Those modifications are the ones could be made automatic through regular expression mutation. However, for the other half of the regular expressions, the changes are much lager. This information suggests that when generating regular expression mutants, we should also consider ways larger changes and/or changes in multiple locations.

4.3.2 Semantic Similarity

Within each edit chain, we look at each edit and classify it according to the semantic evolution types in Figure 2.4. In order to compute the similarity between regular expressions, we adopt an approach

Table 4.2 Edit types in GitHub and Video dataset when $k = 500$

Dataset		Disjoint	Overlap	Equivalent	Reduction	Expansion	Total
GitHub	count	44	17	22	20	106	209
	(%)	21.05	8.13	10.53	9.57	50.72	100.00
Video	count	125	32	36	43	56	292
	(%)	42.81	10.96	12.33	14.73	19.18	100.00

Table 4.3 The distribution of Intersection, Addition, Removal percentages over all types of regular expression edits in GitHub and Video dataset when k is 500.

Dataset		Mean	Min&10%	25%	50%	75%	90%&Max
GitHub	Intersection	56.62	0.00	29.30	70.37	83.05	100.00
	Addition	38.98	0.00	5.06	26.39	57.58	100.00
	Removal	27.71	0.00	0.00	0.00	52.56	100.00
Video	Intersection	35.78	0.00	0.00	3.87	73.09	100.00
	Addition	56.42	0.00	0.00	79.75	100.00	100.00
	Removal	54.73	0.00	0.00	53.49	100.00	100.00

from prior work [23] and use Rex [104].

For strings generated by Rex, we chose to generate up to $k = 500$ strings for each regular expression, as described in Section 2.1.4.4. This is 25% larger than prior work, which generated 400 strings to compute clusters of similar regular expressions [23].

4.3.2.1 Rex-generated Strings

With $k = 500$, Rex tries five times with different seeds to generate 500 matching strings for every valid regular expression. The number of matching strings for regular expression r_i can be different from that for r_{i+1} because Rex could not guarantee to generate exactly 500. For the semantic distance between r_i and r_{i+1}, it requires that both are valid regular expressions and Rex are able to generate matching strings for both of them. Due to the invalid regular expressions next to the valid ones and chains of only one valid regular expressions, the total number of regular expressions involved in the semantic distance is fewer than the ones for which Rex can generate strings.

For the GitHub dataset, Rex generated matching strings for 441 out of the 446 valid regular expressions; 272 have 500 rex-generated matching strings while the other 169 have fewer than 500 matching strings. On average there are 432 matching strings generated for each regular expression.

In total, 209 edits on 146 edit chains are studied for the GitHub dataset.

In the Video dataset, Rex generated matching strings for 393 out of 674 valid regular expressions; 102 regular expressions have 500 strings and the other 291 ones have fewer than 500 strings. On average there are 270 Rex-generated matching strings per regular expression. In total, 292 edits on 47 edit chains are studied for the Video dataset.

4.3.2.2 Results

Table 4.2 shows the number of different edit types in the GitHub and Video datasets. The most common edit is *expansion* for GitHub (50.72%) and *disjoint* for Video (42.81%). Table 4.3 shows the distribution of intersection, addition, and removal metrics for every regular expression edit in both datasets. The average intersection value for an edit in the GitHub dataset is 56.62%, indicating that more than half of the language from an edited regular expression is sourced from its predecessor. This makes sense as a majority of the edits fall in the *expansion* classification (Table 4.2). In the Video dataset, only 35.78% of the language of a regular expression is sourced from its predecessor, likely lower because of the high frequency of *disjoint* edits. In the GitHub dataset, the relatively small addition and removal values indicate that the semantic change is relatively small per edit, whereas with the addition and removal values in the Video dataset are larger, on average, indicating larger semantic changes per edit.

The GitHub dataset edits represent the situations where the regular expression being modified are close to the targeted scope of strings because of their small number of edits and a high percentage of edit intersection. This indicates that the semantic edits are relatively small.

The differences can also be observed in Figure 4.2 which visualizes the distributions of metrics per semantic edit type. In all edit types, the intersection value of GitHub dataset is higher than that of Video dataset while the addition and removal values of GitHub dataset are always lower than of Video dataset. In the *expansion* edit type for GitHub, the average addition is 29.02%; for the video analysis, the addition is 41.52% on average. *Reduction* edits in GitHub remove an average of approximately one-third (35.55%) of the language, whereas the Video reductions remove an average of 47.08%. Average intersection numbers for *overlap*, *reduction*, and *expansion* in the GitHub edits are 45.08%, 64.45%, and 70.98% whereas the average intersection numbers for these three edit types in the Video edits are 36.67%, 52.92%, and 58.48%.

Refactoring Analysis

We did a further study on pairs of regular expressions classified as *equivalent*. In the GitHub dataset, 19 out of the 22 pairs are correctly classified. For the three misclassified cases, one changes the repe-

tition time from `[a-z0-9_-]{1,64}` to `[a-z0-9_-]{1,120}`. This misclassification is because the length of strings generated by Rex is less than 13. The other two cases change special characters in the character class, and Rex does not use those special characters in the string generation.

Among the 19 truly equivalent pairs, eight pairs are related to unnecessary character escaping (e.g., from `([\\+\\-])+(.*)` to `([+\\-])+(.*)` and three pairs are related to changes in capturing group representation (e.g., from `.* \\{.*\\}` to `(.*) (\\{.*\\})`. One removes capital characters from `[aA][sS]` and expresses the case sensitiveness with flag `(?i)AS`, and two adds and removes capital characters in `(?i)[0-9a-fA-F]` and `(?i)[0-9a-f]` while regular flag 'i' is specified for case-insensitive matching. One changes character repetition boundary `[0-9]{1,}` to greedy operator `[0-9]+`, hence improving improves the understandability according to a regular expression comprehension study [24]. One changes the literal parentheses in character class from `[(][)]` to escaped characters `\\(\\)`. One changes the ordering on options surrounding an OR operator, from `([wdhms]|ms)` to `(ms|[wdhms])`. For full matches (as is the case with Pattern.matches), these are identical. The final two pairs change whitespace around a `.*`, from `(?im)^dry-run:\\s*(.*)\\s*` to `(?im)^dry-run:(.*)` and from `(?im)^dry-run:(.*)` to `(?im)^dry-run:(.*)\n*`. Effectively, these are all equivalent.

In the Video dataset, 35 of the 36 pairs are correctly classified. The only misclassified case changes the whitespace characters from `00Z*([a-zA-Z\\s]*)` to `00Z*([a-zA-Z ]*)`. It is correctly classified as *reduction* when $k = 1000$.

Among the 35 truly equivalent pairs, 12 pairs are related to changes in capturing group representation, nine pairs are related to unnecessary character escaping, three are about adding or deleting anchors (e.g., from `^[.]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$` to `[.]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$`[3], four about changing OR operator alternatives which does not impact matching behaviors, such as removing one option from `(.*)|(6.35.)` to `(.*)`. One removes character class of single element from `[A-Za-z0-9][\s]` to `[A-Za-z0-9]\s`. Two pairs manipulate duplicated characters in the character class between `[(1|3|5|7|9)+(2|4|6|8|0)]` and `[(1||3||5||7||9)+(2||4||6||8||0)]` possibly due to misconceptions of the differences between `[]` and `()`. Another edit changes from `.*[02468][13579].*` to `.*([02468][13579])+.*`. Although + is added, additional digits are accepted by `.*` at the beginning of these two regular expressions. One changes from `[a-zA-Z-']{.*)total[0-9]` to `[a-zA-Z-']+(.*)total[0-9]`. Although + is added, additional characters are accepted by `(.*)` in the middle of these two regular expressions. In the modification from `^[a-zA-Z0-9 \\t]*$` to `^[a-zA-Z0-9 \\d \\t]*$`, the regular language does not change because `\\d` is equivalent to `0-9` which exists already in the

[3]Since Pattern.matches(String regex, CharSequence input), String.matches(String regex), and Matcher.matches() by default match the entire input string to the regular expression and anchors do not affect the matching results

character class. The final pair changes from
`(((1||3||5||7||9)+(2||4||6||8||0))* |((2||4||6||8||0)+(1||3||5||7||9))*)` to
`(((1||3||5||7||9)(2||4||6||8||0))+|((2||4||6||8||0)(1||3||5||7||9))+)`. This is because `a||b` contains tree alteratives: a, b, and empty strings and `(1||3||5||7||9)` matches not only odd digits but also empty strings. The changes of repetitions in these two regular expressions are thus counteracted by the empty strings.

Summary: Compared to the GitHub edits, the edits in the Video dataset tend to make larger semantic changes. GitHub edits represent small adjustments of the regular expression close to targeted scope.

4.4 RQ5: Regular Expression Feature Changes

In this section, we present our results and analysis of regular expression language feature changes in the edit chains. This can inform the features to focus on during mutation testing or program repair.

The feature vector we use contains the 35 most frequently used features in Java regular expressions. All feature explanations except *LIT* (literal character) are defined in the work of Chapman and Stolee [23]. Features are extracted using the PCRE parser.

For the GitHub dataset, we started with 262 edits; 3 were removed due to Java runtime errors (Section 4.2.2) and three were removed due to PCRE parsing errors, leaving us with 256 edits for analysis. For the Video dataset, we started with 647 edits; 85 were removed due to runtime errors and four were removed due to PCRE parse errors, resulting in 558 edits for analysis.

4.4.1 Feature Vector Edits

We first calculated the number of regular expressions in which each feature appears. For the 4,054 regular expressions in GitHub and 660 regular expressions in Video Feature that can be parsed by PCRE. Table 4.4 shows the results of frequency analysis for the top 25 features in the *Freq* column for the GitHub and Video Datasets.

For 237 of the GitHub edits and 536 of the Video edits, the feature vector (e.g., Figure 2.6) changed. F_{add} and F_{remove} are listed, alongside the number of edits impacted (*nR*). For example, 123 edits in GitHub added a literal (LIT), and 53 added a KLEENE star (KLE). The ADD feature is the third most commonly seen feature in the GitHub dataset. It is added into 34 regular expressions and ranked as the sixth most frequently added feature while it is removed from 27 regular expressions and ranked as the second most frequently removed features. Assuming that the predecessor in a regular expression edit has a fault of some sort, these details can help inform the types of changes

to make to a regex feature vector during fault injection for mutation testing.

Overall for every feature in both datasets, there are more regular expressions which added it than the ones which removed it. From Table 4.5 we can find that feature frequency is different between GitHub and Video. For example, on the average edit, 3.71 features are added to a regular expression and 2.52 are removed. For the video dataset, these values are smaller, with 2.6o features added and 1.98 removed. These details can help inform the frequencies of changes to make to a feature vector during fault injection for mutation testing or repair.

4.4.2 Feature Vector Non-Edits

The feature vector used in this study does not reflect the positions of features, nor the scope of the changes. For example, the modification from `[\\D]{2}` to `[\\D]{5}` does not change the number of feature SNG but it changes the repetition time of SNG from '2' to '5'. Similarly, the change from `(^\\r\\f\\n)` to `^(\\r\\f\\n)` changes the content of capturing group to exclude'`^`', but the feature vector remains the same.

There were 19 edits in the GitHub dataset and 22 in the Video dataset for which the vectors did not change. On further inspection, the most common modification is related to backslash for character escaping (e.g., from `\t\n` to `\\t\n`), impacting 10 edits in the GitHub dataset and 13 edits in the Video dataset. Other common modifications change characters to other characters (e.g., from `\\{([\\w\\.]*)\\}` to `\\(([\\w\\.]*)\\)`), switch the order of characters (e.g., from `\f\\s` to `\\s\f`), change repetition times (e.g., from `[a-z0-9_-]{1,64}` to `[a-z0-9_-]{1,120}`), change the content and position of capturing group (e.g., from `(^\\r\\f\\n)` to `^(\\r\\f\\n)`), change characters in the character class (i.e., from `^[a-zA-Z0-9 \\\\]*$` to `^[a-zA-Z0-9 \\t]*$` or change alternation option (e.g., from `^(\\r|\\n)` to `^(\\r|\\f)`).

4.4.3 Summary

The regular expression edits usually involve changing multiple features; adding features is more common than removing. The frequency of various features being added and being removed can be used to construct new regular expression mutation operator and guide mutation generation process. However, there are also edits that do not impact the feature vectors; escaping characters is the most common edits do not result in changes in the feature vector.

4.5 Discussion

In this work, we have looked at the evolution of regular expressions from two perspectives, syntactic and semantic, and in two contexts, using GitHub and developers solving tasks.

4.5.1 Implications

Most literal regular expressions in GitHub do not evolve (Section 4.2), and yet, bug reports related to regular expressions abound [96]. This may indicate that buggy regular expressions are simply removed from source code, or another solution to avoid the use of a regular expression.

Yet, regular expressions are under-tested [107], and most string-generation efforts focus on generating test inputs within the language of the regular expression (e.g., [104]). For those regular expressions that do evolve, 50% of the edits in GitHub are expansion edits (Section 4.3), indicating that often the original regular expression language is too restrictive. In generating test inputs, there is a need for test strings that lie outside the language of the original regular expression. One approach to this is fault injection via mutation [6]. However, to do this effectively, these faults need to be reflective of edits that developers make to regular expressions.

For those regular expressions that do evolve, 50% of the edits have a syntactic distance of six or fewer characters; these are most amenable to mutation testing (Section 4.3). Edits also impact multiple language features (Table 4.5).

For the Video dataset, the edits tend to be smaller in terms of character modifications (Table 4.1), but larger in terms of semantic distancing (Table 4.3 shows the intersection for Video edits is smaller than for GitHub edits). Furthermore, the edits to the feature vector tend to be smaller for the Video dataset (Table 4.5). The Video dataset edit chains are also longer than the GitHub edit chains (Section 4.1). Even though the Video dataset participants were largely novices, this indicates that developers likely go through many smaller iterations of edits on the regular expressions before finding one to commit.

4.5.2 Threats to Validity

Internal: We measure similarity using a string-generation approach that provides an approximate measure of similarity. We also observed in Section 4.3.2 that while Rex is a well-used and well-cited tool, sometimes it did not generate strings long enough and strings for uncommon characters. Such tool limitations have an impact on the accuracy of our results.

The regular expressions from the video analysis were collected manually. Each video was transcribed by two graduate students and merged to address any inconsistencies.

We collect regular expressions from GitHub using the current version of a project at the top of the chain. Regular expressions that are removed previously in the edit history will not be included in our dataset.

External: The regular expressions collected in this project reflect a relatively small sample, in one language (Java), and may not generalize. Further, the Video dataset was collected in a lab environment and may not reflect how developers actually compose regular expressions. While further study is needed, we do note that the common language features are consistent with prior work that explored regular expressions in another language [23].

The literal regular expressions we collected are restricted to Pattern.compile(...). Regular expressions with explicit flags to that method are excluded as well. Other Java methods which also accept literal regular expressions are not included in GitHub dataset.

4.6 Summary

In this work, we explore how regular expressions evolve through two lenses, GitHub commits and tested regular expressions during problem-solving tasks (called the Video dataset). We find that the GitHub regular expression edits are larger syntactically, but produce smaller semantic changes. The edit chains in the Video dataset are longer than the GitHub edit chains, indicating that developers may go through multiple iterations prior to committing a regular expression to a repository. The most common change to the scope of the regular expression in GitHub is to expand the matching language, which motivates the use of mutation operators to generate strings outside the original language for testing. Our results provide insights on the types and frequencies of edits that occur in regular expressions and can be used to guide mutation operators that reflect developer practices.

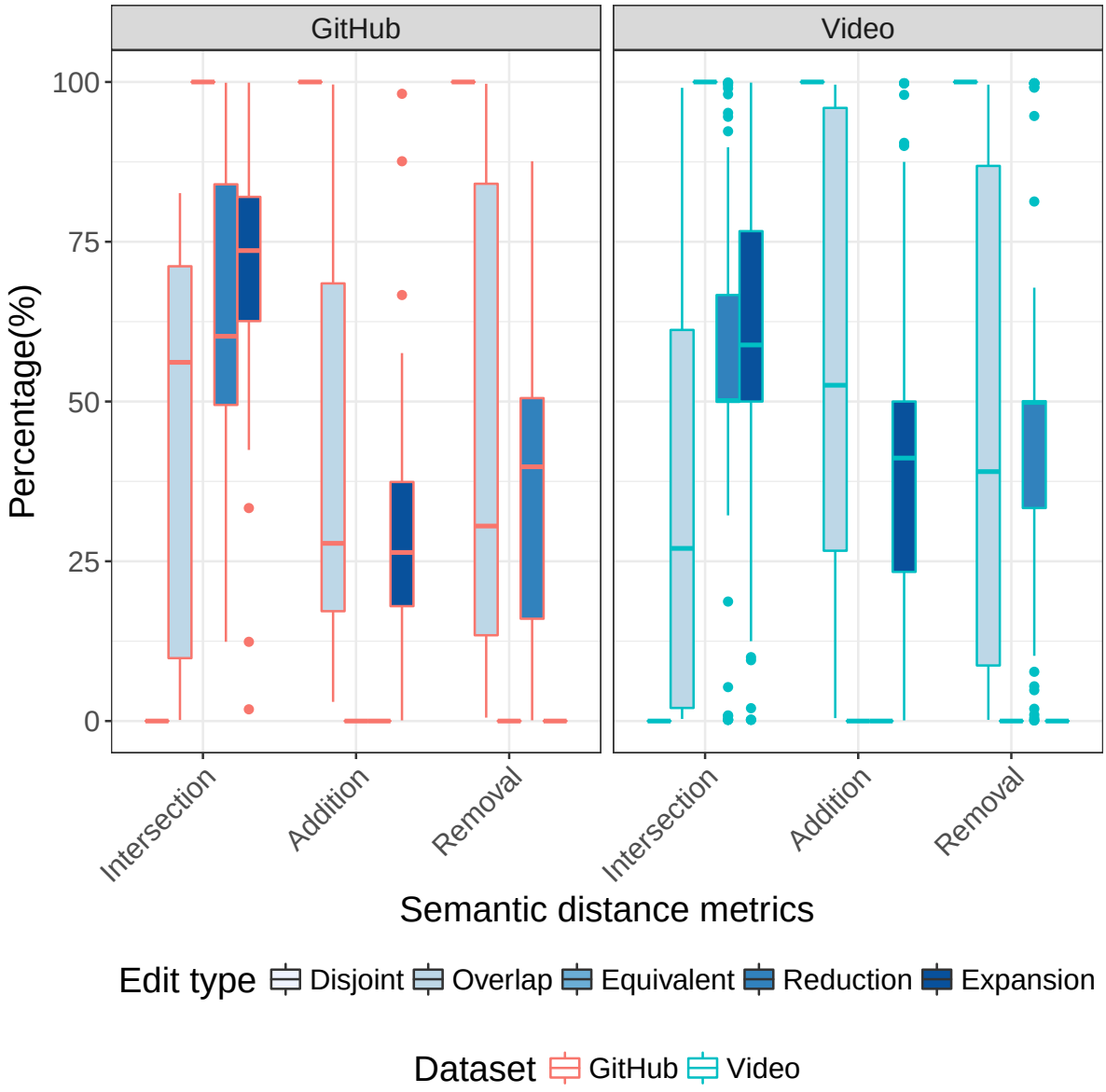


Figure 4.2 The distribution of Add, Remove, and Overlap percentages over disjoint, equal, overlap, subset and superset regular expression edits in GitHub and Video dataset when k is 500.

Table 4.4 Statistics of the language features in GitHub and Video dataset ranked by the number of regular expressions in which features present (Freq), features are added, and features are removed.

rank	GitHub					Video				
	Freq	Add	nR	Remove	nR	Freq	Add	nR	Remove	nR
1	LIT	LIT	123	LIT	62	LIT	LIT	191	LIT	144
2	CG	KLE	53	ADD	27	CG	CG	95	CG	63
3	ADD	QST	42	KLE	24	ANY	KLE	76	KLE	53
4	KLE	ANY	36	QST	23	KLE	ANY	55	ANY	40
5	CCC	CG	34	CCC	22	ADD	ADD	54	ADD	37
6	ANY	ADD	34	CG	21	CCC	CCC	33	CCC	24
7	RNG	CCC	33	ANY	17	WSP	WSP	27	WSP	23
8	STR	RNG	24	DEC	12	RNG	OR	24	OR	15
9	END	OR	20	RNG	11	OR	STR	23	STR	15
10	DEC	NCG	13	END	8	WRD	DEC	21	WRD	12
11	QST	NWSP	9	STR	8	DEC	LZY	16	RNG	11
12	NCCC	NCCC	9	NWSP	7	END	QST	15	DEC	10
13	WSP	WSP	8	OR	6	STR	END	13	END	10
14	OR	WRD	8	NCG	5	LZY	RNG	12	NCCC	10
15	WRD	DEC	7	LWB	5	QST	WRD	12	LZY	8
16	LZY	END	6	DBB	4	WNW	WNW	11	WNW	8
17	SNG	STR	5	LZY	3	NCCC	NCCC	10	QST	7
18	NCG	LZY	4	NCCC	2	SNG	SNG	10	SNG	3
19	NWSP	DBB	3	WSP	2	LKB	LKB	4	LKB	3
20	DBB	OPT	3	WRD	2	BKR	NWRD	3	NCG	3
21	OPT	NDEC	2	SNG	2	NCG	NDEC	2	NLKA	2
22	LWB	LKA	2	LKA	1	LWB	NCG	2	LKA	2
23	WNW	LWB	1	OPT	0	NLKA	BKR	2	LWB	1
24	LKA	SNG	1	NDEC	0	NDEC	LWB	2	NWNW	1
25	NWRD	NWRD	0	NWRD	0	LKA	NLKA	2	NDEC	0

Table 4.5 Distribution of regular expression feature changes among 256 edits in GitHub dataset and 558 edits in Video dataset.

Dataset		Mean	Min	10%	25%	50%	75%	90%	Max
GitHub	F_{add}	3.71	0	0	1	2	5	9	37
	F_{remove}	2.52	0	0	0	0	2	7	37
	F_{add}+F_{remove}	6.23	0	1	2	4	8	15	39
Video	F_{add}	2.60	0	0	0	1	3	6	73
	F_{remove}	1.98	0	0	0	1	2	5	66
	F_{add}+F_{remove}	4.58	0	1	1	2	5	11	75

CHAPTER

5

PROPOSED WORK

In Chapter 5 we first propose two strategies to correct regular expressions: 1) regular expression mutation and 2) regular expression misuse and replacement. And then propose a continuous work on further research of regular expression coverage. The time schedule of completing these two pieces of work can be found in Table 1.1 in Section 1. At the end of this chapter, we also present some other opportunities for future work.

5.1 Regular Expression Mutation

This is a continuous work of understanding regular expression evolution. With the knowledge we gained through exploring regular expression evolution, we can analyze evolved regular expressions as regex mutants via certain mutation operators. Similar to the mutation used in genetic programming for automatic program repair, I conjecture that regular expression mutants play equivalent role for repairing specifically buggy regular expressions. It can be divided into three main piece of work: 1) characterizing regular expression bugs, 2) studying of empirical regular expression mutation patterns according to regular expression bug fixes, and 3) apply the mutants into automatic regular expression repair and possibly measure the effectiveness of this method.

5.1.1 Artifacts

In order to explore the regular expression bug characteristics and mutation patterns, we have searched bug databases for large bug reporting systems, specifically Bugzilla, GitHub, and Apache JIRA. We are going to filter bugs related to regular expressions with searching keywords "regular expression", "regex", and "regexp". The artifacts of regex-related bugs are going to be used in both extracting regular expression patterns and testing the effectiveness of regular expression repair. For the best purpose of validating, it is import to keep regex-related bugs of multiple different languages.

5.1.2 RQ6: Regular Expression Bug Study

The goal of this study is to discover the regex-related bugs whose promising solution is to change the regular expression itself. Some bugs are solved by revising regular expression itself, some others are solved by transforming the regular expression into other forms such as string operations, some even by restricting input strings for matching. In this study we are going to characterize regex-related bugs given the bug causes and their impacts. Possibly, we will also spot possible solutions to them. Below we give three types of regular expression bugs along with their most likely solutions. Other types includes *escape characters*, *case sensitivity*, *special characters*, *match order of alternations*, *requirement and environment changes*, and so on.

5.1.2.1 Catastrophic Backtracking

Backtracking is the most difficult type of regular expression.related bugs. Backtracking happens in the regular expression engines of Nondeterministic Finite Automaton (NFA) implementation. When NFA decides the states it should take, it tries one state a time. If the rest matching process arrives an error state, it backtracks to where it choose the state and selects another state to try. Backtracking usually takes exponential time to complete. It also consumes a lot of computing resources (e.g., CPU and Memory) and therefore slows down the program responses [83]. Sometimes, backtracking in regular expression could even cause crashes of system and software (e.g., browsers, websites, etc) [109]. It can be exploit for ReDoS attacks as well.

Two possible solutions are suggested to this type of bugs: a) Use a regular expression engine implemented in DFA. While the NFA implementation takes regular expression engines exponential time to finish, the DFA version takes linear time. However, it is not always feasible to choose regular expression engines. b) Avoid the regular expression features which causes backtracking. Backtracking is only used for some of the regular expression engines. Finding equivalent regular expression without these features could avoid the backtracking. According to a study of ReDoS attacks [29], the most adopted approach by developers is revising problematic regular expressions.

5.1.2.2 Strings of Zero Length and Long String

If a valid regular expression is given to match a zero-length string, some regular expression engines will try to read a character infinitely. If the regular expression is given a very long string, its execution time is linear to the string length in DFA implemented regular expression engine. This situation is common in regular expression containing wildcard * and +. Long String to match a regular expression could also be used for ReDoS attacks and impact software performance. Strings of zero length may even cause system hangs.

In this type of regular expression bugs, three types of solutions are plausible: 1) restricting the length of input strings within a threshold; 2) changing the regular expression from infinite repetitions (i.e., * and +) to bounded repetitions; or 3) replacing the regular expression by some other implementations, such as proposed work in Section 5.2.

5.1.2.3 Refactoring for Performance Optimization

According to the Java implementation details, every time the method **String.matches(String regex)** is invoked for regular expression matching, a regular expression object of class **java.util.Pattern** must be created first. If the matching happens in a loop, this type of regex matching cause duplicated object creation. In order to avoid unnecessary object creation of class **java.util.Pattern**, developers optimize to initiate the **java.util.Pattern** object out of the loop and out of the method body. This type of regular expression bugs are usually fixed by refactoring the source code and thus does not touch the regular expression at all.

5.1.3 RQ7: Regular Expression Mutation Patterns

The goal of this study is to reveal the empirical regex evolution patterns. Those patterns could be simple mutation operators or be combination of multiple mutation operators. Different from the mutation operators in existing research studies [22, 100], we are not exploring fundamental regular expression features but looking through the bug patches of regex-related bugs and analyze the patterns used for evolve the regular expression. With prior knowledge, we are aware of four aspects of regular expression mutation: 1) one step of regular expression mutation usually involve more than one regular expression feature; 2) one step of regular expression mutation could change the same feature multiple times; 3) a regular expression bug fix may need a couple steps of regular expression mutation, and 4) the change of a regular expression feature can be found commonly in multiple regular expression mutation steps. Below is examples to illustrate these pieces of knowledge.

5.1.3.1 Regular Expression Mutation Examples

In one bug patch[1], the change from / to * is applied to more than one regular expression locations. There are two similar regular expressions defined in two different Java files:
`^rtmps?://([^/:]+)(?::(\\d+))*/([^/]+)/?([^/]*)$` and
`^rtsps?://([^/:]+)(?::(\\d+))*/([^/]+)/?([^/]*)$`. Both of them were changed from `([^/]*)` to `([^*]*)`.

In another bug[2], there are three commits in order to fix it. The first commit changed `\\w` from `alluxio.master.security.impersonation.(\\w+).users` into `[a-zA-Z_0-9-]` from `alluxio.master.security.impersonation.([a-zA-Z_0-9-]+).users`. The escaping was applied in the second commit and thus the regex became `alluxio\\.master\\.security\\.impersonation\\.([a-zA-Z_0-9-\\.]+)\\.users`. In this mutation step, escaping character was applied five times. Another escaping dot `\\.` was added into `[a-zA-Z_0-9-]`. The last commit added a special character `@` and the final regular expression turned to be `alluxio\\.master\\.security\\.impersonation\\.([a-zA-Z_0-9-\\.@]+)\\.users`.

5.1.4 RQ8: Regular Expression Repair Through Mutation

The goal of this study is to measure the effectiveness of applying regular expression mutation into regular expression repair. We are going to use the program repair technique of Generate-and-Validate [61]. Additionally, we will build a repair benchmark for regular expressions. It consists of buggy regexes from the bug reports, correct regular expressions, and the test strings which could reveal the fault of the buggy regexes. Based on related works on building benchmarks [62], the bug reports involved in regex repair is different from the ones used for extracting mutation patterns. We propose to separate the artifacts of regex-related bugs into groups according to the programming language of their source code. For example, bugs in Java projects are used for extracting mutation patterns and bugs in Python projects are used for measuring regex repair.

5.2 Regular Expression Misuse and Replacement

As mentioned above in Section 5.1.2, some regular expression bugs are fixed by replacing regular expressions with other implementations. This represents the problem of regular expression misuse. By exploring a python dataset of around 13,500 regular expressions, we found that many regular expressions are purely character string (e.g. "`get_fabric_name`" and "`set_fabric_sense_len)(`")

[1] https://github.com/pedroSG94/rtmp-rtsp-stream-client-java/issues/208
[2] https://github.com/Alluxio/alluxio/pull/7571

or partly character strings (e.g. "`Revision: (.+)`" and "`cygwin.*`). As shown in section 5.1.2, a major reason for regular expression backtracking and performance issue is because the abuse of * and +. Here we hypothesize that many regular expressions in the software projects are misused and unnecessary and that a large percentage of regular expressions can be replaced with simple string operations. For example, instead of searching "`get_fabric_name`" in a string *s* using `re.search('get_fabric_name',s)`, we can simply use `if 'get_fabric_name' in s`. Similarity, we can use `s.startswith('cygwin')` to replace `re.match('cygwin.*',s)`. Although the problem of regular expression misuse can be easily confirmed, we need to systematically address this problem through research questions as below.

5.2.1 RQ9: Understandability of String Operations

Replacing a regular expression with its implementation of string operations is preferable if the string operations are more understandable than the original regular expressions. In this study, we are going to collect a set of regex-related bugs which are fixed by changing regular expressions into string operations. Similar to the regular expression comprehension study [24], a questionnaire can be made from the collected data and then presented to developers for human study of understandability. The results will be analyzed to answer this research question.

5.2.2 RQ10: Understandability of Automatically Transformed String Operations

If string operations turn out to be more understandable, a research tool is needed in order to detect misused regexes and to replace them with their equivalent string operations. In order to know whether string operations automatically transformed from the regular expression are more understandable, we will explore source code to find misused regular expressions and then file bug reports along with their string operations as the bug fixes. The comparison between regular expression understandability and the understandability of string operations transformed by the automatic tool will be based on the developers feedback.

5.3 Further Exploration of Regular Expression Testing Coverage

This research is a continuous work of Chapter 3. The testing coverage criteria can be complemented with *Path Coverage*. There are five parts of this research: the path feasibility in the DFA of the regexes, the maximum testing coverage for each coverage criteria, the minimum test string size for maximum testing coverage, and testing coverage by mutated regular expressions.

5.3.1 RQ11: DFA Path Feasibility

The testing criteria based on DFA are not always possible because some paths may be infeasible. For example, empty strings lead to zero percentage testing coverage, and hard-coded regular expression matching can only cover part of the DFA paths. This study will be conducted through code analysis and DFA graph analysis.

5.3.2 RQ12: Maximal DFA Testing Coverage

Given the fact of infeasible DFA paths, the testing coverage may not be able to achieve 100%. In this study we will explore the maximal testing coverage by the DFA, the maximal testing coverage with only matching inputs, and the maximal testing coverage with only non-matching inputs.

5.3.3 RQ13: Minimal Test Suite of Strings

Although a regular expression may represent a large or infinite number of strings, there exists a smallest size of testing inputs which can still achieve the best testing coverage. This test suite will satisfy not only the maximum testing coverage but also different types of characters along the DFA state transition. For example, if characters `a-z0-9_$` are acceptable, then the test suite need to satisfy not only characters but also digits and special characters. For range `a-z`, it also satisfies both 'a' and 'z' as they are the ends of the range. For special character $ which is also used as end anchors, it is required to satisfy escaped character `\$` as well.

5.3.4 RQ14: Regular Expression Mutation

This research will be conducted in order to measure whether regular expression mutation generates better testing strings for regular expression testing. It consists of 1) measurement of mutated regular expression quality; and 2) comparison of testing coverage of strings generated by this regex mutation tool and other tools. Additionally, this research question could also involve new metrics of regular expression similarity. For example, using randomly generated strings to measure the semantic similarity and calculating similarity through comparing the Abstract Syntax Tree (AST) or DFA of the regexes.

5.4 Opportunities For Future Work

Mutation Testing: One big motivation of studying regular expression evolution is to apply empirical knowledge to mutation testing of regular expressions. The difference between GitHub and Video

dataset suggest we try different mutation strategies according to the stage of software development. We can define mutation operators depending on the changes among regular expression feature changes and inside each regular expression feature. The priority of various mutants can be ranked according to their syntactic and semantic distance.

Regular Expression Comprehension: Most of the equivalent edits (Section 4.3.2.2) are not reflected in prior work, with one exception (i.e., `[0-9]{1,}` to `[0-9]+` is the same as a L1 to L3 transformation [24]). This provides future opportunities to assess comprehension of edits that reflect source code history.

String-generation Tool: The intersection of regular expression edits is less than 60%, indicating the string-generation tool should generate a certain percentage of strings not matching to the regular expression but matching to its mutants.

CHAPTER

6

RELATED WORK

In Chapter 6 we present regular expression related work in several aspect: regular expression application, regular expression security, regular expression genetic generation, regular expression comprehension, and so on.

6.1 Regular Expression Application

As a query language, lightweight regular expressions are pervasive in search. For example, regular expressions are used as queries in a data mining framework [16]. Another area regular expressions are frequently seen and employed are security-oriented. Regular expressions can be used to prevent and detect SQL injections [111]. They are also research topics in database intrusion detection [63]. Regular expression is popular in DNA sequence matching [39] as well. Regular expressions have been also used for test case generation [4, 40, 42, 102], and as specifications for string constraint solvers [54, 103]. Flipping this around, recent approaches have used mutation to generate test strings for regular expressions themselves [6].

6.2 Regular Expression Security

One common misconception is that all regular expression languages are *regular languages* which can be represented using DFA, and so they are easy to model, easy to describe formally and execute in O(n) time. In fact, many regular expression matching engines run in exponential time in order to support useful features such as lazy quantifiers, capturing groups, look-aheads and back-references [74]. These features make regular expression engines sometimes inevitably to backtrack DFAs, which consumes more resources and takes more time. In the worst case, regular expressions can be exploited for denial-of-service (DoS) attacks, which is therefore called Regular Expression Denial of Service (ReDoS).

As a potential security vulnerability, some research has been focused on the security impact of ReDoS attacks. As a tool, ReScue [93] can craft regular expression attacks. One work statically analyzed regular expressions to check ReDoS attacks [56]. Studies have shown that ReDoS are common in programs [29], especially in javascript-based web servers [97].

6.3 Regular Expression Comprehension

Regular expression understandability has not previously been studied directly, though prior work has suggested that regexes are hard to read and understand [23] and noted that there are tens of thousands of bug reports related to regexes [96]. To aid in regex creation and understanding, tools have been developed to support more robust creation [96], to allow visual debugging [15], or to help programmers complete regex strings [79]. Other research has focused on removing the human from the creation process by learning regular expressions from text [9, 67].

Code smells in object-oriented languages were introduced by Fowler [36]. Researchers have studied the impact of code smells on program comprehension [1, 31], finding that the more smells in the code, the harder the comprehension. Code smells have been extended to other language paradigms including end-user programming languages [48, 49, 98, 99]. Using community standards to define smells has been used in refactoring for end-user programmers [98, 99].

Regular expression refactoring has not been studied directly, though refactoring literature abounds [46, 72, 82]. Refactoring for conformance to the community in end-user programs [98, 99] has been proposed previously. The closest to regex refactoring comes from research recent work that uses genetic programming to optimize regexes for runtime performance while maintaining their behavior in the matching language [26]. Similarly, other research has focused on expediting regular expressions processing on large bodies of text [11], similar to refactoring for performance.

6.4 Regular Expression Testing

Software test coverage can be measured at different levels of granularity, such as method, statement, branch, system, integration, module, and unit (e.g., [2, 65, 69, 86, 114]).

Symbolic execution [3, 18, 19, 110] is one way to generate inputs and to obtain program test coverage at the level of branches. There are many tools for automated test generation [37, 84, 112]. For example, Reggae [66] aims to mitigate the large space exploration issues in generating test inputs for programs with regular expressions.

Static analysis to reduce errors in building regular expressions by using a type system to identify errors like `PatternSyntaxExceptions` and `IndexOutOfBoundsExceptions` at compile time [96].

6.5 Regular Expression Generation

6.5.1 String Generations for Regular Expression

Rex [104] generates testing inputs for the regular expression according to its SFA representation. `brics` [73] generates inputs by traversing the DFA and building strings from the smallest bytes to the largest bytes of every DFA states. Some string generation tools need user-specified string length [42, 53, 73]. Some string solvers [53] and tools for generating testing inputs which use string solvers [42, 108] build finite-state automata based on string constraints.

EGRET [59] is focused on generating unexpected test strings to expose the regular expression errors, but it is based on common mistakes when creating regular expression rather than maximizing test coverage of regular expressions. MUTREX [6] employs distinguishing strings which can separate a mutated regular expression from the original one to expose system faults.

6.5.2 Regular Expression Generation

Regular expression generation research includes algorithms and tools to generate test cases with regular expressions [4, 5, 102], generating regular expressions for DNA sequences [58] and matching patterns [22] [44] [80]. Enhancing regular expression pattern matching and processing speed is a common focus as well [10, 14, 17, 26]. The regular expression can be learned through genetic programming provided with a large number of labeled strings [12, 13, 67] Genetic programming has also been applied [26] to find equivalent alternative regular expressions which exhibit improved performances. For example, ReLIE [67] shows an algorithm of extracting regular expression from information and [9] presents another extraction algorithm for noisy unstructured text.

6.6 Regular Expression Evolution

6.6.1 Source Code Mining

Mining properties of open source repositories is a well-studied topic. Exploring language feature usage by mining source code has been studied extensively for Smalltalk [20, 21], JavaScript [91], Python [23], and Java [33, 45, 68, 85], and more specifically, Java generics [85], Java reflection [68], and Python regular expressions [23]. One study of source code evolution focuses on several popular open source programs and identifies the code changes through AST matching [78]. However, the prior work in regular expression mining looked at the most recent version of the regular expressions, not at their edit histories.

The *code churn* metric represents the number of lines of code that were added/deleted or changed. This concept is was introduced as a means to measure the impact of code changes [76] and has been used to predict software failures and defects [43, 50, 57, 71, 77, 94]. The classic definition of code churn is the absolute value of a code delta between two sequential builds. For regular expressions, we borrow the concept of code churn and measure regular expression evolution by calculating the Levenshtein distance. We do not, however, have a measure of regular expression faultiness to tie together distance with fault-proneness; exploring that is left for future work.

6.6.2 Language Features

The language features and their usage were explored in diverse prior work by mining source code in programming languages, such as Java [33] [85], JavaScript [91]. It is common for the programming language to have new language features as it evolves, and Dyer et al. investigated how these new features are adopted by programmers once released and the most frequently used features [33]. They concluded that the adoption rate of Annotation Use is exceptionally high and about half of the usage is `@Override` annotation. Generic Variable declarations are popular among developers as well, and the top three types are `List`, `ArrayList` and `Map`. These results are consistent with the work of Parnin et al [85].

Chapman and Stolee also studied regular expressions in Python [23]. They explored regular expression behavioral similarity among projects [23], as well as the influential factors to the comprehension for regular expressions [24]. They also pointed out the top three language features appear in projects are ADD (one-or-more repetition with plus sign), CP (a capture group with parentheses) and KLE (zero-or-more repetition with star sign).

6.6.3 Regular Expression Similarity

Regular expression similarity has been studied, but tool support is sparse. Rot, et al. proved regular expression language equivalence and the inclusion of DFAs that represent a subset of regular expression features [92], though no implementation is available. Dulucq, et al., defined tree edit distances [32] that can be applied to regular expression parse trees, which complements recent work proving that parse tree subsumption implies language subsumption [47]. Wang, et al. worked to cluster patterns by syntactic similarity [105] and others have worked to enumerate strings of regular expressions [70, 104]. These efforts toward computing and understanding regular expression similarity largely lack implementations.

6.7 Regular Expression Tools

6.7.1 General-purposed Regular Expression Tools

Visualizations debugging tools [**regexpr**, 15, 81] are developed to aid regular expression comprehension, and may provide some explanation for low test coverage of regular expressions in source code, that is, developers use online tools instead.

6.7.2 DFA-related Regular Expression Tools

With respect to the finite automaton constructed from regular expressions, `brics` [73] contains a DFA implementation with very limited operations; while RE2 [27, 28] provides a DFA implementation which runs much faster than traditional regular expression engines in Java, Perl, and Python. Rex [104] builds a symbolic representation of finite automata (SFA).

CHAPTER

7

CONCLUSION

This work presents research on regular expression testing coverage and regular expression evolution. As regular expression evolution is the first step towards regular expression repair, we propose research of regular expression mutation in order to fix incorrect regular expressions. We also propose regular expression replacement to remove misused regular expressions. Moreover, we propose a further exploration on regular expression testing coverage according to discussed problems.

To our knowledge, we present the first research to evaluate fine-grained coverage metrics for regular expressions. We explore coverage over the DFA representation of a regular expression. It is also the first research to explore how often regular expressions are tested in a large set of the software projects. We show the coverage metrics of regular expressions from 1,225 GitHub Java Maven projects and found that over 80% of *FullMatch* functions are not tested and that most of the tested regular expressions have a low edge and edge-pair coverage. We also show that with the help of the regular expression tool Rex it is possible to improve the regular expression testing coverage by adding input strings, but that there is an upper bound for this type of improvement.

Regarding to regular expression evolution, we explore how regular expressions evolve through two lenses, GitHub commits and tested regular expressions during problem-solving tasks (called the Video dataset). We find that the GitHub regular expression edits are larger syntactically, but produce smaller semantic changes. The edit chains in the Video dataset are longer than the GitHub

edit chains, indicating that developers may go through multiple iterations prior to committing a regular expression to a repository. The most common change to the scope of the regular expression in GitHub is to expand the matching language, which motivates the use of mutation operators to generate strings outside the original language for testing. Our results provide insights on the types and frequencies of edits that occur in regular expressions and can be used to guide mutation operators that reflect developer practices.

Two strategies are proposed in this work for regular expression repair. One is to apply empirical regular expression mutation patterns in order to automatically repair regex-related bugs. The other is to avoid error-prone regular expressions by converting them into string operations. This work also proposes an extension of regular expression testing coverage for better understanding the limitations and possible improvements of regular expression testing criteria. With these proposed pieces of work, we hope that our research could help developers better test regular expressions and ease the burden of solving regex-related bugs.

BIBLIOGRAPHY

[1] Abbes, M. et al. “An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension”. *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE. 2011, pp. 181–190.

[2] Ammann, P. & Offutt, J. *Introduction to software testing*. Cambridge University Press, 2016.

[3] Anand, S., Păsăreanu, C. & Visser, W. “JPF–SE: A symbolic execution extension to java pathfinder”. *Tools and Algorithms for the Construction and Analysis of Systems* (2007), pp. 134–138.

[4] Anand, S. et al. “An Orchestrated Survey of Methodologies for Automated Software Test Case Generation”. *J. Syst. Softw.* **86**.8 (2013), pp. 1978–2001.

[5] Arcaini, P., Gargantini, A. & Riccobene, E. “Fault-based test generation for regular expressions by mutation”. *Software Testing, Verification and Reliability* (), e1664.

[6] Arcaini, P., Gargantini, A. & Riccobene, E. “Mutrex: A mutation-based generator of fault detecting strings for regular expressions”. *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on*. IEEE, 2017, pp. 87–96.

[7] Arcaini, P., Gargantini, A. & Riccobene, E. “Interactive Testing and Repairing of Regular Expressions”. *IFIP International Conference on Testing Software and Systems*. Springer, 2018, pp. 1–16.

[8] Arslan, A. N. “Multiple sequence alignment containing a sequence of regular expressions”. *Computational Intelligence in Bioinformatics and Computational Biology, 2005. CIBCB’05. Proceedings of the 2005 IEEE Symposium on*. IEEE, 2005, pp. 1–7.

[9] Babbar, R. & Singh, N. “Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text”. *Proceedings of the fourth workshop on Analytics for noisy unstructured text data*. ACM, 2010, pp. 43–50.

[10] Backurs, A. & Indyk, P. “Which Regular Expression Patterns Are Hard to Match?” *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 457–466.

[11] Baeza-Yates, R. A. & Gonnet, G. H. “Fast Text Searching for Regular Expressions or Automaton Searching on Tries”. *J. ACM* **43**.6 (1996), pp. 915–936.

[12] Bartoli, A. et al. “Automatic generation of regular expressions from examples with genetic programming”. *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*. ACM, 2012, pp. 1477–1478.

[13] Bartoli, A. et al. “Inference of regular expressions for text extraction from examples”. *IEEE Transactions on Knowledge and Data Engineering* **28**.5 (2016), pp. 1217–1230.

[14] Becchi, M. & Crowley, P. “Efficient Regular Expression Evaluation: Theory to Practice”. *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS ’08. San Jose, California: ACM, 2008, pp. 50–59.

[15] Beck, F. et al. “Regviz: Visual debugging of regular expressions”. *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 504–507.

[16] Begel, A., Khoo, Y. P. & Zimmermann, T. “Codebook: Discovering and Exploiting Relationships in Software Repositories”. *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. Cape Town, South Africa: ACM, 2010, pp. 125–134.

[17] Brodie, B. C., Taylor, D. E. & Cytron, R. K. “A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching”. *33rd International Symposium on Computer Architecture (ISCA’06)*. 2006, pp. 191–202.

[18] Bucur, S. et al. “Parallel symbolic execution for automated real-world software testing”. *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 183–198.

[19] Cadar, C., Dunbar, D., Engler, D. R., et al. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.” *OSDI*. Vol. 8. 2008, pp. 209–224.

[20] Callaú, O. et al. “How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk”. *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 23–32.

[21] Callaú, O. et al. “How (and Why) Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk”. *Empirical Software Engineering* **18**.6 (2013), pp. 1156–1194.

[22] Cetinkaya, A. “Regular expression generation through grammatical evolution”. *Proceedings of the 9th annual conference companion on Genetic and evolutionary computation*. ACM, 2007, pp. 2643–2646.

[23] Chapman, C. & Stolee, K. T. “Exploring regular expression usage and context in Python”. *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 282–293.

[24] Chapman, C., Wang, P. & Stolee, K. T. “Exploring regular expression comprehension”. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 405–416.

[25] Chiba, S. “Javassist-a reflection-based programming wizard for Java”. *Proceedings of OOPSLA’98 Workshop on Reflective Programming in C++ and Java*. Vol. 174. 1998.

[26] Cody-Kenny, B. et al. “A search for improved performance in regular expressions”. *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 1280–1287.

[27] Cox, R. “Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...)” *URL:*`http//swtch.com/~rsc/regexp/regexp1.html` (2007).

[28] Cox, R. “Regular expression matching in the wild”. *URL:*`http//swtch.com/~rsc/regexp/regexp3.html` (2010).

[29] Davis, J. C. et al. “The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale”. *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2018.

[30] De Moura, L. & Bjørner, N. “Z3: An efficient SMT solver”. *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[31] Du Bois, B. et al. “Does god class decomposition affect comprehensibility?” *IASTED Conf. on Software Engineering*. 2006, pp. 346–355.

[32] Dulucq, S. & Touzet, H. “Analysis of tree edit distance algorithms”. *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2003, pp. 83–95.

[33] Dyer, R. et al. “Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: ACM, 2014.

[34] *exrex 0.5.2 documentation*. `https://exrex.readthedocs.io/`.

[35] Ficara, D. et al. “An improved DFA for fast regular expression matching”. *ACM SIGCOMM Computer Communication Review* **38**.5 (2008), pp. 29–40.

[36] Fowler, M. *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[37] Fraser, G. et al. “Does automated white-box test generation really help software testers?” *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM. 2013, pp. 291–301.

[38] Friedl, J. E. *Mastering regular expressions*. " O’Reilly Media, Inc.", 2002.

[39] Galassi, U. & Giordana, A. “Learning regular expressions from noisy sequences”. *International Symposium on Abstraction, Reformulation, and Approximation*. Springer. 2005, pp. 92–106.

[40] Galler, S. J. & Aichernig, B. K. “Survey on Test Data Generation Tools”. *Int. J. Softw. Tools Technol. Transf.* **16**.6 (2014), pp. 727–751.

[41] *Generex: A Java library for generating String from a regular expression*. `https://github.com/mifmif/Generex/`.

[42] Ghosh, I. et al. "JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings". *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 992–1001.

[43] Giger, E., Pinzger, M. & Gall, H. C. "Comparing fine-grained source code changes and code churn for bug prediction". *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM. 2011, pp. 83–92.

[44] González-Pardo, A. et al. "A Case Study on Grammatical-Based Representation for Regular Expression Evolution". *Trends in Practical Applications of Agents and Multiagent Systems*. Ed. by Demazeau, Y. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 379–386.

[45] Grechanik, M. et al. "An Empirical Investigation into a Large-scale Java Open Source Code Repository". *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '10. New York, NY, USA: ACM, 2010.

[46] Griswold, W. G. & Notkin, D. "Automated Assistance for Program Restructuring". *ACM Trans. Softw. Eng. Methodol.* **2**.3 (1993), pp. 228–269.

[47] Henglein, F. & Nielsen, L. "Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation". *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 385–398.

[48] Hermans, F., Pinzger, M. & Deursen, A. van. "Detecting Code Smells in Spreadsheet Formulas". *Proc. of ICSM '12*. 2012, pp. 409–418.

[49] Hermans, F., Pinzger, M. & Deursen, A. van. "Detecting and refactoring code smells in spreadsheet formulas". English. *Empirical Software Engineering* (2014), pp. 1–27.

[50] Hovsepyan, A. et al. "Software Vulnerability Prediction Using Text Analysis Techniques". *Proceedings of the 4th International Workshop on Security Measurements and Metrics*. MetriSec '12. Lund, Sweden: ACM, 2012, pp. 7–10.

[51] Hutchings, B. L., Franklin, R. & Carver, D. "Assisting network intrusion detection with reconfigurable hardware". *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*. IEEE. 2002, pp. 111–120.

[52] Inozemtseva, L. & Holmes, R. "Coverage is Not Strongly Correlated with Test Suite Effectiveness". *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 435–445.

[53] Kiezun, A. et al. “HAMPI: a solver for string constraints”. *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM. 2009, pp. 105–116.

[54] Kiezun, A. et al. “HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars”. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **21**.4 (2012), p. 25.

[55] King, J. C. “Symbolic execution and program testing”. *Communications of the ACM* **19**.7 (1976), pp. 385–394.

[56] Kirrage, J., Rathnayake, A. & Thielecke, H. “Static analysis for regular expression denial-of-service attacks”. *International Conference on Network and System Security*. Springer, 2013, pp. 135–148.

[57] Knab, P., Pinzger, M. & Bernstein, A. “Predicting Defect Densities in Source Code Files with Decision Tree Learners”. *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR ’06. Shanghai, China: ACM, 2006, pp. 119–125.

[58] Langdon, W. B. & Harrison, A. P. “Evolving Regular Expressions for GeneChip Probe Performance Prediction”. *Parallel Problem Solving from Nature – PPSN X*. Ed. by Rudolph, G. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1061–1070.

[59] Larson, E. & Kirk, A. “Generating evil test strings for regular expressions”. *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE, 2016, pp. 309–319.

[60] Le, X. B. D., Lo, D. & Le Goues, C. “History Driven Program Repair”. *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE, 2016, pp. 213–224.

[61] Le Goues, C. et al. “Genprog: A generic method for automatic software repair”. *Ieee transactions on software engineering* **38**.1 (2012), p. 54.

[62] Le Goues, C. et al. “The ManyBugs and IntroClass benchmarks for automated repair of C programs”. *IEEE Transactions on Software Engineering* **41**.12 (2015), pp. 1236–1256.

[63] Lee, S. Y., Low, W. L. & Wong, P. Y. “Learning fingerprints for a database intrusion detection system”. *European Symposium on Research in Computer Security*. Springer, 2002, pp. 264–279.

[64] Levenshtein, V. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. *Soviet Physics Doklady* **10** (1966), p. 707.

[65] Li, N., Praphamontripong, U. & Offutt, J. “An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage”. *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE. 2009, pp. 220–229.

[66] Li, N. et al. “Reggae: Automated test generation for programs using complex regular expressions”. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 515–519.

[67] Li, Y. et al. “Regular expression learning for information extraction”. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2008, pp. 21–30.

[68] Livshits, B., Whaley, J. & Lam, M. S. “Reflection Analysis for Java”. *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS'05. Berlin, Heidelberg: Springer-Verlag, 2005.

[69] Malaiya, Y. K. et al. “Software reliability growth with test coverage”. *IEEE Transactions on Reliability* **51**.4 (2002), pp. 420–426.

[70] Mcilroy, M. D. “Enumerating the Strings of Regular Languages”. *J. Funct. Program.* **14**.5 (2004), pp. 503–518.

[71] Meneely, A. et al. “Predicting Failures with Developer Networks and Social Network Analysis”. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '08/FSE-16. Atlanta, Georgia: ACM, 2008, pp. 13–23.

[72] Mens, T. & Tourwé, T. “A Survey of Software Refactoring”. *IEEE Trans. Soft. Eng.* **30**.2 (2004), pp. 126–139.

[73] Møller, A. *dk.brics.automaton – Finite-State Automata and Regular Expressions for Java*. `http://www.brics.dk/automaton/`. 2017.

[74] *MSDN - Matching Behavior*. `https://msdn.microsoft.com/en-us/library/0yzc2yb0.aspx`. 2015.

[75] Munaiah, N. et al. “Curating GitHub for engineered software projects”. *Empirical Software Engineering* **22**.6 (2017), pp. 3219–3253.

[76] Munson, J. C. & Elbaum, S. G. “Code Churn: A Measure for Estimating the Impact of Code Change”. *Proceedings of the International Conference on Software Maintenance*. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 24–.

[77] Nagappan, N. & Ball, T. “Use of Relative Code Churn Measures to Predict System Defect Density”. *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: ACM, 2005, pp. 284–292.

[78] Neamtiu, I., Foster, J. S. & Hicks, M. “Understanding source code evolution using abstract syntax tree matching”. *ACM SIGSOFT Software Engineering Notes* **30**.4 (2005), pp. 1–5.

[79] Omar, C. et al. “Active Code Completion”. *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 859–869.

[80] O’Neill, M. & Ryan, C. “Grammatical evolution”. *IEEE Transactions on Evolutionary Computation* **5**.4 (2001), pp. 349–358.

[81] *Online regex tester, debugger with highlighting for PHP, PCRE, Python, Golang and JavaScript*. `https://regex101.com/`.

[82] Opdyke, W. F. “Refactoring object-oriented frameworks”. PhD thesis. University of Illinois at Urbana-Champaign, 1992.

[83] *Outage Postmortem - July 20, 2016*. `http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016`.

[84] Pacheco, C. & Ernst, M. D. “Randoop: feedback-directed random testing for Java”. *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM. 2007, pp. 815–816.

[85] Parnin, C., Bird, C. & Murphy-Hill, E. “Adoption and Use of Java Generics”. *Empirical Softw. Engg.* **18**.6 (2013).

[86] Piwowarski, P., Ohba, M. & Caruso, J. “Coverage measurement experience during function test”. *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press. 1993, pp. 287–301.

[87] *RegExr: Learn, Build, and Test RegEx*. `https://regexr.com/`.

[88] *Regldg: a regular expression grammar language dictionary generator*. `https://regldg.com/`.

[89] *Regular expression visualizer using railroad diagrams*. `https://regexper.com/`.

[90] *Rex @ rise4fun from Microsoft*. `http://rise4fun.com/rex/`.

[91] Richards, G. et al. “An Analysis of the Dynamic Behavior of JavaScript Programs”. *SIGPLAN Not.* **45**.6 (2010).

[92] Rot, J., Bonsangue, M. & Rutten, J. “Proving Language Inclusion and Equivalence by Coinduction”. *Inf. Comput.* **246**.C (2016), pp. 62–76.

[93] Shen, Y. et al. “ReScue: crafting regular expression DoS attacks”. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 225–235.

[94] Shin, Y. et al. “Evaluating Complexity, Code Churn, and Developer Activity Metrics As Indicators of Software Vulnerabilities”. *IEEE Trans. Softw. Eng.* **37**.6 (2011), pp. 772–787.

[95] Sipser, M. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston, 2006.

[96] Spishak, E., Dietl, W. & Ernst, M. D. “A type system for regular expressions”. *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. ACM, 2012, pp. 20–26.

[97] Staicu, C.-A. & Pradel, M. “Freezing the web: A study of redos vulnerabilities in javascript-based web servers”. *27th USENIX Security Symposium (USENIX Security 18)(Baltimore, MD, 2018), USENIX Association*. 2017.

[98] Stolee, K. T. & Elbaum, S. “Refactoring pipe-like mashups for end-user programmers”. *International Conference on Software Engineering*. Waikiki, Honolulu, HI, USA, 2011.

[99] Stolee, K. T. & Elbaum, S. “Identification, impact, and refactoring of smells in pipe-like web mashups”. *IEEE Transactions on Software Engineering* **39**.12 (2013), pp. 1654–1679.

[100] Tarlao, F. “Genetic Programming Techniques for Regular Expression inference from Examples” (2017).

[101] *The Bro Network Security Monitor*. https://www.bro.org/. 2015.

[102] Tillmann, N., Halleux, J. de & Xie, T. “Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger”. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. Vasteras, Sweden: ACM, 2014, pp. 385–396.

[103] Trinh, M.-T., Chu, D.-H. & Jaffar, J. “S3: A Symbolic String Solver for Vulnerability Detection in Web Applications”. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1232–1243.

[104] Veanes, M., De Halleux, P. & Tillmann, N. “Rex: Symbolic regular expression explorer”. *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE. 2010, pp. 498–507.

[105] Wang, H. et al. “Clustering by Pattern Similarity in Large Data Sets”. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’02. Madison, Wisconsin: ACM, 2002, pp. 394–405.

[106] Wang, P., Gina, R. & Stolee, K. T. “Exploring Regular Expression Evolution”. *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019.

[107] Wang, P. & Stolee, K. T. “How well are regular expressions tested in the wild?” *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2018, pp. 668–678.

[108] Wassermann, G. et al. “Dynamic test input generation for web applications”. *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM. 2008, pp. 249–260.

[109] *Why does this Javascript match() function Crash My Browser?* `https://stackoverflow.com/questions/12825950/why-does-this-javascript-match-function-crash-my-browser`.

[110] Xie, T. et al. “Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution.” *TACAS*. Vol. 3440. Springer. 2005, pp. 365–381.

[111] Yeole, A. & Meshram, B. “Analysis of different technique for detection of SQL injection”. *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*. ACM, 2011, pp. 963–966.

[112] Zhang, S. et al. “Combined static and dynamic automated test generation”. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM. 2011, pp. 353–363.

[113] Zhao, H. et al. “Fully automatic wrapper generation for search engines”. *Proceedings of the 14th international conference on World Wide Web*. ACM. 2005, pp. 66–75.

[114] Zhu, H., Hall, P. A. & May, J. H. “Software unit test coverage and adequacy”. *Acm computing surveys (csur)* **29**.4 (1997), pp. 366–427.