Exploring Regular Expression Evolution

Peipei Wang
North Carolina State University
Raleigh, NC, USA
pwang7@ncsu.edu

Gina R. Bai
North Carolina State University
Raleigh, NC, USA
rbai2@ncsu.edu

Kathryn T. Stolee
North Carolina State University
Raleigh, NC, USA
ktstolee@ncsu.edu

***Abstract*—Although there are tools to help developers understand the matching behaviors between a regular expression and a string, regular-expression related faults are still common. 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.**

In this work, we explore 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.

***Index Terms*—Regular expressions, evolution, empirical studies**

I. Introduction

Regular expressions are employed in data validation, classification, and extraction (e.g., [1]–[3]). Understanding and writing regular expressions [4], [5] requires both knowledge and skills. A single character mistake could cause drastically different regular expression matching behaviors. However, over 80% of regular expressions written in GitHub projects are not tested [6], indicating that developers either do not test regular expressions or use external tools rather than test cases.

Regular expression test generation tools exist for regular expressions [7]–[11]. These tools enumerate members of the regular expression language but do not enumerate non-matching strings. Recent research efforts have explored a fault-based approach to generating regular expression tests through mutation testing [12]; 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 mature research area, the focus on regular expression mutation is recent [12]–[16]. However, these efforts have defined the mutation operators in an ad hoc manner, without consideration of how regular expressions actually evolve in practice. Hence, the faults injected might not be representative of actual edits.

In this paper, we explore how regular expressions evolve over time. Beyond shedding light on the types of edits to consider in fault-based test generation, the history of source code development is valuable in guiding program repairs [17]. Given the fault-proneness of regular expressions [18], and that developers under-test their regular expressions [6], understanding 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 within the regular expression's language.

We study regular expression evolution on two datasets collected separately from two different contexts: Github data and Video data. 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 reveals 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.

This work makes the following contributions:

- 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.
- Insights into how regular expressions are edited syntactically and semantically over time, using several metrics: Levenshtein distance, semantic distance, and feature changes.

Our results shows that:

- 95% of literal regular expressions in GitHub (i.e., that do not contain variables) do not evolve (RQ1),
- Approximately half the edits in both datasets contain six or fewer character modifications (RQ2),
- Over half the edits in the GitHub dataset expand the scope of the language whereas the most common edits in the

Video dataset create a disjoint language (RQ2), and

- Most edits in the GitHub and Video datasets involve adding and/or removing 4-6 language features (RQ3).

While most regular expressions do not change, understanding the common changes, and the changes to the language, provides valuable insights for designing mutation operators to assess the quality of test suites and for repairing faulty regular expressions, two promising directions of future work.

II. Motivation

While we show later that a majority of regular expressions do not evolve (see Section VI), prior work indicates that regular expression bug reports abound [19] and that regular expressions are under-tested [6]. 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 at once. This provides evidence that understanding evolution is important for test generation and repair.

A. 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 [20], 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 tracking so the scope of the regular expression is not changed by the mutation. In another example [21], regular expression `(\\W|\\d|_)` is changed to `[^A-Za-z]` for better regular expression readability. (To clarify, `\\W` is equivalent to the negated character class `[^A-Za-z_0-9]`, and `\\d` is equivalent to `[0-9]`; making `[0-9]` valid in the desired character class; the underscore is treated similarly).

These provide evidence that not all edits modify the behavior, and thus not all testing efforts should focus on matching behavior [6], but rather on performance and understandability.

B. Regular Expression Feature Changes

The state-of-the-art literature on fault-injection [22] and fixing regular expressions [13] uses simple faults. However, fixing regular expressions in bug reports often involves more than one feature. For example [20], the regular expression

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

is changed 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, and 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.

III. Research Questions

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 programmers solving regular expression tasks. Our research questions are:

RQ1: *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 appears multiple times.

RQ2: *How similar is a regular expression to its predecessor syntactically and semantically?*
We explore syntactic similarity with the Levenshtein distance [23] between regular expression strings. Regarding semantic similarity, for a regular expression r_2 that evolves from r_1, we empirically measure the overlap between the languages (i.e., $L(r_2)$ and $L(r_1)$).

RQ3: *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.

IV. Analysis

A series of edited regular expressions over time is called an *edit chain*. 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"$\}$

Next, we describe how the 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 a discussion of the limitations.

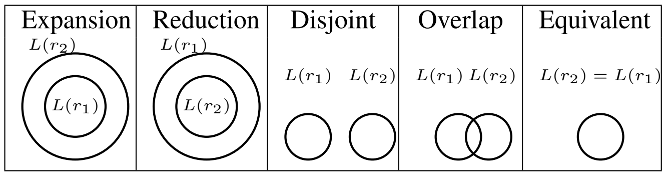

Fig. 1. Types of semantic evolution from r_1 to r_2

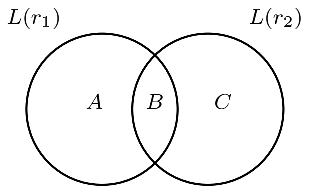

Fig. 2. Notation for sets A, B and C to compute the semantic evolution from r_1 to r_2.

A. 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 [16] and thus chose Levenshtein distance.

Levenshtein distance [23] 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 $\rightarrow$ {), four additions (i.e., **0,2**}) and one removals (i.e., ~~?~~) as follows: c~~a~~{**0,2**}a~~?~~a?b

B. 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 [5], 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. Formally:

$A = L(r_1) \setminus L(r_2)$
$B = L(r_1) \cap L(r_2)$, 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 1[1]. 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*. 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*.

[1]Prior work used mutation for test case generation and defined similar relationships between regular expressions, omitting disjoint [22].

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) 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:

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

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 compute the percentage of $L(r_1)$ 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 $L(r_2)$.

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 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 $L(r_1)$.

C. Language Features

The language features used in a regular expression may evolve as the regular expression evolves. In the example

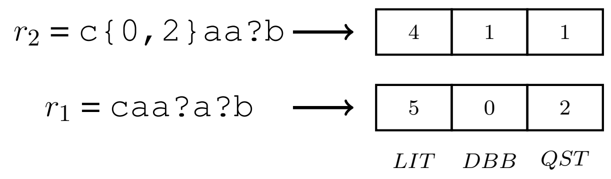

Fig. 3. Example r_1 and r_2 parsed into feature vectors. LIT = Literal, DBB = Double-Bounded, QST = Questionable

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 [5], Figure 3 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 3, 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.

D. Implementation

For the regular expressions, we use an ANTLR-based PCRE (Perl Compatible Regular Expressions) parser[1] to extract features used in a regular expression to create a feature vector. We use the class Levenshtein Distance provided in the Java library of Apache Commons Text [24].

For semantic distance, we approximate the language for $L(r)$ by generating strings for r using Rex [7]. 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 strings which could successfully match r. We tried $k = 100$, $k = 200$, and $k = 500$; because the difference in the distribution of regular expression edit types are negligible, we choose to present the results with $k = 500$.

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.

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

E. 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.

For infinite languages, and languages larger than the upper bound on strings generated (i.e., $k = 500$), the approach described in Section IV-B 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 IV). 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.

V. 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 developers solving regular expression tasks in an IDE, providing a low-level view of evolution during problem-solving tasks.

A. 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.

1) Collecting Regular Expressions in Latest Version: Our data collection starts within the 1,114 Java projects used in prior work on testing regular expression [6]. 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.

2) Building the Edit Chains: 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.

B. 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.

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 made publicly available[1].

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.

[1]https://github.com/wangpeipei90/VideoRegexTasks/.

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) *and then* submitted by participants for execution (i.e., at least one test is run). In this way, a node in an edit chain is created at every test suite execution.

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.

VI. RQ1: Regular Expression Evolution Characteristics

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

A. Edit Frequency of Regular Expressions

In the GitHub dataset, the 3,962 edit chains contain 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 a regular expression is committed to a GitHub repository, edits are rare; 95% of the literal regular expressions were not edited.

B. 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 and thrown 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. 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 appear in the early stages of evolution.

TABLE I
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

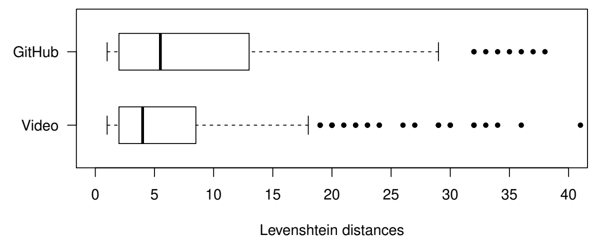

Fig. 4. The distribution of Levenshtein distances for 262 GitHub edits (with distance > 0) and 647 Video edits (with distance > 0).

C. *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 ($i < j < k$) 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. This behavior is not seen as commonly in the GitHub dataset.

VII. RQ2: SYNTACTIC AND SEMANTIC EVOLUTION

We explore RQ2 regarding syntactic and semantic evolution.

A. *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 user in a newer version[2]. The original and modified regular expressions are, respectively,

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

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

TABLE II
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

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).

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.

Figure 4 shows the distribution of Levenshtein distances among the edits in both dataset; Table I details the distributions of the Levenshtein distances. 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.

B. *Semantic Similarity*

Within each edit chain, we look at each edit and classify it according to the semantic evolution types in Figure 1. In order to compute the similarity between regular expressions, we adopt an approach from prior work [5] and use Rex [7]. We generate up to $k = 500$ strings for each regular expression, as described in Section IV-D.

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

TABLE III
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

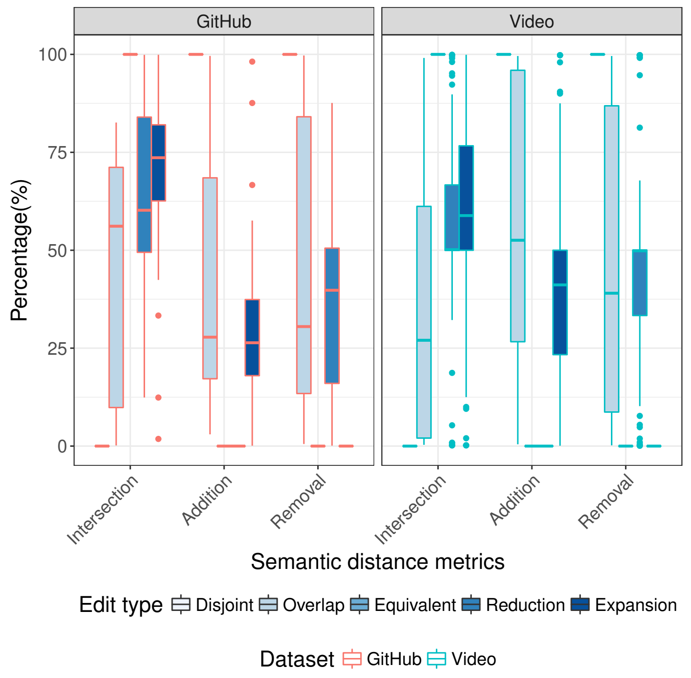

Fig. 5. The distribution of Add, Remove, and Overlap percentages over disjoint, equal, overlap, subset and superset regular expression edits in GitHub and Video dataset when k is 500.

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.

2) Results: Table II shows the number of different edit types in the GitHub and Video datasets. The most common edit is *expansion* for GitHub (50.72%); *disjoint* is the most common edit for the Video dataset (42.81%). Table III 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 II). 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 5 which visualizes the distributions of metrics per semantic edit type. Because *disjoint* and *equivalent* has either no intersection or are completely the same, these two edit types are shown as horizontal lines at 0% and 100%. 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 repetition time from `[a-z0-9_-]{1,64}` to `[a-z0-9_-]{1,120}`. This misclassification is because the length of strings generated by Rex appears to be 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 [4]. 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-.]+$` to `[.]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$`[1], 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 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 include larger semantic changes. GitHub edits represent small adjustments of the regular expression close to targeted scope.

[1]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

TABLE IV
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 (ADD), AND FEATURES ARE REMOVED (REMOVE).

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

ADD: one-or-more repetition (`a+`) QST: zero-or-one repetition (`a?`)
KLE: zero-or-more repetition (`a*`) DBB: double bounded repetition (`a{2,4}`)
SNG: exactly n repetition (`a{3}`) OR: logical or (`a|b`) LIT: literal character (`a`)

VIII. RQ3: 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 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 [5].

For the GitHub dataset, we started with 262 edits; 3 were removed due to Java runtime errors (Section VI-B) and three were removed due to PCRE parsing errors, leaving us with 256 edits. 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.

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 IV 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 3) 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

TABLE V
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

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 V 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.

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))`.

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.

IX. DISCUSSION

In this work, we look at the evolution of regular expressions from two perspectives, syntactic and semantic, and in two contexts, using GitHub projects with commit histories and developer during problem-solving tasks.

A. Implications

Most literal regular expressions in GitHub do not evolve (Section VI), and yet, bug reports related to regular expressions abound [19]. This may indicate that our results are only limited to literal regular expressions. As we only studied literal regular expressions which still exist in the latest code versions, it does not cover the ones which are removed in previous commits.

Yet, regular expressions are under-tested [6], and most string-generation efforts focus on generating test inputs within the language of the regular expression (e.g., [7]). For those regular expressions that do evolve, 50% of the edits in GitHub are expansion edits (Section VII), 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 [12]. However, to do this effectively, these faults need to be reflective of edits that developers make to regular expressions. The analysis of semantic changes in regular expression evolution suggests that in mutation testing we should focus on meaningful mutants which have a high semantic overlap with the original one. We should also pay more attention to regular expressions with multiple faults since a typical regular expression edit involves a few feature changes.

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 VII). Edits also impact multiple language features (Table V).

For the Video dataset, the edits tend to be smaller in terms of character modifications (Table I), but larger in terms of semantic distancing (Table III 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 V). The Video dataset edit chains are also longer than the GitHub edit chains (Section V). 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.

B. Threats to Validity

Internal: We measure similarity using a string-generation approach that provides an approximate measure of similarity. We also observed in Section VII-B 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 [5].

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.

C. 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 features 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 VII-B2) 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 [25]). 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.

X. Related Work

Theoretical regular expression evolution has been studied in research. The regular expression can be created from a large number of labeled strings [14]–[16] or generated according to an existing one [26]–[28]. However, in these studies, mutation operators are defined according to genetic programming or grammatical evolution. The actual mutations performed by developers are unknown and not considered. Thus, this paper explores empirical regular expression changes and establishes the basis for the later study on practical regular expression mutation operators.

Another well-studied topic is source code evolution. Software evolves through a number of source code changes and the evolution has been analyzed at levels of granularity different from added or deleted lines [29], [30]. Early studies are focused on identifying semantic and syntactic code changes [31]–[33]. Recent studies are focused on predicting defects or vulnerabilities through code changes. For example, a file with many changes is more likely to be vulnerable than an unchanged file. The concept of the *code churn* metric was introduced as a means to measure the impact of code changes [34]. Together with other information (e.g., complexity, developer activity metrics), code churn are used to predict security vulnerability, software failures and defects [35]–[40]. Some other study in source code evolution includes code clones [41], [42] and crosscutting concerns [43]–[46]. 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 edit distance with fault-proneness; exploring that is left for future work.

Also related to syntactic and semantic evolution is the regular expression similarity. Rot, et al. proved regular expression language equivalence and the inclusion of DFAs that represent a subset of regular expression features [47], though no implementation is available. Dulucq, et al., defined tree edit distances [48] that can be applied to regular expression parse trees, which complements recent work proving that parse tree subsumption implies language subsumption [49]. Wang, et al. worked to cluster patterns by syntactic similarity [50], and others have worked to enumerate strings of regular expressions [7], [51]. These efforts toward computing and understanding regular expression similarity are valuable but lack available implementations.

Other regular expression research includes algorithms and tools to generate test cases with regular expressions [22], [52], [53], generating regular expressions for DNA sequences [54] and matching patterns. Enhancing regular expression pattern matching and processing speed is a common focus as well [55]–[58]. Regular expression features, refactoring, and comprehension have been the subject of recent work [5], [25], but exploring how regular expressions evolve is new to the literature.

XI. Conclusions

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 syntactically larger but semantically smaller than the Video dataset. 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.

Acknowledgment

This work is supported in part by NSF-SHF #1714699, #1749936 and #1645136.

References

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

[2] A. S. Yeole and B. B. Meshram, "Analysis of different technique for detection of sql injection," in *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ser. ICWET '11. New York, NY, USA: ACM, 2011, pp. 963–966. [Online]. Available: http://doi.acm.org/10.1145/1980022.1980229

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

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

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

[6] P. Wang and K. T. Stolee, "How well are regular expressions tested in the wild?" in *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018.

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

[8] A. Møller, "dk.brics.automaton – finite-state automata and regular expressions for Java," 2017, `http://www.brics.dk/automaton/`.

[9] "exrex 0.5.2 documentation," https://exrex.readthedocs.io/.

[10] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A solver for word equations over strings, regular expressions, and context-free grammars," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, pp. 25:1–25:28, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2377656.2377662

[11] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 105–116.

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

[13] ——, "Interactive testing and repairing of regular expressions," in *IFIP International Conference on Testing Software and Systems*. Springer, 2018, pp. 1–16.

[14] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Jagadish, "Regular expression learning for information extraction," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2008, pp. 21–30.

[15] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Inference of regular expressions for text extraction from examples," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1217–1230, 2016.

[16] A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, and E. Sorio, "Automatic generation of regular expressions from examples with genetic programming," in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*. ACM, 2012, pp. 1477–1478.

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

[18] "Outage postmortem - july 20, 2016," http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[19] E. Spishak, W. Dietl, and M. D. Ernst, "A type system for regular expressions," in *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, ser. FTfJP '12. New York, NY, USA: ACM, 2012, pp. 20–26. [Online]. Available: http://doi.acm.org/10.1145/2318202.2318207

[20] "Improve regex to avoid backtrack and to use non-capturing groups," https://github.com/SonarSource/sonar-css/pull/110/.

[21] "Fixed the regex," https://github.com/CaszGamerMD/NootSpeak/pull/14/.

[22] P. Arcaini, A. Gargantini, and E. Riccobene, "Fault-based test generation for regular expressions by mutation," *Software Testing, Verification and Reliability*, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1664

[23] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[24] "Apache commons text," https://commons.apache.org/proper/commons-text/.

[25] C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 405–416. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155616

[26] A. Cetinkaya, "Regular expression generation through grammatical evolution," in *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 2643–2646. [Online]. Available: http://doi.acm.org/10.1145/1274000.1274089

[27] A. González-Pardo, D. F. Barrero, D. Camacho, and M. D. R-Moreno, "A case study on grammatical-based representation for regular expression evolution," in *Trends in Practical Applications of Agents and Multiagent Systems*, Y. Demazeau, F. Dignum, J. M. Corchado, J. Bajo, R. Corchuelo, E. Corchado, F. Fernández-Riverola, V. J. Julián, P. Pawlewski, and A. Campbell, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 379–386.

[28] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 349–358, Aug 2001.

[29] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[30] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories." in *MSR*, vol. 7, 2007, p. 14.

[31] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90. New York, NY, USA: ACM, 1990, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/93542.93574

[32] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[33] D. Jackson, D. A. Ladd *et al.*, "Semantic diff: A tool for summarizing the effects of modifications." in *ICSM*, vol. 94, 1994, pp. 243–252.

[34] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 24–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853326

[35] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 13–23. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453106

[36] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.81

[37] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 83–92. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985456

[38] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 119–125. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138012

[39] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, ser. MetriSec '12. New York, NY, USA: ACM, 2012, pp. 7–10. [Online]. Available: http://doi.acm.org/10.1145/2372225.2372230

[40] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE ’05. New York, NY, USA: ACM, 2005, pp. 284–292. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062514

[41] E. Duala-Ekoko and M. P. Robillard, “Tracking code clones in evolving software,” in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 158–167.

[42] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, “An empirical study on the maintenance of source code clones,” *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.

[43] E. L. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher, “Managing crosscutting concerns during software evolution tasks: An inquisitive study,” in *Proceedings of the 1st international conference on Aspect-oriented software development*. ACM, 2002, pp. 120–126.

[44] B. Adams, Z. M. Jiang, and A. E. Hassan, “Identifying crosscutting concerns using historical code changes,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 305–314.

[45] L. Bergmans and M. Aksit, “Composing crosscutting concerns using composition filters,” *Communications of the ACM*, vol. 44, no. 10, pp. 51–57, 2001.

[46] S. Rastkar, G. C. Murphy, and A. W. Bradley, “Generating natural language summaries for crosscutting source code concerns,” in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 103–112.

[47] J. Rot, M. Bonsangue, and J. Rutten, “Proving language inclusion and equivalence by coinduction,” *Inf. Comput.*, vol. 246, no. C, pp. 62–76, Feb. 2016. [Online]. Available: https://doi.org/10.1016/j.ic.2015.11.009

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

[49] F. Henglein and L. Nielsen, “Regular expression containment: Coinductive axiomatization and computational interpretation,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’11. New York, NY, USA: ACM, 2011, pp. 385–398. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926429

[50] H. Wang, W. Wang, J. Yang, and P. S. Yu, “Clustering by pattern similarity in large data sets,” in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’02. New York, NY, USA: ACM, 2002, pp. 394–405. [Online]. Available: http://doi.acm.org/10.1145/564691.564737

[51] M. D. Mcilroy, “Enumerating the strings of regular languages,” *J. Funct. Program.*, vol. 14, no. 5, pp. 503–518, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1017/S0956796803004982

[52] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, “An orchestrated survey of methodologies for automated software test case generation,” *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.02.061

[53] N. Tillmann, J. de Halleux, and T. Xie, “Transferring an automated test generation tool to practice: From pex to fakes and code digger,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’14. New York, NY, USA: ACM, 2014, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642941

[54] W. B. Langdon and A. P. Harrison, “Evolving regular expressions for genechip probe performance prediction,” in *Parallel Problem Solving from Nature – PPSN X*, G. Rudolph, T. Jansen, N. Beume, S. Lucas, and C. Poloni, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1061–1070.

[55] A. Backurs and P. Indyk, “Which regular expression patterns are hard to match?” in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, Oct 2016, pp. 457–466.

[56] M. Becchi and P. Crowley, “Efficient regular expression evaluation: Theory to practice,” in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS ’08. New York, NY, USA: ACM, 2008, pp. 50–59. [Online]. Available: http://doi.acm.org/10.1145/1477942.1477950

[57] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A scalable architecture for high-throughput regular-expression pattern matching,” in *33rd International Symposium on Computer Architecture (ISCA’06)*, 2006, pp. 191–202.

[58] B. Cody-Kenny, M. Fenton, A. Ronayne, E. Considine, T. McGuire, and M. O’Neill, “A search for improved performance in regular expressions,” in *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 1280–1287.