Exploring Regular Expression Feature Usage in Practice and the Impact on Tool Design

Carl Chapman and Kathryn T. Stolee
Department of Computer Science
Iowa State University
{carl1978, kstolee}@iastate.edu

***Abstract*—Regular expressions are used frequently in programming languages for form validation, ad-hoc file searches, and simple parsing. Due to the popularity and pervasive use of regular expressions, researchers have created tools to support their creation, validation, and use. Each tool has made design decisions about which regular expression features to support, and these decisions impact the usefulness of the tools and their power. Yet, these decisions are often made with little information as there does not exist an empirical study of regular expression feature usage to inform these design decisions.**

In this paper, we explore regular expression feature usage, focusing on how often features are used and the diversity of regular expressions from syntactic and semantic perspectives. To do this, we analyzed nearly 4,000 open source Python projects from GitHub and extracted nearly 14,000 unique regex patterns that were used for analysis. We also map the most common features used in regular expressions to those features supported by four common regex engines from industry and academia, brics, Hampi, Re2, and Rex. Our results indicate that the most commonly used regular expression features are also supported by popular research tools and that programmers frequently reinvent the wheel by writing identical or nearly identical regular expressions in different ways. We concluded by discussing the implications of omitting certain features in a tool's design and outline several directions of future work.

I. Introduction

Regular expressions (regexes) are an abstraction of keyword search that enables the identification of text using a pattern instead of an exact keyword. There is a saying about regexes: 'now you have two problems'. A skilled programmer can quickly solve problems such as form validation and parsing text using regular expressions. Regular expression languages also enable a valuable text search/string specification technique used frequently within text editors (e.g., emacs), command line tools (e.g., grep, sed) and IDEs (e.g., the search feature in the Eclipse IDE). Although regexes are powerful and versatile, they can be hard to understand, maintain, and debug, resulting in tens of thousands of bug reports [18].

Due in part to their pervasive use across programming languages and how susceptible regexes are to error, many researchers and practitioners have developed tools to support more robust creation [18] or to allow visual debugging [6]. To remove the human in the loop, other research has focused on learning regular expressions from text [4], [13]. Beyond supporting regular expression usage, the applications of regular expressions in research include test case generation [2], [9], [10], [20], solvers for string constraints [11], [21], and as queries in a data mining framework [7] or on the semantic web [12]. Regexes are also employed in critical missions like mysql injection prevention [23] and network intrusion detection [16], or in more diverse applications like DNA sequencing alignment [3].

In writing tools to support regular expressions, tool designers make decisions about which features to support and which not to support. These decisions are sometimes made casually and may be dependent on the regular expressions the designers happen to have experience with, the designers have seen in the wild, or the complexity of the implementation. The goal of this work is to bring more context and information about regular expression feature usage so these design decisions can be better informed.

In fact, this paper emerges out of a need to understand which features can be reasonably included in or excluded from a tool that supports regular expressions. For some features that could involve more complexity, such as lazy evaluation, it is important to understand the impact of omitting such features. In the absence of empirical research into how regular expressions are used in practice, this work emerged.

In this paper, to motivate the study of regular expressions in general, we explore how regular expressions are used in practice. For example, we measure how frequently regular expressions appear in projects, and after doing a feature analysis (e.g., kleene star, character classes, and capture groups are all features), we further measure how often such features appear in regular expressions and in projects. Then, we compare the features to those supported by four common regex support tools, brics [15], hampi [11], Rex [22], and RE2 [17]. We then explore the features not supported by these tools and, using a semantic analysis to cluster similar regular expressions, we explore the impact of omitting those features. Our results indicate that these tools support all of the top six most common features and that some of the omitted features, such as the lazy quantifier, are used in over 35% of projects containing regular expressions. The contributions of this work are:

- An empirical analysis of the usage of regular expressions in 3,898 open-source Python projects
- An analysis of how frequently features are used, a mapping of which features are omitted from common regular expression tools, and a discussion of the impact of ignoring those features
- A discussion on the semantic similarity of regular expressions in practice, what use cases are important to users, and identification of opportunities for future

work in supporting programmers in writing regular expressions.

The rest of the paper is organized as follows. Section II motivates this work by discussing research in supporting programmers in the use, creation, and validation of regular expressions. Section III presents the research questions and study setup for exploring regular expressions in the wild. Results are in Section IV followed by a discussion in Section V and a conclusion in Section VII.

II. Related Work

Regular expressions have been a focus point in a variety of research objectives. From the user perspective, tools have been developed to support more robust creation [18] or to allow visual debugging [6]. Building on the perspective that regexes are difficult to create, other research has focused on removing the human from the creation process by learning regular expressions from text [4], [13].

Regarding applications, regular expressions have been used for test case generation [2], [9], [10], [20], and solvers for string constraints [11], [21]. Regexes are also employed in critical missions like mysql injection prevention [23] and network intrusion detection [16], or in more diverse applications like DNA sequencing alignment [3] or querying RDF data [1], [12].

As a query language, lightweight regular expressions are pervasive in search. For example, some data mining frameworks use regular expressions as queries (e.g., [7]). Efforts have also been made to expedite the processing of regular expressions on large bodies of text [5].

Within standard programming languages, regular expressions libraries are very common, yet there are differences between languages in the features that they support. For example, Java supports possessive quantifiers like `'ab*+c'` (here the '+' is modifying the '*' to make it possessive) whereas Python does not.

Since regular expression languages vary somewhat in their syntax and feature set, researchers and tool designers have typically had to pick what features to include or exclude. Thus, researchers and tool designers face a difficult design decision: supporting advanced features is always more expensive, taking more time and potentially making the tool or research project too complex and cumbersome to execute well. A selection of only the simplest of regex features is common in research papers and automata libraries, but this limits the applicability/relevance of that work in the real world.

In this work, we perform a feature analysis on regular expressions used in the wild and compare that set to the features supported by four popular regular expression tools. Research tools like Hampi [11], and Rex [22], and commercial tools like brics [15] and RE2 [17], all use regular expressions for various task. Hampi was developed in academia and uses regular expressions as a specification language for a strong constraint solver. Rex was developed by Microsoft Research and generates strings for regular expressions that can be used in several applications, such as test case generation [2], [20]. Brics is an open-source package that creates automata from regular expressions for manipulation and evaluation. RE2 is

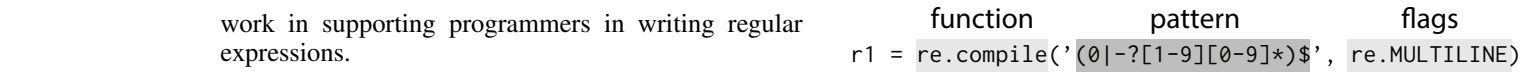

Fig. 1. example of one regex utilization

an open-source tool created by Google to power Code Search with a more efficient regex engine. While there are many regular expression tools available, in this work, we focus on the features support for these four tools, which offer diversity across developers (i.e., Microsoft, Google, open source, and academia) and across applications. Further, as the focus of this work is on tool designers and we wanted to perform a feature analysis, these four tools and their features are well-documented, allowing for easy comparison.

Mining properties of open source repositories is a well-studied topic, focusing, for example, on API usage patterms [14] and bug characterizations [8]. To our knowledge, this is the first work to mine and evaluate regular expression usages from existing software repositories. Related to mining work, regular expressions have been used to form queries in mining framework [7], but have not been the focus of the mining activities.

III. Study

To understand how programmers use regular expressions in Python projects, we scraped 3,898 Python projects from GitHub, and recorded regex usages for analysis as described in Section III-B. Throughout the rest of this paper, we employ the following terminology:

Utilization: A *utilization* occurs whenever a developer uses a regex engine in a project. We detect utilizations by recording all calls to the `re` module in Python. Within a particular file in a project, a utilization is composed of a function, a pattern and 0 or more flags. Figure 1 presents an example of one regex utilization, with key components labeled. Specifically, `re.compile` is the function call, `(0|-?[1-9][0-9]*)$` is the regex string, or pattern, and `re.MULTILINE` is an (optional) flag. Thought of another way, a regular expression utilization is one single invocation of the `re` library in a project.

The utilization in Figure 1 will compile a regex object in the variable `r1` from the pattern `(0|-?[1-9][0-9]*)$`, with the `$` token matching at the end of each line because of the `re.MULTILINE` flag. The pattern in this *utilization* will match if it finds a zero at the end of a line, or a (possibly negative) integer at the end of a line (i.e., due to the `-?` sequence denoting zero or one instance of the `-`).

Pattern: A *pattern* is extracted from a utilization, as shown in Figure 1. In essence, it is a string, but more formally it is an ordered series of regular expression language feature tokens.

Notice that because the vast majority of regex features are shared across most all-purpose languages, a Python pattern will (almost always) behave the same when used in other languages, such as Java, C#, Javascript, or Ruby, whereas a utilization is not universal in the same way (i.e., it may not compile in other languages, even with small modifications to function and flag names).

In this work, we primarily focus on patterns since they are cross-cutting across languages and are the primary way of specifying the matching behavior for every utilization. Next, we describe the research questions and how the data set was collected and analyzed.

A. Research Questions

Our overall research goal is to understand how regular expressions and regular expression features are used in practice. We aim to answer the following research questions:

RQ1: How is the `re` module used in Python projects?

To address this research question, we measure how often any calls are made to the `re` module per file and per project in Python projects.

Furthermore, we measure the frequency of usage for calls to the 8 functions of the `re` module (`re.compile`, `re.search`, `re.match`, `re.split`, `re.findall`, `re.finditer`, `re.sub` and `re.subn`) in Python projects scraped from GitHub.

We also measure usage of the 8 flags (`re.DEFAULT`, `re.IGNORECASE`, `re.LOCALE`, `re.MULTILINE`, `re.DOTALL`, `re.UNICODE`, `re.VERBOSE` and `re.DEBUG`) of the `re` module.

Further, to provide context as to the overlap among regular expression strings used in Python, we explore the most common regex patterns across all utlizations.

RQ2: Which regular expression language features are most commonly used in python?

We consider regex language features to be tokens that specify the matching behavior of a regex pattern, for example, the `+` in `ab+`. All studied features are listed and described in Section III-B with examples.

To measure feature usage, we parse Python regular expression patterns using Bart Kiers' PCRE parser[1], as described in Section III-B. We then count the number of usages of each feature per project, per file and as a percent of all distinct regular expression patterns.

RQ3: What is the impact of *not* supporting various regular expression features on tool users and designers?

To address, this question, we use semantic analysis to illustrate the impact of missing features on a tool's applicability by identifying what each feature (or group of features) is commonly used for.

At a high level, our semantic analysis clusters regular expressions by their behavioral similarity. We will briefly illustrate our definition of behavioral similarity using some unspecified patterns A and B. Consider a set 'mA' of 100 strings that pattern A matches. If pattern B matches 90 of the 100 strings in the set mA, then B is 90% similar to A. Now consider another set of 100 strings, 'mB' that pattern B matches. If pattern A only matches 50 of the strings in 'mB', then A is 50% similar to B. We use similarity scores to create a similarity matrix as shown in Figure 2. In row A, column B

```
Pattern A matches 100/100 of A's strings
Pattern B matches  90/100 of A's strings
Pattern A matches  50/100 of B's strings
Pattern B matches 100/100 of B's strings
```

	A	B
A	1.0	0.9
B	0.5	1.0

Fig. 2. A Similarity Matrix Created by Counting Strings Matched

we see that B is 90% similar to A. Int row B, column A, we see that A is 50% similar to B. Each pattern is always 100% similar to itself, since it matches all 100 of its own strings by their definition.

Using the similarity matrix, clusters of regexes with similar behavior are discovered using Markov Clustering[2]. These clusters are used to see how programmers implement regular expressions that match similar strings and interpret what a feature is used for. We chose the mcl clustering tool because it offers a fast and tunable way to cluster items by similarity and it is particularly useful when the number of clusters is not known *a priori*.

Next, we describe in greater detail how the corpus of regex patterns was built, how features were analyzed, and how the clustering was performed.

B. Building the Corpus

Github is a popular project hosting site containing over 100,000 Python projects. The GitHub API assigns an integer identifier to each repository and can be used to clone relevant repositories for analysis. Using the http://api.github.com/repositories?since=N interface page, we launched 32 scrapers to find repositories containing Python code. The Github interface provides information about the first 100 repository IDs since the `N` value on a single results page. Each scraper used this information to identify repositories containing Python. When a scraper was done with one page, it continued on to the next 100 repository IDs by using the interface again with `N` now equal to the last repository ID on the current page. Using this process, each scraper paged through the next available 1,000 repositories, cloning and scanning Python projects as they were found. Each scraper started at a different `N` value, with the first scraper starting at 0. Scraper start indices were spaced by 262,144 so as to investigate within the first 8 million repositories. At the time scraping was performed, the highest repoID was over 32 million, so we were cloning projects in the lowest fourth of the available space of IDs. After this process was complete, 3,898 Python projects had been cloned and scanned.

For each project, we used Astroid[3] to build the AST of each Python file and find utilizations of Python's `re` module. This ensured that all utliizations of the `re` module were captured for analysis.

Using git, each project's commit history was scanned at 20 evenly-spaced commits. If the project had fewer than 20 commits, then all commits were scanned. The most recent commit was always included, and the spacing between all other chosen commits was determined by dividing the remaining number of commits by 19 (rounding as needed).

[1] https://github.com/bkiers/PCREParser

[2] http://micans.org/mcl/

[3] https://bitbucket.org/logilab/astroid

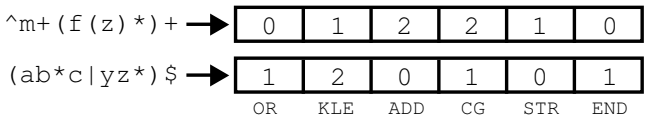

Fig. 3. Two Patterns Parsed into Feature Vectors

Within one project, we define a duplicate utilization as a utilization having the same function, pattern and flags within the same file (same relative path). We ignored duplicate utilizations across project versions to protect against over-counting the same utilization as we rewind the project through its history. We observed and recorded 53,894 non-duplicate regex utilizations in 3,898 projects.

C. Extracting Patterns

As the focus of this study is regex features, our analysis targets the patterns. Thus, we ignore the 12.7% of utilizations using flags that can alter regex behavior. An additional 6.5% of utilizations contained patterns that could not be compiled because the pattern was non-static (e.g., used some runtime variable), or because of other unknown parsing failures.

The remaining 80.8% (43,525) utilizations were collapsed into 14,113 distinct pattern strings using sql. Each of the pattern strings was pre-processed by removing Python quotes(`'\\W'` becomes `\\W`), unescaping escaped characters (`\\W` becomes `\W`) and parsing the resulting unescaped string using an ANTLR-based, open source PCRE parser released by Bart Kiers[4].

This parser was unable to support 0.5% (76) of the patterns due to unsupported unicode characters. Another 0.2% (27) of the patterns used regex features that we have chosen to exclude in this study because they did not appear often enough (e.g., Reference Conditions). The 13,912 distinct pattern strings that remain were each assigned a weight value equal to the number of distinct projects the pattern appeared in. We refer to this set of weighted, distinct pattern strings as the *corpus*.

D. Analyzing Features

For each escaped pattern, the PCRE-parser produces a tree of feature tokens, which is converted to a vector by counting the number of each token present in the tree. For a simple example, consider the patterns in Figure 3. The pattern `'^m+(f(z)*)+'` contains four different types of tokens. It contains the kleene star (KLE), which is specified using the asterisk `'*'` character, additional repetition (ADD), which is specified using the plus `'+'` character, capture groups (CG), which are specified using pairs of parenthesis `'(...)'` characters, and the start anchor (STR), which is specified using the caret `'^'` character at the beginning of a pattern.

Once all patterns were transformed into vectors, we examined each feature independently for all patterns, tracking the number of patterns it was in, and the size of the sets of projects and files that the patterns containing the features appeared in at least once.

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

Once the statistics about feature appearance in projects, files and patterns was established, we mapped the features from the corpus to those features supported by the four regular expression engines described in Section II: brics, hampi, RE2, and Rex. To create the tool mappings, we consulted documentation for each of the selected regular expression engines. For brics, we collected the set of supported features using the formal grammar[5]. For hampi, we manually inspected the set of regexes included in the `lib/regex-hampi/sampleRegex` file within the hampi repository[6] (this may have been an overestimation, as this included more features than specified by the formal grammar[7]). For RE2, we used the supported feature documentation[8]. For Rex, we were able to use trial and error because we tried to parse all patterns with Rex, and Rex provides good error feedback when a feature is unsupported.

E. Clustering and Semantic Analysis

We are interested in what behaviors users are trying to get when using regexes, and we know that the exact same behavior, or very similar behavior can be specified in many ways. For example, the three patterns `'\W'`, `'[^\w]'`, `'[^a-zA-Z0-9_]'` all specify the same matching behavior. Even if we create a slightly different pattern that matches one more character, for example `'[\W_]'`, most strings that match the first three equivalent patterns will also match the different pattern.

Rex can be used to generate a set of strings that will all match a pattern. For each of the 13,912 distinct patterns, we use Rex to generate a set of at least 384 *matching strings*. If Rex rejects the pattern, or fewer than 384 strings are generated, we do not include that pattern in the similarity analysis. The number 384 was selected to balance the runtime of the similarity analysis with the precision of the calculations. Since Rex does not support all the features present in the corpus, we could only generate sets of matching strings for 9,727 (70%) of the 13,912 patterns in the corpus (omitted features are indicated in Table III as described in Section IV-C). At least one pattern from the 9727 patterns that Rex was able to generate matching strings for can be found in 1375 of the 1645 projects containing at least one utilization. This means that 270 projects that contained at least one utilization did not contain any Rex-compatible patterns.

As explained in Section III-A, we used these sets of matching strings to measure the similarity between regular expressions and create a behavioral similarity matrix. We will refer to a cell of this matrix with row index `i` and column index `j` as `M[i][j]`. For each pattern at index `i`, we used Rex to create a set of matching strings which we will refer to as `matching_strings_i`. Then for every pattern at index `j`, we set the value of `M[i][j]` equal to the fraction of strings in `matching_strings_i` that the pattern at index `j` matched.

We illustrate the process used to create a text file specifying the edges of the graph to cluster in Figure 4. The

[5]http://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html

[6]https://code.google.com/p/hampi/downloads/list

[7]http://people.csail.mit.edu/akiezun/hampi/Grammar.html

[8]https://re2.googlecode.com/hg/doc/syntax.html

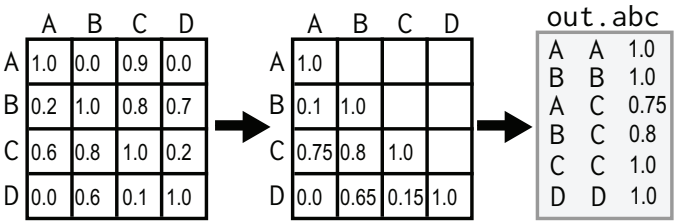

Fig. 4. Creating A Similarity Graph From A Similarity Matrix

leftmost matrix represents all similarity values between the four regexes: A, B, C, and D. The central matrix represents the average of cells reflected across the diagonal of the matrix. For example the value of row A, column C (0.9) represents the similarity of C to strings generated by Rex for regex A. The value of row C column A (0.6) represents the similarity of A to strings generated by Rex for regex C. The average of the two values is 0.75, which goes into row C, column A in the half-matrix. For every value of 0.75 or greater, an edge is written to *out.abc*. Note that pairs DB and DC are omitted from the final file because their similarities are lower than the threshold. The *out.abc* generated file is fed as input to the Markov Clustering Algorithm, and the top 100 clusters are catagorized by inspection into six categories of behavior (see Section IV-C).

Markov clustering can be tuned using many parameters, including inflation and filtering out all but the top-k edges for each node. After exploring the quality of the clusters using various tuning parameter combinations, the best clusters (by inspection) were found using an inflation value of 1.8 and k=83.

We note that there was an operational error in pulling patterns from our database prior to the similarity analysis and clustering, so that 224 patterns (2.3%) of the 9,727 patterns were omitted. These were duplicate patterns that were quoted differently (for example `'\W'` and `"\W"`). The result of this error is a slight underestimate in number of projects per pattern (and per cluster), and a slight over-estimate in the pattern, file and project statistics shown in Table III. We do not believe that this error affects our conclusions.

IV. Results

In this section, we present the results of each research question.

A. RQ1: How is the `re` module used in Python projects?

To address this research question, we look at regex utilizations, flags, and the most frequently observed pattern strings.

1) Regex Utilizations per Project: Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one regex utilization. For context about how saturated these projects were with utilizations, we consider how many utilizations were observed per project, how many files the average scanned project contained, how many of those files contained utilizations, and how many utilizations occurred per file, as shown in Table I.

The average project contained 32 utilizations, and the maximum number of utilizations was 1,427. The project with

TABLE I. How Saturated are Projects with Utilizations? (RQ1)

source	Q1	Avg	Med	Q3	Max
utilizations per project	2	32	5	19	1,427
files per project	2	53	6	21	5,963
utilizing files per project	1	11	2	6	541
utilizations per file	1	2	1	3	207

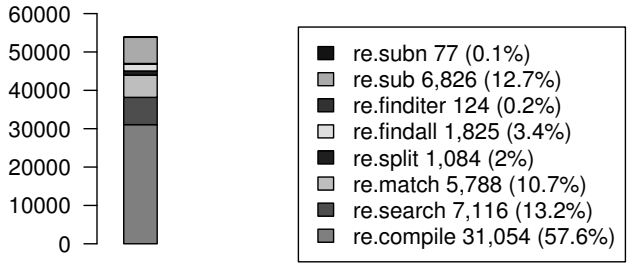

Fig. 5. How often are the 8 re functions used? (RQ1)

the most utilizations is a C# project[9] that maintains a collection of source code for 20 Python libraries, including larger libraries like `pip`, `celery` and `ipython`. These larger Python libraries contain many utilizations.

From Table I, we also see that each project had an average of 11 files containing any utilization, and each of these files had an average of 2 utilizations. As we scanned 3,898 projects, we would expect to have seen $11*2*3898 = 85,756$ regex usages, which is higher than the actual 53,894 usages observed.

2) Usage Frequency of `re` *Module Functions:* The number of projects that use each of the `re` functions are shown in Figure 5. The y-axis denotes the total utilizations, with a maximum of 53,894. The `re.compile` function encompasses 57.6% of all utilizations, presumably because each usage of those functions can accept a regex object compiled using `re.compile` as an argument.

3) Usage Frequency of `re` *Module Flags:* When considering flag use, we excluded the default flag, which is built into the `re` module, and present internally whenever no flag is used. Of all utlizations, 87.3% had no flag, or explicitly specified the default flag (which is equivalent). The debug flag, which

[9]https://github.com/Ouroboros/Arianrhod

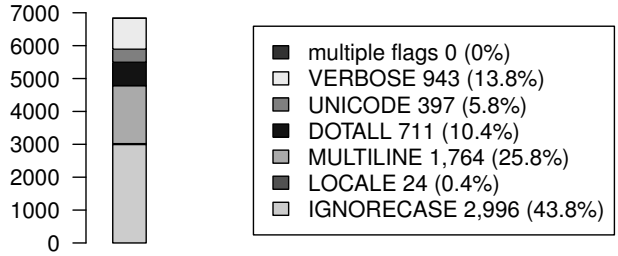

Fig. 6. Which behavioral flags are used? (RQ1)

TABLE II. TOP 10 PATTERNS BY NPROJECTS (RQ1)

pattern	nProjects
`'\s+'`	181
`'\s'`	78
`'\d+'`	70
`'[\x80-\xff]'`	69
`'\nmd5_data = {\n([^}]+)}'`	69
`'\\(.)'`	67
`'([\\"]|[^\ -~])'`	66
`'(-?(?:0|[1-9]\d*))(\.\d+)?([eE][-+]?\d+)?'`	61
`'[^]]+?\] +([0-9.]+): (\w+) <-(\w+)'`	60
`'.*rlen=([0-9]+)'`	57

causes the `re` regex engine to display extra information about its parsing process, was never observed.

Figure 6 presents the number of projects in which each flag appears. Of all behavioral flags used, ignorecase (43.8%) and multiline (25.8%) were the most frequently used. It is also worth noting that although multiple flags can be combined using a bitwise or, this was never observed.

4) Most Frequently Observed Patterns: Table II contains the most frequently used patterns, ordered by the number of projects that the pattern appears in. The patterns are quoted to show the presence of spaces, if any. For brevity, we will only elaborate on the top four patterns, and we will reference the feature set in Table III, as described in Section IV-B.

The first pattern, `'\s+'`, uses the WSP character class feature. This character class represents one or more whitespace characters, defined as spaces, tabs, newlines, vertical whitespace, carriage returns or form feeds. The + (the ADD feature) at the end of the pattern means that it must match one or more whitespace characters. The pattern `'\s+'` is often used to split sentences into separate words which may have more than one space between them, or contain tabs or other types of whitespace.

Interestingly, the second most common pattern, `'\s'`, also uses the WSP character. In this case, it does not match several whitespace characters (though it would match the first of several).

The third pattern, `\d+`, uses the DEC character class, which is composed of the digits from 0 to 9. Like the first pattern, the third uses the ADD feature to match one or more digits.

The fourth pattern, `[\x80-\xff]`, uses the CCC feature to create a custom character class, and the RNG feature to specify a range of characters. When hexadecimal values are used within a character class like this (and the UNICODE flag is not active), it signifies the corresponding characters in an ASCII lookup table.

5) Summary of Results for RQ1: Only about half of the projects sampled contained any utilizations, and many of these utilizations came from python module source code that had been copied into projects, not code written by the users. Most utilizations used the `re.compile` function to compile a regex object before actually using the regex to find a match. Most utilizations did not use a flag to modify matching behavior. The most frequently observed patterns were used to match whitespace and digits.

B. RQ2: Which regular expression language features are most commonly used in python?

Table III displays feature usage from the corpus and relates it to four major regex related projects. Only features appearing in at least 10 projects are listed. The first column, *rank*, lists the rank of a feature (relative to other features) in terms of the number of projects in which it appears. The next column, *code*, gives a succinct reference string for the feature, and is followed by a *description* column that provides a brief comment on what the feature does. The *example* column provides a short example of how the feature can be used. The next four columns map to the four major research projects chosen for our investigation (see Section IV-C). We indicate that a project supports a feature with the '●' symbol, and indicate that a project does not support the feature with the '○' symbol.

The next six columns contain three pairs of usage statistics. The first pair contains the number and percent of *patterns* that a feature appears in, out of the 13,912 patterns that make up the corpus. The next pair of columns contain the number and percent of *files* that a feature appears in out of the 18,549 files scanned that contain at least one utilization. The last pair of columns contain the number and percent of *projects* that a feature appears in out of the 1645 projects scanned that contain at least one utilization.

One notable omission from Table III is the literal feature, which is used to specify matching any specific character. An example pattern that contains only one literal token is the pattern `'a'`. This pattern only matches the lowercase letter 'a'. The literal feature was found in 97.5% of patterns. We consider the literal feature to be ubiquitous in all patterns, and necessary for any regex related tool to support, and so exclude it from Table III and the rest of the feature analysis.

1) Feature Usage: The eight most commonly used features, ADD, CG, KLE, CCC, ANY, RNG, STR and END, appear in over half the projects. The remaining 26 features appear in less than half of the projects containing utilizations.

Table III is sorted by the number of projects a feature appears in, but if the table were instead sorted by the number of patterns or files that a feature appears in, the ranking order would be different. CG is more commonly used in patterns than the highest ranked feature (ADD) by a wide margin (over 8%). CG is also more commonly used with respect to how many files it appears in by 2.3%, while only being present in 12 fewer projects (0.7%) than ADD.

2) Feature Support in Regex Projects: Of the four projects selected for analysis, RE2 supports the most features (28 features) followed by hampi (25 features), Rex (21 features), and brics (12 features). All projects support the 8 most commonly used features except brics, which does not support STR or END. All projects support NCC, OR, and the four less common repetition features: QST, SNG, DBB and LWB.

No projects support the four look-around features LKA, NLKA, LKB and NLKB. RE2 and hampi support the LZY, NCG, PNG and OPT features, whereas brics and Rex do not.

TABLE IV. AN EXAMPLE CLUSTER (RQ3)

index	pattern	nProjects
1	`'\s*,\s*'`	54
2	`','`	30
3	`'\s*,'`	16
4	`' ,\s*'`	13
5	`' *, *'`	12
6	`',\S'`	5
7	`',.*$'`	3
8	`'(\S+)\s*,\s*'`	2
9	`',+'`	1
10	`',\ ?'`	1
10	`',\s*(\S)'`	1
11	`'\s*(,)\s*'`	1
12	`'\s*\,\s*'`	1

Brics is the only project that does not support any of the six default character classes (WSP, DEC, WRD, NWSP, NWRD, NDEC) - the rest of the projects support all of those features. RE2 is the only project to support the WNW, ENDZ and NWNW features. RE2 also supports the PNG feature (which allows you to name capturing groups), but does not support the BKRN feature which is necessary to refer back to the named capture group.

3) Summary of Results for RQ2: We found that the eight most common features are found on over 50% of the projects. We also identify brics as the project supporting the fewest features and RE2 as the project supporting the most features, and identify groups of features supported or not supported by the four regex projects.

C. RQ3: What is the impact of not supporting various regular expression features on users and tool designers?

When tool designers are considering what features to include, data about user behaviors is valuable. Our clustering technique helps to discern these behaviors by looking beyond the structural details of specific patterns and seeing trends in actual matching behavior. We are also able to find out what features are being used in these behavioral trends so that we can make assertions about why certain features are important.

From 9503 distinct patterns, the MCL clustering technique identified 514 clusters with 2 or more patterns, and 7214 clusters of size 1. Recall that only pairs of patterns with a similarity level of 0.75 were included in the matrix passed to MCL. The average size of clusters larger than contains 3.7 patterns. Each pattern belongs to exactly one cluster.

Table IV provides an example of a smaller behavioral cluster, representing 13 patterns, with at least one pattern from this cluster present in 100 different projects. At first glance this cluster may seem to revolve around the '`\s*`' parts of these patterns, but actually this cluster was formed because each of these patterns has a comma literal, and other details did not interfere with matching the Rex-generated strings with commas in them.

The smallest pattern in Table IV is the single comma literal `','` at index 1. This smallest pattern gives the a good idea of what all the patterns within it have in common. A shorter pattern will tend to have less extraneous behavior because it is specifying less behavior. And yet in order for the smallest pattern to be clustered with other patterns, it had to match most of the `matching_strings` created by Rex from another pattern within the cluster, and so we assume that*the smallest pattern is a good representation of the cluster*.

For the rest of this paper, a cluster will be represented by one of the shortest patterns it contains, followed by the number of projects it appears in, so the cluster in Table IV will be represented as `','(100)`. We manually mapped the top 100 clusters into each of 6 behavioral categories, omitting 11 clusters that did not fit into any of these catagories, and 29 clusters composed of long strings (example: `set_fabric_sense_len\)\()` which were in the top 100 because 28 projects that were scanned were forked linux kernels.

Next, we define the six categories and provide examples from the relevant clusters.

1) Single Literal Characters: This category contains 19 clusters. Each of the clusters center around a single literal character. For example, three of the top clusters in this category include: `'\\'(110)`, `','(100)`, and `':'(91)`.

2) Default Character Classes: This category contains 12 clusters. Each of the clusters revolves around the use of a default character class. For example, three of the top clusters in this category are: `'\s'(277)`, `'\W'(208)`, and `'\d'(193)`.

3) User-Defined Character Classes: This category contains 10 clusters. Each of the clusters center around user defined character classes. For example, three of the top clusters in this category are: `'[a-zA-Z]'(138)`, `'[^!-~]'(122)`, and `'[&<]'(50)`.

4) Matching Whole Strings: This category contains 8 clusters. Each of the clusters begins with the STR anchor and ends with the END anchor, requiring the entire input string to match the pattern. For example, three of the top clusters in this category include: `'^\d+$'(78)`, `'^\w+$'(74)`, and `'^\s*$'(59)`.

5) Parsing Angle Bracket Contents: This category contains 5 clusters. Each of the clusters contains a pair of angle brackets that contain a repeating character class. It appears that these clusters are being used to recognize or capture the contents of the angle brackets. For example, three of the top clusters in this category include: `'<.+>'(63)`, `'<!\s+([^<>]*)>'(35)`, and `'(<[^<>]*)/>'(35)`.

6) Capturing Variable Assignments: This category contains 4 clusters. Each of the clusters contain an equals symbol and some pattern on either side of it, which appears to be a variable on the left of the equals sign and a value on the right. This type of cluster is very likely used to capture the value of the variable assignment. For example, three of the top clusters in this category are: `'\nmd5_data = {\n([^}]+)}'(69)`, `'.*rlen=([0-9]+)'(57)`, and `'coding[:=]\s*([-\w.]+)'(48)`.

TABLE III. HOW FREQUENTLY DO FEATURES APPEAR IN PROJECTS, AND WHICH FEATURES ARE SUPPORTED BY FOUR MAJOR REGEX PROJECTS? (RQ2)

rank	code	description	example	brics	hampi	Rex	RE2	nPatterns	% patterns	nFiles	%files	nProjects	% projects
1	ADD	one-or-more repetition	`z+`	●	●	●	●	6,122	44	9,330	50.3	1,209	73.5
2	CG	a capture group	`(caught)`	●	●	●	●	7,248	52.1	9,759	52.6	1,197	72.8
3	KLE	zero-or-more repetition	`.*`	●	●	●	●	6,104	43.9	8,323	44.9	1,100	66.9
4	CCC	custom character class	`[aeiou]`	●	●	●	●	4,581	32.9	7,808	42.1	1,027	62.4
5	ANY	any non-newline char	`.`	●	●	●	●	4,708	33.8	6,394	34.5	1,006	61.2
6	RNG	chars within a range	`[a-z]`	●	●	●	●	2,698	19.4	5,196	28	849	51.6
7	STR	start-of-line	`^`	○	●	●	●	3,660	26.3	5,622	30.3	847	51.5
8	END	end-of-line	`$`	○	●	●	●	3,258	23.4	5,549	29.9	828	50.3
9	NCCC	negated CCC	`[^qwxf]`	●	●	●	●	1,970	14.2	4,027	21.7	777	47.2
10	WSP	\t \n \r \v \f or space	`\s`	○	●	●	●	2,908	20.9	4,812	25.9	764	46.4
11	OR	logical or	`a|b`	●	●	●	●	2,161	15.5	4,039	21.8	711	43.2
12	DEC	any of: 0123456789	`\d`	○	●	●	●	2,385	17.1	4,366	23.5	694	42.2
13	WRD	[a-zA-Z0-9_]	`\w`	○	●	●	●	1,457	10.5	3,004	16.2	652	39.6
14	QST	zero-or-one repetition	`z?`	●	●	●	●	1,922	13.8	3,821	20.6	647	39.3
15	LZY	as few reps as possible	`z+?`	○	●	○	●	1,318	9.5	2,291	12.4	606	36.8
16	NCG	group without capturing	`a(?:b)c`	○	●	○	●	813	5.8	1,748	9.4	404	24.6
17	PNG	named capture group	`(?P<name>x)`	○	●	○	●	934	6.7	1,517	8.2	354	21.5
18	SNG	exactly n repetition	`z{8}`	●	●	●	●	623	4.5	1,359	7.3	340	20.7
19	NWSP	any non-whitespace	`\S`	○	●	●	●	490	3.5	788	4.2	271	16.5
20	DBB	$n \leq x \leq m$ repetition	`z{3,8}`	●	●	●	●	384	2.8	692	3.7	242	14.7
21	NLKA	sequence doesn't follow	`a(?!yz)`	○	○	○	○	137	1	503	2.7	184	11.2
22	NWRD	non-word chars	`\W`	○	●	●	●	97	0.7	315	1.7	169	10.3
23	LWB	at least n repetition	`z{15,}`	●	●	●	●	97	0.7	337	1.8	167	10.2
24	WNW	word/non-word boundary	`\b`	○	○	○	●	248	1.8	438	2.4	166	10.1
25	LKA	matching sequence follows	`a(?=bc)`	○	○	○	○	114	0.8	360	1.9	159	9.7
26	OPT	options wrapper	`(?i)CasE`	○	●	○	●	232	1.7	378	2	154	9.4
27	NLKB	sequence doesn't precede	`(?<!x)yz`	○	○	○	○	102	0.7	321	1.7	139	8.4
28	LKB	matching sequence precedes	`(?<=a)bc`	○	○	○	○	82	0.6	262	1.4	120	7.3
29	ENDZ	absolute end of string	`\Z`	○	○	○	●	91	0.7	154	0.8	94	5.7
30	BKR	match the i^{th} CG	`\1`	○	○	○	○	60	0.4	129	0.7	84	5.1
31	NDEC	any non-decimal	`\D`	○	●	●	●	36	0.3	92	0.5	58	3.5
32	BKRN	references PNG	`\g<name>`	○	●	○	○	17	0.1	44	0.2	28	1.7
33	VWSP	matches U+000B	`\v`	○	○	●	●	13	0.1	16	0.1	15	0.9
34	NWNW	negated WNW	`\B`	○	○	○	●	4	0	11	0.1	11	0.7

D. Summary of Results for RQ3

We used the behavior of individual patterns to form clusters, and identified six main categories that clusters belonged to. Overall, we see that many clusters are defined by the presence of particular tokens, such as the comma for the cluster in Table IV. These six categories define what users are doing with regexes at a high level: using default character classes, defining their own character classes, matching single characters, parsing variable assignments, parsing the contents of brackets, or matching whole lines. One of the six common cluster categories, *capturing variable assignments*, has a very specific purpose of parsing source code files. This shows a very specific and common use of regular expressions in practice.

V. DISCUSSION

This research has explored how regular expressions are used in practice. In this section, we discuss the implications of these empirical findings on tool designers and users of regex tools and opportunities for future work.

A. Implications For Tool Designers of Omitting a Feature

We observe that, although the clusters were generated based on behavioral similarity, they are often centered around the presence of certain features. Thus, omitting those features in a tool's implementation often omits an entire space of regular expression behaviors.

1) STR, END: The endpoint anchor features STR and END are the only way to specify that the entire input string has to

1) literals, sequences of tokens
2) CCC (NCCC, RNG, WSP, DEC, WRD, NWSP, NDEC, NWRD)
3) CG (without back-references)
4) DBB (ADD, KLE, QST, SNG, LWB)
5) STR, END
6) OR
7) LZY
8) NCG
9) NLKA, LKA, NLKB, LKB
10) WNW, NWNW
11) ENDZ
12) BKR

Fig. 7. A Suggested Feature Implementation Priority List

match a pattern. Consider the following example, comparing the pattern `'^\s*$'` to the pattern `'\s*'` when looking for input strings devoid of content. Because it does not have endpoint anchors, the pattern `'\s*'` matches all inputs, since there are always at least zero whitespace characters in every input. But with the endpoint anchors, only inputs that contain nothing but whitespace will match, allowing the user to determine if a string is devoid of content.

From Table III we know that STR and END features are present in over half of the scanned projects containing utilizations - further evidence of the importance of these features. The brics library does not support this feature, which is a missed opportunity for many developers who could otherwise have used brics to model their regexes that use STR and END.

2) LZY: The LZY feature modifies the behavior of the repetition features (i.e., ADD, KLE, QST, DBB, LWB, SNG) by forcing them to use as few repetitions as possible for a match. The default behavior for the repetition features is to use as many repetitions as possible for a match. Consider trying to find the shortest sequence of binary characters starting and ending with a 1 within the binary sequence: `1010001`. The pattern `'1.*1'` will match the entire sequence whereas the pattern `'1.*?1'` will match `101` because the KLE repetition feature was modified by the LZY feature. There is no way to obtain shortest matches without the LZY feature. LZY is present in over 36% of scanned projects with utilizations (see Table III), and yet was not supported by two of the four major regex projects we looked at.

B. Key User Behaviors to Support

Here we offer a discussion of the main behaviors that are important for tool designers to support.

1) Prioritized List of Features: Based on our analysis of feature usage in RQ2 and the data presented in Table III, we have produced a prioritized list of features (Figure 7) to include in a tool meant to support as many regexes as possible.

The features NCCC, RNG, WSP, DEC, WRD, NWSP, NDEC, NWRD are all included as the second priority because by implementing CCC, these features are available as specific instances of CCC. Consider how the DEC feature can be simulated using CCC, for example $\backslash d \equiv [0123456789]$. Likewise, any use of the RNG feature can be simulated using CCC, for example: $[a-f] \equiv [abcdef]$.

Similarly, the features ADD, KLE, QST, SNG and LWB are available once the DBB feature is implemented. Consider how every use of the ADD feature can be simulated with a specific use of the DBB feature, for example: $a+ \equiv a\{1, MAX\}$. In the same way, any use of the SNG feature can be simulated using the DBB feature, for example: $a\{5\} \equiv a\{5,5\}$.

2) Capturing Specific Content: As mentioned in section IV-B1, the CG feature is the most frequently used feature in terms of patterns and files (see Table III). The ability to capture some part of a match provides a powerful tool to programmers. As mentioned in Section IV-D, capturing values assigned to variables was one of the main categories of clusters observed. All four of the major regex tools support the CG feature. Any non-trivial tool or research that hopes to be applicable to regex use in practice must treat the CG feature as especially important.

3) Use of the Default Character Classes: The pattern language for Python and most major regex engines support a few default character classes (and their negations) which we have described as the features ANY, DEC, WSP, WRD, NDEC, NWSP, NWRD. Throughout this analysis it was obvious that these default character classes were widely used. As mentioned in section V-B1, all of these default character classes can be simulated using the CCC feature, and the CCC feature is one of the most basic features that is essential for all regex tools to support. For tools that do not support the default character classes, this is a significant obstacle for users trying to test or model the regexes that they already have.

4) Using `'.*'` to Count Lines Containing a Pattern: Text files containing one unit of information per line are common in a wide variety of applications (for example log and csv files). Out of all the patterns observed during this study, the most ubiquitous sub-pattern was `'.*'`, usually present near the end of the pattern. Out of the 13,912 patterns in the corpus, 3444 (24%) contained `'.*'` at least once. One reasonable explanation for this tendency to put `'.*'` at the end of a pattern is that users want to disregard all matches after the first match on a single line in order to count how many distinct lines the match occurs on. In Figure 8, we illustrate this phenomenon with two lines of Python code.

The first line initializes the string `fruitDiary`, which represents some file with many lines, with each line containing one unit of data. In this example, each line contains a list of the fruits that were eaten on that day. Two apples are recorded on the first line of `fruitDiary`, zero apples are on the second line, and one apple is on the third line. Overall, we see that at least one apple was eaten on two of the three days.

The second line of code finds the length of the list of matches found using the `re.findall` function. The pattern `'Apple.*'` will match any character sequence that starts with the string `Apple` and then the `'.*'` portion of the pattern will match every character except the newline character. The two matches are highlighted in Figure 8. Because the `'.*'` portion of the pattern matches everything up to the newline, the second Apple on the first line is ignored. This allows the number of matches to be a count of lines containing `Apple` at least once.

```
fruitDiary = "Day1: Apple, Orange, Apple, Peach\n
             Day2: Kumquat, Berry, Pear, Peach\n
             Day3: Orange, Pear, Apple, Pear"
len(re.findall('Apple.*',fruitDiary)) # 2
```

Fig. 8. An Example of Using .* to Count Lines Containing 'Apples'

Tools that intend to be applicable to regex use in practice must support both the ANY and KLE features, and should give special care to maintain the expected line-counting behavior for the `'.*'` idiom.

C. Opportunities For Future Work

Based on our findings, there are many opportunities for future work.

1) Re-Evaluating the WRD Character Class: One surprising result of our clustering is that, behaviorally speaking, the negation of the word class NWRD was used in 208 projects, while the word class itself was used in only 114 projects. After inspecting several projects using the patterns found in this behavioral cluster, we concluded that most users are trying to sanitize arbitrary strings that must conform to a system character set requirement, such as requirements for filenames. For example, a user might replace all NWRD matching characters with the '_' to guarantee that an arbitrary string can be used as a filename. We also considered the largest cluster using custom character classes (`'[^ -~]'(122)`) and concluded that users are constructing a more permissive version of the NWRD character class, to allow more non-letter, non-digit characters than just the '_' in their sanitized strings. We hypothesize that this is because modern systems allow more special characters in filenames than systems allowed when the WRD character class originated. More research is needed to determine if a more modern WRD class could be useful, and if so, what characters set is preferred.

2) How Does Developer Experience Influence Regex Composition?: The project with the most files containing utilizations observed is Arianrhod[10] which is a Japanese Anime game, mostly written in C# (over 18K files), but containing 3404 Python files. Most of these Python files are source code for various libraries. Of these library files, 15.9% (541) contain at least one utilization, which is more than the 11% average. This indicates that more experienced developers (those capable of developing core Python libraries) may be more likely to use regexes than less experienced developers. More research is needed to determine if experienced developers are more likely to use regexes, and what differences in feature set and composition can be observed when compared to regexes composed by less experienced developers.

3) Regexes need refactoring: When clustering by behavior, we observed many clusters with more than 10 patterns all specifying nearly the same behavior. We don't know why different users choose to implement the same behavior in different ways. Out of the 13,912 patterns in the corpus, 1977 (14%) used the RNG feature to specify the range `0-9`, even though this is exactly equivalent to using `\d` . Future work is needed to determine why users specify digits using the RNG feature when an equivalent default character class already exists, as has been done for other refactoring work (e.g., [19]).

[10]https://github.com/Ouroboros/Arianrhod

VI. Threats to Validity

The threats to validity of this work stem primarily from sampling bias, tool limitations, and language selection.

We mined only 3,898 Python projects from GitHub, which is small in comparison with the over 100,000 available Python projects. The projects were mined using the GitHub API which sorts the projects by creation date. By using the API, the goal was to reduce any sampling bias introduced by the researchers.

We did not scrape all commits in every project for regular expression utilizations, rather, we grabbed each project every 20 commits. It is possible that in between the scanned commits, a regular expression utilization was added and then removed, leading to fewer utilizations in our final data set.

In extracting patterns for analysis, we omit utilizations that contain flags since flags provide refinements on the functionality of the pattern as it is used in the project. Given that only 12.7% of all utilizations used flags, the impact on the clusters should be minimal.

Regular expression patterns were clustered using strings generated by the Rex tool. This introduces two threats to validity. First, we assume that the strings generated by Rex are reasonably diverse to help characterize the regular expression behavior. To mitigate this threat, Rex generated at least 384 strings per regular expression. Second, since the similarity between two regular expressions was calculated empirically using Rex rather than through analysis, we may have over-estimated the similarly. For this reason, in our clustering algorithm, we require a similarity level of 0.75 so that we do not over-estimate the similarity between regular expressions.

The final threat to validity comes from the fact that we only explore regular expressions in Python projects so these results many not generalize to other languages. Future work will replicate this study in other languages and compare the results.

VII. Conclusion

Regular expressions are used frequently in programming projects. In our analysis of nearly 4,000 Python projects scraped from GitHub, we observed that over 42% contained a regular expression. The most commonly found regular expressions deal with matching whitespace or digits. In analyzing the features used in regular expressions, we find that the most common is the + token. When clustering regexes based on semantic similarity, we observe that programmers frequently create identical regular expressions using slightly different patterns. After mapping the features supported by four popular regex tools from academia and industry, we observe that most of the most popular features are also supported. However, those unsupported can have an impact on the tool users.

Acknowledgment

This work is supported in part by NSF SHF-1218265, NSF SHF-EAGER-1446932, and the Harpole-Pentair endowment at Iowa State University.

REFERENCES

[1] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending sparql with regular expression patterns (for querying rdf). *Web Semant.*, 7(2):57–73, Apr. 2009.

[2] 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.*, 86(8):1978–2001, Aug. 2013.

[3] A. 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*, pages 1–7, Nov 2005.

[4] R. Babbar and N. Singh. Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text. In *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data*, AND '10, pages 43–50, New York, NY, USA, 2010. ACM.

[5] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, Nov. 1996.

[6] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf. Regviz: Visual debugging of regular expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 504–507, New York, NY, USA, 2014. ACM.

[7] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[8] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 82–91, New York, NY, USA, 2014. ACM.

[9] S. J. Galler and B. K. Aichernig. Survey on test data generation tools. *Int. J. Softw. Tools Technol. Transf.*, 16(6):727–751, Nov. 2014.

[10] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang. Jst: An automatic test generation tool for industrial java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, Piscataway, NJ, USA, 2013. IEEE Press.

[11] 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.*, 21(4):25:1–25:28, Feb. 2013.

[12] J. Lee, M.-D. Pham, J. Lee, W.-S. Han, H. Cho, H. Yu, and J.-H. Lee. Processing sparql queries with regular expressions in rdf databases. In *Proceedings of the ACM Fourth International Workshop on Data and Text Mining in Biomedical Informatics*, DTMBIO '10, pages 23–30, New York, NY, USA, 2010. ACM.

[13] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 21–30, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[14] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[15] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[16] The Bro Network Security Monitor. https://www.bro.org/, May 2015.

[17] RE2. https://github.com/google/re2, May 2015.

[18] 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*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM.

[19] K. T. Stolee and S. Elbaum. Identification, impact, and refactoring of smells in pipe-like web mashups. *IEEE Trans. Soft. Eng.*, 39(12):1654–1679, 2013.

[20] 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*, ASE '14, pages 385–396, New York, NY, USA, 2014. ACM.

[21] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, New York, NY, USA, 2014. ACM.

[22] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 498–507, Washington, DC, USA, 2010. IEEE Computer Society.

[23] 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*, ICWET '11, pages 963–966, New York, NY, USA, 2011. ACM.