Exploring Regular Expression Comprehension

Carl Chapman*
Sandia National Laboratories
Albuquerque, NM
carlallenchapman@gmail.com

Peipei Wang
Department of Computer Science
North Carolina State University
pwang7@ncsu.edu

Kathryn T. Stolee
Department of Computer Science
North Carolina State University
ktstolee@ncsu.edu

***Abstract*—The regular expression (regex) is a powerful tool employed in a large variety of software engineering tasks. However, prior work has shown that regexes can be very complex and that it could be difficult for developers to compose and understand them. This work seeks to identify code smells that impact comprehension. We conduct an empirical study on 42 pairs of behaviorally equivalent but syntactically different regexes using 180 participants and evaluate the understandability of various regex language features. We further analyze regexes in GitHub to find the community standards or the common usages of various features. We found that some regex expression representations are more understandable than others. For example, using a range (e.g., `[0-9]`) is often more understandable than a default character class (e.g., `[\d]`). We also found that the DFA size of a regex significantly affects comprehension for the regexes studied. The larger the DFA of a regex (up to size eight), the more understandable it was. Finally, we identify smelly and non-smelly regex representations based on a combination of community standards and understandability metrics.**

I. Introduction

Regular expressions (regexes) are used fundamentally in string searching and substitution tasks, such as word searching, text editing, file parsing, user input validation, and access controls. More advanced uses can be seen in search engines [1], database querying [2], and network security [3]–[5].

Recent research has suggested that regular expressions are hard to understand, hard to compose, and error prone [6]. Given their frequent appearances in software source code and the difficulty of working with them, some effort has been put into easing the burden on developers by providing environments that make regexes easier to understand. Some tools provide debugging environments which explain string matching results and highlight the parts of regex patterns which match a certain string [7], [8]. Other tools present graphical representations (e.g., finite automata) of the regular expressions [9], [10]. Still, others can automatically generate strings according to a given regular expression [11], [12] or automatically generate regexes according to a given list of strings [13], [14]. The commonality of such tools and techniques provides evidence that developers need help with regex composition and comprehension.

In software engineering, code smells have been found to hinder understandability of source code [15], [16]. Once removed through refactoring, the code becomes more understandable, easing the burden on the programmer. In regular expressions, such code smells have not yet been defined, perhaps in part because it is not clear what makes a regex difficult to understand or maintain. This is one of the goals of this work, to explore language features that impact comprehension and begin to identify code smells in regexes.

In regular expressions as in source code, there are multiple ways to express the same semantic concept. For example, the regex, `aa*` matches an "a" followed by zero or more "a", and is equivalent to `a+` , which matches one or more "a". That is, both regexes match the same *language* but are expressed using different syntax. What is not clear is which representation, `aa*` or `a+`, is more easily understood.

In this work, we focus on identifying regex comprehension smells. We identify equivalence classes of regex representations that provide options for concepts such as double-bounds in repetitions (e.g., `a{1,2}`, `a|aa`) or character classes (e.g., `[0-9]`, `[\d]`). Based on an empirical study measuring regex comprehension on 42 pairs of regexes using 180 participants, as well as an empirical study of nearly 14,000 regexes and their features, we identify smelly and non-smelly regex representations. For example, `aa*` is more smelly than `a+`, based on feature usage frequency in source code (conformance to community standards) and understandability. Our contributions are:

- An empirical study to evaluate regex comprehension with 180 participants for studying regex understandability,
- Identification of five types of equivalence classes and 18 corresponding representations for regular expressions, and
- Identification of smelly and non-smelly regex representations to optimize 1) understandability and 2) conformance to community standards, backed by empirical evidence.

Despite the frequent usage of regexes in source code [17], this is the first work to explore regex comprehension.

II. Research Questions

In this work, we use the term ***regex representation*** to refer to the syntactic expression of a regular expression. A ***feature*** is a structural component of a regular expression (e.g., Kleene star: `*` or custom character class: `[1-5]`). An ***equivalence class*** is a group of behaviorally equivalent regular expressions. To explore regex comprehension, we answer the following research questions:

* This work was done while this author was at Iowa State University.

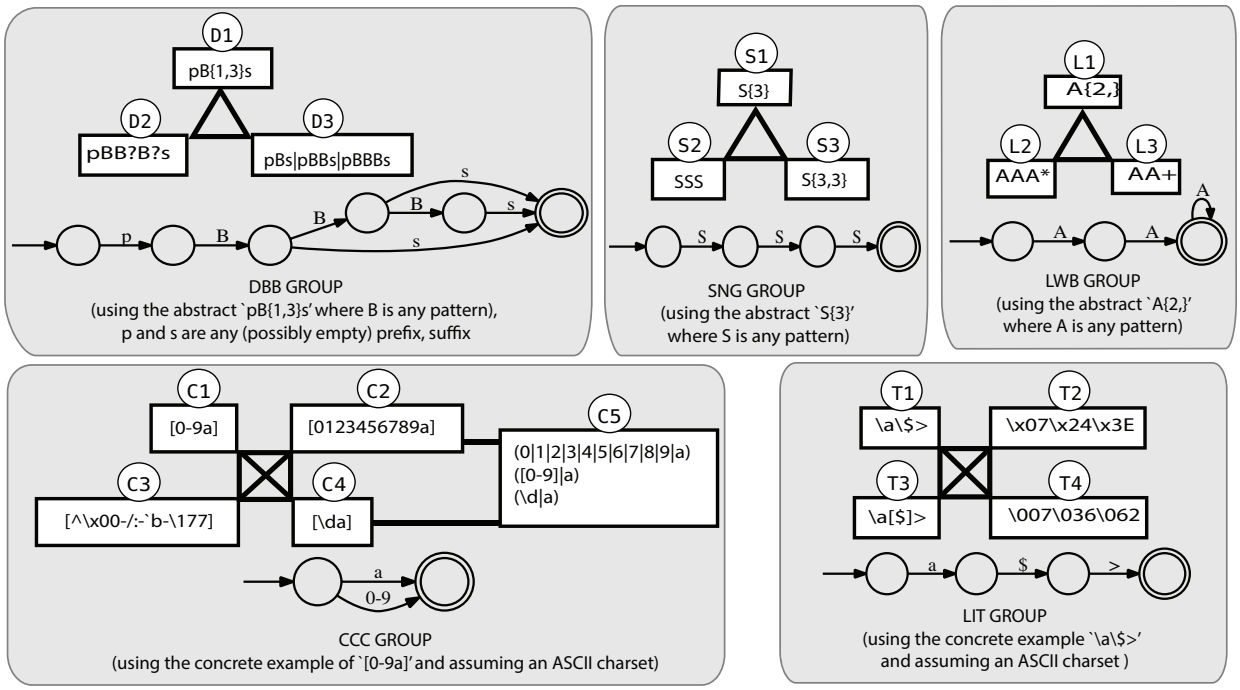

Fig. 1: Types of equivalence classes based on language features. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal. We use concrete regexes along with their Deterministic Finite Automaton (DFA) in the representations for illustration. However, the A's in the LWB group (or B's in DBB group, S's in SNG group, and so forth) abstractly represent any pattern that could be operated on by a repetition modifier (e.g., literal characters, character classes, or groups). The same is true for the literals used in all the representations.

RQ1: *Which regex representations are most understandable?* To answer RQ1, we conduct a study in which programmers are presented with a regex and asked comprehension questions about its matching behavior. By comparing accuracy between regexes that match the same language but are expressed using different representations (e.g., `tri[a-f]3` and `tri(a|b|c|d|e|f)3`), we can measure understandability and identify code smells.

We also explore factors that may impact comprehension, namely regex string length, regex DFA size, and the equivalence class representation. This analysis requires identification of equivalence classes for regexes. By inspecting a Python regex dataset of nearly 14,000 regexes [17], we formed an initial set of five types of equivalence classes to explore.

RQ2: *Which regex representations have the strongest community support based on frequency?* To answer RQ2, we explore the publicly available regex dataset [17] and use the presence and absence of language features as a proxy for community support, where more frequently-used features are assumed to be more understandable.

RQ3: *Which regex representations are most desirable (i.e., least smelly) based on both community support and understandability?* Based on RQ1 and RQ2, we identify smelly and non-smelly regex features based on a combination of comprehension metrics and community support.

III. Equivalence Classes

To explore understandability, we defined an initial set of equivalence classes for regexes. Using the publicly available behavioral clusters of Python regexes [17], we manually identified several representations that appeared in many of the larger clusters. While they are not a complete set of equivalence classes, this is the first work to explore regex understandability, and these equivalence classes provide an initial testbed for exploration.

Figure 1 shows five types of equivalence classes in grey boxes and examples of behaviorally equivalent representations in white boxes with identifiers in white circles. For example, LWB is a type of equivalence class with representations L1, L2, and L3. Regexes `AAA*` and `AA+` map to L2 and L3, respectively.

Each equivalence class is accompanied by a deterministic finite automata (DFA) representing the behavior of the example regexes. For example, with the SNG group, each of the regexes accepts strings with a sequence of exactly three 'S' characters. The accept state is marked by a double-circle. Next, we describe each equivalence class group.

A. Custom Character Class Group

The Custom Character Class (CCC) group has regex representations that use the custom character class language feature or can be represented by such a feature. A custom character class matches a set of alternative characters. For example, the regex `c[ao]t` will match strings "cat" and "cot" because, between the `c` and `t`, there is a custom character class, `[ao]`, that specifies either `a` or `o` (but not both) must be selected. We use the term *custom* to differentiate these classes from the default character classes, : `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`,

provided by most regex libraries, though the default classes can be encapsulated in a custom character class.

C1: Any pattern that contains a (non-negative) custom character class with a range feature like `[a-f]` as shorthand for all of the characters between 'a' and 'f' (inclusive) belongs to C1.

C2: Any pattern that contains a (non-negative) custom character class without any shorthand representations, specifically ranges or defaults (e.g., `[012]` is in C2, but `[0-2]` is not).

C3: Any pattern with a character class expressed using negation, indicated by a caret (i.e., `^`) followed by a custom character class. For example, the pattern `[^ao]` matches every character *except* `a` or `o`.

C4: Any pattern using a default character class such as `\d` or `\W` within a (non-negative) character class.

C5: These can be transformed into custom character classes by removing the ORs and adding square brackets (e.g., `(\d|a)` in C5 is equivalent to `[\da]` in C4). All custom character classes expressed as an OR of length-one sequences, including defaults or other custom classes, are in C5[1].

Note that a pattern can belong to multiple representations. For example, `[a-f\d]` belongs to both C1 and C4.

B. Double-Bounded Group

The Double-Bounded (DBB) group contains all regex patterns that use some repetition defined by a (non-equal) lower and upper boundary. For example, `pB{1,3}s` represents a `p` followed by one to three sequential `B` patterns, then followed by a single `s`. This matches "pBs", "pBBs", and "pBBBs".

D1: Any pattern that uses the curly brace repetition with a lower and upper bound, such as `pB{1,3}s`.

D2: Any pattern that uses the questionable (i.e., `?`) modifier implies a lower-bound of zero and an upper-bound of one (and hence is double-bounded).

D3: Any pattern that has a repetition with a lower and upper bound and is expressed using ORs (e.g., `pB{1,3}s` becomes `pBs|pBBs|pBBBs` by expanding on each option in the boundaries).

Patterns can belong to multiple representations (e.g., `(a|aa)X?Y{2,4}` belongs to all three nodes: `Y{2,4}` maps to D1, `X?` maps to D2, and `(a|aa)` maps to D3).

C. Literal Group

In the Literal (LIT) group, all patterns that are not purely default character classes must use literal tokens. We use the ASCII charset in which all characters can be expressed using hex and octal codes such as `\xF1` and `\0108`, respectively.

T1: Patterns that do not use any hex, wrapped, or octal characters, but use at least one literal character. Special characters are escaped using the backslashes.

T2: Any pattern using a hex token, such as `\x07`.

T3: Any pattern with a literal character wrapped in square brackets. This style is used most often to avoid using a backslash for a special character in the regex language, for example, `[{]` which must otherwise be escaped like `\{`.

T4: Any pattern using an octal token, such as `\007`.

Patterns often fall in multiple of these representations. For example, `abc\007` includes literals `a`, `b`, and `c`, and also octal `\007`, thus belonging to T1 and T4. Not all transformations are possible in this group. If a hex representation used for a character is not on the keyboard, a transformation to T1 or T3 is infeasible.

D. Lower-Bounded Group

The Lower-Bounded (LWB) group contains patterns that specify only a lower boundary on repetitions. This can be expressed using curly braces with a comma after the lower bound but no upper bound. For example, `A{2,}` will match "AA", "AAA", "AAAA", and any number of A's greater or equal to 2. In Figure 1, we chose the lower bound repetition threshold of 2 for illustration; in practice this could be any number, including zero.

L1: Any pattern using this curly braces-style lower-bounded repetition (i.e., {}) belongs to node L1.

L2: Any pattern using the Kleene star (i.e., *), which means zero-or-more repetitions.

L3: Any pattern using the additional repetition (i.e., +). For example, `T+` means one or more `T`'s.

Patterns often fall into multiple nodes in this equivalence class. For example, with `A+B*`, `A+` maps it to L3 and `B*` maps it to L2.

E. Single-Bounded Group

The Single-Bounded (SNG) equivalence class contains three representations in which each regex has a fixed number of repetitions of some element. The important factor distinguishing this group from DBB and LWB is that there is a single finite number of repetitions, rather than a bounded range on the number of repetitions (DBB) or a lower bound on the number of repetitions (LWB).

S1: Any pattern with a single repetition boundary in curly braces belongs to S1. For example, `S{3}`, states that S appears exactly three times in sequence.

S2: Any pattern that is explicitly repeated two or more times and could use repetition operators.

S3: Any pattern with a double-bound in which the upper and lower bounds are same belong to S3. For example, `S{3,3}` states `S` appears a minimum of 3 and maximum of 3 times.

The pattern `fa[lmnop][lmnop][lmnop]` is a member of S2 as `[lmnop]` is repeated three times, and it could be transformed to `fa[lmnop]{3}` in S1 or `fa[lmnop]{3,3}` in S3.

IV. Understandability Study (RQ1)

This study presents programmers with regexes and asks comprehension questions. By comparing the understandability

[1]An OR cannot be directly negated, it there is no edge between C3 and C5

TABLE I: Matching metric example

String	`'RR*'`	Oracle	P1	P2	P3	P4
1	"ARROW"	✓	✓	✓	✓	✓
2	"qRs"	✓	✓	×	×	?
3	"R0R"	✓	✓	✓	?	-
4	"qrs"	×	✓	×	✓	-
5	"98"	×	×	×	×	-
	Score	1.00	0.80	0.80	0.50	1.00

✓ = match, ×= not a match, ? = unsure, – = left blank

of semantically equivalent regexes that match the same language but have different syntactic representations, we aim to identify understandability code smells. This study was implemented on Amazon's Mechanical Turk with 180 participants. A total of 60 regexes were evaluated, constructing 42 pairs of regex comparison. Each regex pattern was evaluated by 30 participants.

A. Metrics

We measure the understandability of regexes using two complementary metrics, *matching* and *composition*. These are referred to as the *comprehension metrics*. For a deeper look at the data to gain a better understanding of factors that impact comprehension, we also compute *regex length* and *DFA size* for each regex.

Matching: Given a pattern and a set of strings, a participant determines by inspection which strings will be matched by the pattern. There are four possible responses for each string, *matches*, *not a match*, *unsure*, or blank. An example[2] from our study is shown in Figure 2.

The percentage of correct responses, disregarding blanks and unsure responses, is the matching score. For example, consider regex pattern `'RR*'`, the five strings shown in Table I, and the responses from four participants in the *P1*, *P2*, *P3* and *P4* columns. The *Oracle* indicates the first three strings match and the last two do not; *P1* answers correctly for the first three strings and the fifth, but incorrectly on the fourth, so the matching score is $4/5 = 0.80$. *P2* incorrectly thinks that the second string is not a match, so the score is also $4/5 = 0.80$. *P3* marks 'unsure' for the third string and so the total number of attempted matching questions is 4. *P3* is incorrect about the second and fourth string, so they score $2/4 = 0.50$. For *P4*, we only have data for the first and second strings, since the other three are blank. *P4* marks 'unsure' for the second string so only one matching question has been attempted; the matching score is $1/1 = 1.00$.

Blanks were incorporated into the metric because questions were occasionally left blank in the study. Unsure responses were provided as an option so not to bias the results through blind guessing. These situations did not occur very frequently. Out of 1,800 questions (180 participants * 10 questions each), only 1.8%(32) were impacted by a blank or unsure response (never more than four out of 30 responses per pattern).

Composition: Given a pattern, a participant composes a string they think it matches (question 7.F in Figure 2). If the participant is accurate, a composition score is 1, otherwise 0. For example, given the pattern `(q4fab|ab)` from our study, the string, "xyzq4fab" matches and gets a score of 1, but the string, "acb" does not match and gets a score of 0.

[2]Task instructions are also available: https://github.com/wangpeipei90/RegexSmells/blob/master/questionnaire.pdf

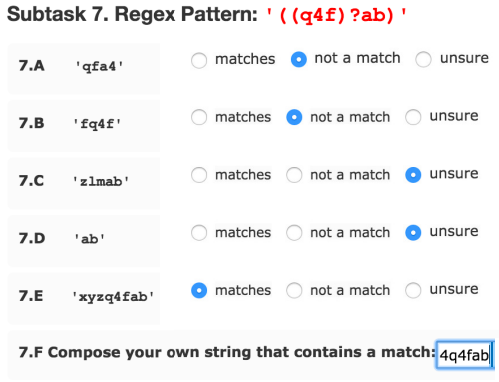

Fig. 2: Questions from one pattern in one HIT

To determine the match between a string and a pattern, the pattern is compiled using the *re.compile* module in Python. An instance of *re.RegexObject* `m` is created using the compiled pattern. *m.search()* returns an instance of *re.MatchObject* `m2` with the string given as the input to this function. If `m2` is not *None*, then that string was a match and scored 1; otherwise it scored 0.

Regex Length: Given a pattern, the regex length is computed by its literal string length. For example, regexes `\072` and `ab*c` are both length four.

DFA Size: Given a pattern,To compute the size of minimal DFA, we run both *brics* [18] and *Rex* [12] on each regex, and manually check their results to guarantee their correctness.

B. Design

We implemented this study on Amazon's Mechanical Turk (MTurk), a crowdsourcing platform where requesters create human intelligence tasks (HITs) for completion by workers.

Worker Qualification: Qualified workers had to answer four of the five basic regex questions correctly. These questions were multiple-choice and asked the worker to analyze the following patterns: `a+`, `(r|z)`, `\d`, `q*`, and `[p-s]`.

Tasks: Guided by the patterns in the corpus, we created 60 regex patterns that were grouped into 26 semantic equivalence groups. There were 18 groups with two regexes targeting various edges in the equivalence classes. The other eight groups had three regexes each. In total there are 42 pairs of patterns. In this way, we can draw conclusions by comparing representations since the regexes evaluated were semantically equivalent.

To form the semantic groups, we took a regex from the corpus, matched it to a representation in Figure 1, trimmed it down so it contained little more than just the feature of interest, and then created other regexes that are semantically equivalent but belong to other nodes in the equivalence class. For example, a semantic group with regexes `((q4f){0,1}ab`, `((q4f)?ab)`, and `(q4fab|ab)` belong to D1, D2, and D3, respectively. A group with regexes `([0-9]+)\.([0-9]+)` and `(\d+)\.(\d+)` is intended to evaluate the edge between

TABLE II: 3-factor ANOVA with average matching or composition accuracy as dependent variables, considering representation (rep), DFA size (dfa_size), and regex length (len) as independent variables

		Average Matching		Average Composition	
	Df	F value	Pr(>F)	F value	Pr(>F)
dfa_size	1	7.632	0.0153 *	10.084	0.00674 **
len	1	3.325	0.0896 ·	0.001	0.98161
rep	15	2.062	0.0921 ·	1.224	0.35538
dfa_size:len	1	1.002	0.3339	1.384	0.25907
dfa_size:rep	14	0.709	0.7355	0.920	0.56075
len:rep	10	0.924	0.5397	0.599	0.79054
dfa_size:len:rep	3	1.163	0.3589	0.678	0.58002
Residuals	14				

$\cdot\alpha = 0.10$ $*\alpha = 0.05$ $**\alpha = 0.01$ $***\alpha = 0.001$

C1 and C4. We note that if we only used regexes from the corpus, we would have had regexes with different semantics at each node, or with additional language features, which would make the comparisons of the targeted features difficult.

For each of the 26 semantic groups, we created five strings for the study, where at least one matched and at least one did not match. These were used to compute the matching metric. Once all the patterns and matching strings were collected, we created tasks for the MTurk participants as follows: randomly select a pattern from 10 of the 26 semantic groups. Randomize the order of these 10 patterns, as well as the order of the matching strings for each pattern. After adding a question asking the participant to compose a string that each pattern matches, this creates one task on MTurk, such as the example in Figure 2. This process was completed until each of the 60 regexes appeared in 30 HITs, resulting in a total of 180 total unique HITs.

Implementation: Workers were paid $3.00 for successfully completing one and only one HIT. The average completion time for accepted HITs was 682 seconds (11 mins, 22 secs). A total of 54 HITs were rejected: 48 had too many blank responses, four were double-submissions by same workers, one did not answer composition questions, and one missed data of 3 questions. Rejected HITs were returned to MTurk to be completed by others.

Participants: In total, there were 180 participants. A majority were male (83%). Most had at least an Associates degree (72%), were at least somewhat familiar with regexes (87%), and had prior programming experience (84%).

C. Analysis

We computed a matching and composition score for each regex based on the 30 participant responses. The average analysis or average composition is computed by averaging the associated 26-30 values for each metric for each of the 60 regexes (fewer than 30 values were used if all the responses in a matching question were a combination of blanks and unsure).

Of the original 42 pairs, we report scores for 41. Due to a design flaw, the regexes evaluated, `\..*` and `\.+`. were not semantically equivalent (the former is missing an escape and should be `\.\.*`), so this was omitted from the data. In the end, we analyzed 58 regexes that cover 17 edges from Figure 1.

To gain a better understanding of why some regexes may be more understandable than others, we also look at the impact of the representation from Figure 1, regex length, and DFA size[3] on the comprehension metrics. Note that we retain all 60 regexes for this analysis as we are looking at the properties of regexes individually. We conduct two three-factor analysis of variances (ANOVAs) with matching accuracy and composition accuracy as the dependent variables. We also conduct the correlation analysis between these three factors and the composition metrics. We use Spearman's Rank-Order Correlation because we have no priori knowledge about the distributions of the factors. Since the regex representations are categorical data, these are excluded from the correlation analysis.

D. Results

The ANOVA in Table II shows that DFA size significantly affects both the average matching accuracy and the average composition at $\alpha = 0.05$ and $\alpha = 0.01$, respectively. The length and representation from Figure 1 each significantly affect the average matching accuracy at $\alpha = 0.10$. Since the DFA sizes vary across the pairwise comparisons within a representation, we present our results for matching and composition using each of the 41 pairs of regexes separately, rather than in aggregate over the equivalence class edges explored.

For the comprehension metrics, Table III presents the results. Each row represents a *Pair* of regex evaluated by study participants. The representations for the regexes per Figure 1 are shown in the *Edge* column, which is how the table is sorted. The *Regex 1* and *Regex 2* columns identify the regexes used in the study, mapping to the first and second representations in the *Edge* column, respectively. *Match1* is the average matching for *Regex 1* and *Match2* is the average matching for *Regex 2*. Using the Mann-Whitney test of means, the *sigM* column following tests if there is a significant difference between the accuracies. The *Comp1* column presents the percentage of the string responses for that were in fact correctly matched by *Regex 1*. *Comp2* presents the same information, except for *Regex 2*. The following *sigC* column uses a test of two proportions to identify if the percentage of the participants who correctly composed a string for Regex 1 is significantly different than the percentage who correctly composed a string for Regex 2.

To illustrate, consider pair 16 in Table III. One pair of regexes was `([]{])` and `(\{|\})` in C4 and C5, respectively, with average matching scores of 78.79% and 70.33% and average composition scores of 50.00% and 86.67%, respectively. The difference between the composition scores is significant at $\alpha = 0.01$, yet the difference between the accuracies is not. In fact, the representation C5 was more understandable in that participants could more effectively compose a string that it would match, but C4 is more understandable in that participants could more easily determine

[3]Note that the study was not specifically designed for regex length and DFA size

TABLE III: Pairwise comparisons of regexes. Each matching or composition value is computed based on approximately 30 data points from 30 study participants

Pair	Edge	Regex 1	Regex 2	Match1(%)	Match2(%)	SigM	Comp1(%)	Comp2(%)	SigC
1	**C1** – C2	`tri[a-f]3`	`tri[abcdef]3`	94.00	93.17		83.33	83.33	
2	**C1** – C2	`no[w-z]5`	`no[wxyz]5`	93.33	87.17		86.67	86.67	
3	C1 – **C3**	`no[w-z]5`	`no(w|x|y|z)5`	93.33	93.67		86.67	96.67	
4	C1 – **C4**	`([0-9]+)\.([0-9]+)`	`(\d+)\.(\d+)`	90.17	94.44		83.33	93.33	
5	**C1** – C4	`xg1([0-9]{1,3})%`	`xg1(\d{1,3})%`	82.67	81.33		76.67	66.67	
6	**C1** – C4	`[a-f]([0-9]+)[a-f]`	`[a-f](\d+)[a-f]`	91.17	83.33		80.00	70.00	
7	C1 – **C4**	`&([A-Za-z0-9_]+);`	`&(\w+);`	81.90	82.59		56.67	66.67	
8	**C1** – C4	`1q[A-Za-z0-9_][A-Za-z0-9_]`	`1q\w\w`	86.00	78.11		83.33	70.00	
9	**C1** – C4	`tuv[A-Za-z0-9_]`	`tuv\w`	89.17	86.00		83.33	70.00	
10	**C1** – C5	`tri[a-f]3`	`tri(a|b|c|d|e|f)3`	94.00	86.11		83.33	80.00	
11	C2 – C4	`[\t\r\f\n ]`	`[\s]`	82.99	92.41	·	3.33	0.00	
12	**C2** – C5	`tri[abcdef]3`	`tri(a|b|c|d|e|f)3`	93.17	86.11		83.33	80.00	
13	C2 – **C5**	`no[wxyz]5`	`no(w|x|y|z)5`	87.17	93.67		86.67	96.67	
14	C3 – C4	`[^0-9A-Za-z]`	`[\W_]`	64.50	61.00		46.67	53.33	
15	C3 – **C4**	`[^0-9]`	`[\D]`	58.00	73.33		63.33	73.33	
16	C4 – C5	`([}{])`	`(\{|\})`	78.79	70.33		50.00	86.67	**
17	C4 – **C5**	`([:;])`	`(:|;)`	81.38	94.00		46.67	46.67	
18	**D1** – D2	`((q4f){0,1}ab)`	`((q4f)?ab)`	82.93	79.25		50.00	40.00	
19	**D1** – D2	`(dee(do){1,2})`	`(deedo(do)?)`	84.83	77.17		66.67	60.00	
20	D1 – **D3**	`((q4f){0,1}ab)`	`(q4fab|ab)`	82.93	84.50		50.00	60.00	
21	D1 – D3	`(dee(do){1,2})`	`(deedo|deedodo)`	84.83	90.00		66.67	63.33	
22	D2 – **D3**	`((q4f)?ab)`	`(q4fab|ab)`	79.25	84.50		40.00	60.00	
23	D2 – **D3**	`(deedo(do)?)`	`(deedo|deedodo)`	77.17	90.00	*	60.00	63.33	
24	L2 – L3	`zaa*`	`za+`	86.67	90.67		70.00	50.00	
25	L2 – **L3**	`RR*`	`R+`	86.00	91.56		66.67	66.67	
26	S1 – S2	`%([0-9A-Fa-f]{2})`	`%([0-9a-fA-F][0-9a-fA-F])`	77.78	73.44		50.00	60.00	
27	S1 – **S2**	`&d([aeiou]{2})z`	`&d([aeiou][aeiou])z`	91.28	95.34		83.33	83.33	
28	S1 – S2	`fa[lmnop]{3}`	`fa[lmnop][lmnop][lmnop]`	87.17	88.00		83.33	73.33	
29	T1 – T2	`xyz[_\[\]‘\^\\]`	`xyz[\x5b-\x5f]`	77.78	78.67		86.67	56.67	*
30	**T1** – T2	`t[:;]+p`	`t[\x3a-\x3b]+p`	94.33	88.59		80.00	63.33	
31	T1 – T3	`(\$\{)\d+(:[^}]+\})`	`([$][{])\d+(:[^}]+[}])`	81.61	75.18		63.33	73.33	
32	T1 – **T3**	`t\.\$+\d+\*`	`t[.][$]+\d+[*]`	88.67	94.00		56.67	73.33	
33	**T1** – T3	`\{\$(\d+\.\d)\}`	`[{][$](\d+[.]\d)[}]`	93.28	89.33		70.00	66.67	
34	**T1** – T4	`xyz[_\[\]‘\^\\]`	`xyz[\0133-\0140]`	77.78	71.35		86.67	33.33	***
35	**T1** – T4	`t[:;]+p`	`t[\072\073]+p`	94.33	90.00		80.00	70.00	
36	**T1** – T4	`(\{|\})`	`([\0175\0173])`	70.33	54.40	·	86.67	30.00	***
37	**T1** – T4	`([}{])`	`([\0175\0173])`	78.79	54.40	**	50.00	30.00	
38	**T1** – T4	`(:|;)`	`([\072\073])`	94.00	65.77	**	46.67	23.33	
39	**T1** – T4	`([:;])`	`([\072\073])`	81.38	65.77	·	46.67	23.33	
40	**T2** – T4	`xyz[\x5b-\x5f]`	`xyz[\0133-\0140]`	78.67	71.35		56.67	33.33	
41	T2 – **T4**	`t[\x3a-\x3b]+p`	`t[\072-\073]+p`	88.59	90.00		63.33	70.00	

$\cdot\alpha = 0.10$ $*\alpha = 0.05$ $**\alpha = 0.01$ $***\alpha = 0.001$

which of a set of strings would be matched by C4. Thus, neither representation is bolded in the *Edge* column since there is a conflict. If both comprehension metrics indicated a preferred representation, that representation is bolded (e.g., C4 in pair 15). Ties are broken by deferring to the other metric. For example, there's a tie in composition for pair 17, but matching indicates a preference for C5. Therefore C5 is bolded.

For pairs 16, 29, 34, and 36, the difference in composition is significant at $\alpha < 0.05$, indicating differences favoring C5 over C4, T1 over T2, and twice favoring T1 over T4. For pairs 23, 37, and 38, the difference in matching is significant with $\alpha < 0.05$, indicating differences favoring D3 over D2 and twice favoring T1 over T4. Interestingly, for pairs 16 and 29, while the differences in composition are significant, there is a conflict between the composition metrics. Further investigation is needed to understand in what circumstances the metrics are in conflict with one another. Recall that participants were able to select *unsure* for whether a string is matched by a pattern. From a comprehension perspective, this indicates some level of confusion. For each pattern, we counted the number of responses containing at least one unsure. Overall, the highest number of unsure responses came from T4 and T2, which have octal and hex representations of characters. The least number of unsure responses were in L3 and D3. These results mirror the understandability analysis, as T4 and T2 are generally lower in comprehension, and L3 and D3 are generally higher.

While the ANOVA indicates that variance in matching is due to all three factors, representation, DFA size, and regex length, it is not entirely clear why. Variance in composition is

impacted by DFA size only. Between DFA size and composition, there is a strong, positive correlation at $\alpha = 0.01$ with $\rho = 0.354$. At first, this result may seem counter-intuitive, but considering that larger DFAs may represent more constrained regex languages (i.e., languages that accept fewer strings), these may be easier to compose a string for. However, as the explored DFA size range was between two and eight nodes, these results may not generalize to larger regexes. None of the other correlations are significant with $\alpha < 0.05$.

E. Summary

Matching and composition are impacted by DFA size, and matching is also impacted by regex length and representation, showing some support that the representation of the regex impacts comprehension. The larger the DFA, the easier it was for the community to generate strings that match it. There also appears to be a clear trend favoring T1 over T4. Representations D3 and C5 are also preferred. While C1 is favored comparisons against C2, C4, and C5, none are significant.

V. Community Support Study (RQ2)

The goal of this evaluation is to understand how frequently each of the regex representations appears in source code, as a way to identify community standards code smells [19], [20].

A. Artifacts

We analyzed an existing corpus of regexes collected from Python code in GitHub projects [17]. This dataset has 13,597 distinct (non-duplicate) regex patterns from 1,544 projects.

This corpus was created by analyzing static invocations to the Python `re` library. Consider the Python snippet:

```
r1 = re.compile('(0|-?[1-9][0-9]*)$', re.MULTILINE)
```

The function *re.compile* compiles the regex `(0|-?[1-9][0-9]*)$` into `r1`, an object of re.RegexObject. *re.MULTILINE* is a flag that changes the matching behavior from the default one line to multiple lines. This particular regex will match strings with any integer at the end of a line ("`-?`" indicates the integer may be negative).

B. Metrics

We measure community support by matching regexes in the corpus to representations in Figure 1 and by counting the *patterns* and *projects*. These are referred to as the *community standards metrics*. A regex can belong to multiple representations and to multiple projects since the corpus tracks its duplicates.

C. Analysis

To match patterns to representations, we either used the PCRE parser to parse features of patterns or extracted token streams of them. The choice depends on the characteristics of the representation. Our analysis code is available on GitHub[4]. The details of this process are described as follows.

[4]https://github.com/wangpeipei90/RegexSmells

Presence of a Feature: For representations that require a particular feature, we used the PCRE parser to decide membership. This applies to C3, D1, D2, L1, L2, L3, S1, and S3.

Features and Pattern: Identifying D3 requires an OR containing at least two entries with a sequence repeated N times in one entry and the same sequence repeated N+1 times in another entry. We first looked for a sequence of N repeating groups with an OR-bar (i.e., `|`) next to them on a side. This produced a list of 113 candidates and we narrowed them down manually to 10 actual members.

T1 requires that no characters are wrapped in brackets or are hex or octal characters, which matches over 91% of the patterns analyzed; T2 requires a literal character with a hex structure; T3 requires that a single literal character is wrapped in a custom character class (a member of T3 is always a member of C2); T4 requires a literal character with a Python-style octal structure.

Token Stream: C1 requires a non-negative class of characters whose ASCII codes are consecutive; C2 requires a custom character class which does not use ranges or defaults; C4 requires the presence of a default character class within a custom character class; C5 requires an OR of length-one sequences (literal characters or any character class); S2 requires a repeated element which could be a character class, a literal character, or a smaller regex encapsulated in parentheses. These representations were treated as sequences of tokens. To identify them, we chose token streams instead of the PCRE parser.

D. Results

Table IV presents the results. *Node* references Figure 1; *Description* briefly describes the representation; *Example* provides a regex from the corpus; *nPatterns* counts the patterns that belong to the representation; *% patterns* shows the percentage of patterns out of 13,597; *nProjects* counts the projects that contain a regex belonging to the representation; *% projects* shows the percentage of projects out of 1,544. For example, D1 is more concentrated in a few projects and T3 is more widespread across projects. This is because comparing to D1, T3 appears in 39 *fewer* patterns but 34 *more* projects.

The pattern frequency is our guide for setting the community standards. For example, since C1 is more prevalent than C2 in both patterns and projects, we could say that C2 is smelly since it could better conform to community standards if expressed as C1.

E. Summary

Based on patterns, the winning representations per equivalence class are C1, D2, T1, L2, and S2. With one exception, these are the same project-based recommendations; L3 appears in more projects than L2, so it is unclear which is smelly.

VI. Desirable Representations (RQ3)

To determine the overall trends in the data, we created and compared total orderings on the representations in each

TABLE IV: How frequently is each regex representation style used?

Node	Description	Example	nPatterns	% patterns	nProjects	% projects
C1	char class using ranges	`^[1-9][0-9]*$`	2,479	18.2%	810	52.5%
C2	char class explicitly listing all chars	`[aeiouy]`	1,903	14.0%	715	46.3%
C3	any negated char class	`[^A-Za-z0-9.]+`	1,935	14.2%	776	50.3%
C4	char class using defaults	`[-+\d.]`	840	6.2%	414	26.8%
C5	an OR of length-one sub-patterns	`(@|<|>|-|!)`	245	1.8%	239	15.5%
D1	curly brace repetition like {M,N} with M<N	`^x{1,4}$`	346	2.5%	234	15.2%
D2	zero-or-one repetition using question mark	`^http(s)?://`	1,871	13.8%	646	41.8%
D3	repetition expressed using an OR	`^(Q|QQ)\<(.+)\>$`	10	.1%	27	1.7%
T1	no HEX, OCT or char-class-wrapped literals	`get_tag`	12,482	91.8%	1,485	96.2%
T2	has HEX literal like `\xF5`	`[\x80-\xff]`	479	3.5%	243	15.7%
T3	has char-class-wrapped literals like [$]	`[$][{]\d+:([^}]+)[}]`	307	2.3%	268	17.4%
T4	has OCT literal like `\0177`	`[\041-\176]+:$`	14	.1%	37	2.4%
L1	curly brace repetition like {M,}	`(DN)[0-9]{4,}`	91	.7%	166	10.8%
L2	zero-or-more repetition using Kleene star	`\s*(#.*)?$`	6,017	44.3%	1,097	71.0%
L3	one-or-more repetition using plus	`[A-Z][a-z]+`	6,003	44.1%	1,207	78.2%
S1	curly brace repetition like {M}	`^[a-f0-9]{40}$`	581	4.3%	340	22.0%
S2	explicit sequential repetition	`ff:ff:ff:ff:ff:ff`	3,378	24.8%	861	55.8%
S3	curly brace repetition like {M,M}	`U[\dA-F]{5,5}`	27	.2%	32	2.1%

equivalence class with respect to the comprehension (RQ1) and community standards (RQ2) metrics.

A. Analysis

Total orderings were represented in directed graphs with representations as nodes and edge directions determined by the metrics: matching and composition of understandability; patterns and projects of community standards. The graphs for comprehension are based on Table III and for community support are based on Table IV. Within each graph, a topological sort created total orderings.

Building the Graphs: In the community standards graph, a directed edge $\overrightarrow{C2C1}$ is used when nPatterns(C1) $>$ nPatterns(C2) *and* nProjects(C1) $>$ nProjects(C2). When there is a conflict between nPatterns and nProjects, as is the case between L2 and L3, an undirected edge $\overline{L2L3}$ is used. For example, Figure 3 shows the graphs for the CCC group; Figure 3b is based on community standards.

A similar process is used to build the graphs based on the comprehension metrics. In Table III, each row maps to an edge in the understandability graph. If the matching and composition results both indicate a favorite (i.e., a bolded representation in the *Edge* column of Table III), there is a directed edge. For example, in Pair 3, the matching and composition metrics for C3 are higher than C1, resulting in a directed $\overrightarrow{C1C3}$ arrow. If one of the metrics is a tie, the other is used to break the tie; in Pair 2, the composition scores are the same but C1 is preferred based on matching, resulting in a $\overrightarrow{C2C1}$. If there is a conflict between the metrics, an undirected edge is used, as is the case with Pair 14, resulting in $\overline{C3C4}$. An example is shown in Figure 3a, which has 17 total edges, 14 of which are directed and three are undirected.

As a general rule, for both graphs, the higher the ratio of incoming edges to total edges, the less smelly the node.

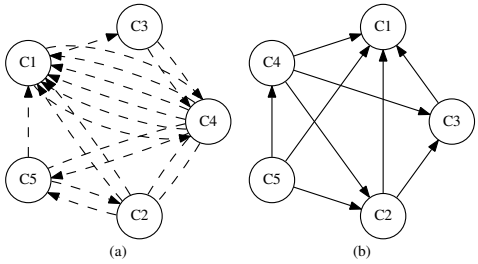

Fig. 3: Trend graphs for the CCC equivalence graph: (a) represent the comprehension analysis (RQ1) and (b) represent the artifact analysis (RQ2)

TABLE V: Topological Sorting, with the left-most position being highest (non-smelly) and the right-most being most smelly

	Understandability	Community
CCC	C1 C5 C3 C4 C2	C1 C3 C2 C4 C5
DBB	D3 D1 D2	D2 D1 D3
LBW	L3 L2	L3 L2 L1
SNG	S2 S1	S2 S1 S3
LIT	T1 T3 T2 T4	T1 T3 T2 T4

Topological Sorting: Once the two graphs are built for each equivalence class type, within each graph, we sort the nodes to identify a (preferably unique) total ordering on the nodes. This ordering represents preferences from the perspective of the comprehension or community standards metrics.

For each node n, we compute the ratio of $in\_deg(n)/deg(n)$ where $in\_deg(n)$ is the number of incoming edges to n, and $deg(n)$ is the total edges touching n. For example, in Figure 3a, $in\_deg(C5) = 2$ and $deg(C5) = 5$. The higher the ratio (that is, the more incoming edges indicating preference), the higher the node appears in the sorted list. For example, with node C1 in Figure 3a, the ratio is $7/10$ since C1 is involved in ten total comparisons and is favored in seven. The ratio for node C2 is $1/5$ as C2 is involved in five comparisons, is preferred in

one, is strictly not preferred in three, and has one with no preference, represented as an undirected edge.

One challenge with this (and any topological sorting approach, such as Kahn's algorithm), is that the total ordering is not necessarily unique and often multiple nodes have similar properties. Thus, we mark ties in order to identify when a tiebreaker is needed. Breaking ties on the community standards graph involves choosing the representation that appears in a larger number of projects, as it is more widespread across the community. Breaking ties in the understandability graph uses the RQ1 results. Based on Table III, we compute the average metrics for each representation and select the winner.

B. Results

The total orderings on nodes for each graph are shown in Table V. For example, given the graphs in Figure 3a and Figure 3b, the topological sorts are `C1 C5 C3 C4 C2` and `C1 C3 C2 C4 C5`, respectively.

Considering both topological sorts, there is a clear winner in each equivalence class except DBB. This is C1 for CCC, L3 for LBW, S2 for SNG, and T1 for LIT. While L3 is the winner for the LBW group, we note that L2 appears in slightly more patterns. DBB is different as the orderings are completely reversed depending on the analysis. Further study is needed on this, as well as LBW and SNG since not all nodes were considered in the understandability analysis.

These results can guide regex design. For example, to match numbers from one to 999, there are (at least) three options: A = `[1-9]|[1-9][0-9]|[1-9][0-9][0-9]`, B = `[1-9][0-9]?[0-9]?`, and C = `[1-9][0-9]{0,2}`. A contains representations {C1, D3, S2}, B contains {C1, D2}, and C contains {C1, D1}. According to Table V, the sorting in understandability is `A>C>B` since `D3>D1>D2`. However, what we usually see in source code are B and C but not A. The reason might be that the representation of A takes more time to type, or may have a longer runtime performance. In another example, we prefer `$[0-9]*.[0-9][0-9]` to `$[0-9]*.[0-9]{2}` in order to match dollar amounts. This is because S2 in the former regex is preferred to S1 in the latter regex, for all metrics.

C. Summary

Having a consistent and clear winner is evidence of a preference with respect to community standards and understandability, and thus provides guidance for potential refactoring. This positive result, that the most popular representation in the corpus is also the most understandable, makes sense as people may be more likely to understand things that are familiar or well documented.

VII. Discussion

Based on our studies, we have identified preferred regex representations that may make regexes easier to understand than their smelly counterparts. Here, we summarize the results.

A. Implications

Our goal in this work was to identify code smells in regular expressions. In an evaluation using humans where we measure comprehension of various regular expressions, we find that it is easier to understand regexes that do not use hex or octal characters, that repetition operators, such as `?` in D2 can decrease comprehension, and that using ranges is preferred to some character classes (e.g., `[A-Za-z0-9_` is often more understandable than `\w`).

In general, the factors that explain differences in comprehension metrics are the DFA size and the representation, where DFA sizes range from two to eight and the representations are as defined in Section 1. The implications of these results are for refactoring tool designers and code maintainers. Opting to use the preferred regex representations, when possible, may increase the understandability of regexes in source code. Since there are differences in regex comprehension based on how regexes are syntactically composed, it also has implications for refactoring tool designers to add refactoring for comprehension, which could enable developers to more easily compose regexes that are easier to understand.

B. Opportunities For Future Work

Equivalence Class Models: We looked at five types of equivalence classes, each with three to five representations. Future work could consider models with more types of equivalence classes, such as:

- Multiline option: `(?m)G\n` ≡ `(?m)G$`
- Case insensitive: `(?i)[a-z]` ≡ `[A-Za-z]`
- Backreference: `(X)q\1` ≡ `(?P<name>X)q\g<name>`

There also may exist critical comprehension differences within a representation. For example, between C1 (e.g., `[0-9a]`) and C4 (e.g., `[\da]`), it could be that `[0-9]` is preferred to `[\d]`, but `[A-Za-z0-9_]` is not be preferred to `[\w]`. By creating a more granular model of equivalence classes and carefully evaluating alternative representations of the most frequently used specific patterns, additional useful smells could be identified.

Automated Improvement: Currently the equivalence classes are identified manually. In future work, we will consider automatically generating the equivalence classes by building behavioral clusters and observing how regex representations differ within those clusters.

C. Threats to Validity

Internal: We measure understandability using two metrics: matching and composition. These measures may not reflect the actual understanding of regex behavior. To mitigate, we used multiple metrics that require reading and writing regexes.

We measure community support using two metrics: patterns and projects. These measures may not reflect the actual desire of the community to use the representations contained within. To mitigate, we use multiple metrics.

Participants evaluated regular expressions on MTurk, which may not be reflective enough of the context in which pro-

grammers would encounter regexes in practice. Further study is needed to determine the impact of the experimental context.

Some regex representations from the equivalence classes were not involved in the understandability analysis and that may have biased the results against those nodes. More complete coverage of the edges in the equivalence classes is needed.

We treated unsure responses as omissions that did not count against the matching scores. Thus, if a participant answered two strings correctly and marked the other three strings as unsure, then this was 2/2 correct, not 2/5. This may have inflated the matching scores, however, less than 5% of the matching scores were impacted by such responses.

External: The regexes used in the evaluation were inspired by those found in Python code, which is just one language that has library support for regexes, and may not generalize to other languages. Furthermore, the DFAs for the regexes ranged in size from two to eight, so the comprehension metrics may not generalize to larger regexes.

Our criteria for membership in a representation may overestimate the opportunities for refactoring to address the smells. For example, `[a-f]` in C1 cannot be refactored to C4 since there does not exist a default character class for that range of characters. The transformation between T4 and T1 may not be possible if the regex matches on non-printable characters, which require hex or octal representation. A finer-grained analysis is needed to identify actual refactoring opportunities from the smells.

Participants in our survey came from MTurk, which may not be representative of people who read and write regexes on a regular basis. However, all participants demonstrated knowledge of regexes through a qualification test. Our survey are done online without supervision and cheating could also be a factor impacting the results.

The results of the understandability analysis may be closely tied to the particular regexes chosen for the experiment. For many of the representations, we had several comparisons. Still, replication with more regex patterns is needed.

VIII. Related Work

Regular expression understandability has not previously been studied directly, though prior work has suggested that regexes are hard to read and understand [17] and noted that there are tens of thousands of bug reports related to regexes [6]. To aid in regex creation and understanding, tools have been developed to support more robust creation [6], to allow visual debugging [21], or to help programmers complete regex strings [22]. Other research has focused on removing the human from the creation process by learning regular expressions from text [13], [14].

Code smells in object-oriented languages were introduced by Fowler [23]. Researchers have studied the impact of code smells on program comprehension [15], [16], finding that the more smells in the code, the harder the comprehension. Code smells have been extended to other language paradigms including end-user programming languages [19], [24]–[26]. Using community standards to define smells has been used in refactoring for end-user programmers [19], [26].

Regular expression refactoring has not been studied directly, though refactoring literature abounds [27]–[29]. Refactoring for conformance to the community in end-user programs [19], [20] has been proposed previously, and is the inspiration behind RQ2 in this work. The closest to regex refactoring comes from research recent work that uses genetic programming to optimize regexes for runtime performance while maintaining their behavior in the matching language [30]. Similarly, other research has focused on expediting regular expressions processing on large bodies of text [31], similar to refactoring for performance. Our work is complementary to these efforts, where our focus is on comprehension and theirs is on performance.

Regarding applications, regular expressions have been used for test case generation [32]–[35], and as specifications for string constraint solvers [11], [36]. Flipping this around, recent approaches have used mutation to generate test strings for regular expressions themselves [37].

Exploring language feature usage by mining source code has been studied extensively for Smalltalk [38], JavaScript [39], Python regular expressions [17], and Java [40]–[43], and more specifically, Java generics [42] and Java reflection [43].

IX. Conclusion

In an effort to find smells that impact regex understandability, we created five types of equivalence classes and used these models to investigate the most common representations and most comprehensible representations per class. The high agreement between the community standards and understandability analyses suggests that one particular representation can be preferred over others in most cases. Based on these results, we recommend using hex to represent unprintable characters in regexes instead of octal and using escape special characters with slashes instead of wrapping them in brackets. Further research is needed into more granular models that treat common specific cases separately.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. This research is supported in part by NSF SHF-EAGER-1446932 and NSF SHF-1645136.

References

[1] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu, "Fully automatic wrapper generation for search engines," in *Proceedings of the 14th international conference on World Wide Web*. ACM, 2005, pp. 66–75.

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

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

[5] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved dfa for fast regular expression matching," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008.

[6] 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

[7] "Online regex tester, debugger with highlighting for php, pcre, python, golang and javascript," https://regex101.com/.

[8] "Regexr: Learn, build, and test regex," https://regexr.com/.

[9] "Regular expression visualizer using railroad diagrams," https://regexper.com/.

[10] "Rex @ rise4fun from microsoft," http://rise4fun.com/rex/.

[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.*, vol. 21, no. 4, pp. 25:1–25:28, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2377656.2377662

[12] 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.

[13] 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*, ser. AND '10. New York, NY, USA: ACM, 2010, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/1871840.1871848

[14] 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*, ser. EMNLP '08. Stroudsburg, PA, USA: Association for Computational Linguistics, 2008, pp. 21–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=1613715.1613719

[15] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 181–190.

[16] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does god class decomposition affect comprehensibility?" in *IASTED Conf. on Software Engineering*, 2006, pp. 346–355.

[17] 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.

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

[19] K. T. Stolee and S. Elbaum, "Refactoring pipe-like mashups for end-user programmers," in *International Conference on Software Engineering*, 2011.

[20] K. T. Stolee and S. Elbaum, "Identification, impact, and refactoring of smells in pipe-like web mashups," *IEEE Trans. Softw. Eng.*, vol. 39, no. 12, Dec. 2013.

[21] 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*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 504–507. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591111

[22] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 859–869. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337324

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

[24] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Proc. of ICSM '12*, 2012.

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

[26] K. T. Stolee and S. Elbaum, "Identification, impact, and refactoring of smells in pipe-like web mashups," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1654–1679, 2013.

[27] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Soft. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1109/TSE.2004.1265817

[28] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[29] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 228–269, Jul. 1993. [Online]. Available: http://doi.acm.org/10.1145/152388.152389

[30] B. Cody-Kenny, M. Fenton, A. Ronayne, E. Considine, T. McGuire, and M. O'Neill, "A search for improved performance in regular expressions," *arXiv preprint arXiv:1704.04119*, 2017.

[31] R. A. Baeza-Yates and G. H. Gonnet, "Fast text searching for regular expressions or automaton searching on tries," *J. ACM*, vol. 43, no. 6, pp. 915–936, Nov. 1996. [Online]. Available: http://doi.acm.org/10.1145/235809.235810

[32] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 992–1001. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486925

[33] S. J. Galler and B. K. Aichernig, "Survey on test data generation tools," *Int. J. Softw. Tools Technol. Transf.*, vol. 16, no. 6, pp. 727–751, Nov. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10009-013-0272-3

[34] 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

[35] 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.

[36] 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*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1232–1243. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660372

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

[38] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger, "How developers use the dynamic features of programming languages: The case of smalltalk," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985448

[39] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," *SIGPLAN Not.*, vol. 45, no. 6, Jun. 2010.

[40] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of ast nodes to study actual and potential usage of java language features," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014.

[41] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010.

[42] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of java generics," *Empirical Softw. Engg.*, vol. 18, no. 6, Dec. 2013.

[43] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for java," in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, ser. APLAS'05. Berlin, Heidelberg: Springer-Verlag, 2005.