Generating Evil Test Strings for Regular Expressions

Eric Larson and Anna Kirk
Computer Science and Software Engineering
Seattle University
Seattle, WA USA
elarson@seattleu.edu and kirka1@seattleu.edu

***Abstract*— Regular expressions are a powerful string processing tool. However, they are error-prone and receive little error checking from the compiler as most regular expressions are syntactically correct. This paper describes EGRET, a tool for generating evil test strings for regular expressions. EGRET focuses on common mistakes made by developers when creating regular expressions and develops test strings that expose these errors. EGRET has found errors in 284 out of 791 regular expressions. Prior approaches to test string generation have traversed all possible paths in the equivalent nondeterministic finite state automaton leading to the generation of too many strings. EGRET keeps the set of test strings to a manageable number: Fewer than 100 test strings were generated for 96% of the regular expressions; a manageable 307 test strings were generated for the most complex regular expression.**

Keywords— testing; regular expressions; test generation

I. INTRODUCTION

Regular expressions are widely used in a variety of computer programming tasks such as processing stylized input, validating data entered into a form on a web page, and searching for different types of text. Despite their widespread use, regular expressions are error-prone. First, the "language" used to specify regular expressions is designed to be compact, using punctuation marks to represent different operations, allowing complex regular expressions to be written succinctly. Second, when the regular expression is compiled at run-time, only limited error-checking related to syntax is done. Most regular expressions are free of syntax errors.

This paper describes the EGRET (Evil Generation of Regular Expression Tests) tool. EGRET takes a regular expression as input and generates test strings. The "evil" aspect of EGRET focuses on generating test strings to expose common errors made by programmers. Some strings generated by EGRET are actually intended to be rejected by the regular expression. EGRET creates two lists of test strings: accepted strings and rejected strings. A user can quickly scan both of these lists and identify strings that are incorrectly classified. Even if no bugs are found, the user has higher confidence that the regular expression is working as intended.

EGRET works by converting the regular expression into a specialized nondeterministic finite state automaton (NFA). The resulting NFA is traversed to obtain a set of basis paths. The strings that correspond to these paths form an initial set of test strings. These strings are processed further to generate additional strings. For each character set, character class, and wildcard, a set of interesting characters is created and one test string will be added for each interesting character. For each repeat quantifier, test strings are generated for different iteration counts of the corresponding substring that are near the boundaries.

The design decisions of EGRET focus on common mistakes for regular expressions. In general, the regular expression language makes it easy to write regular expressions that are overly simple and accept more strings than it should. For example, a poor regular expression that accepts a floating point number is "`[0-9,.]+`". This will properly accept all floating point numbers but it will also accept strings such as "`1,,00`", "`3.4.5`", and simply "`.`". Writing a regular expression that properly rejects poorly formatted entries is more difficult. EGRET helps find these issues by generating strings that are likely to be incorrect. It has generated problematic strings such as "`.@.`" for an e-mail regular expression and "`200)00000000`" for a phone number regular expression.

Since the strings are manually analyzed, the number of test strings generated must be manageable. One approach is to traverse all of the paths in the NFA while limiting the number of iterations of any loops. However, this generates too many paths and resulting strings. EGRET does not suffer from path explosion problem in several ways. First, only a set of basis paths are traversed. The number of basis paths is linear while likely retaining all of the interesting combinations. Second, the number of iterations for repeat quantifiers is limited to values near the boundary. Lastly, a set of interesting characters is selected for each character set that is likely to find bugs without creating a string for each possible character.

This paper makes the following research contributions:

- A technique for generating regular expression test strings targeted at finding bugs. Using EGRET, we were able to find bugs in 284 out of 791 regular expressions extracted from RegExLib.com [11] and six Python programs.
- Optimizations to the technique that minimize the number of test strings that are generated. 96% of the 707 regular expressions generated fewer than 100 test strings. The most complex regular expression generated 307 test strings.
- The EGRET tool and source code is available for use at: http://fac-staff.seattleu.edu/elarson/web/egret.htm.

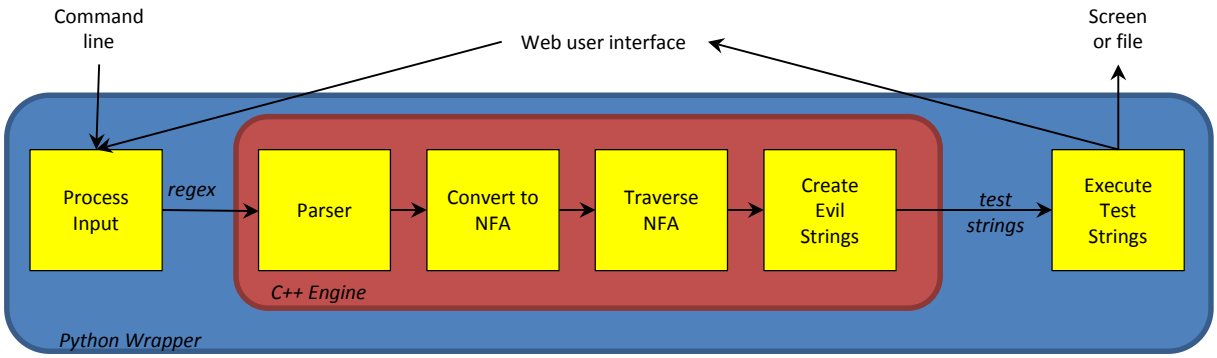


Fig. 1. Process for Generating Evil Test Strings

The remainder of the paper is organized as follows. Section II gives an overview of EGRET while section III gives more details on the test string generation process. Results are discussed in section IV. Section V outlines related work and section VI concludes.

II. Overview

The process of generating evil strings in EGRET is shown in Fig. 1. A Python wrapper takes the regular expression and passes it to a C++ engine. The regular expression engine developed by Bendersky [3] was used as a starting point and has been significantly modified. The engine parses the regular expression and converts it into an equivalent NFA. The NFA is traversed to get a set of basis paths forming the initial set of test strings. The last phase in the engine adds additional evil test strings. Then each test string is tested against the regular expression generating a list of accepted strings and a list of rejected strings. The lists are then displayed to the user. More details of the test string generation process are in Section III.

A. User Interface

EGRET contains a command-line interface and a graphical web interface. They both operate in a similar fashion. A user enters a regular expression and receives a list of accepted string and rejected strings. EGRET may also emit error and warning messages in certain situations. An optional feature allows users to see the contents of any groups for an accepted string. The web interface also allows the user to enter their own test string and determine if the string is accepted or rejected. The command-line interface also supports file input/output.

B. Supported Regular Expression Elements

A summary of what regular expression elements are supported is provided in TABLE I. The table is broken down into three groups: supported, ignored, and unsupported. The C++ engine ignores elements in the ignored group but test strings will be generated anyway. However, the results may be substandard depending on how the ignored element is used. If a regular expression contains an element that is in the unsupported group, EGRET aborts. Regular expressions that contain syntax errors are also rejected.

EGRET only generates strings that are entire string matches but does check that the begin and end anchors are used correctly. The Python wrapper uses the method `fullmatch` to test the strings against the regular expression. Optional flags, such as multiline mode, are not used nor are they supported.

TABLE I. Python Regular Expression Support

Supported	• Alternation: `|` • Repeat quantifiers: `?`, `*`, `+`, `??`, `*?`, `+?` • User-defined repeat quantifiers: `{2,5}` • Wildcards: `.` • Character sets: `[A-Z\s_]` • Character classes: `\w` • Groups: `()` • Anchors: `^`, `$`, `\A`, `\Z`
Ignored	• Word boundaries: `\b`, `\B` • Lookahead / lookbehind assertions
Unsupported	• Non-printable ASCII characters • Non-ASCII characters • Newlines and carriage returns • Flags: `?i`, `?m` • Backreferences • Conditional patterns

III. GENERATING TEST STRINGS

This section describes each of the four phases used in the generation of test strings and concludes with an example.

A. Parser

The parser is responsible for converting the regular expression into a parse tree. The parser also detects regular expressions that are poorly constructed.

There are nine types of nodes in the parse tree:

- ALTERNATION: Represents alternation '`|`'.
- CONCATENTATION: Represents concatenation of two regular expression elements.
- REPEAT: Represents all repeat quantifiers: `?`, `*`, `+`, `??`, `*?`, `+?`, and user-defined repeats quantifier such as `{2,5}`. The lower bound and upper bound (if one exists) are stored with the node. Since EGRET is only focused on whether the entire string is accepted or rejected, the lazy quantifiers such as `*?` are treated the same as the corresponding greedy quantifier.
- GROUP: Represents any set of parentheses. It could be a named group, a nameless group, or non-capturing

parentheses – no distinction is made between the three types in the generation of strings.

- CHARACTER: Represents a specific individual character including escaped characters such as `\(` and characters specified using octal or hexadecimal values.
- CHARACTER SET: Represents any construct that corresponds to a single character but there are several choices. This includes character sets (such as `[A-Z_;+]`), character classes (such as `\w`), and wildcards '`.`'. A character set node retains the list of items that comprise the character set[1] and a flag that indicates if the set is negated (using `^`) or not.
- CARET: Represents the `^` and the `/A` beginning-of-string anchors. Since multiline mode is not supported, these two anchors are equivalent.
- DOLLAR: Represents the `$` and the `/Z` end-of-string anchors. Since multiline mode is not supported, these two anchors are equivalent.
- IGNORED: Represents ignored elements as noted in TABLE I.

B. Convert to NFA

The parse tree is converted into a specialized NFA. A traditional approach (as described in [7]) is used to convert concatenation and the matching of individual characters. A transition between NFA states can either be an individual character, a character set, an anchor (`^` or `$`), or an epsilon transition. Only one transition is generated for a character set, but the set of allowable characters is retained and used when creating evil strings.

The translation of repeating constructs deviates from the traditional approach. Instead of creating a loop in the NFA, special "begin repeat" and "end repeat" states are respectively added to the beginning and end of each repeating construct. These states keep track of the type of construct (`?`, `*`, `+`, `{n,m}`) along with the acceptable ranges for the user-specified repeat quantifier. By representing the loops in this manner, the resulting NFA is a directed acyclic graph.

C. Traverse the NFA

The NFA is traversed to generate an initial set of test strings. A set of basis paths [20] is generated in a depth-first manner using the backtracking algorithm shown in Fig. 2. Generating basis paths takes O(*n*) time where n is the number of decision points. The only decision points in the NFA are due to alternation. If there are *n* alternation nodes, there are *n* + 1 basis paths.

During development, we tried alternative approaches: complete path coverage and state coverage (due to the nature of how the NFA is constructed, transition coverage is identical to state coverage). Complete path coverage generated too many paths to be reasonably analyzed by a person for many of the regular expressions. In Section IV-D, we show results for EXREX [16], a tool that uses full path coverage for traversal. State coverage did not generate enough paths causing the possibility to miss certain problems. This is best illustrated using the example in Section III-E.

```
FIND_BASIS_PATHS(curr_state, path)
1   if curr_state is the final state:
2     add path to list
3     mark all states in path as visited
4   else if curr_state.visited:
5     next_state = lowest-numbered state that has a transition
        from curr_state to next_state
6     FIND_BASIS_PATHS(next_state, path + curr_state)
7   else:
8     for each next_state that has a transition from curr_state
        to next_state:
9     FIND_BASIS_PATHS(next_state, path + curr_state)
```

Fig. 2. Backtracking Algorithm for Generating Basis Paths

For each path, a string is formed by taking the characters from the transitions that comprise the path. For character set transitions, an acceptable character is chosen from the set. When a repeat quantifier is discovered, the substring between the "begin repeat" and "end repeat" states is noted. If the lower bound is zero or one, the substring is added once to the test string. In other words, the test string has one iteration of the substring. If the lower bound is two or more, the substring will be repeated until the lower bound is met.

A path may contain the anchors `^` or `$`. No character is added to string for either anchor. Instead, error checking takes place for these errors:

- A `^` anchor occurs after the first character.
- A `$` anchor occurs before the last character.
- Some, but not all, strings start with an `^`.
- Some, but not all, strings end with an `$`.

The first two errors are caused by a poorly formed regular expression. The last two errors target a common error: forgetting that alternation has lower precedence than the anchors. For example, using the regular expression "`^cat|dog$`" instead of "`^(cat|dog)$`" to match either the string "`cat`" or "`dog`".

The output of this phase is an initial list of test strings that all satisfy the regular expression under two assumptions: (a) there are no errors due to an anchor being in the middle of the string and (b) the regular expression does not contain any ignored elements from TABLE I. We assume both of these assumptions are met in the next section.

D. Create Evil Strings

This phase takes the test strings generated by traversing the NFA and creates additional evil test strings. There are two methods for creating evil strings: (a) altering the number of iterations for each repeat quantifier and (b) changing the character used for a character set. To avoid an explosion of test strings, each repeat quantifier and character set is associated with a single path (and corresponding test string) that contains that element. New test strings are only generated for that path, not all of the paths that contain a particular element.

[1] Character classes can and wildcards must appear outside a character set, not in braces []. In these cases, a character set node with a single item is created.

TABLE II. NUMBER OF ITERATIONS FOR REPEAT QUANTIFIERS

Repeat Quantifier	Number of Iterations
* or {0,}	0, ***1***, 2
+ or {1,}	0, ***1***, 2
? or {0,1}	0, ***1***, 2
{0,n} or {,n}	0, ***1***, n, n + 1
{m,n} (m > 0 and n > m)	m – 1, ***m***, n, n + 1
{m} or {m,m}(m > 0)	m – 1, ***m***, m + 1
{m,} (m > 1)	m – 1, ***m***

TABLE III. CHARACTER SET PROCESSING

Character Set Item	Add to Set	Flags Set
Uppercase character *x*	*x*	Uppercase
Lowercase character *x*	*x*	Lowercase
Digit character *x*	*x*	Digit
Other character *x*	*x*	none
Uppercase range *x-y*	a letter in range	Uppercase
Lowercase range *x-y*	a letter in range	Lowercase
Digit range *x-y*	a digit in range	Digit
Character class \w	an uppercase letter a lowercase letter a digit _ (underscore)	none
Character class \d	digit	none
Character class \s	<space>	none
Character classes \W, \D, \S	an uppercase letter a lowercase letter a digit _ (underscore) <space>	Punctuation
Wildcard .	an uppercase letter a lowercase letter a digit <space>	Punctuation
Any negated character set	an uppercase letter a lowercase letter a digit <space>	Punctuation

To alter the number of iterations for a repeat quantifier, the test string is divided into three parts: a prelude, the repeating substring, and a postlude. Adding new strings is simply concatenating these parts with the proper number of iterations for the repeating substring. The number of iterations depends on the repeat quantifier as summarized in TABLE II. The number of iterations in the initial test string are bold and italicized. Strings containing the other iteration counts are added during this step.[2]

The choice of the number of iterations is inspired by boundary value testing. Many of the iteration counts violate the repeat quantifier causing the string to be rejected by the regular expression.[3] This borrows a key idea from modified condition / decision coverage (MC/DC). Since the rest of string satisfies the regular expression, the number of iterations will independently determine whether the string is accepted or rejected by the regular expression.

Altering character sets is done in a similar manner. The test string is divided into a prelude, the character from the character set, and a postlude. A set of interesting characters is created based on the contents of the character set. For each character in this interesting set, an additional test string is created using that character in place of the original character.

The process of creating the set of interesting characters has two distinct phases. During the first phase, the contents of the character set are scanned and processed according to TABLE III. The *Add to Set* column indicates which characters should be added to the set. The *Flags Set* column indicates which, if any, of these four flags are set: uppercase, lowercase, digits, and punctuation. These flags are used in the second phase to possibly add more characters to the set. When adding characters to the set, there often is a choice of characters such as "a letter in range" or "a digit"; the character with the smallest ASCII value (closest to 'A', 'a', or '0') is chosen. This choice was made to keep the algorithm deterministic during development, testing, and evaluation. We envision adding an option to select the character randomly in the future.

When processing character classes and wildcards, EGRET ensures that at least one character is chosen from each interesting subset. For instance, there will be at least one test string where the \w is replaced with an uppercase letter, another where it's replaced with a lowercase letter, a string where it's replaced with a digit, and a fourth string where it's replaced with an underscore.

EGRET performs checks for poorly-constructed ranges. First, ranges must be confined to only digits, only uppercase letters, or only lowercase letters. Ranges such as A-z cause EGRET to abort with an error. Second, EGRET checks that ranges do not overlap with other ranges or characters specified within the same character set. During evaluation, this error only occurred twice but did flag this poorly written character set: "[1-31]".

The second phase is best illustrated using an example. If the uppercase flag is set, one more uppercase letter will be added to the set if possible. The uppercase letter will be chosen such that it does not match any specified characters and is outside any specified ranges. For example, the character set "[AB-FGR-UW]" will generate the characters: A (for the letter A), B (for the range B-F), G (for the letter G), R (for the range R-U), W (for the letter W), and H (representative uppercase letter that does not match any letter or range). An additional letter will not be added if all 26 uppercase letters are "covered" by either an individual letter or from a range. The most common situation where this occurs is when the entire range A-Z is specified. In this scenario, no additional letters are selected since A-Z covers the entire range of uppercase letters. The same approach is used for the lowercase and digit flags.

The punctuation flag is handled differently. First, it is only set when all the punctuation marks are acceptable: opposite character classes such as \W, wildcards, and negated character sets using ^. When the flag is set, each punctuation mark that appears somewhere in the regular expression, not just in the character set, is added to set of interesting characters. This choice was made because the punctuation marks that appear elsewhere in the regular expression are the punctuation marks that are more likely to need to be excluded. Consider a regular

[2] For nonsensical bounds such as {0,0}, EGRET aborts with an error message.

[3] Unless the string could be accepted using a completely different path through the regular expression. That is why the test strings are tested against the regular expression at the end of the process.

expression "\(.+\)". The '(' and ')' will be both be added to the set of interesting characters leading to the generation of test strings "(()" and "())". If there are no punctuation marks, the "_" is added to the set of interesting characters so that it contains at least one punctuation mark.

This approach specifically targets a common problem – a wildcard is incorrectly used instead of a more restricted character set. To combat this problem, different types of characters and punctuation marks are added for wildcards. A related problem is a negated character set that does not fully specify what needs to be excluded. In addition, not all of the characters in the interesting set result in test strings that are accepted – some strings will be rejected. The goal is to give the user confidence that strings that should be accepted are accepted and strings that should be rejected are indeed rejected.

E. Example

An example of the process of generating strings from a regular expression is shown in Fig. 3. The regular expression, a slightly simplified version of a buggy regular expression from RegExLib.com [11], accepts a United States phone number in the form "(555)-555-5555". The parentheses around the area code are optional and a period '.' can be used instead of a dash '-' to delimit the different parts of the phone number. The regular expression also ensures that area codes do not start with 0 or 1 as US area codes do not begin with those digits.

The first part of Fig. 3 shows the NFA corresponding to the regular expression. The two alternation operators are represented by the two outgoing transitions at states 14 and 24. Epsilon (∈) edges allow transitions between states without consuming any input characters. Other transitions include individual characters ('(', '-' and ')') and character sets (includes[2-9], \d, and the wildcard .). The repeat quantifiers (?, {2}, {3}, and {4}) are associated with a pair of states one representing the beginning and one representing the end. For instance, state 6 is start state for {2} and state 9 is the end state for {2}.

The next step is to generate the basis paths from the NFA. The first path chooses states 15 and 25 at the two decision points. This results in choosing the '-' in both alternation decisions. To form the test string, an acceptable value is chosen for each character set. For ranges, the lowest value is chosen meaning that 2 is chosen for the range [2-9] and 0 is chosen for \d. With these choices, the test string "(200)-000-0000" is formed.

A second path is found by backtracking to state 24 and choosing state 27 instead. This means that a wildcard is used instead of the '-'. For wildcards, the lowercase 'a' is chosen resulting in a problematic test string of "(200)-000a0000". The third and final basis path is created by backtracking to state 14 and choosing state 15 resulting in the test string "(200)a000-0000".

To illustrate the importance of basis paths, imagine the wildcards are actually periods (like the author intended). The author likely intended if the test string has a dash at state 14, there would also be a dash at state 24. Similarly, if the test string has a period at state 14, there should also be a period at state 24. This is a common problem in where the author of the regular expression forgets to use a backreference. Generating paths to obtain state coverage could result in only two paths: one that chooses both dashes and one that chooses both periods. By using basis path coverage, three paths are generated guaranteeing there will be at least one path that chooses a period at one decision point and a dash at the other.

The next phase generates additional evil strings by modifying the iteration counts for repeat qualifiers and by choosing different characters for character sets. For the ? represented in state 3, the corresponding substring for the repeat quantifier is simply the string "(". Based on TABLE II, a string is created with no iterations "200)-000-0000" and a string is created with two iterations "((200)-000-0000". Even though state 3 appears in all three paths, this modification only occurs in one of the paths (path 1 in this case). The same modifications are applied to the ? represented in state 13.

State 9 represents the {2} repeat quantifier. The substring between the begin state (state 6) and the end state is the string "0". In the initial path string, this string is repeated forming the string "00" which satisfies the {2} requirement. New strings are created, again using TABLE II, to create paths that have one iteration "0" and three iterations "000". Similar modifications are applied to states 23 and 33.

A character set transition occurs between states 4 and 5 with the character set [2-9]. According to TABLE III, a value in the range is chosen (2 in this case) and the digit flag is set. By setting the digit flag, an additional value is chosen outside the range: 0 is selected from either 0 or 1. This forms the test string "(000)-000-0000". As with repeat quantifiers, this modification is only applied to one path.

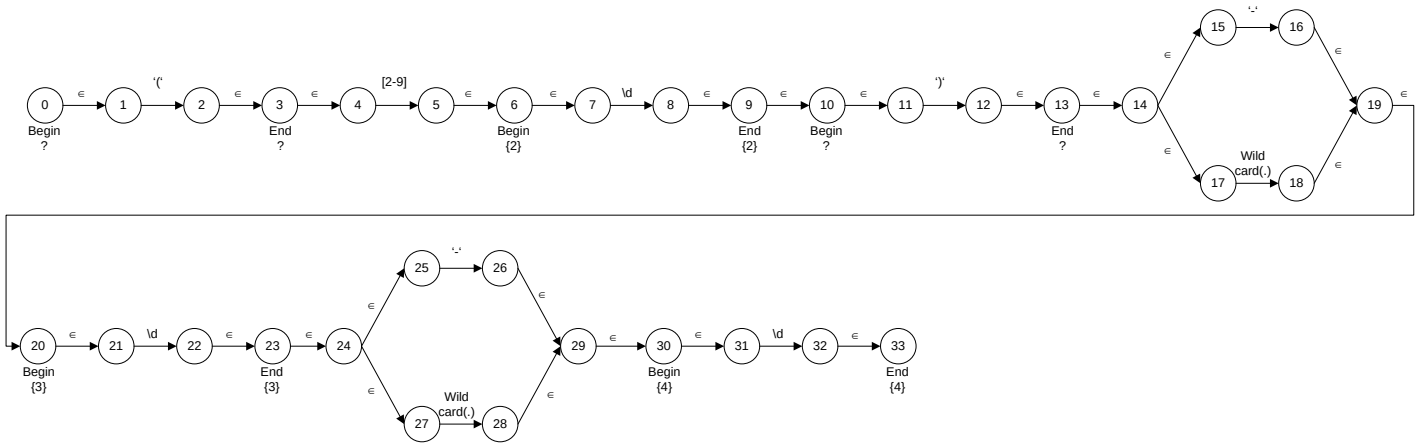
The wildcard between states 17 and 18 requires several different characters to be used. First, TABLE III indicates different strings using an uppercase letter ('A'), a lowercase letter ('a'), a digit ('0'), and a space. In addition, the punctuation flag is set. This means that all punctuation marks that appear elsewhere in the regular expression must be used in place of the wildcard in a test string. In this example, the punctuation marks consist of '(', ')', and '-'. The initial string uses the lowercase letter 'a' so six new strings are created by replacing the 'a' with each of these: 'A', '0', ' ', '(', ')', and '-'. The string with the '-' is a duplicate of the initial test string "(200)-000-0000" so it is discarded. The same set of characters is used to replace the wildcard between states 27 and 28.

The three \d character classes do not require any modification as the interesting set of characters only has one character: the digit '0'. Since the initial test string already uses this character, no additional test strings are generated.

In total, 21 additional test strings are created during the evil string generation phase. These 21 strings are added to the 3 initial test strings to create a list of 24 test strings. The 24 strings are tested against the regular expression to determine if they match or not. The list of accepted strings and rejected strings are displayed to the user.

Regular expression: `\(?[2-9]\d{2}\)?(-|.)\d{3}(-|.)\d{4}`

NFA:



Basis paths:

1: 0-…-14-15-16-19-…-24-25-26-29-…-33
2: 0-…-14-15-16-19-…-24-27-28-29-…-33
3: 0-…-14-17-18-19-…-24-25-26-29-…-33

Initial Strings:

`(200)-000-0000`
`(200)-000a0000`
`(200)a000-0000`

Generate Additional Strings:

State 3 (path 1): Replace '`(`' with '' and '`((`'	0 and 2 iterations for `?`
State 5 (path 1): Replace '`2`' with '`0`'	one digit outside `[2-9]`
State 9 (path 1): Replace '`00`' with '`0`' and '`000`'	1 and 3 iterations for `{2}`
State 13 (path 1): Replace '`)`' with '' and '`))`'	0 and 2 iterations for `?`
State 18 (path 3): Replace '`a`' with '`A`', '`0`', '` `', '`(`', '`)`', and '`-`'	uppercase letter, digit, space, and each punctuation mark for wildcard `.`
State 23 (path 1): Replace '`000`' with '`00`' and '`0000`'	2 and 4 iterations for `{3}`
State 28 (path 2): Replace '`a`' with '`A`', '`0`', '` `', '`(`', '`)`', and '`-`'	uppercase letter, digit, space, and each punctuation mark for wildcard `.`
State 33 (path 1): Replace '`0000`' with '`000`' and '`00000`'	3 and 5 iterations for `{4}`

Accepted Strings:

```
(200) 000-0000
(200)(000-0000
(200))000-0000
(200)-000 0000
(200)-000(0000
(200)-000)0000
(200)-000-0000
(200)-00000000
(200)-000A0000
(200)-000a0000
(200)0000-0000
(200)A000-0000
(200)a000-0000
(200-000-0000
200)-000-0000
```

Rejected Strings:

```
((200)-000-0000
(000)-000-0000
(20)-000-0000
(200))-000-0000
(200)-00-0000
(200)-000-000
(200)-000-00000
(200)-0000-0000
(2000)-000-0000
```

Fig. 3. Example for Generating Test Strings

By scanning the list of accepting strings, the user should be able to identify they forgot to escape the period so the period is treated as a wildcard. As a result it permits phone numbers with 11 digits, as denoted by the string "`(200)0000-0000`", to be incorrectly accepted. The user can also see that improperly formatted phone numbers "`(200-000-0000`" and "`200)-000-0000`" are accepted.

The rejected strings do not reveal any bugs in this example. However, they do provided the user confidence that phone numbers with area codes that start with 0 or 1 are rejected since "`(000)-000-0000`" is rejected. It also assures the user that a phone number with the incorrect number of digits will be rejected (once the wildcard problem is fixed).

IV. Results

To evaluate the approach, we used regular expressions from the regular expression library RegExLib.com [11]. The library has eight primary categories as shown in TABLE IV. We extracted every regular expression from these eight categories. Some of the regular expressions were not written

TABLE IV. REGULAR EXPRESSIONS USED IN EVALUATION

		Regexes	Tested	Not supported
RegExLib Library Category	address / phone	104	98	6
	dates / time	135	117	18
	email	38	33	5
	markup / code	63	50	13
	misc	173	150	23
	numbers	107	102	5
	string	91	73	18
	uri	74	69	5
Python Program	advas (0.2.5)	16	16	0
	beautifulsoup4 (4.3.2)	7	7	0
	pychecker (0.8.19)	4	4	0
	pymetrics (0.8.1)	6	5	1
	tables (3.1.1)	8	8	0
	Trac (1.0.1)	63	59	4
	Library total	785	692	93
	Programs total	104	99	5
	Total	***889***	***791***	***98***

TABLE V. TYPES OF REGULAR EXPRESSION BUGS

Bad Range: Any range that includes anything that is not a letter or digit.

Example: `^[D-d][K-k]-[1-9]{1}[0-9]{3}$`
Error message: `ERROR: Range is likely incorrect: D-d`

Anchor Usage: Incorrect use of ^ and $ anchors.

Example: `^[-+]?\d+(\.\d+)?|[-+]?\.\d+?$`
Error message:
```
WARNING: some but not all strings start with a ^
String with ^ anchor:    +0.0
String with no ^ anchor: +.0
WARNING: some but not all strings end with a $
String with $ anchor:    +.0
String with no $ anchor: +0.0
```

Character Set: Using regex elements incorrectly in a character set or used braces instead of parentheses.

Example (taken from a time regular expression):
`[AM|PM|am|pm]{2,2}`
Incorrectly accepted strings: `Mm`, `|m`

Delimiter Mismatch: Not having consistent delimiters or incomplete parentheses, braces, or brackets.

Example: `href[ ]*=[ ]*('|\")([^\"'])*('|\")`
Incorrectly accepted strings: `href = 'a", href = "a'`

Negation: Using negation in a character set (using `^`) or using an opposite character class (such as `\D`) but not removing all possibilities.

Example:
`href=[\"\']?((?:[^>]|[^\s]|[^"]|[^']))+)[\"\']?`
Incorrectly accepted string: `href='''`

Wildcard: Using a wildcard when some possibilities should be excluded or forgetting that the "." is a wildcard.

Example (email address, `\x2E` is a period): `(\w+?@\w+?\x2E.+)`
Incorrectly accepted string: `a@a..`

Wrong Repeat: Using the wrong repeat quantifier.

Example (matching US currency with either 0 or 2 digits past the decimal point): `^\d+(?:\.\d{0,2})?$`
Incorrectly accepted string: `0.0`

for Python – these were converted to Python regular expressions. A few regular expressions were excluded because they contained an element that was not supported in Python (the most common being Unicode categories such as `\p{Letter}`). There is a total of 785 regular expressions that are supported in Python. Of these 785 regular expressions, 93 contain an element that is not supported by our tool (see TABLE I). Testing was carried out using the 692 remaining regular expressions.

In addition, we manually extracted regular expressions from six different Python programs as shown in TABLE IV. We did not include regular expressions for substitutions, splits, or regular expressions that involved a (non-constant) variable. From these six programs, 104 regular expressions were extracted. Five regular expressions contained unsupported elements resulting in 99 regular expressions to test.

A. Bugs Found

The bug finding capability was evaluated by looking at the output generated by EGRET. First, any warning or error messages were investigated. A bug occurs if the message clearly points to a bug. Second, the generated strings were analyzed. A bug occurs if any of the accepted strings should be rejected and any of the rejected strings should be accepted. Bugs were divided into seven categories plus an *other* classification for bugs that don't fall into one of the earlier categories. TABLE V describes the seven categories. For each category, an actual buggy regular expression is given along with the offending error message or generated test string. The first two errors in TABLE V (*bad range* and *anchor usage*) are exclusively detected using error messages. *Character set* bugs are most often detected by analyzing test strings but could be detecting an error message in some cases. The other four types of bugs are exclusively caught by analyzing test strings – no error messages exists. Bugs found in the *other* category were all caught by analyzing test strings except for four regular expressions that contained syntax errors.

The act of determining bugs is subjective. The first three categories (*bad range*, *anchor usage*, *character set*) in TABLE V were straightforward to evaluate – they all refer to mistakes that are more similar to syntax errors than logical flaws in the regular expression. These errors are also easy to fix when detected. The fourth category (*delimiter mismatch*) was also straightforward to detect but the bugs are often not that significant. For example, it may not matter if the regular expression accepts poorly formatted phone numbers such as "555-555.5555" or "(555555-5555" provided the correct number of digits are provided. The last three categories (*negation*, *wildcard*, *wrong repeat*) along with all the bugs in the *other* classification are more subjective.

Fortunately, each regular expression in RegExLib.com contains a brief description. Unfortunately, these descriptions often lacked detail on what precisely should be accepted or rejected. We often had to infer what the author was to trying to accomplish with a given regular expression. For issues found in the actual Python programs, we did not investigate the source code further to determine if additional checks were present to mitigate the bug. In all cases that we detected a bug, we felt it was something worthy of additional investigation.

TABLE VI shows the number of bugs found across the different categories and programs: 284 of the 791 (36%) regular expressions contained a bug. Note that several regular expressions contained multiple bugs so the total number of bugs (347) is higher than the number of the buggy regular

TABLE VI. BUGS FOUND

Category / Program	Regexes Tested	Buggy Regexes	Bad Range	Anchor Usage	Character Set	Delimiter Mismatch	Negation	Wild card	Wrong Repeat	Other
address / phone	98	40	4	5	6	20	3	3	7	6
dates / time	117	59	2	2	18	26	0	1	5	26
email	33	12	3	0	1	1	2	2	0	5
markup / code	50	23	0	0	0	9	7	10	1	4
misc	150	51	4	8	6	11	9	8	5	12
numbers	102	23	0	2	0	0	0	4	6	13
string	73	31	2	4	9	2	2	1	2	12
uri	69	26	4	1	1	2	4	3	1	11
advas	16	0	0	0	0	0	0	0	0	0
beautifulsoup	7	3	0	0	0	0	1	1	0	1
pychecker	4	1	0	0	0	1	0	0	0	0
pymetrics	5	0	0	0	0	0	0	0	0	0
tables	8	0	0	0	0	0	0	0	0	0
Trac	59	15	3	0	0	0	3	8	1	0
Regex Library	692	265	19	22	41	71	27	32	27	89
Programs	99	19	3	0	0	1	4	9	1	1
Total	***791***	***284***	***22***	***22***	***41***	***72***	***31***	***41***	***28***	***90***

expressions (284). For each of the seven categories, there were at least 20 bugs. The most common mistake was a delimiter mismatch – many due to regular expressions that accept incorrectly formatted phone numbers and dates. These results demonstrate that EGRET is effective at finding bugs.

Of the 347 bugs, 48 were detected using an error message and 299 were detected by analyzing the test strings. Of the 48 detected by error message: 22 are anchor usage errors, 22 are bad ranges, and 4 are cases where the regular expression has a syntax error. There are three false alarms, situations where the error message did not refer to a bug. Two of these cases were situations where an anchor usage warning was emitted but the author intended to have some matches that match the beginning of the line and some that do not. The third false alarm signaled a character set with overlapping ranges. While the character set was sloppily written, it was indeed correct.

The remaining 299 bugs were detected by analyzing the generated strings. All but three bugs were detected by finding a string that was accepted but should have been rejected. Three bugs were detected by finding a string that was rejected that should have been accepted. Two bugs are *wrong repeat* bugs and the third bug was in the *other* category. Note that, by definition, *wrong repeat* is the only type of the seven bug types in that could be detected in this manner.

Here is one of the faulty regular expressions:

```
^(\d{1,3}'(\d{3}')*\d{3}(\.\d{1,3})?|
\d{1,3}(\.\d{3})?)$
```

According to the description, this regular expression can accept up to three digits past the decimal point. However, the string "`0.00`" is generated and rejected by the regular expression. The error is the last `{3}` should be `{1,3}`. Even though only three bugs were detected using rejected strings, there is the potential for more bugs. For example, several floating point number regular expressions allowed "`.0`" for a number but rejected "`0.`" (or vice versa). There was not enough information in the description to determine if this was the intended behavior or a bug. A user creating a regular expression for floating point values can look at the accepted and rejected values and determine whether both sets look correct. The use of whitespace was another situation where looking at both accepting and rejecting strings was helpful.

In addition, 59 regular expressions contained an ignored element from TABLE I. Out of these 59 regular expressions, 13 (22%) were considered buggy. As noted earlier, the tool generates substandard results when a regular expression contains an unsupported element so this lower percentage is not surprising.

B. Strings Generated

In this section, we explore the number of strings generated by the tool. For this section, we only consider the 707 fully-supported regular expressions; the 59 regular expressions that contain an ignored element and the 25 regular expressions that abort with an error message are not considered. Fig. 4 divides the regular expressions into histogram buckets based on how many test strings were generated.

Most regular expressions generated a relatively few number of strings. Over half (55%) of the regular expressions generated fewer than 20 strings, 80% generated fewer than 40 strings, and 96% generated fewer than 100 strings. Fourteen regular expressions generated more than 130 strings. The most strings generated by a regular expression was 307 strings.

The number of strings generated is loosely proportional to the number of tokens in the regular expression. The scatterplot in Fig. 5 shows this relationship. As you can see, many regular expressions have a small number of tokens and subsequently a small number of generated strings. A higher number of tokens results in a higher number of paths but the types of regular expression elements also plays a role in the number of test strings. Fig. 5 also shows that the number of test strings is roughly linear with respect to the number of tokens. This shows that our approach scales to large regular expressions.

Fig. 4 also shows the number of buggy regular expressions for each histogram bucket. Not surprisingly, there were relatively fewer bugs for smaller regular expressions. Of the 202 regular expressions with 9 or fewer test strings, 24 (12%) were buggy. Over half (115 of 201 or 57%) of the regular expressions with at least 30 test strings contained a bug. This

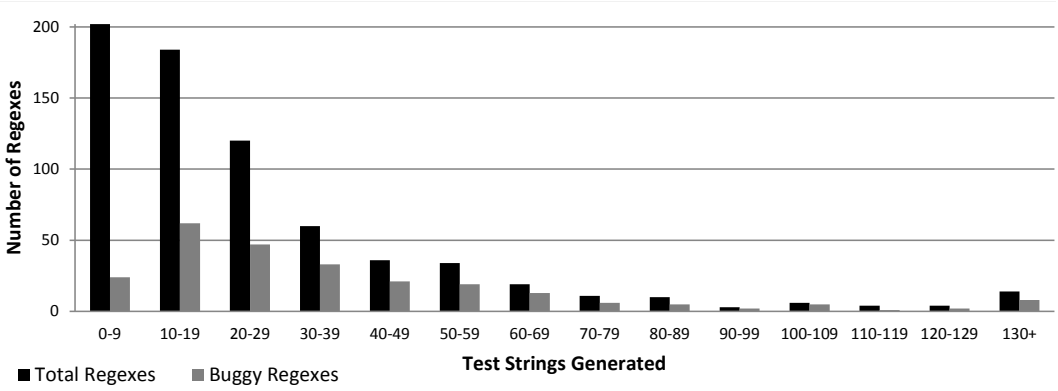


Fig. 4. Number of Test Strings Generated

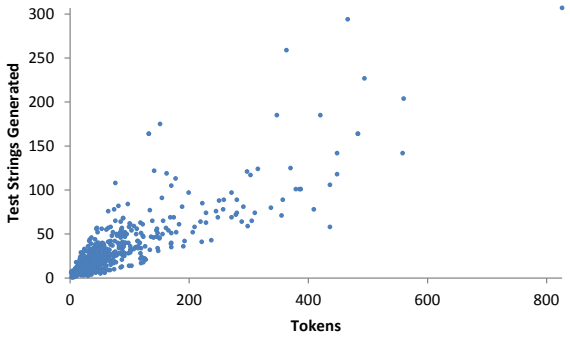


Fig. 5. Tokens vs. Test Strings Generated

indicates that writing complex regular expressions correctly is hard, thereby motivating the need for a tool like EGRET. These figures show that, while having a moderate number of test strings is helpful, it is not necessary to have an excessive number of test strings in order to find bugs.

C. Performance

Performance was measured for each regular expression by calculating the time it took EGRET to run in command line mode on a Linux machine with a 3.4 GHz Intel I7-4770 Quad Core processor with 8 GB of memory. In each case, EGRET took less than a second. The most complex regular expression in our study (the point in the upper-right corner of Fig. 5) consists of 826 tokens. This regular expression generated the most strings (307) and was the slowest (0.3 seconds).

Another aspect of performance is the time it takes to manually process the list of the regular expressions and look for items that are incorrect. During our evaluation, we manually looked at generated tests for each regular expression and informally noted how long it took to analyze the two lists. Our biggest challenge was figuring out what the regular expression was precisely trying to accomplish. For most buggy regular expressions, strings that should not be accepted stuck out and were easy to identify. Even with regular expressions that had over 100 generated test strings, it was generally straightforward to analyze the lists. A more formal usability study with likely users is left as future work.

D. Comparison to EXREX

EGRET was compared to another regular expression string generation tool EXREX [16]. The comparison looks at bug finding capabilities and number of strings generated.

EXREX converts the regular expression into an NFA like EGRET but with two key differences. First, repeat quantifiers follow a normal NFA construction that creates a loop in the NFA. The second key difference is that each edge is either an epsilon transition or an individual character. Character classes and sets will add one edge for each valid character. For wildcards, opposite character classes, and negated character sets, the initial set is the range of ASCII values from 32 (space) to 122 (lowercase 'z'). This includes all digits, uppercase letters, lowercase, and most punctuation marks.[4] To generate the strings, EXREX simply generates strings for each unique path. For loops formed by repeat quantifiers, the number of iterations is capped by a user-defined parameter. The default limit is 20 but we set the parameter to 2 during this evaluation for a more fair comparison.

The number of generated test strings is massive as seen in TABLE VII. For instance, a regular expression consisting of a single wildcard "`.`" generates 91 test strings, one for each of the ASCII characters from 32 to 122. The regular expression "`.+`" generates 8,372 strings (91 one-character strings + 91^2 two-character strings). Looking at TABLE VII, only 98 of the regular expressions generated fewer than 100 test strings. 193 regular expressions generated over one million test strings and many were several orders of magnitude greater than one million. One regular expression generated 6.33×10^{49} paths.[5]

The set of test strings generated by EXREX was infeasible to manually analyze. To gauge the bug finding capabilities, we created a test suite that contains simple regular expressions with one to two buggy regular expressions for each type of bug in TABLE V. This experiment found that EXREX is capable of finding all bug types except for *anchor usage* since anchors are essentially ignored. However, the large number of paths

[4] Unlike EGRET, it excludes {, |, }, and ~ which have ASCII values 123-126 but it would not be difficult to update EXREX to include these values.

[5] EXREX has a mode that will print out the number of strings instead of generating all of them.

TABLE VII. NUMBER OF STRINGS GENERATED BY EXREX

Number of Strings	Regexes
1-9	35
10-99	63
100-999	94
1,000-9,999	123
10,000-99,999	92
100,000-999,999	64
1,000,000+	193
ERROR	19

generated will make it difficult for users to identify incorrectly accepted strings especially for *negation* and *wildcard* bugs since the regular expressions associated with those bugs introduce a large number of transitions in the underlying NFA. Since EXREX does not generate any rejected strings, it could miss bugs that rely on rejected strings being accepted.

During our evaluation, we detected one instance where EXREX generated a test string that clearly indicated a bug that was not generated by EGRET. It was for the regular expression for floating point numbers: "`\d*\.\d*`". EXREX generated the incorrect string of "`.`". This was not generated for EGRET since it never generates a string where both `\d*` elements correspond to zero iterations. Due to the large number of strings generated by EXREX, it was infeasible to perform a systematic study to determine if other such cases existed.

V. RELATED WORK

There are many tools available for testing regular expressions. EXREX [16] generates all accepted test strings subject to a limit by repeat quantifiers, by traversing the equivalent NFA. EGRET is compared to EXREX in Section IV-D. Another tool for generating regular expression strings is at Uttool [5]. It randomly generates test strings for a regular expression. The user specifies the number of strings to generate. Other tools [2][12][13] are used to test regular expressions but the strings must be provided by the user. A nice feature of these tools is they display a visual representation of how the test string matches the regular expression. These tools complement EGRET in that when a problematic test string is generated, these tools can be used to debug the regular expression. RegExpert [4] provides a graphical representation of the corresponding automata for a regular expression.

Significant research attention has been placed on automatically generating tests, typically using symbolic execution approaches [14][17]. Reggae [9] describes an approach on how to incorporate regular expressions into the test generation process. Their goal is to obtain branch coverage so they are most concerned with quickly generating a string that is accepted by the regular expression. Rex [19] is a general-purpose solver of regular expressions constraints. It uses a symbolic finite automaton [18], an NFA with symbolic transitions that represent character sets and ranges, similar to the NFA used in EGRET. Instead of generating test strings, the symbolic finite automaton is used within a constraint solver to check feasibility. Performance and scalability are concerns with constraint solvers so the underlying automata are optimized and minimized. Aydin et al. [1] developed a technique to create test strings based on vulnerabilities. A vulnerability signature is represented using an automaton. The automaton is traversed to create test strings using different coverage criteria: state coverage, transition coverage, and path coverage. HAMPI [8] is a string constraint solver that allows users to specify constraints in a variety of ways including using regular expressions.

Additional research in regular expressions includes a type system designed by Spishak et al. [15]. The type system is used to catch regular expression syntax errors and cases where an invalid group number is used during extraction. Liu and Miao [10] describe how to create test cases from a regular expression and describe how to decompose the regular expression when it is too long. They use regular expressions derived from finite-state models as a form of model-based testing. However, their approach could be applied to normal regular expressions as well. Hodován et al. [6] analyze regular expressions that appear in JavaScript scripts on web pages. They found that many of the regular expressions are frequently used and showed that caching compiled regular expressions improves performance.

VI. CONCLUSION AND FUTURE WORK

This paper describes EGRET, a tool for generating test strings for a regular expression. Developers can manually scan the list of strings to determine if a string is incorrectly accepted or incorrectly rejected. Using EGRET, we found 284 out of 791 regular expressions to be buggy. EGRET improves upon existing work by generating a moderate amount of strings that are likely to focus on mistakes made by the developer. 307 strings were generated for the most complex regular expression; fewer than 100 test strings were generated for 96% of the regular expressions.

There are several avenues of future work. The most obvious place to start is to increase the level of support for regular expressions. Based on how frequently they occurred in the regular expressions used in the study, it is worth adding support for backreferences, lookahead and lookbehind assertions, and flags. We also would like to support regular expressions from other languages beyond Python. Second, we would like to better explore some of the tradeoffs EGRET makes in terms of keeping the number of generated strings low. We already noted one missed bug; a more systematic study is needed. Lastly, EGRET could be expanded to detect and/or warn about certain types of errors instead of relying on a test string that exposes the error. For instance, the character set "`[A|B|C]`" is likely incorrectly constructed because the `|` appears twice. However, error checking of this sort would need to be carefully constructed to minimize false alarms.

In addition, more improvements to the user interface are planned. One modification includes the ability to save test strings since the faulty string may not be generated again after a regular expression is rewritten during debugging. A usability study with likely users is also needed.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] A. Aydin, M. Alkhalaf, and T. Bultan, "Automated Test Generation from Vulnerability Signatures," in 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, 2014, pp. 193–202.

[2] 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, 2014, pp. 504–507.

[3] E. Bendersky, "Finite State Machines and Regular Expressions." [Online]. Available: http://www.gamedev.net/page/resources/_/technical/general-programming/finite-state-machines-and-regular-expressions-r3176.

[4] I. Budiselic, S. Srbljic, and M. Popovic, "RegExpert: A Tool for Visualization of Regular Expressions," in EUROCON 2007 - The International Conference on "Computer as a Tool," 2007, pp. 2387–2389.

[5] "Generate String from/Match Regular Expression Online - Uttool." [Online]. Available: http://uttool.com/text/regexstr/default.aspx. [Accessed: 20-Oct-15]

[6] R. Hodován, Z. Herczeg, and A. Kiss, "Regular Expressions on the Web," in 12th IEEE International Symposium on Web Systems Evolution (WSE), 2010, pp. 29–32.

[7] J. E. Hopcroft, R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., 2006.

[8] 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. 1–28, Nov. 2012.

[9] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Reggae: Automated Test Generation for Programs Using Complex Regular Expressions," in 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009, pp. 515–519.

[10] P. Liu and H. Miao, "A New Approach to Generating High Quality Test Cases," in 2010 19th IEEE Asian Test Symposium, 2010, pp. 71–76.

[11] "RegExLib.com." [Online]. Available: http://www.regxlib.com.

[12] "RegExr: Learn, Build, & Test RegEx." [Online]. Available: http://www.regexr.com.

[13] "Regular Expressions 101." [Online]. Available: https://regex101.com.

[14] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), 2005, pp. 263–272.

[15] 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, 2012, pp. 20–26.

[16] A. Tauber, "EXREX - regular expression string generator." [Online]. Available: https://github.com/asciimoo/exrex.

[17] N. Tillmann and J. de Halleux, "Pex–White Box Test Generation for .NET," Tests Proofs - Lect. Notes Comput. Sci., vol. 4966, pp. 134–153, 2008.

[18] M. Veanes, "Applications of Symbolic Finite Automata," Implement. Appl. Autom., vol. 7982, pp. 16–23, Jul. 2013.

[19] M. Veanes, P. de Halleux, and N. Tillmann, "Rex: Symbolic Regular Expression Explorer," in 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 498–507.

[20] A. H. Watson and T. J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," NIST Special Publication 500-235, Sept. 1996.