

# Automatic Checking of Regular Expressions

Eric Larson  
Seattle University  
Seattle, WA USA  
elarson@seattleu.edu

**Abstract**—Regular expressions are extensively used to process strings. The regular expression language is concise which makes it easy for developers to use but also makes it easy for developers to make mistakes. Since regular expressions are compiled at run-time, the regular expression compiler does not give any feedback on potential errors. This paper describes ACRE – Automatic Checking of Regular Expressions. ACRE takes a regular expression as input and performs 11 different checks on the regular expression. The checks are based on common mistakes. Among the checks are checks for incorrect use of character sets (enclosed by `[]`), wildcards (represented by `.`), and line anchors (`^` and `$`). ACRE has found errors in 283 out of 826 regular expressions. Each of the 11 checks found at least seven errors. The number of false reports is moderate: 46 of the regular expressions contained a false report. ACRE is simple to use: the user enters a regular expressions and presses the check button. Any violations are reported back to the user with the incorrect portion of the regular expression highlighted. For 9 of the 11 checks, an example accepted string is generated that further illustrates the error.

**Keywords**—*regular expressions; software bug detection; testing*

## I. INTRODUCTION

Regular expressions are a popular mechanism for processing strings that are in a particular format such as dates, email addresses, and phone numbers. Regular expressions are used in searching, validating data on a web page form, and data processing. Regular expressions are specified using a concise language with punctuation marks representing different operations.

Like most programming languages, the regular expression language has nuances that can lead to bugs. One problem is that very few regular expressions fail to compile. For instance, while unbalanced parentheses will cause a failure, unbalanced braces `[]` will not. Another problem is that some symbols have different meanings in different situations. The `^` symbol could refer to the “beginning of the string”, a negated character set, or a `^` symbol depending on where it is relative to other elements in the regular expression. These rules can be confusing and easy to forget during development.

It is important that regular expressions are correct. Faulty regular expressions could reject strings that are acceptable leading to incorrect behavior. Regular expressions could also accept strings that should be rejected. A program could also operate incorrectly or crash as a result of an incorrectly accepted string. An extreme case is a web form that uses a bad regular expression to validate data that could possibly be exploited to intentionally crash the server.

This paper describes ACRE (Automatic Checking of Regular Expressions), a tool that detects mistakes in regular

expressions. ACRE is simple to use, the user enters a regular expression and presses the check button. Any errors are reported back to the user. ACRE is similar to the warnings given by many high-level language compilers. Since regular expressions are typically compiled at run-time, it is not appropriate for these compilers to give warnings.

ACRE consists of a set of 11 checkers summarized in TABLE I. The checkers focus on common mistakes when developing regular expressions. When an error is reported, the portion of the regular expression that triggered the error is highlighted. Nine checkers also give an example string that is accepted and likely not intended to be accepted. Four checkers also give a suggested fix.

Four of the checkers analyze character sets. Character sets are surrounded by `[]` and contain a list of characters. The character set matches one character that is in the list. In addition, the character sets can contain ranges such as `A-Z` and character classes such as `\d` which both serve as a shorthand for common sets of characters. One problem with character sets is the meaning of several characters changes when they are inside a character versus outside a character set. In most cases, a punctuation mark inside a character set matches that punctuation mark. For instance, a `|` matches a vertical bar `|` inside a character set but is used for alternation outside the character set. Developers can forget this and sometimes use the `|` inside the character set to separate the cases. One of the checkers in ACRE looks for this case. A few punctuation marks have different meanings inside the character set. The hyphen `-` matches a hyphen outside the character set but is used to represent a range (in most cases) inside a character set. Another checker in ACRE looks at properly specified ranges.

Another source of problems is the wildcard which can match any character. Often some characters, especially punctuation marks used as delimiters or separators, should be excluded. One checker detects likely cases where the wildcard is too wild and doesn’t exclude characters. A related checker analyzes situations where punctuation marks are likely incorrectly repeated.

Other checkers look for mismatched and/or unbalanced parenthesis, braces, double quotes, and single quotes. One checker detects cases where the regular expression appears to be accepting numbers but ends up accepting strings that have no digits. Two checkers ensure appropriate usage of the `^` and `$` line anchors.

Our prior approach to detecting regular expression errors used test generation. EGRET [17] generated a set of test strings that are accepted and rejected by the regular expression. The user has to scan the list of strings to determine if an error occurred. ACRE improves this process by reporting checker

TABLE I. ACRE CHECKER SUMMARY

Checker	Applied To	Example	Fix
Bad Range	Character Set	No	Yes
Separator in Character Set	Character Set	Yes	Yes
Duplicate Character	Character Set	No	No
Lone Brace in Character Set	Character Set	Yes	No
Optional Brace	Path	Yes	No
Duplicate Punctuation Only Character Set	Path	Yes	No
Wildcard Next to Punctuation	Path	Yes	Yes
Repeat Punctuation	Path	Yes	No
Digit Too Optional	Path	Yes	No
Anchor in the Middle	Path	Yes	No
Consistent Anchor Usage	Set of Paths	Yes	Yes

violations directly, removing the requirement to scan the list of strings. The downside to ACRE is that the checkers are narrow and will miss bugs that are not specifically detected by the checker. With only 11 individual checkers, ACRE was able to catch 75% of the possible bugs that could be detected using test generation.

This paper makes the following research contributions:

- A description of 11 checkers that can be used to find regular expression bugs.
- An evaluation of the checkers. ACRE was able to find 283 bugs out of 826 regular expressions extracted from RegExLib.com [21] and six Python programs.
- ACRE is available for use, including the source code, at <http://fac-staff.seattleu.edu/elarson/web/regex.htm>.

The remainder of the paper is organized as follows. Section II gives an overview of the ACRE engine. Section III describes each of the 11 checkers in detail. Section IV describes limitations. Results are discussed in section V. Section VI outlines related work and section VII concludes.

## II. ACRE ENGINE

ACRE operates on Python regular expressions and uses a modified version of the EGRET engine [17]. The regular expression is read from the command line or web interface. Then the regular expression is compiled. If the compilation fails, ACRE ends with the compilation error message. Very few regular expressions have compiler errors. The most common cases for compiler errors are unbalanced parentheses, ranges that are out of order (such as Z-A), and incorrectly formed Python-specific extensions that have more complicated syntax.

The regular expression is parsed and converted into a specialized acyclic NFA using a modified version of a regular expression to NFA conversion tool developed by Bendersky [4]. A traditional approach [15] is used to convert concatenation and the matching of individual characters. However, there are several key differences in the specialized NFA compared to the traditional approach. First, character sets, which can match multiple characters, occupy a single edge instead of creating an edge for each individual character. The contents of the character set is retained for the checkers.

Character classes and the wildcard are considered to be character sets. Second, repeating quantifiers introduce a "begin repeat" edge immediately before the repeating subexpression and an "end repeat" edge immediately after. All forms of repeating quantifiers are expressed this way:  $*$ ,  $+$ ,  $?$ ,  $\{n\}$ ,  $\{m,n\}$ . The lower bound and upper bound (if one exists) are retained. A loop is not formed – the resulting NFA is a directed acyclic graph. Third, the begin anchor ( $^$ ), end anchor ( $$$ ), and backreferences (such as  $\backslash 1$ ) are considered to be special characters and also appear on edges like normal characters. Fourth, groups or parentheses are discarded as the NFA already demonstrates proper precedence. Lastly, ignored elements are converted to epsilon transitions. Word boundaries ( $\backslash b$ ), flags, lookahead assertions, lookbehind assertions, and conditional patterns are considered to be ignored elements. These ignored elements are not used by the checkers. All of these ignored elements are zero-width; they do not match or consume input characters. TABLE II summarizes how each regular expression element is processed in the conversion to the NFA.

Once the NFA is constructed, it is traversed to generate a set of paths such that each element in the regular expression is present in at least one path. A path consists of an ordered set of edges in the NFA.

The checker phase analyzes each path independently. TABLE I shows how the 11 checkers can be classified on how they are applied: four checkers are applied to individual character sets, six checkers are applied to individual paths, and one checker is applied to the set of paths. ACRE applies the four character set checkers to any character sets present in the path and then applies the six path checkers to the path itself. The one checker that is applied to the set of paths is executed at the end. The individual checkers are independent of one another and can be executed in any order. The 11 individual checkers are described in section III.

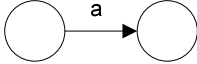
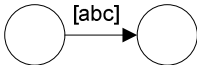
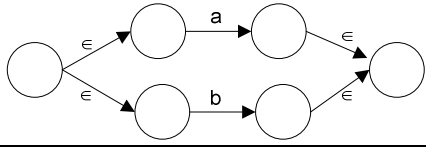
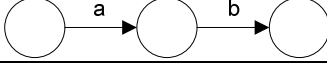
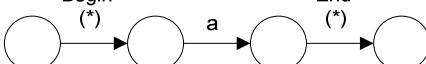
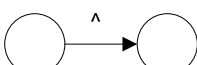

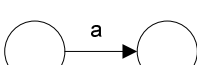
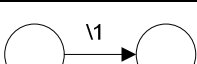
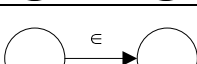
Paths often overlap. Character sets are only checked once – on the first path that has the character set. The character set checks are not applied if encountered on subsequent paths. The individual path checkers are applied to each path possibly leading to duplicate error messages. If the same error occurs in the same location, only one error message is reported. Multiple reports are given if the same error occurs in multiple locations or if different errors occur in the same location.

Any violations are reported back to the user. Most violations include an example string that should be accepted by the regular expression. In these cases, the example string is matched against the regular expression. If the example string is not accepted, the example string is suppressed but the error is still reported without the example string. Cases where this may occur are described in Section IV.

## III. ACRE CHECKERS

The 11 checkers were derived after analyzing the different types of bugs detected using test generation from EGRET [17]. Many of the bugs were common across different regular expressions and we hypothesized that checkers could automatically detect some of these common errors. We developed heuristics based on these errors but soon discovered

TABLE II. CONSTRUCTING THE NFA

Regex Element	Corresponding NFA
CHARACTER a	
CHARACTER SET [abc]	
ALTERNATION a b	
CONCATENTATION ab	
REPETITION a*	
BEGIN ANCHOR ^	
END ANCHOR \$	
GROUP (a)	 (group is discarded)
BACKREFERENCE \1	
IGNORED \b	 (ignored element is discarded)

that multiple checkers were needed for some checkers because there are many ways of writing regular expressions. After implementing a heuristic within a checker, it was evaluated by comparing the results to EGRET. Many checkers needed refinement to either improve bug detection capabilities and/or reduce false alarms.

This section describes each of the 11 checkers. Examples of the error reports are given in TABLE III. All of the examples are actual bugs detected by ACRE except for the *Anchor in the Middle Checker* (all regexes with this error are long and difficult to comprehend). Some of the other regular expressions were slightly edited for brevity.

#### A. Bad Range Checker

The bad range checker detects invalid ranges within a character set. One example of a bad range is A-z. This is interpreted as a range from ASCII character A to ASCII character z. While it includes all of the uppercase letters and all of the lowercase letters, the range includes six punctuation marks that are between the uppercase and lowercase letters in

the ASCII encoding. Another similar type of bad range is D-d. It includes all of the characters from D to d including the uppercase letters E-Z and the lowercase letters a-c.

Another bad range is +=. This often appears in a set of punctuation marks where the user wants to include the -. The compiler interprets this as a range, not as three separate punctuation marks.

The bad range checker works by analyzing each range and determining whether it is a good range or not. A range is considered good if it meets one of the criteria: (a) only lowercase letters, (b) only uppercase letters, (c) only digits, or (d) only non-displayable ASCII characters (0-31). The last case is commonly used in negated character sets where the user wants to exclude all non-displayable characters. Any range that is not considered good will be reported as a violation. The regular expression compiler catches bad ranges where the lower bound has a greater ASCII value than the upper bound such as F-A. Since ACRE immediately rejects regular expressions that do not compile, this checker does not consider those cases.

When the violation is reported, a suggested fix is provided. The fix depends on the type of bad range. For punctuation ranges, the - is moved to the end of the character set. Ranges such as D-d are replaced with Dd. The range A-z is replaced with A-Za-z. ACRE checks other ranges in the character set such that repetition is avoided.

#### B. Separator in Character Set Checker

This checker detects cases where the user likely incorrectly used a | in a character set as alternation. The | in a character set matches the | character. This checker also reports violation when a , is also likely to be used a separator of items.

The checker reports a violation if one or more of the following cases are true (the last two cases only apply to character sets that are not negated):

- Separator (| or ,) appears multiple times in the character set: [a,b,c]
- Separator (| or ,) is the only punctuation mark and it appears in the middle of the character set: [dog|cat]
- Character set consists of three items and the second (middle) item is a separator (| or ,): [a|b]

When ACRE reports a violation, an example string and suggested fix are displayed to the user. In the example string, the separator (| or ,) is intentionally chosen as the character that matches the character set in question. In most cases, the fix is simply the character set with the separators removed: [A,B,C] becomes [ABC]. However, cases such as 0|9 or 0,9 will be corrected to 0-9. In addition, if the separator is a | and the character set does not contain any ranges, the fix will replace character set braces with parentheses creating an alternation as demonstrated in the example in TABLE III. Note that the fix is not perfect because the {2,2} would have to be removed but sufficiently demonstrates what the problem is.

TABLE III. EXAMPLE CHECKER REPORTS

<i>Bad Range Checker</i>	<code>^[0-9]{4}\s{0,2}[a-zA-Z]{2}\$</code> VIOLATION (bad range): The fragment A-z is interpreted as a range ...Regex: <code>^[0-9]{4}\s{0,2}[a-zA-Z]{2}\$</code> ...Suggested fix: <code>[a-zA-Z]</code>
<i>Separator in Character Set Checker</i>	<code>(^[0-9] [0-1][0-9] [2][0-3]):([0-5][0-9])\s{0,1}([AM PM am pm]{2,2})\$</code> <i>time</i> VIOLATION (charset sep): Likely use of   in character set for alternation ...Regex: <code>(^[0-9] [0-1][0-9] [2][0-3]):([0-5][0-9])\s{0,1}([AM PM am pm]{2,2})\$</code> ...Suggested fix: <code>(AM PM am pm)</code> ...Example accepted string: <code>0:00   </code>
<i>Duplicate Character Checker</i>	<code>^((([1-9]) (0[1-9]) (1[0-2]))\)/((0[1-9]) ([1-31]))\)/((\d{2}) (\d{4}))\$</code> <i>date</i> VIOLATION (duplicate char): Duplicate characters in character set: 1 ...Regex: <code>^((([1-9]) (0[1-9]) (1[0-2]))\)/((0[1-9]) ([1-31]))\)/((\d{2}) (\d{4}))\$</code>
<i>Lone Brace in Character Set Checker</i>	<code>^[{\(\)}[0-9a-fA-F]{8}[-]?([0-9a-fA-F]{4}[-]?){3}[0-9a-fA-F]{12}[\)]?}\$</code> VIOLATION (lone brace): Found ( in charset but not ), could lead to unbalanced ( ...Regex: <code>^[{\(\)}[0-9a-fA-F]{8}[-]?([0-9a-fA-F]{4}[-]?){3}[0-9a-fA-F]{12}[\)]?}\$</code> ...Example accepted string: <code>{aaaaaaaa-aaaa-aaaa-aaaaaaaaaaaa}</code> VIOLATION (lone brace): Found ) in charset but not (, could lead to unbalanced ( ...Regex: <code>^[{\(\)}[0-9a-fA-F]{8}[-]?([0-9a-fA-F]{4}[-]?){3}[0-9a-fA-F]{12}[\)]?}\$</code> ...Example accepted string: <code>{aaaaaaaa-aaaa-aaaa-aaaaaaaaaaaa}</code>
<i>Optional Brace Checker</i>	<code>^[0-9]([ -])?(\([0-9]{3}\) [0-9]{3})([ -])?([0-9]{3}([ -])?[0-9]{4})\$</code> <i>phone number</i> VIOLATION (optional brace): Optional ( and ) found - accepts strings that have one but not the other ...Regex: <code>^[0-9]([ -])?(\([0-9]{3}\) [0-9]{3})([ -])?([0-9]{3}([ -])?[0-9]{4})\$</code> ...Example accepted string: <code>0 (000 000 0000</code>
<i>Duplicate Punctuation Only Character Set Checker</i>	<code>href=[\"'\"]?(?:[&gt;] [\s] ["'] [''])+ [\"'\"]?</code> VIOLATION (duplicate punc charset): Duplicate character set of punctuation marks can lead to mismatched punctuation usage ...Regex: <code>href=[\"'\"]?(?:[&gt;] [\s] ["'] [''])+ [\"'\"]?</code> ...Example accepted string: <code>href="a'</code>
<i>Wildcard Next to Punctuation Checker</i>	<code>^.+@[^\.]*\.[a-z]{2,}\$</code> <i>email address</i> VIOLATION (wild punctuation): Wildcard may wish to exclude adjacent punctuation mark @ ...Regex: <code>^.+@[^\.]*\.[a-z]{2,}\$</code> ...Suggested fix: <code>[^@]</code> ...Example accepted string: <code>@aevil.aa</code>
<i>Repeat Punctuation Checker</i>	<code>(^\+?\-? *[0-9]+)([0-9 ]*)\$</code> <i>number</i> VIOLATION (repeat punctuation): Punctuation mark may be repeated two or more times: , ...Regex: <code>(^\+?\-? *[0-9]+)([0-9 ]*)\$</code> ...Example accepted string: <code>+ - 0,, ,0</code>
<i>Digit Too Optional Checker</i>	<code>^\d*\.?\d*\$</code> <i>floating point number</i> VIOLATION (digit too optional): Digit range allows for zero digits causing a string with no digits to be accepted ...Regex: <code>^\d*\.?\d*\$</code> ...Example accepted string: <i>empty string</i>
<i>Anchor in the Middle Checker</i>	<code>^\\$d\.\d\d\$</code> <i>money amount</i> VIOLATION (anchor middle): Generated string has \$ anchor in the middle: 0.00 ...Regex: <code>^\\$d\.\d\d\$</code>
<i>Consistent Anchor Usage Checker</i>	<code>^\d{5}-\d{4} \d{5}\$</code> <i>zip code</i> VIOLATION (anchor usage): Some but not all strings start with a ^ anchor ...String with ^ anchor: 00000-0000 ...String with no ^ anchor: 00000 ...Suggested fix: <code>^\d{5}-\d{4} \d{5}\$</code>

### C. Duplicate Character Checker

The duplicate character checker detects cases where a character is specified twice in a character set. This suggests a poorly written character set that is likely not meeting the intentions of the user.

A character is considered to be part of a character set if it is explicitly mentioned as a character or part of a range. Character classes such as `\d` or `\w` are ignored since sometimes users use multiple character classes that overlap such as `\d` and `\w` but the overlap doesn't affect the correctness.

The duplicate character checker does not trigger an error if the character set contains a bad range and/or a separator error as those checkers give more precise error warnings. When ACRE reports the violation, the duplicate characters are specified.

### D. Lone Brace in Character Set Checker

This checker is concerned with balancing "braces". The set of braces include parentheses `()`, curly braces `{ }`, and square braces `[ ]`. The purpose of this checker is to detect cases where strings with unbalanced braces are incorrectly accepted. A violation is reported if a character set contains the left brace but does not contain the corresponding right brace.

For instance, a violation is reported if a character set contains a `(` but does not contain a `)`. The reported violation includes an example string that is accepted where the `(` matches the character set and `)` is avoided from being selected whenever possible. It is often the case that there are two character sets, one containing `(` and one containing `)`. The example string will choose the `(` from the first character set and choose something other than `)` in the second character set<sup>1</sup>. In addition, a second error is reported on the second character set which will choose `)` for the second character set but something other than `(` from the first character set.

### E. Optional Brace Checker

This checker is also concerned with balancing braces (parentheses, curly braces, and square braces). This checker looks for braces that are optional such as `\( ? or \[ { 0, 1 }`. If a brace is optional, it can lead to accepting strings with unbalanced braces.

A violation is reported if a path contains an optional left or right brace. If the path contains both an optional left brace and the corresponding optional right brace, ACRE will highlight both parts of the regular expression. The generated example string will include the left brace but exclude the right brace demonstrating the imbalance.

A violation is also reported if the path only includes one optional brace. The example string is generated like the previous checker: The optional character is selected and the corresponding brace is not selected whenever possible.

---

<sup>1</sup>Internally, ACRE treats character sets with a single character, such as `[ ( ]`, as individual characters (not character sets). This means there are always at least two characters in a character set.

### F. Duplicate Punctuation Only Character Set Checker

This checker detects situations where identical character sets that contain only punctuation marks appear on a path. Punctuation marks are often used as delimiters of fields in text and typically the same delimiter should be used throughout the string. As an example, in some cases, the user has the choice of using quotation marks `"` or single quotes `'` to enclose a string. Either choice is acceptable as long as the same choice is made at the beginning and the end.

The checker works by looking at each character set in the path. If the character set allows something other than punctuation marks or spaces, the character set is ignored. If the character set contains only punctuation, the contents are recorded. If another character set is encountered along the path with the exact same contents, then a violation is signaled. Note that the identical character sets must be on the same path. It is not an error if the identical character sets appear in the regular expression but are not on the same path due to alternation. This is common and acceptable situation.

There are two exceptions where violations are not signaled even though they meet the above requirements. First, while space characters are acceptable, there has to be at least one punctuation mark – character sets such as `[ \t ]` are ignored. Second, the character sets `[-+]` or `[+-]` are also ignored. For these character sets the `+` and `-` refer to positive or negative signs and not being used as delimiters. This rule reduces false alarms for regular expressions that match numbers in scientific notation or a list of numbers that may be positive or negative.

The example string generated with the error chooses one punctuation mark for the first character set and a different punctuation mark for the second character set.

### G. Wildcard Next to Punctuation Checker

The regular expression wildcard `.` matches any character. While this is convenient to use, it is often the case where some characters should be excluded. This checker detects cases where a wildcard is adjacent to a punctuation mark. Since punctuation marks are often used as separators between different parts of the string, the user may not want the wildcard to match the punctuation mark.

The checker works by looking for wildcards on the path. If there is a wildcard, both the previous character before the wildcard and the next character after the wildcard is further analyzed. If the adjacent character is a punctuation mark, a violation is reported. The adjacent character only refers to explicit characters but those explicit characters could be nested within a repeat quantifier. If the item adjacent to the wildcard is a character set, wildcard, line anchor, or the beginning/ending of the regular expression, no violation is reported.

The checker also analyzes negated character sets using `^`. If a negated character set is adjacent to a punctuation mark (using the same rules described above) and the punctuation mark is not excluded from the character set, a violation is reported.

The violation report includes an example string where the adjacent punctuation mark is used to match the wildcard (or negated character set) creating a string that has at least two

consecutive punctuation marks. In addition, a fix is shown. For wildcards, a negated character set with the punctuation mark is shown. For negated character sets, the punctuation mark is added to the set of disallowed characters.

#### H. Repeat Punctuation Checker

This checker also detects cases where punctuation may be incorrectly repeated. A violation occurs if a repeat quantifier that allows more than one but not a fixed limit such as `{2}` is applied to one of the following:

- punctuation character
- (non-negated) character set only containing punctuation marks
- (non-negated) character set containing an explicit period and/or explicit comma (explicit means it is intentionally specified by the user and not part of a range nor part of a character class such as `\D`)

The goal is to detect errors where consecutive punctuation is not permitted. As noted before, punctuation is often used as a separator and typically not used consecutively. Periods and commas are specifically called out because they are almost always used as separators. By allowing repetition, this could suggest a poorly written regular expression that accept numbers have consecutive commas or decimals points or email address with nothing but periods before and/or after the `@`.

The example string generated with this report shows a string where three consecutive punctuation marks match the repeat quantifier.

#### I. Digit Too Optional Checker

The digit too optional checker focuses on regular expressions that match some form of number. A common mistake is to incorrectly accept strings with no digits. This checker has two phases. In the first phase, it looks for cases where digits are optional. This includes cases: `\d?`, `\d*`, `\d{0,}`, and `\d{0,n}`. The character sets `[0-9]` or `[1-9]` can be used in place of `\d` in the above examples. If a path contains one of these elements, the checker moves into the second phase. In the second phase, an example string is generated in such a way that each optional repeat quantifier (such as `*` or `?`) is matched zero times. If the resulting string does not have any digits then an error is reported.

The violation includes the generated example string during the second phase of the checker. In many violations (including the example in TABLE III), the generated example string is the empty string.

#### J. Anchor in the Middle Checker

The anchor in the middle checker detects cases where a line anchor (`^` or `$`) appears in the middle of a regular expression such that characters could appear before and after the line anchor. This is a poorly written regular expression that leads to rejecting strings that should be accepted.

The checker traverses the path using two flags that are initially off: character flag and end anchor flag. When a character or character set edge is processed, the character flag

is set; the regular expression has matched at least one character at that point in the regular expression. When an end anchor `$` is encountered, the end anchor flag is set. An error occurs if a begin anchor `^` edge is encountered after the character flag is set. This signifies that a begin anchor is encountered after at least one character. It is also an error if a character or character set edge is encountered after the end anchor flag is set. This signifies that at least one character is part of the regular expression after the end anchor.

The error report highlights both the line anchor and either the character before (for `^`) or the character after (for `$`). An example string is shown but is actually not accepted by the regular expression due to the misplaced anchor.

#### K. Consistent Anchor Usage Checker

The consistent anchor usage looks at the entire set of paths. An error is reported if some of the paths start with a begin anchor `^` and some paths do not. Similarly, an error is reported if some paths end with an end anchor `$` and some paths do not. The most common situation where this occurs is when developers forget that the concatenation of line anchors has higher precedence than alternation. For instance, `^dog|cat$` matches strings that start with `dog` or end with `cat`. If the user only wanted to match the word `dog` or the word `cat`, then this is the correct regular expression: `^(dog|cat)$`

The violation report gives two accepted strings: one with an anchor and one without the anchor. It also gives a suggested fix where all line anchors are removed and the entire regular expression is wrapped within a parentheses surrounded by line anchors: `^(...)$`

## IV. LIMITATIONS

ACRE currently only supports regular expressions from Python. Python was chosen due to familiarity and that the regular expression syntax is very similar to Perl like many other regular expression implementations. More specifically, regular expressions in Python use the same metacharacters and character classes as Perl but have a different set of extensions with differing syntax. These extensions are largely ignored by ACRE and are not used by any of the checkers. As such, ACRE could be easily extended to other regular expression implementations, especially those that are similar to Perl.

ACRE supports all Python regular expression elements except Unicode characters beyond normal ASCII characters. The lack of support of Unicode characters was due to lack of support in the original EGRET engine that ACRE uses. We plan to add support for Unicode characters in the future. Currently, if a regular expression has such a Unicode character, ACRE will abort with an error message.

Word boundaries (`\b`), flags, lookahead assertions, lookbehind assertions, and conditional patterns are ignored during checking. Presence of these elements could lead to false alarms. False alarms can also occur because the checks are written to point out likely errors with the regular expression. Occasionally a check may report a violation that is not actually an error based on user intention.

TABLE IV. REGULAR EXPRESSIONS USED IN EVALUATION

		Num Regex	Eval	Compile Error	No Support
RegExLib Library Category	address / phone	105	99	5	1
	dates / time	135	125	10	0
	email	38	34	4	0
	markup / code	63	59	3	1
	misc	168	153	3	12
	numbers	107	102	2	3
	string	91	82	2	7
	uri	74	70	4	0
Python Program	advas (0.2.5)	16	16	0	0
	beautifulsoup4 (4.3.2)	7	7	0	0
	pychecker (0.8.19)	4	4	0	0
	pymetrics (0.8.1)	6	6	0	0
	tables (3.1.1)	8	8	0	0
	Trac (1.0.1)	63	61	0	2
	Library total	781	724	33	24
	Programs total	104	102	0	2
	<b>Total</b>	<b>885</b>	<b>826</b>	<b>33</b>	<b>26</b>

ACRE generates example strings for many of the checkers. These example strings are intended to be accepted by the regular expression to illustrate the error. There are three cases where the string is not accepted by the regular expression and subsequently suppressed (the error is still reported without the example string). The first case is when a regular expression contains one of the ignored elements listed above (such as a word boundary) since the test generation module also ignores these elements. A second case is when regular expression has a line anchor in the middle. When this occurs, the regular expression often rejects all strings. The last case is when the regular expression has a backreference and the error is in the group captured by the backreference.

## V. RESULTS

To evaluate the approach, we used regular expressions from the regular expression library RegExLib.com [21]. 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 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 regular expressions (the most common being Unicode categories such as `\p{Letter}`). A total of 781 regular expressions that are supported in Python. Of these 781 regular expressions, 33 did not successfully compile and were immediately rejected by ACRE. In addition, 24 contained a non-ASCII Unicode character not currently supported by ACRE. Evaluation was carried out using the 724 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. Two regular expressions contained non-ASCII Unicode characters resulting in 102 regular expressions. In total, 826 regular expressions were used during the evaluation.

### A. Bugs Found

The bug finding capability was evaluated by looking at the output generated by ACRE. Each violation is analyzed to determine whether the report referred to a bug or not.

The act of determining bugs is subjective. 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 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 V shows the number of bugs found across the different categories and programs: 283 of the 826 (34%) regular expressions contained a bug. Several regular expressions contained multiple bugs so the total number of bugs (355) is higher than the number of the buggy regular expressions (283). If a regular expression had multiple reports from the same checker, this was considered to be just one bug. In most cases, it was same bug replicated in multiple locations within the regular expression – the nature of the regular expression language does cause portions of the regular expression to be replicated. If a regular expression had bugs reported from different checkers, the bugs were considered to be separate bugs.

Each of the checker found at least seven bugs and four checkers found at least 40 bugs: *Wildcard Next to Punctuation Checker* (86), *Repeat Punctuation Checker* (66), *Separator in Character Set Checker* (52), and *Duplicate Punctuation Only Character Set Checker* (40). These results show the checkers are effective at finding errors in regular expressions.

The *Wildcard Next to Punctuation Checker* found several issues in the *markup / code* and *uri* categories that included errors involving duplicate `>>` in HTML code or quotes inside quotes such as `""`. The checker also found 24 errors in the Trac. Trac uses a `/` as delimiter and it is possible to match strings consecutive `//` in many cases.

The *Repeat Punctuation Checker* found several bugs that matched email addresses. Many of the bugs were due to accepting strings with several consecutive periods for the user name and/or email domain. The *Duplicate Punctuation Only Character Set Checker* found several issues with mismatched delimiters for regular expression that match phone numbers and dates. The *Digit Too Optional Checker*, not surprisingly, found 19 bugs in regular expressions that tried to match numbers but also accept strings with no digits.

The checkers were less effective at finding bugs in regular expressions in Python programs. Only four of the eleven checkers found bugs in Python programs with the *Wildcard Next to Punctuation Checker* accounting for most of the bugs. We consider the *Separator in Character Set Checker*,

TABLE V. BUGS FOUND

Category / Program	Regexes	Buggy	Bad Range	Sep   or ,	Dup Char	Lone Brace	Opt Brace	Dup Punc Sets	Wild Card	Repeat Punc	Digit Too Opt	Anchor Middle	Anchor Usage
address / phone	99	35	3	7	1	5	15	8	2	7	2	1	5
dates / time	125	44	0	24	2	1	0	21	0	2	0	6	1
email	34	19	0	0	0	0	2	1	2	15	1	0	0
markup / code	59	25	0	1	1	1	0	3	19	4	0	0	1
misc	153	39	2	7	1	1	5	3	12	6	4	0	8
numbers	102	23	0	1	0	0	1	0	0	2	19	0	2
string	82	25	2	11	2	1	0	2	4	3	0	0	5
uri	70	34	2	1	1	1	1	0	15	21	0	0	1
advas	16	0	0	0	0	0	0	0	0	0	0	0	0
beautifulsoup	7	3	0	0	0	0	0	1	2	1	0	0	0
pychecker	4	3	0	0	0	0	0	1	2	1	0	0	0
pymetrics	6	4	0	0	0	0	0	0	4	0	0	0	0
tables	8	0	0	0	0	0	0	0	0	0	0	0	0
Trac	61	29	3	0	0	1	0	0	24	4	0	0	0
Regex Library	724	244	9	52	8	10	24	38	54	60	26	7	23
Programs	102	39	3	0	0	1	0	2	32	6	0	0	0
<b>Total</b>	<b>826</b>	<b>283</b>	<b>12</b>	<b>52</b>	<b>8</b>	<b>11</b>	<b>24</b>	<b>40</b>	<b>86</b>	<b>66</b>	<b>26</b>	<b>7</b>	<b>23</b>

*Duplicate Character Checker*, *Anchor in the Middle Checker*, and *Consistent Anchor Usage Checker* to target very obvious mistakes (while the other checkers catch somewhat more subtle errors). None of these four checkers found any errors in the Python programs. We conclude that ACRE is less useful, but still has value, in regular expressions that appear in released code that has been tested. We feel that ACRE is more useful when a developer is creating the regular expression and the overall number of bugs found reflect that effectiveness for that particular use case.

### B. False Alarms

All 11 checkers are designed to highlight possible bugs but false alarms are possible where a violation does not actually refer to a bug. TABLE VI shows the number of false alarms by program and by checker. 46 of the 826 regular expressions (6%) had at one least false alarm. Of those 46 regular expressions, 13 also had bugs, leaving 33 regular expressions (4%) with only false alarms.

Regular expressions with multiple false alarms are counted in the same manner as bugs. Five regular expressions contained false alarms from two different checkers culminating in 51 different false alarms over 46 regular expressions.

The *Wildcard Next to Punctuation Checker* with 20 false alarms and the *Repeat Punctuation Checker* with 13 false alarms account for most of the false alarms. However, these checkers also detected the most bugs so it is likely an acceptable tradeoff. Five other checkers had two to six false alarms while four checkers had no false alarms.

There were some common false alarm cases. For the *Wildcard Next to Punctuation Checker*, there were three common cases: a) `\\.` is used to represent an escaped character. The escaped character can certainly be used to represent a backslash as `\\` is a legal and commonly-used escaped character. b) File systems allow consecutive `/` within a file path. c) Source code comments can start with the character used to designate the comment. For instance, `/**` is the start of a valid C comment. A common false alarm in the *Repeat*

TABLE VI. FALSE ALARMS

Category / Program	False Alarms	Checker	False Alarms
address / phone	1	bad range	0
dates / time	1	separator in char set	0
email	4	duplicate char	2
markup / code	10	lone brace	2
misc	7	optional brace	0
numbers	4	duplicate punc char set	2
string	3	wildcard next to punc	20
uri	6	repeat punctuation	13
advas	0	digit too optional	6
beautifulsoup4	2	anchor in the middle	0
pychecker	1	consistent anchor usage	6
pymetrics	1	<b>Total</b>	<b>51</b>
tables	1		
Trac	5		
Library total	36		
Programs total	10		
<b>Total</b>	<b>46</b>		

*Punctuation Checker* is regular expression that matches (or a portion matches) alphanumeric strings with periods and based on the description, having multiple periods is acceptable. In all six *Digit Too Optional* false alarm cases, ACRE properly indicated that an empty string was accepted but the empty string was explicitly allowed based on the description that accompanied the regular expression.

One concern in bug detection systems that can produce false alarms is that it could require significant time to manually analyze the results. ACRE's output is designed to allow the user to quickly determine whether a violation is actually a bug. The entire regular expression is displayed with the faulty part highlighted. In most cases, it only took us a few seconds to determine whether a report was a false alarm or not analyzing the results. In the few cases that took longer, we had to take time to determine the intent of the user. An actual user would have a better idea of what the regular expression is precisely trying to match.



### C. Comparison to Test Generation

Our prior work found bugs in regular expressions using test generation with a tool called EGRET [17]. EGRET generates strings from the regular expression that are both accepted and rejected from the strings focusing on common mistakes from users. Users have to scan the list of strings and determine whether there is a string that should be accepted or not.

We compare our approach to a theoretical version of EGRET that includes the example strings generated from ACRE in addition to any strings generated from EGRET. Nine of the eleven checkers generate an example string. While ACRE does not generate a test string for bad ranges<sup>2</sup>, a test string could be created for a bad range that selects an unintended character to represent the character set with the bad range. To summarize, this theoretical approach to test generation should be able to capture all of the bugs that ten of the eleven checkers encountered. This accounts for 275 of the 283 bugs detected. Two issues could lower this number: a) Implementation issues such as handling trickier constructs such as lookahead assertions and backreferences are difficult to implement in a test generation environment; it is easier for the checkers in ACRE to ignore these constructs. b) Humans are required to look at the resulting list of generated strings and notice there is an error. It is difficult to quantify either of these.

The *Duplicate Character Checker* is the one checker that is not easily replicated in a test generation system. The checker found eight bugs. Those regular expressions were run in EGRET to determine if a bug could be found and we discovered that four of the eight cases resulted in a generated test string that demonstrates the error. Consequently, four bugs detected by the checker were unable to be detected by EGRET.

Test generation systems will always have the ability to find more bugs because the user that is analyzing the test strings knows that the string should accept or not accept. This knowledge is lost in the checkers which are narrow, focused on a particular problem. For instance, assume a regular expression is trying to match dates. If it accepts strings that don't look like a date, this is going to be readily apparent when test strings are generated. The error will be missed by ACRE unless it specifically contains one of the errors that ACRE is looking for.

The same regular expressions that ACRE used were run in EGRET and were manually analyzed. EGRET found 94 buggy regular expressions that were missed by ACRE. Looking at these 94 regular expressions, we find that most of them require specific knowledge of what types of strings the regular expression accepts. Additional checkers could be created for a few of these regular expressions but the errors are less common than those found by the current checkers.

To summarize, ACRE and EGRET combined were able to detect 377 buggy regular expressions. ACRE alone was able to find 283 of 377 bugs. This means that ACRE is able to quickly find 75% of the bugs using a set of 11 checkers. Using ACRE is actually easier for the user since it doesn't required looking

at several test strings and provides specific information about the error to aid in debugging.

## VI. RELATED WORK

ACRE is inspired by a variety of lint tools including LCLint [7], Splint [8], PCLint [10], DLint [11], and Android Lint [12]. Like ACRE, these lint tools employ a variety of checks, most of them are relatively simple, that detect likely programming mistakes. CodeSonar [13] implements a number of source code checks and allows user to implement their own custom checkers. Compilers such as gcc [9] incorporate checks as warnings. To the best of our knowledge, ACRE is the first tool that applies these type of checks to regular expressions.

There are several tools that can be used to test regular expressions including RegViz [3], RegExr [20], and Regular Expressions 101 [22]. These tools allow the user to enter a regular expression and a test string. The tool will determine where the test string is accepted or rejected by the regular expression. These tools contain a graphical interface that provides a visual representation of how the test string matches the regular expression. RegExpert [5] includes a graphical representation of the automata corresponding to the regular expression.

Test generation is a popular approach to finding bugs in regular expressions. EGRET [17] generates evil test strings given a regular expression. The evil test strings are generated based on common programming mistakes and kept to a moderate number so the user can easily scan the list of strings. EXREX [24] also generates test strings. It generates all accepted test strings limiting unbounded repeat quantifiers (such as  $*$ ) to a fixed number of iterations. Reggae [18] can generate create tests for functions that include regular expressions. Liu and Miao [19] describe how to create test cases from a regular expression and decompose the regular expression when it is too long.

Research groups have shown that writing and comprehending regular expressions is difficult. Chapman et al. [6] performed a study on functionally equivalent but syntactically different regular expressions. They found that certain elements were easier to comprehend than others and created a set of guidelines for writing easy to comprehend guidelines. ACRE could be expanded to ensure that any stylistic guidelines are met or possibly rewrite the regular expression using a set of guidelines. Hollmann and Hanenberg [14] found that users were more able to quickly understand a regular expression in a visual representation than the standard textual representation. Bartoli et al. [1] developed a system that automatically creates a regular expression based on examples. Then, (Bartoli et al. [2]) a study was performed comparing the performance of the system with human developers. They found the system to have similar performance in both time and accuracy with an experienced developer. Both the system and the developers made mistakes further illustrating the need for a tool such as ACRE.

Other relevant work on regular expressions include a type system created by Spishak et al. [23] for programs that use regular expressions to capture regular expression syntax errors and invalid group numbers during extraction. Lämmel and

---

<sup>2</sup>We feel that a test string is not necessary for the bad range checker, the error report and fix are sufficient.

Zaytsev [16] use checks similar to ACRE to detect and correct errors in grammars while extracting a grammar from a text-based source such as a web page. Grammars, like regular expressions, use parentheses and vertical bars as metacharacters and occasionally are specified incorrectly. Rex [25] is a general-purpose solver of regular expressions constraints for use in program analysis and bug detection tools.

## VII. CONCLUSION AND FUTURE WORK

This paper describes the ACRE tool that automatically checks regular expressions. ACRE consists of a set of 11 checkers that focus on common mistakes made by developers. ACRE is easy to use – users simply enter a regular expression and press the check button. Out of 826 regular expressions, ACRE found 283 buggy regular expressions and 46 contained false alarms.

In the future, additional checkers could be added to ACRE. One specific example is a checker that looks at alternation and cases where one or more of the choices is unnecessary such as `-|\.|\s`. In this case, the wildcard (which probably should have been an escaped period) makes the other choices unnecessary indicating a poorly-written regular expression that is likely incorrect.

Another direction of future work is to refine the checkers within ACRE to reduce false alarms. Can ACRE recognize cases where an alphanumeric string is being used and eliminate false alarms within the *Repeat Punctuation Checker* while still finding bugs? We would like more checkers to provide suggested fixes. For the *Optional Brace Checker*, a regular expression could be suggested that requires both or none of the braces.

## ACKNOWLEDGMENT

The author would like to thank the anonymous reviewers for their valuable comments. The author would also like to thank Tyler Hartje for his contributions to the ACRE engine and initial versions for two of the checkers used in this project.

## REFERENCES

- [1] A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet, and E. Sorio, "Automatic Synthesis of Regular Expressions from Examples," *IEEE Comput.*, vol. 47, no. 12, pp. 72–80, Dec. 2014.
- [2] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Can a Machine Replace Humans in Building Regular Expressions? A Case Study," *IEEE Intell. Syst.*, vol. 31, no. 6, pp. 15–21, Nov. 2016.
- [3] 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.
- [4] E. Bendersky, "Finite State Machines and Regular Expressions." [Online]. Available: <https://www.gamedev.net/articles/programming/general-and-gameplay-programming/finite-state-machines-and-regular-expressions-r3176>. [Accessed: 14-Jun-2018].
- [5] I. Budiselic, S. Sribljic, 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.
- [6] C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 405–416.
- [7] D. Evans, J. Guttag, J. Horning, Y. M. Tan, D. Evans, J. Guttag, J. Horning, and Y. M. Tan, "LCLint: a tool for using specifications to check code," *ACM SIGSOFT Softw. Eng. Notes*, vol. 19, no. 5, pp. 87–96, Dec. 1994.
- [8] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, 2002.
- [9] Free Software Foundation, "GCC, the GNU Compiler Collection." [Online]. Available: <https://gcc.gnu.org/>. [Accessed: 14-Jun-2018].
- [10] Gimple Software, "PC-lint." [Online]. Available: <http://www.gimpel.com/html/pcl.htm>. [Accessed: 14-Jun-2018].
- [11] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "DLint: dynamically checking bad coding practices in JavaScript," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, 2015, pp. 94–105.
- [12] Google Developers, "Improve your code with lint checks." [Online]. Available: <https://developer.android.com/studio/write/lint>. [Accessed: 14-Jun-2018].
- [13] GrammaTech, "GrammaTech CodeSonar." [Online]. Available: <https://www.grammotech.com/products/codesonar>. [Accessed: 14-Jun-2018].
- [14] N. Hollmann and S. Hanenberg, "An Empirical Study on the Readability of Regular Expressions: Textual Versus Graphical," in *2017 IEEE Working Conference on Software Visualization (VISST)*, 2017, pp. 74–84.
- [15] 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.
- [16] R. Lämmel and V. Zaytsev, "Recovering grammar relationships for the Java Language Specification," *Softw. Qual. J.*, vol. 19, no. 2, pp. 333–378, Jun. 2011.
- [17] E. Larson and A. Kirk, "Generating Evil Test Strings for Regular Expressions," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 309–319.
- [18] 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.
- [19] 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.
- [20] "RegExr: Learn, Build, & Test RegEx." [Online]. Available: <http://www.regexr.com>. [Accessed: 14-Jun-2018].
- [21] "RegExLib.com." [Online]. Available: <http://www.regexlib.com>. [Accessed: 14-Jun-2018].
- [22] "Regular Expressions 101." [Online]. Available: <https://regex101.com>. [Accessed: 14-Jun-2018].
- [23] 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.
- [24] A. Tauber, "EXREX - regular expression string generator." [Online]. Available: <https://github.com/asciimoo/exrex>. [Accessed: 14-Jun-2018].
- [25] 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.