MutRex: a mutation-based generator of fault detecting strings for regular expressions

Paolo Arcaini
Charles University
Faculty of Mathematics and Physics, Czech Republic
Email: arcaini@d3s.mff.cuni.cz

Angelo Gargantini
DIGIP
University of Bergamo, Italy
Email: angelo.gargantini@unibg.it

Elvinia Riccobene
Department of Computer Science
University of Milan, Italy
Email: elvinia.riccobene@unimi.it

***Abstract*—Regular expressions (regexes) permit to describe set of strings using a pattern-based syntax. Writing a correct regex that exactly captures the desired set of strings is difficult, also because a regex is seldom syntactically incorrect, and so it is rare to detect faults at parse time. We propose a fault-based approach for generating tests for regexes. We identify fault classes representing possible mistakes a user can make when writing a regex, and we introduce the notion of *distinguishing string*, i.e., a string that is able to witness a fault. Then, we provide a tool, based on the automata representation of regexes, for generating distinguishing strings exposing the faults introduced in mutated versions of a regex under test. The basic generation process is improved by two techniques, namely *monitoring* and *collecting*. Experiments show that the approach produces compact test suites having a guaranteed fault detection capability, differently from other test generation approaches.**

I. Introduction

Regular expressions (regexes) provide a synthetic and abstract way to precisely identify a set of strings (i.e., a *language*) using a pattern-based syntax. By using a regex, programmers can specify whether a given string belongs or not to a desired set of strings. They are commonly used to validate data (like in web forms), or to match and extract sub-strings in possibly long texts. Regexes are used in different contexts, as, for example, DNA sequencing alignment [5], intrusion detection in networks [1], or prevention of MySQL injection [24].

However, writing a correct regex is a difficult task. Regexes are often hard to understand, also because their syntax is rather compact and tolerant. Moreover, the regex syntax varies across different programming languages. Several studies show that bugs in regexes occur quite often [14], [20]. Only a limited form of syntax checking can be performed and most regexes are free of syntax errors.

To ease the design of regexes, several techniques and tools have been developed. Some tools try to detect errors by static checking [20] and others by visual debugging [6]. Erwig and Gopinath propose a range of independent, complementary representations that can serve as explanations of regular expressions [9]. Most approaches and tools are still based on generation of strings used to validate regexes [14], [16], [17], [21], [22].

A common assumption is that while writing a regex can be very difficult, evaluating if a given string belongs to the desired set of strings can be easily carried out by a human (provided that the set of strings to be assessed is of reasonable size). For this reason, two approaches are often used. The first one consists in providing a set of *labeled* strings (i.e., strings together with their evaluation) and then apply some learning algorithm in order to extract a regex [17]; the second one consists in generating a meaningful set of examples from a given regex r (i.e., words that are accepted and words that are not accepted by r) and show them to the designer in order to validate r [14].

The approach we here propose (and the operation of its supporting tool MutRex) belongs to the second stream of works, since it is based on the generation of a set of *meaningful* strings together with their evaluation, and, in order to validate the regex, the user is asked for inspecting all the strings to confirm or refuse the string evaluation.

However, the novelty refers to the process used for strings generation. Indeed, MutRex first produces some mutants from a regex r according to some fault classes representing common mistakes that are made when writing regexes; then, for each mutant m, it generates a string s able to *distinguish* m from r, i.e., a string that is evaluated differently in r and m (in terms of mutation testing, s *kills* m). The set of generated strings is a test suite able to detect all the seeded faults. The basic approach is improved by two techniques, monitoring and collecting, that permit to obtain more compact test suites.

Our approach advances with respect to the existing literature in these directions:

- w.r.t. regex *testing* (like [14], [16], [17], [21], etc.), we introduce several mutation operators and propose an original notion of *fault* coverage that can be used to measure the quality of the examples the tester uses to check if the regex captures exactly its intended meaning.
- w.r.t. automatic examples *generation* (like [14], [22]), we introduce a way to generate, as examples, only those strings that are able to identify (or *distinguish*) the regex under test from all its faulty mutations. This can significantly reduce the number of strings for which the user must assess their validity and/or increase the confidence of the regex correctness.

The paper is organized as follows. Sect. II introduces some background on regular expressions, and Sect. III presents the fault classes and the corresponding mutation operators we devised. Sect. IV introduces the process we propose to

Name	Derivation rule
union	intersection ["|" union]
intersection	concatenation ["&" intersection]
concatenation	repeatexp [concatenation]
repeatexp	repeatexp "?" (zero or one occurrence) | repeatexp "*" (zero or more occurrences) | repeatexp "+" (one or more occurrences) | repeatexp "{"n"}" (n occurrences) | repeatexp "{"n",}" (n or more occurrences) | repeatexp "{"n"," m"}" (n to m occurrences) | complement
complement	"~" complement | charclassexp
charclassexp	"[" [charclass]+ "]" (character class) |"[^" [charclass]+ "]" (negated char class) | simpleexp
charclass	char "-" char (character range)
simpleexp	char "." (any single character) "#" (the empty language) "@" (any string)
special chars	\w (word character) \d (digit) \s (whitespace character)
not supported	^, $, \A, \Z (anchors) \b, \B (word boundaries) (?<= ... (lookahead / lookbehind assertions)

Table I: BNF of the regexes supported by MUTREX

generate some strings that distinguish a regex from its mutants, and that must be inspected by the user to validate the regex. Sect. V presents the concept of mutation score for regexes and describes a way to compute it for a given test suite and a set of mutants. Sect. VI presents the experiments we performed, Sect. VII discusses some related work, and Sect. VIII concludes the paper.

II. BACKGROUND

Definition 1 (Regular expression)**.** A regular expression (*regex*) r is a sequence of *constant*s, defined over an alphabet Σ, and *operator* symbols. The regex characterizes a set of *words* $\mathcal{L}(r) \subseteq \Sigma^*$ (i.e., a *language*).

As Σ we support the Unicode alphabet (UTF-16); the supported grammar is shown in Table I. Note that our approach applies to regexes that can be represented as automata.

Definition 2 (Acceptance)**.** A string s is *accepted* by a regex r iff $s \in \mathcal{L}(r)$ (i.e., s is a word of $\mathcal{L}(r)$).

We will also use r as a predicate: $r(s) = \mathit{true}$ if $s \in \mathcal{L}(r)$, and $r(s) = \mathit{false}$ otherwise.

Normally, acceptance is computed by converting r to an automaton $\mathcal{R}$, and then checking whether $\mathcal{R}$ accepts the string. Fig. 1 reports an example of automaton accepting all the words of the regex [a-zA-Z_][a-zA-Z0-9_]*. In this paper, we use the library dk.brics.automaton [18] to transform a regex r into an automaton $\mathcal{R}$ (by means of function `toAutomaton`(r)) and perform standard operations on it, as those described as follows.

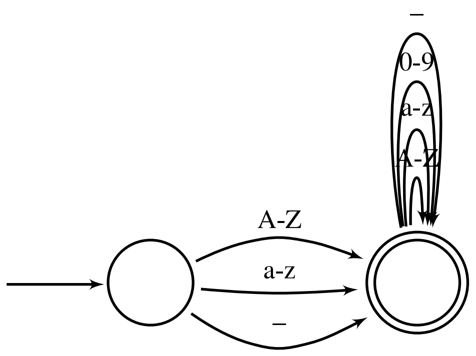


Figure 1: Automaton of regex [a-zA-Z_][a-zA-Z0-9_]*

Let $\mathcal{R}$ be the automaton of a regex r, and $\mathcal{L}(\mathcal{R})$ the language accepted by the automaton $\mathcal{R}$. Two unary operators on $\mathcal{R}$ are:

- *Complement*: $\mathcal{R}^{\complement}$ accepts all the strings not accepted by $\mathcal{R}$, i.e., $\mathcal{L}(\mathcal{R}^{\complement}) = \Sigma^* \setminus \mathcal{L}(\mathcal{R})$.
- *Word selection*: `pickAword`$(\mathcal{R})$ returns a string s accepted by $\mathcal{R}$, i.e., $s \in \mathcal{L}(\mathcal{R})$.

Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be the automata of regexes r_1 and r_2; three binary operators on $\mathcal{R}_1$ and $\mathcal{R}_2$ are:

- *Intersection*: $\mathcal{R}_1 \cap \mathcal{R}_2$ accepts the strings accepted by both automata, i.e., $\mathcal{L}(\mathcal{R}_1 \cap \mathcal{R}_2) = \mathcal{L}(\mathcal{R}_1) \cap \mathcal{L}(\mathcal{R}_2)$.
- *Union*: $\mathcal{R}_1 \cup \mathcal{R}_2$ accepts all the strings accepted by at least one of the two automata, i.e., $\mathcal{L}(\mathcal{R}_1 \cup \mathcal{R}_2) = \mathcal{L}(\mathcal{R}_1) \cup \mathcal{L}(\mathcal{R}_2)$.
- *Symmetric difference*: $\mathcal{R}_1 \oplus \mathcal{R}_2 = (\mathcal{R}_1^{\complement} \cap \mathcal{R}_2) \cup (\mathcal{R}_1 \cap \mathcal{R}_2^{\complement})$ accepts the strings accepted by only one of the two automata, i.e., $\mathcal{L}(\mathcal{R}_1 \oplus \mathcal{R}_2) = \mathcal{L}(\mathcal{R}_1) \setminus \mathcal{L}(\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_2) \setminus \mathcal{L}(\mathcal{R}_1)$.

We will also write $s \in \mathcal{R}$ for a string s accepted by the automaton $\mathcal{R}$ (i.e., $s \in \mathcal{L}(\mathcal{R})$).

III. FAULT CLASSES AND MUTATION OPERATORS

Mutation has been applied to various artifacts, mainly programs, but also specifications and grammars (see Sect. VII). In this paper, we apply mutation to regexes. As suggested by [23], to introduce a set of *mutation operators* producing useful mutations, we have tried to identify potential fault classes. We have browsed some Internet sites and read several books ([11] is the main reference) which explain the common mistakes programmers make when defining regexes, and we found the following fault classes and defined the following mutation operators. In our setting, a mutation operator is a function that given a regex r, returns a list of regexes (called *mutants*) obtained by mutating r. Every mutation *slightly* modifies the regex r under the assumption that the programmer has defined r close to the correct version (competent programmer hypothesis [12]).

We identified three families of faults: *single character faults* and *character class faults* are respectively related to wrong uses of single characters and character classes, *other faults* are instead related to wrong uses of the multiplicity and of the negation operator.

A. Single character faults

***Case Change** (CC):* Regexes are case sensitive. However, a user could use them without being aware of this, or (s)he could simply use the wrong case (either upper or lower). This operator mutates a regex r by changing the case of characters appearing in r: a mutant is created for each character of r not used in a character class, and a mutant is created for each character class (both chars are changed at the same time).

Example 3. A user could be interested in accepting all strings starting with 'a' or 'A'. However, (s)he wrongly writes regex `a[a-z]*` that only accepts words starting with 'a'. CC produces the mutant `A[a-z]*` that only accepts strings starting with 'A'. Note that the mutant is not yet what the user has in mind, but it allows to see that there is a problem in the original regex. The other mutant produced by CC is `a[A-Z]*`.

***Case Addition** (CA):* This operator is similar to CC, but makes both lower and upper cases possible when only one of the two is used: for each character of the regex r not used in a character class it creates a mutant with the other case of the char added as alternative; for each character class, instead, the mutant adds as alternative a new character class having the extremes of the interval in the other case.

Example 4. Considering the same example of Ex. 3, operator CA would mutate regex `a[a-z]*` in mutants `(a|A)[a-z]*` and `a[a-zA-Z]*`.

***Metachar To Char** (M2C):* In a regex, some characters can be interpreted as metachars or chars depending on the context, and there is no mandatory special way to identify metachars. For example, character '-' is interpreted as a metachar only in character classes, otherwise it matches the normal dash character. A user may want to use a char c, but (s)he could wrongly use c as metachar. This operator transforms a regex r containing a char c interpreted as a metachar in a regex r' in which c is interpreted as a char. Note that the way to mutate r depends on the metachar c: for example, M2C applied to '-' removes the character class where '-' is used.

Example 5. The user writes the regex `[0-9]{3}.[0-9]{3}` for matching a sequence of three digits, followed by a dot, followed by other three digits. However, in the regex, '.' is a metachar and strings like "123A456" are accepted. M2C produces the mutant `[0-9]{3}\.[0-9]{3}` that correctly expresses what the user had in mind.
Another example is when the user writes `[a-b]+` for matching strings as "a", "b", "-", "a-", "-b", ... However, the regex accepts only strings as "a", "b", "ab", "ba", ... M2C produces the mutant `(a|-|b)+`.

***Char To Metachar** (C2M):* In other cases, instead, the designer wants to use a metachar c, but, because of the context, c is interpreted as a simple char. This operator transforms a regex r containing a c interpreted as a char in a regex r' in which c is interpreted as a metachar. Similarly to M2C, the way to mutate r depends on the metachar c.

Example 6. The designer, in order to match strings of three chars, writes the regex `\.{3}`, where `\.` is a normal char and, therefore, only string "..." is accepted. C2M produces the mutant `.` that correctly accepts all three char strings.

B. Character class faults

***Character Class Creation** (CCC):* Character classes are delimited by square brackets []. A user may want to use a character class and forgets (or ignores) the parentheses. Given a regex $r = c_1 - c_2$, the operator mutates r in $r' = [c_1 - c_2]$.

Example 7. A user could have wrongly written `0-9+` to accept all the sequences of digits; the CCC operator produces the correct regex `[0-9]+`.

***Character Class Addition** (CCA):* The user could have forgotten a given interval in a set of character classes. Given a regex $r = [cc_1 \dots cc_n]$, the operator creates a mutant $r' = [cc_1 \dots cc_n cc_{new}]$ for each cc_{new} not present in r; cc_{new} can be `a-z`, `A-Z`, or `0-9`.

Example 8. Given the regex `[a-z]`, CCA creates mutants `[a-zA-Z]` and `[a-z0-9]`.

***Range Modification** (RM):* The user could have specified a too tight or too broad interval. Given a regex $r = [c_1 - c_2]$, the operator produces a mutant in which c_1 or c_2 is increased or decreased (if it is still a valid char).

Example 9. Given the regex `[f-m]`, RM creates mutants `[e-m]`, `[g-m]`, `[f-l]`, and `[f-n]`.

***Character Class Restriction** (CCR):* The user could have written a regex that is too permissive since it accepts characters that should not [17]. Given a regex $r = [cc_1 \dots cc_n]$, the operator creates a mutant $r' = [cc_1 \dots cc_{i-1} cc_{i+1} \dots cc_n]$ for each cc_i (i.e., it removes an interval from the character class).

Example 10. Given the regex `[a-zA-Z0-9]`, CCR creates mutants `[A-Z0-9]`, `[a-z0-9]`, and `[a-zA-Z]`.

***Prefix Addition** (PA):* Sometimes all the characters of a string s satisfy some constraints, except for the first character in s that must satisfy additional constraints; for example, identifiers in most programming languages cannot start with a number. This mutation operator, given a repeat expression regex $r = [cc_1, \dots, cc_n]$m (being m the multiplicity), introduces a prefix that is one of the different character classes $cc_1, \dots, cc_n$ used in r. The obtained mutants are as follows: $[cc_1][cc_1, \dots, cc_n]$m, $[cc_2][cc_1, \dots, cc_n]$m, …, $[cc_n][cc_1, \dots, cc_n]$m.

Example 11. The regex `[a-zA-Z0-9]*` accepts any character as first char, while if the correct regex accepted only an alphabetic char as first one, the PA mutation `[a-zA-Z][a-zA-Z0-9]*` is correct.

***Character Class Negation** (CCN):* Regex $[\hat{}cc_1 \dots cc_n]$ matches any character that is not listed in the character classes $cc_1, \dots, cc_n$. The designer could have forgotten the ^ symbol and written $[cc_1 \dots cc_n]$. CCN introduces symbol ^; it creates a mutant in which all the character classes are excluded (i.e,

[^$cc_1 \ldots cc_n$]), and a mutant for each character class cc_i in which only cc_i is excluded (i.e, [cc_1] | ... | [^cc_i] | ... | [cc_n]).

Example 12. The regex `[a-zA-Z]` is mutated in three alternative regexes, namely: `[^a-zA-Z]`, `[^a-z]|[A-Z]`, and `[a-z]|[^A-Z]`.

Negated Character Class to Optional *(NCCO):* Another common error regarding the use of a negated character class [^cc] is that it requires "to match a character that is not listed" and not "to not match what is listed" [11]. Therefore, a negated character class still requires to match a character. A user could misunderstand the semantics of the operator, thinking that it simply excludes the character; in some cases, for example at the end a word, (s)he could be interested in accepting also no character. Operator NCCO makes ^ optional, i.e., it mutates [^cc] in [^cc]?.

Example 13. Suppose that the designer wants to find all the words having a 'q' not followed by an 'u' [11]; (s)he may (wrongly) write `.*q[^u]` that, however, requires to always have a character different from 'u' after 'q': word "Iraq" would not be accepted. Applying NCCO would produce `.*q[^u]?` that also accepts "Iraq".

C. Other faults

Negation Addition *(NA):* In a regex $r = r_1 r_2 \ldots r_n$, the user could forget a negation ^. The operator creates a mutant adding a negation wherever possible in r, i.e., $\hat{}r_1 r_2 \ldots r_n$, $r_1 \hat{}r_2 r_n$, ..., $r_1 r_2 \ldots \hat{}r_n$.

Example 14. Given the regex `[A-Z][a-z]`, NA creates mutants `[^A-Z][a-z]` and `[A-Z][^a-z]`.

Quantifier Change *(QC):* The user could have used the wrong cardinality. The operator mutates each simple repeat quantifier in another simple quantifier; moreover, for each user-defined quantifier, it creates a mutant in which n (or m) is increased and a mutant in which it is decreased.

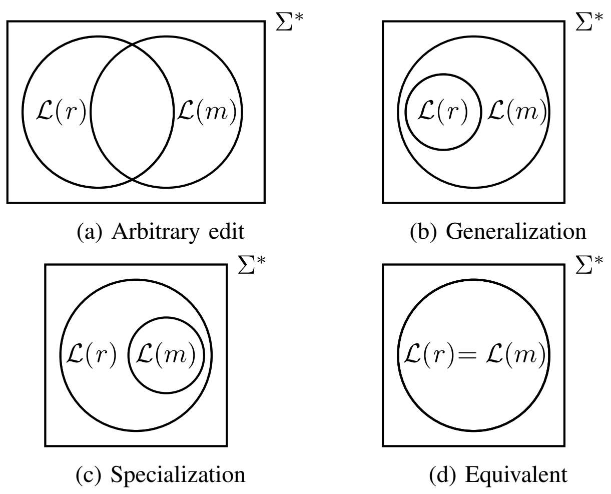
Example 15. A user could have wrongly written the regex `[0-9]*` to accept all the sequences of digits (that, however, also accepts the empty string); the QC operator would produce the correct regex `[0-9]+`.

D. Classification of mutations

A mutant m of a regex r can modify the accepted language in four ways, as shown in Fig. 2:

- *Arbitrary edit*: some words are added to the language and other words are removed.
- *Generalization*: some words are added to the language and no word is removed.
- *Specialization*: some words are removed from the language and no word is added.
- *Equivalent*: the mutant is equivalent and so the two languages are the same.

Example 16 (Classification examples). r' = `[a-z][A-Z]*` is a CC mutant of r = `[a-z][a-z]*`; it is an arbitrary edit: for



(a) Arbitrary edit (b) Generalization (c) Specialization (d) Equivalent

Figure 2: Classification of mutations

example, "aA" is accepted only by r', and "aa" is accepted only by r.

r' = `[a-z]*` is a QC mutant of r = `[a-z]+`; r' is a generalization of r because it accepts the same language plus the empty word "". Conversely, r is a specialization of r'.

r' = `[a-z][a-z]*` is a QC mutant of r = `[a-z][a-z]+`; r' is an equivalent mutant of r because it accepts the same language.

Some mutation operators guarantee to always produce mutants of the same type; for example, operator PA (adding a prefix to a regex) can only reduce the set of accepted strings and, therefore, it always produces specializations.

An alternative way to define mutation operators

In the previous section, we have introduced several mutation operators by *semantic* fault classes, i.e., mistakes done by programmers which misunderstood regex meaning and semantics. A simple alternative way to define mutation operators for regexes, would be ignoring fault classes and simply defining a unique syntactic variation by substituting a character c in a regex r with another character c'. However, this would lead to a great number of mutants (and, therefore, a great number of tests) and would require the use of higher order mutants to capture even simple faults, like, for example, the programmer forgets the character class "[" "]". As it will be apparent in the following sections, we want to keep the number of mutants small in order to limit the number of strings generated by MUTREX.

IV. GENERATION OF FAULT-DETECTING STRINGS

We suppose that a user has written a regex r and (s)he wants to validate it over some strings; the user may have wrongly written the regex, but (s)he can correctly assess whether a string should be accepted or not ((s)he can act as oracle).

We here aim at generating some *meaningful* strings that must be shown to the user for regex validation.

Algorithm 1 genDs: Generation of a distinguishing string

```
Require: $r_1$, $r_2$, two regexes
1: function genDs($r_1$, $r_2$)
2:     $\mathcal{R}_1 \leftarrow$ toAutomaton($r_1$)
3:     $\mathcal{R}_2 \leftarrow$ toAutomaton($r_2$)
4:     return genDs($\mathcal{R}_1$, $\mathcal{R}_2$)
5: end function

Require: $\mathcal{R}_1$, $\mathcal{R}_2$, two automata
1: function genDs($\mathcal{R}_1$, $\mathcal{R}_2$)
2:     $\mathcal{U} \leftarrow \mathcal{R}_1 \oplus \mathcal{R}_2$
3:     if $\mathcal{U} \neq \emptyset$ then
4:         return pickAword($\mathcal{U}$)
5:     else
6:         return null
7:     end if
8: end function
```

A. Distinguishing strings

Definition 17 (Distinguishing string)**.** Given two regexes r_1 and r_2, we say that the string s is *distinguishing* if it is a word of the symmetric difference between r_1 and r_2, i.e.,

$$s \in \mathcal{L}(r_1 \oplus r_2) = \mathcal{L}(r_1) \setminus \mathcal{L}(r_2) \cup \mathcal{L}(r_2) \setminus \mathcal{L}(r_1)$$

A distinguishing string is accepted by r_1 and not by r_2, or vice versa. We name as *positive* the distinguishing strings that are accepted by r_1, and as *negative* those accepted by r_2.

Alg. 1 shows an algorithm that produces a distinguishing string between two regexes r_1 and r_2. The function builds two automata $\mathcal{R}_1$ and $\mathcal{R}_2$ for the two regexes and then builds the automaton $\mathcal{U}$ corresponding to their symmetric difference. If $\mathcal{U}$ is not empty (i.e., the two regexes are not equivalent), the function randomly picks a word of $\mathcal{U}$ and returns it as distinguishing string; otherwise, it returns `null`.

B. Generation of distinguishing strings

We developed the tool MutRex[1] that, given a regex r, generates a set of mutants and asks the user to evaluate the distinguishing strings generated between r and its mutants.

The generation of the distinguishing strings is shown in Alg. 2. The approach builds a set of mutants *muts* using the mutation operators described in Sect. III, and then, for each mutant m, tries to build a distinguishing string for r and m (over their automata $\mathcal{R}$ and $\mathcal{M}$) using function `genDs` shown in Alg. 1: if m and r are not equivalent, a distinguished string exists and it is added to the set *DSs*. The set *DSs* is returned at the end of the algorithm.

The approach in Alg. 2 can be further improved by two techniques: monitoring and collecting.

1) Monitoring: Monitoring consists in keeping track of the previously generated distinguishing strings *DSs* and, after generating a mutant, checking whether the mutant is distinguished by any $ds \in DSs$; if this is the case, we avoid computing a new

[1]The tool is available as web service at http://cs.unibg.it/mutrex.

Algorithm 2 MutRex: generation of a distinguishing set

```
Require: $r$: initial regex
1: $\mathcal{R} \leftarrow$ toAutomaton($r$)
2: $muts \leftarrow$ mutate($r$)
3: $DSs \leftarrow \emptyset$
4: for $m \in muts$ do
5:     $\mathcal{M} \leftarrow$ toAutomaton($m$)
6:     $ds \leftarrow$ genDs($\mathcal{R}$, $\mathcal{M}$)
7:     if $ds \neq$ null then        ▷ $r$ and $m$ are not equivalent
8:         $DSs \leftarrow DSs \cup \{ds\}$
9:     end if
10: end for
11: return $DSs$
```

Algorithm 3 MutRex: generation of a distinguishing set using monitoring

```
Require: $r$: initial regex
1: $\mathcal{R} \leftarrow$ toAutomaton($r$)
2: $muts \leftarrow$ mutate($r$)
3: $DSs \leftarrow \emptyset$
4: for $m \in muts$ do
5:     $\mathcal{M} \leftarrow$ toAutomaton($m$)
6:     if $\neg\exists ds \in DSs$: $ds \in \mathcal{R} \oplus \mathcal{M}$ then        ▷ monitoring
7:         $ds \leftarrow$ genDs($\mathcal{R}$, $\mathcal{M}$)
8:         if $ds \neq$ null then
9:             $DSs \leftarrow DSs \cup \{ds\}$
10:        end if
11:    end if
12: end for
13: return $DSs$
```

distinguishing string and we continue from the next mutant. This permits us to limit the number of strings that are provided to the user. The modified algorithm is shown in Alg. 3. At line 6, the algorithm checks whether there exists an already generated distinguishing string that can also distinguish r and the new mutant m.

2) Collecting: Collecting consists in trying to generate a string that distinguishes r from a set of mutants $m_1, \ldots, m_n$. A string ds distinguishes r from a set of mutants $m_1, \ldots, m_n$ if ds is accepted by r and not accepted by any mutant in $m_1, \ldots, m_n$, or if ds is not accepted by r and accepted by all the mutants in $m_1, \ldots, m_n$. Therefore, the distinguishing string is a word of one of these two automata:

$$\mathcal{D}^+ = \mathcal{R} \cap \bigcap_{i=1}^{n} \mathcal{M}_i^{\complement}$$
$$\mathcal{D}^- = \mathcal{R}^{\complement} \cap \bigcap_{i=1}^{n} \mathcal{M}_i$$

being $\mathcal{R}$, $\mathcal{M}_1$, …, $\mathcal{M}_n$ the automata of r, m_1, …, m_n. We name $\mathcal{D}^+$ and $\mathcal{D}^-$ as *distinguishing automata*, and we further name $\mathcal{D}^+$ as *positive* and $\mathcal{D}^-$ as *negative*. The proposed approach iteratively builds positive and negative distinguishing automata by collecting different mutants. In order to check whether a distinguishing automaton $\mathcal{D}$ is positive or negative, we introduce the predicate $isPos(\mathcal{D})$ that is true if $\mathcal{D}$ is positive, false otherwise.

Algorithm 4 MUTREX: generation of a distinguishing set using collecting

```
Require: $r$: initial regex
 1: $muts \leftarrow \texttt{mutate}(r)$
 2: $\mathcal{R} \leftarrow \texttt{toAutomaton}(r)$
 3: $DA \leftarrow \emptyset$
 4: for $m \in muts$ do
 5:     $\mathcal{M} \leftarrow \texttt{toAutomaton}(m)$
 6:     $mAdded \leftarrow false$
 7:     for $\mathcal{D} \in DA$ do
 8:         if $isPos(\mathcal{D}) \wedge \mathcal{D} \cap \mathcal{M}^{\complement} \neq \emptyset$ then
 9:             $\mathcal{D} \leftarrow \mathcal{D} \cap \mathcal{M}^{\complement}$
10:             $mAdded \leftarrow true$
11:             break
12:         end if
13:         if $\neg isPos(\mathcal{D}) \wedge \mathcal{D} \cap \mathcal{M} \neq \emptyset$ then
14:             $\mathcal{D} \leftarrow \mathcal{D} \cap \mathcal{M}$
15:             $mAdded \leftarrow true$
16:             break
17:         end if
18:     end for
19:     if $\neg mAdded$ then
20:         if $\mathcal{R} \cap \mathcal{M}^{\complement} \neq \emptyset$ then
21:             $DA \leftarrow DA \cup \{\mathcal{R} \cap \mathcal{M}^{\complement}\}$
22:         else if $\mathcal{R}^{\complement} \cap \mathcal{M} \neq \emptyset$ then
23:             $DA \leftarrow DA \cup \{\mathcal{R}^{\complement} \cap \mathcal{M}\}$
24:         end if
25:     end if
26: end for
27: $DSs \leftarrow \emptyset$
28: for $\mathcal{D} \in DA$ do
29:     $DSs \leftarrow DSs \cup \{\texttt{pickAword}(\mathcal{D})\}$
30: end for
31: return $DSs$
```

The approach is shown in Alg. 4. The algorithm first creates a set of distinguishing automata DA, initially empty. Then, for each mutant m, it (randomly) iterates over all the automata $\mathcal{D}$ in DA to find an already existing automaton that can distinguish $\mathcal{M}$ (the automaton of m):

- if $\mathcal{D}$ is positive and $\mathcal{M}^{\complement}$ can be conjuncted with $\mathcal{D}$ (i.e., $\mathcal{D} \cap \mathcal{M}^{\complement}$ is not empty), then the conjunction is performed and the search terminates (lines 8-11);
- otherwise, it checks whether $\mathcal{D}$ is negative and $\mathcal{M}$ can be conjuncted (i.e., $\mathcal{D} \cap \mathcal{M}$ is not empty); if this is the case, the search terminates (lines 13-16);
- if neither $\mathcal{M}^{\complement}$ nor $\mathcal{M}$ can be conjuncted with any existing distinguishing automaton, a new distinguishing automaton is created. It first tries to create a positive distinguishing automaton (lines 20-21) and, if not possible, it tries to build a negative distinguishing automaton[2] (lines 22-23). If both cannot be created, it means that $\mathcal{R}$ and $\mathcal{M}$ are equivalent.

[2]Note that we give precedence to positive distinguishing automata. A different approach could be to randomly choose the precedence.

Algorithm 5 MUTREX: Computing mutation score

```
Require: $S$: set of strings
 1: $muts \leftarrow \texttt{mutate}(r)$
 2: $killedMuts \leftarrow \emptyset$
 3: for $m \in muts$ do
 4:     if $\exists s \in S\colon killed(r, m, s)$ then
 5:         $killedMuts \leftarrow killedMuts \cup \{m\}$
 6:     else if $m \equiv r$ then
 7:         $muts \leftarrow muts \setminus \{m\}$
 8:     end if
 9: end for
10: return $|killedMuts|/|muts|$
```

After the iteration over all the mutants, each non-equivalent mutant has been added to a distinguish automaton. The algorithm builds the set of distinguishing strings by taking a word from each distinguishing automaton (lines 28-29).

V. COMPUTING THE MUTATION SCORE OF A STRING SET

The MUTREX framework can be used also to measure the quality of a set of strings for a generic regex. As already observed, a common way to validate a regex r, is to generate a set of meaningful strings from r (both accepted and rejected by r), and submit these strings to the user. If the user confirms that the strings are correctly labeled as valid and invalid, the regex r is considered validated. The quality of the string set is crucial: if there are too *banal* strings, these may lead to a false confidence in the correctness of r.

We can use mutants to assess the meaningfulness of a set of strings. We introduce, as in mutation testing, the concepts of killing a mutant and mutation score.

Definition 18 (Killed mutant)**.** A mutant m of r is *killed* by a string s iff m evaluates s differently from r, i.e., s is a distinguishing string for r and m. Formally:

$$killed(r, m, s) \iff s \in \mathcal{L}(r \oplus m)$$

Example 19. If r is `[a-zA-Z]` and the CCR mutant m is `[a-z]`, then the string "A" kills m, while the string "a" does not.

Given a set of strings S for a regex r, we can use the MUTREX function $\texttt{mutate}(r)$ as follows.

Definition 20 (Mutation score)**.** The mutation score of a set of strings S for a regex r is defined as follows:

$$\frac{|\{m \in \texttt{mutate}(r)\colon (\mathcal{L}(r) \neq \mathcal{L}(m) \wedge \exists s \in S\colon killed(r, m, s))\}|}{|\{m \in \texttt{mutate}(r)\colon \mathcal{L}(r) \neq \mathcal{L}(m)\}|}$$

The mutation score is the ratio of non-equivalent mutants that are killed by at least a string in S. We can compute the mutation score using the algorithm shown in Alg. 5. The algorithm iterates over all mutants, collects those that are killed by at least an s in S (lines 4-5), and identifies the equivalent ones (removed by the set of mutants at line 7). At the end, the mutation score is returned. Note that the test generation approach of MUTREX (see Algs. 2, 3, and 4) guarantees to generate test suites having mutation score 1 (by definition of distinguishing string).

The set of string S can come from different sources. It could have been obtained from some tools for generating tests for regexes, as EGRET [14]. In this case, MUTREX is used to evaluate the fault detection capability of the generated strings. We will present an experiment is this direction in Sect. VI-A.

S could instead be the set of strings from which the user synthesized the regex r (for example, by the technique presented in [17]). In this case, the mutation score tells how well S represents r; a low mutation score means that probably S is not sufficient to guess the right regex and more labeled strings should be added to S in order to synthesize a better regex.

VI. EXPERIMENTS

As benchmarks we have taken 33 regexes from different books and websites[3]. They are used to detect dates, e-mail addresses, URLs, credit card numbers, US phone numbers, palindrome words, floating point numbers, and car identification numbers.

Fig. 3 shows the size of each regex in terms of length and number of operators. The benchmarks are representative of different kinds of regexes, from small ones, long less than 100 characters and containing less than 20 operators, to big ones, longer than 100 characters and with more than 20 operators.

All the experiments have been executed on a Linux PC with Intel(R) Core(TM) i7 CPU and 4 GB of RAM.

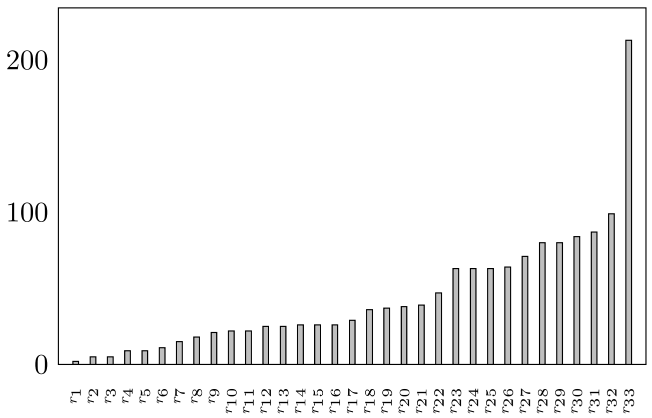
RQ1 How many mutants are generated by each mutation operator?

Fig. 4 shows, for each mutation operator, the number of mutants generated for all benchmarks. The operator that produces more mutants is CCR (that removes an interval from a character class) since character classes are widely used in regexes. In general, operators that mutate common operators (e.g., CCN negating a character class, RM modifying the range of an interval, CCA adding a new interval, ...) produce several mutants. Operators that rely on the presence of seldom used operators, instead, produce few mutants: NCCO, for example, applies to negated character classes that are not commonly used by developers.

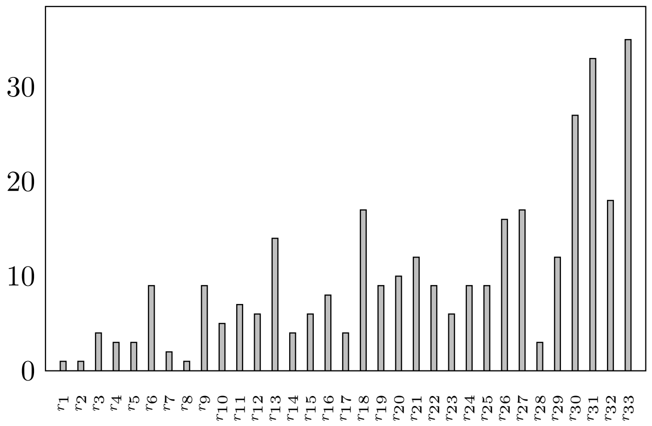
RQ2 How are mutants distributed in the classes presented in Sect. III-D?

Fig. 5 shows, for each mutation class, the classification of its mutants in generalizations, specializations, arbitrary edits, and equivalent mutants. As expected, PA only produces specializations: indeed, adding a prefix to a regex can only reduce the set of accepted strings. In our experiments, C2M always produces generalizations because it usually mutates a normal dot \. in the metachar . and, therefore, it will accept more strings (having any character for the metachar). Table II reports the total number of mutant types, and their percentage over all the mutations. We can see that most of the mutants are specializations: this means that mutation operators tend to

[3]All benchmarks can be downloaded from http://cs.unibg.it/mutrex/mutation2017experiments.txt.



(a) Length



(b) Number of operators

Figure 3: Benchmarks size

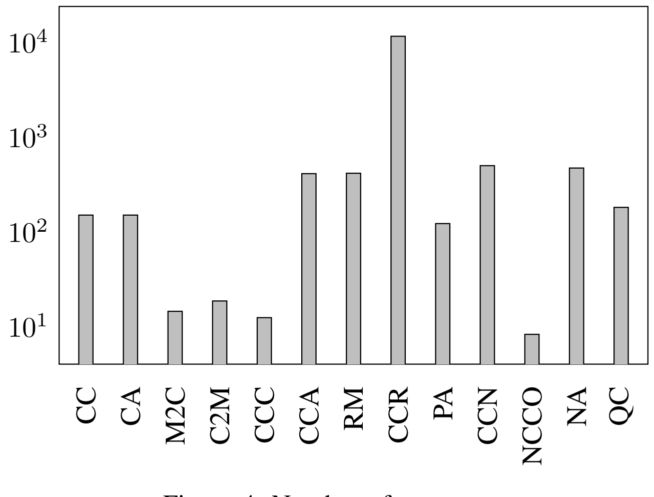


Figure 4: Number of mutants

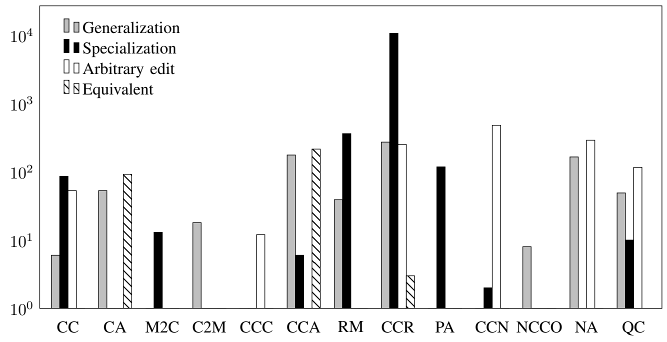


Figure 5: Mutants classification – Single operator classes

Table II: Mutants classification – All operators

	Generalization	Specialization	Arbitrary edit	Equivalent	Total
Sum	787	11250	1203	310	13550
Ratio	5.8%	83.0%	8.9%	2.3%	

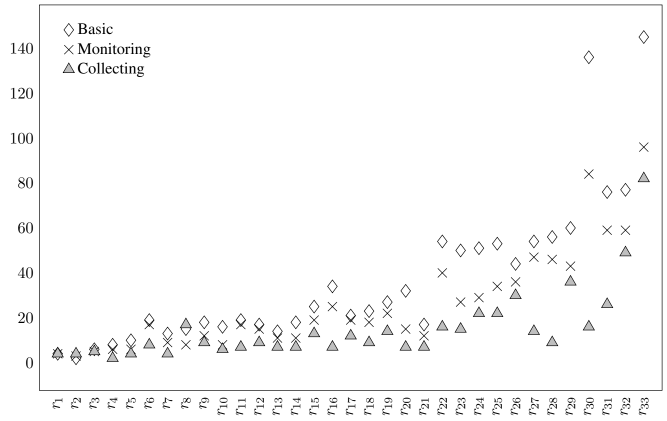


Figure 6: Test suite size

add further constraints to the regex under test. Only 2.3% of mutants are equivalent: this means that the equivalent mutants are not a big problem for regex testing (moreover, detecting equivalent mutants is decidable in this context).

RQ3 How big are the fault-detecting test suites?

Fig. 6 reports the sizes of the generated test suites for all the benchmarks and for the three generation techniques: the *basic* one shown in Alg. 2, the one using *monitoring* shown in Alg. 3, and the one using *collecting* shown in Alg. 4. For all the three techniques, the size of the test suite grows linearly with the size of the regex under test (in the plot, regexes are ordered according to their length). As expected, collecting permits to obtain the most compact test suites. On average, the size of the test suite obtained with collecting is half of the size of the test suite obtained with the basic approach. Also monitoring permits to reduce the size of the test suite, but not as well as collecting; this fact has been observed also in other contexts [2]. The total size of the test suites for all the benchmarks is reported in Table III.

Table III: Comparison of techniques

Technique	Size	Time (sec)
Basic	1214	662
Monitoring	862	1258
Collecting	499	3827

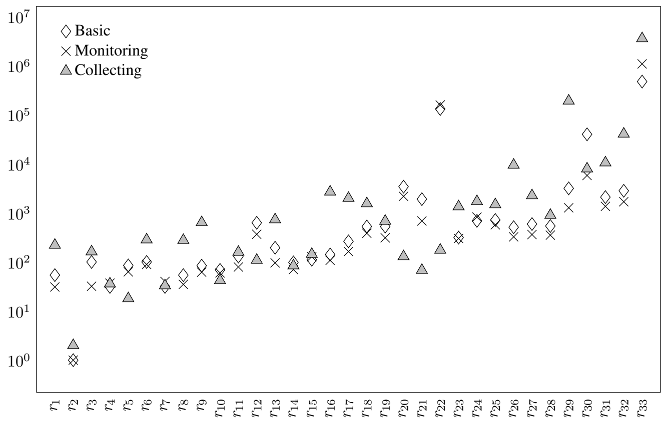


Figure 7: Generation time (ms)

RQ4 How much time is required by the generation techniques?

The time taken for generating the test suites is shown in Fig. 7. Table III reports the total time over all the benchmarks; monitoring takes the double of time of basic, and collecting is three times slower than monitoring.

RQ5 Is monitoring effective?

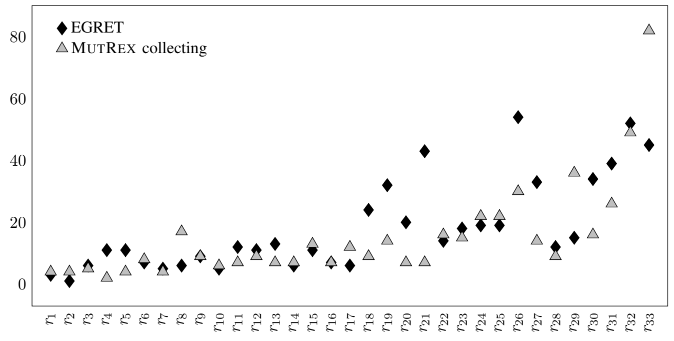
W.r.t. the basic approach, monitoring permits to always obtain smaller test suites (see Fig. 6), but it takes on average the double of time (see Table III). Note that monitoring is particularly effective on big regexes that produce a lot of mutants: in these cases, the basic approach produces a lot of tests for killing all the mutants, and monitoring permits to significantly reduce the number of tests needed.

RQ6 Is collecting effective?

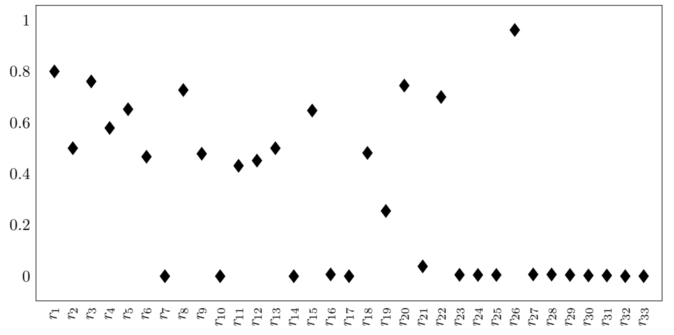
In terms of size, collecting is effective w.r.t. both the basic approach and the monitoring approach. However, it is the slowest technique (three times slower than the monitoring approach). Note that the bottleneck of the approach is the addition of a new automaton to a distinguishing automaton (lines 9 and 14 in Alg. 4). As future work, we plan to devise heuristics to improve the scalability of the approach; a solution could be to limit the number of automata that can be conjuncted together in a distinguishing automaton.

A. *Comparison with other string generator tools*

We are interested in studying how MUTREX can be compared with other string generator tools. As comparison, we have chosen EGRET (Evil Generation of Regular Expression Tests) which is a very recent tool [14], freely available, and it promises to generate the most "evil" strings. We have the following two main research questions.



(a) Size



(b) Mutation score

Figure 8: EGRET

RQ7 Does MUTREX generate more strings than EGRET?

Fig. 8a reports the size of the tests produced by EGRET (compared with the size of MUTREX when using collecting). The data show that EGRET and MUTREX produce test set of comparable size. For the entire benchmark set, EGRET produces 603 strings (against 499 of MUTREX when using collecting and 862 when using monitoring).

RQ8 Is EGRET able to detect faults as MUTREX?

For this question, we can use the mutation score presented in Sect. V. Recall that MUTREX always generates test suites having mutation score 100%; we are interested in knowing to what extent other test generation tools target the fault classes we defined. Fig. 8b reports the mutation score of the string sets produced by EGRET. On average, the mutation score is 31% with a maximum of 96% and a minimum of 0%. Especially tests for long regexes have a very low mutation score. This means that most of the faults we have presented can pass undetected if one relies on EGRET to validate a regex.

VII. Related work

Mutation is a well known technique in the context of software artifacts. It has been mainly applied to programming languages, but also at the design level to formal specifications [3], [4], [7], [10], [15], [23]. In particular, Offutt et al. [19] argue that mutation analysis is a test generation method that create inputs from syntactic descriptions and it is applicable to any software artifact on the base of its syntactic description. They view mutation as an implementation of *grammar-based testing*, in the sense that a syntactic description such as a grammar is used to create tests. Our way of using mutation for testing regexes perfectly reflects this concept of grammar-based testing.

Although no specific mutation operators have ever been defined for regexes, there exist approaches based on suitable (grammar) transformations, some on regexes, some on strings accepted by regexes, which resemble mutation. Among these approaches, Li et al. [17] introduce ReLIE, a learning algorithm that, starting from a plausible initial regex and a set of labeled strings, tries to learn the correct regex. ReLIE works on a set of regex transformations, a sort of regex mutation, to obtain a set of candidate regexes. Similarly, MUTREX allows to learn the intended regex by a set of strings accepted by mutated regex.

Closer to our approach and based on string transformations, is the approach presented in [14] to generate *evil* strings. EGRET is a tool for generating test strings for a regex. It takes a regex as input and generates two lists of test strings: strings *accepted* by the regex, and strings *rejected*. A user can manually scan both these lists and identify strings that are incorrectly classified: incorrectly accepted or incorrectly rejected. In this way, (s)he can have confidence that the regex works as intended. Generated strings work, therefore, as *evil* since they expose errors commonly made by programmers. As for MUTREX, the approach is based on converting regexes into nondeterministic finite state automata, generating strings able to expose possible common mistakes, involving the user as oracle to decide regex correctness. Furthermore, EGRET generation of evil strings applies mutation on strings accepted by the regex (differently from MUTREX that mutates the regex): altering the number of iteration for each repeat operators and changing the character used for a character set. However, MUTREX improves EGRET in terms of fault detection capability: when a user detects a faulty string, (s)he also knows the error made in writing the regex (that is the error induced by the mutation operator), and, as shown by our experiments on the mutation score of strings generated by EGRET (see Sect. VI-A,), especially for long regexes, faults can pass undetected if one relies on EGRET evaluation.

Different tools have been developed for testing regexes. They are based on random generation of test strings, and most of them are available on line, as EXREX, Regex101, RegExr, RegEx Testing, to cite a few. MUTREX can be used in combination with these tools to evaluate the fault detection capability of the set of generated strings, as discussed for EGRET in Sect. VI-A.

Other approaches have been defined for testing purposes, still based on strings generation, which exploit the symbolic automaton representation. Veanes et al. [22] have developed Rex, a general-purpose solver of regexes constraints. Rex translates a given regex into a symbolic representation of a finite automaton, i.e., an NFA where transitions are labeled by formulas representing set of characters rather than single characters. A symbolic finite automaton is represented in terms of a set of axioms that describe the acceptance conditions of a string by the automaton, and an SMT solver (Z3) is used for satisfiability checking. Since the SMT solver is able to generate a *model* as witness of the satisfiability check, Rex can be used to build strings accepted by the automaton/regex: the model represents an accepted string. Rex is used in combination with Pex [21] for dealing with regex constraints in parameterized unit tests. Reggae [16] in a tool to generate string test inputs that are accepted by a given regex. The goal is to perform branch coverage for programs that use complex string operation including regex operations. It exploits dynamic symbolic execution based on path exploration. Kiezun et al. [13] propose HAMPI, a solver for string constraints over bounded string variables. Users of HAMPI specify constraints using regexws, context-free grammars, equality between string terms, and typical string operations such as concatenation and substring extraction. HAMPI then finds a string that satisfies all the constraints or reports constraints unsatisfiability.

VIII. Conclusions

In this paper, we have presented a fault-based test generation approach for regular expressions (regexes). We have introduced a set of fault classes (and corresponding mutation operators) representing possible mistakes developers do when writing a regex, and proposed a technique for generating distinguishing strings, i.e., strings able to distinguish a regex from its mutants. The generation approach consists in generating a distinguishing string for each mutant; the basic approach is improved (in terms of test suite size) by two techniques, monitoring and collecting. Experiments show that the two techniques permit to obtain compact test suites having a guaranteed maximal mutation score. Moreover, we also found that existing test generation approaches produce test suites with a low mutation score, i.e., they are not able to capture faults of most of the classes we proposed.

As future work, we plan to devise a technique that, once a fault is detected, tries to repair the faulty regex. Moreover, we plan to improve the computation time of the collecting technique by devising some heuristics that avoid to build too big distinguishing automata. In order to speed up the process, we also plan to study whether approaches as [8], considering mutants as part of a family, are applicable also in our context. Finally, we plan to evaluate whether higher-order mutants can give any benefit, although heuristics to limit the number of mutants should be applied in this case.

Acknowledgment

The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S.

References

[1] The Bro network security monitor, 2015. `https://www.bro.org/`.

[2] P. Arcaini, A. Gargantini, and E. Riccobene. How to optimize the use of SAT and SMT solvers for test generation of Boolean expressions. *The Computer Journal*, 58(11):2900–2920, 2015.

[3] P. Arcaini, A. Gargantini, and E. Riccobene. Using mutation to assess fault detection capability of model review. *Software Testing, Verification and Reliability*, 25(5-7):629–652, 2015.

[4] P. Arcaini, A. Gargantini, and P. Vavassori. Generating tests for detecting faults in feature models. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, April 2015.

[5] A. N. Arslan. Multiple sequence alignment containing a sequence of regular expressions. In *2005 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, pages 1–7, Nov 2005.

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

[7] S. D. R. S. De Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. L. De Souza. Mutation Testing Applied to Estelle Specifications. *Software Quality Control*, 8:285–301, December 1999.

[8] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 655–666, New York, NY, USA, 2016. ACM.

[9] M. Erwig and R. Gopinath. *Explanations for Regular Expressions*, pages 394–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[10] S. C. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 220 –229, nov 1994.

[11] J. E. F. Friedl. *Mastering regular expressions (3. ed.)*. O'Reilly, 2006.

[12] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.

[13] A. Kiezun, V. Ganesh, S. Artzi, P. Guo, P. Hooimeijer, and M. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering andMethodology-TOSEM*, 21(4), 2012.

[14] E. Larson and A. Kirk. Generating evil test strings for regular expressions. In *Software Testing, Verification and Validation (ICST), 2016 IEEE 9th International Conference on*, April 2016.

[15] T.-C. Lee and P.-A. Hsiung. Mutation Coverage Estimation for Model Checking. In F. Wang, editor, *Automated Technology for Verification and Analysis*, volume 3299 of *Lecture Notes in Computer Science*, pages 354–368. Springer Berlin / Heidelberg, 2004.

[16] N. Li, T. Xie, N. Tillmann, J. d. Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 515–519. IEEE Computer Society, 2009.

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

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

[19] J. Offutt, P. Ammann, and L. L. Liu. Mutation testing implements grammar-based testing. In *Proceedings of the Second Workshop on Mutation Analysis*, MUTATION '06, pages 12–, Washington, DC, USA, 2006. IEEE Computer Society.

[20] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM.

[21] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. In *International conference on tests and proofs*, pages 134–153. Springer, 2008.

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

[23] M. R. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal*, 8(4):211–224, Jul 1993.

[24] A. S. Yeole and B. B. Meshram. Analysis of different technique for detection of SQL injection. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, pages 963–966, New York, NY, USA, 2011. ACM.