SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–27
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Fault-based test generation for regular expressions by mutation

Paolo Arcaini[1*], Angelo Gargantini[2] and Elvinia Riccobene[3]

[1] *Charles University, Faculty of Mathematics and Physics, Czech Republic*
[2] *Department of Management, Information and Production Engineering, Università degli Studi di Bergamo, Italy*
[3] *Dipartimento di Informatica, Università degli Studi di Milano, Italy*

SUMMARY

Regular expressions are used to characterize sets of strings (i.e., languages) using a pattern-based syntax. They are applied in different contexts as, for example, data validation in web forms. However, writing a regular expression that exactly captures the desired set of strings could be particularly difficult, and techniques are sought to validate regular expressions or test their use in applications. A common means to regular expression validation and testing is the generation of a set of *labeled* strings (i.e., strings together with their evaluation).
We here propose a *fault-based approach for generating strings* usable as tests for regular expressions. We define some fault classes representing mistakes that could be made when writing a regular expression, and we introduce the notion of *distinguishing string*, i.e., a string that is able to expose a fault. Given a regular expression, our approach generates a test suite composed of distinguishing strings that are able to detect possible faults in the regular expression. We present different versions of the approach, which provide different results in terms of test suite size and generation time. Experiments show that the proposed approach can generate compact test suites and that, using suitable optimizations, the generation time is reasonable.
Exploiting the proposed fault classes, we use the notion of *mutation score* to assess the ability of a generic set of strings in exposing possible faults contained in the regular expression under test. A comparison with other test generation tools in terms of mutation score, size, and generation time shows the advantages and limits of our approach. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: regular expression; fault class; mutation testing; distinguishing string; mutation score

1. INTRODUCTION

Regular expressions are used for different purposes as data validation, lexical analysis, or string matching in texts. Regular expressions have been applied in several different contexts [1] as, to name a few, intrusion detection in networks [2], prevention of MySQL injection [3], and DNA sequencing alignment [4].

Regular expressions provide programmers with a pattern-based syntax that permits to precisely characterize a set of strings (i.e., a *language*). However, writing a correct regular expression that exactly captures the desired set of strings could be particularly difficult, due to their compact and rather tolerant syntax. Moreover, the syntax checking that can be performed over a regular expression is rather limited and most regular expressions are free of syntax errors: therefore, it is not common to discover faults at parsing time. Several studies show that regular expressions are quite often faulty, i.e., they do not exactly describe the intended language [5, 6]: they could accept words that should not be accepted, or refuse words that should be accepted, or both.

*Correspondence to: Paolo Arcaini, Charles University, Malostranské náměstí 25, 118 00 Praha 1, Czech Republic. E-mail: arcaini@d3s.mff.cuni.cz

Copyright © 0000 John Wiley & Sons, Ltd.

Therefore, several techniques and tools have been proposed for easing the design of regular expressions and for their validation and testing. For example, some approaches attempt to discover errors using static checking [5], others using visual debugging [7]; a different approach [8] proposes to transform a regular expression in a more intelligible format that should allow the developer to understand whether the regular expression is what (s)he had in mind.

However, most of the approaches are based on the use of *labeled strings*, i.e., strings with their evaluation, either accepted or rejected [9, 10, 6, 11, 12]. Labeled strings are exploited in two ways. The first one applies to a given set of labeled strings (possibly provided by the developer) some learning algorithm in order to extract a regular expression [11]. The second one consists in generating, from the stated regular expression r, a set of labeled strings that can be used for different purposes: shown to the developer in order to validate r [6] (as done in the web implementation of our tool), used for regression testing (i.e., r is known to be correct and the generated strings are used to test an alternative implementation of r), used to provide particular inputs to the program where r is embedded in order to try to make the program crash or to expose wrong behaviors, and for testing regular expression evaluators (like those presented by Becchi et al. [13]).

The approach proposed here consists in the generation of a set of *fault detecting* labeled strings. The novelty lies in the process used for string generation that is driven by a notion of fault coverage and exploits *mutation*. The approach first generates mutants from a regular expression r, according to some fault classes representing common mistakes that can be made when writing regular expressions; then, for each mutant m, it generates a string s that *distinguishes* m from r, i.e., a string that is evaluated differently by r and m. The set of generated distinguishing strings is a test suite able to detect all the seeded faults.

The main original contribution of our approach w.r.t. other approaches that generate tests [6, 12], is that during generation we directly aim at the detection of faults. Targeting faults has some advantages: in regular expression validation, the developer should be able (observing a limited number of strings) to increase her/his confidence in the regular expression correctness (i.e., absence of faults); in testing of programs involving regular expressions, the generated strings should be able to trigger program executions that are more likely to be faulty. The *basic* approach is improved by two techniques, *monitoring* and *collecting*, that permit to obtain more compact test suites.

We have already proposed the three aforementioned test generation approaches (i.e., *basic*, *monitoring*, and *collecting*) [14]: we found that the approach that allows to obtain the smallest test suite is *collecting* that directly generates tests for targeting multiple test requirements (i.e., for distinguishing multiple mutants). The technique, however, is computationally expensive. Therefore, in this paper, we provide two optimized versions of the collecting algorithm: one that parallelizes the collection process, and another one that limits the number of test requirements that are considered together.

Moreover, we also propose a way to assess the *mutation score* (i.e., the fault coverage) of a generic test suite of regular expressions (possibly coming from different sources): it consists in the percentage of faults (those described by our fault classes) the test suite can detect. Such approach could be used to evaluate existing test generation tools [6, 12], or also the test suites generated by our approach in case it cannot complete the whole generation due to time constraints (indeed, if the approach terminates correctly, it has maximum mutation score by definition). Such notion of fault coverage could be also exploited to evaluate the quality of strings from which regular expressions are synthesized [11].

The paper is organized as follows. Sect. 2 introduces some background on regular expressions, and Sect. 3 presents the fault classes and the corresponding mutation operators we devised. Sect. 4 introduces the approach we propose to generate some strings that distinguish a regular expression from its mutants, describing the three main algorithms and the optimizations of the collecting algorithm. Sect. 5 presents the concept of mutation score for regular expressions and describes a way to compute it for a given set of strings and set of mutants. Sect. 6 describes the tool MUTREX, and Sect. 7 presents the experiments we performed. Sect. 8 discusses possible threats to the validity of our approach, Sect. 9 discusses some related work, and Sect. 10 concludes the paper.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

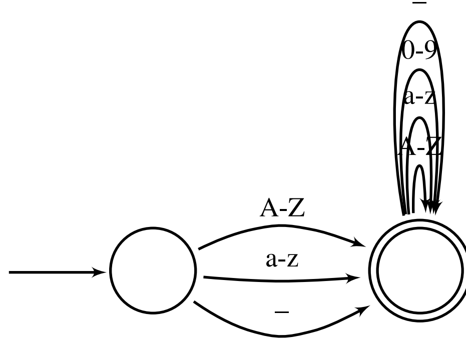


Figure 1. Automaton of regular expression [a-zA-Z_][a-zA-Z0-9_]*

2. BACKGROUND

We here recall some basic definitions regarding regular expressions, which are relevant for our approach. Note that, in our context, regular expressions are intended as valid regular expressions from formal automata theory, i.e., only regular expressions that describe regular languages. The proposed approach is indeed based on the use of the finite automata accepting the language described by the regular expressions.

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

As Σ we support the Unicode alphabet (UTF-16); the supported grammar is shown in Table I.

Definition 2 (Acceptance)
A string s is *accepted* by a regular expression 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 shows an example of an automaton that accepts all the words of the regular expression [a-zA-Z_][a-zA-Z0-9_]*. In this paper, we use the library dk.brics.automaton [15] to transform a regular expression r into an automaton $\mathcal{R}$ (by means of function `toAutomaton`(r)) and perform standard operations on it, as those described hereafter.

Definition 3 (Automaton unary operators)
Let $\mathcal{R}$ be the automaton of a regular expression r, and $\mathcal{L}(\mathcal{R})$ the language accepted by the automaton $\mathcal{R}$. The following are *unary operators* on $\mathcal{R}$:

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

Definition 4 (Automata binary operators)
Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be the automata of regular expressions r_1 and r_2. The following are binary operators on $\mathcal{R}_1$ and $\mathcal{R}_2$:

- *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*: $\mathcal{R}_1 \oplus \mathcal{R}_2 = (\mathcal{R}_1 \cap \mathcal{R}_2^{\mathsf{C}}) \cup (\mathcal{R}_1^{\mathsf{C}} \cap \mathcal{R}_2)$ accepts the strings accepted by only one of the two automata, i.e., $\mathcal{L}(\mathcal{R}_1 \oplus \mathcal{R}_2) = \mathcal{L}(\mathcal{R}_1) \setminus \mathcal{L}(\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_2) \setminus \mathcal{L}(\mathcal{R}_1)$.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Table I. BNF of the regular expressions supported by MUTREX

<table>
<tr><th>Name</th><th></th><th>Derivation rule</th><th></th></tr>
<tr><td>regular expression</td><td>::=</td><td>union</td><td></td></tr>
<tr><td>union</td><td>::=</td><td>intersection [“|” union]</td><td></td></tr>
<tr><td>intersection</td><td>::=</td><td>concatenation [“&” intersection]</td><td></td></tr>
<tr><td>concatenation</td><td>::=</td><td>repeatexp [concatenation]</td><td></td></tr>
<tr><td rowspan="7">repeatexp</td><td>::=</td><td>repeatexp “?”</td><td>(zero or one occurrence)</td></tr>
<tr><td>|</td><td>repeatexp “*”</td><td>(zero or more occurrences)</td></tr>
<tr><td>|</td><td>repeatexp “+”</td><td>(one or more occurrences)</td></tr>
<tr><td>|</td><td>repeatexp “{”n“}”</td><td>(n occurrences)</td></tr>
<tr><td>|</td><td>repeatexp “{”n“,}”</td><td>(n or more occurrences)</td></tr>
<tr><td>|</td><td>repeatexp “{”n“,”m“}”</td><td>(n to m occurrences)</td></tr>
<tr><td>|</td><td>complement</td><td></td></tr>
<tr><td>complement</td><td>::=</td><td>“~” complement | charclassexp</td><td></td></tr>
<tr><td rowspan="3">charclassexp</td><td>::=</td><td>“[” charclasses “]”</td><td>(character class)</td></tr>
<tr><td>|</td><td>“[^” charclasses “]”</td><td>(negated char class)</td></tr>
<tr><td>|</td><td>simpleexp</td><td></td></tr>
<tr><td>charclasses</td><td>::=</td><td>charclass [charclasses]</td><td>(sequence of charclass)</td></tr>
<tr><td>charclass</td><td>::=</td><td>char [“-” char]</td><td>(character range)</td></tr>
<tr><td rowspan="7">simpleexp</td><td>::=</td><td>char</td><td></td></tr>
<tr><td>|</td><td>“.”</td><td>(any single character)</td></tr>
<tr><td>|</td><td>“#”</td><td>(the empty language)</td></tr>
<tr><td>|</td><td>“@”</td><td>(any string)</td></tr>
<tr><td>|</td><td>“"”<Unicode string without double-quotes>“"”</td><td>(a string)</td></tr>
<tr><td>|</td><td>“()”</td><td>(empty string)</td></tr>
<tr><td>|</td><td>“(” union “)”</td><td>(precedence override)</td></tr>
<tr><td rowspan="5">char</td><td>::=</td><td><Unicode character></td><td>(a single non-reserved character)</td></tr>
<tr><td>|</td><td>“\w”</td><td>(word character)</td></tr>
<tr><td>|</td><td>“\d”</td><td>(digit)</td></tr>
<tr><td>|</td><td>“\s”</td><td>(whitespace character)</td></tr>
<tr><td>|</td><td>“\” <Unicode character></td><td>(a single character- except w,d,s)</td></tr>
<tr><td rowspan="3">not supported</td><td></td><td>^, $, \A, \Z</td><td>(anchors)</td></tr>
<tr><td></td><td>\b, \B</td><td>(word boundaries)</td></tr>
<tr><td></td><td>(?<= ...</td><td>(lookahead / lookbehind assertions)</td></tr>
</table>

3. FAULT CLASSES AND MUTATION OPERATORS

Mutation is a well known technique in the context of software artifacts, mainly programs, but also specifications and grammars (see Sect. 9). It consists in introducing, into an artifact, small changes, called *mutations*, which represent typical mistakes that developers could make. These faults are deliberately seeded into the original artifact in order to obtain a set of faulty variations called *mutants*. A transformation rule generating a mutant from the original artifact is known as *mutation operator*. Mutation is used for validation purposes and for removing faults from artifacts.

Here, we aim to apply mutation to regular expressions for test generation. In our setting, a mutation operator is a function that given a regular expression r, returns a list of regular expressions (called *mutants*) obtained by mutating r. Every mutation *slightly* modifies the regular expression

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table II. Recap Mutation Operators

<table>
<tr><th>Fault</th><th>Mutation Operator</th><th>Original regular expression</th><th>Some mutated regular expressions</th></tr>
<tr><td colspan="4">Single character faults</td></tr>
<tr><td>wrong upper/lower case</td><td>Case Change (CC)</td><td>a[a-z]*</td><td>A[a-z]* a[A-Z]* a[A-z]*</td></tr>
<tr><td>missing upper/lower case</td><td>Case Addition (CA)</td><td>a[a-z]*</td><td>(a|A)[a-z]* a[a-zA-Z]*</td></tr>
<tr><td>char used as metachar</td><td>Metachar To Char (M2C)</td><td>[0-9]{3}.[0-9]{3}</td><td>((0|-)|9){3}.[0-9]{3}
[0-9]{3}\.[0-9]{3}
[0-9]{3}.((0|-)|9){3}</td></tr>
<tr><td>metachar used as char</td><td>Char To Metachar (C2M)</td><td>\.{3}</td><td>.{3}</td></tr>
<tr><td colspan="4">Character class faults</td></tr>
<tr><td>missing char class spec</td><td>Character Class Creation (CCC)</td><td>(0-9)+</td><td>([0-9])+</td></tr>
<tr><td>missing spec of char interval</td><td>Character Class Addition (CCA)</td><td>[a-z]</td><td>[a-zA-Z] [a-z0-9]</td></tr>
<tr><td>wrong character class specification</td><td>Character Class Modification (CCM)</td><td>[az]
[a-z]</td><td>[a-z]
[az]</td></tr>
<tr><td>wrong spec of char class limits</td><td>Range Modification (RM)</td><td>[f-m]</td><td>[e-m] [g-m]
[f-l] [f-n]</td></tr>
<tr><td>over-spec of char set</td><td>Character Class Restriction (CCR)</td><td>[a-zA-Z0-9]</td><td>[A-Z0-9] [a-z0-9] [a-zA-Z]</td></tr>
<tr><td>missing/wrong constraints on initial char</td><td>Prefix Addition (PA)</td><td>[a-zA-Z0-9]*</td><td>[A-Z0-9][a-zA-Z0-9]*
[a-z0-9][a-zA-Z0-9]*
[a-zA-Z][a-zA-Z0-9]*</td></tr>
<tr><td>missing negation char class</td><td>Character Class Negation (CCN)</td><td>[a-zA-Z]</td><td>[^a-zA-Z]
[^a-z]|[A-Z]
[a-z]|[^A-Z]</td></tr>
<tr><td>wrong negated char class</td><td>Negated Character Class to Optional (NCCO)</td><td>.*q[^u]</td><td>.*q[^u]?</td></tr>
<tr><td colspan="4">Other faults</td></tr>
<tr><td>missing negation</td><td>Negation Addition (NA)</td><td>a
[A-Z][a-z]</td><td>^a
[^A-Z][a-z] [A-Z][^a-z]</td></tr>
<tr><td>wrong quantifier</td><td>Quantifier Change (QC)</td><td>[0-9]*
[a-z]{3}</td><td>[0-9]+ [0-9]?
[a-z]{2} [a-z]{4}
[a-z]{3,} [a-z]{0,3}</td></tr>
</table>

r under the assumption that the programmer has defined r close to the correct version (competent programmer hypothesis [16]).

As suggested by Woodward [17], in order to define mutation operators, we started by identifying possible faults on regular expression definitions. To this aim, we browsed some Internet sites (see Sect. 7.1) and referred to available documentation [18] explaining common mistakes programmers make when writing regular expressions.

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. For each of the detected fault classes, we defined suitable mutation operators. Table II summarizes possible faults and mutation operators.

3.1. *Single character faults*

3.1.1. Wrong upper/lower case Regular expressions are case sensitive, and a user not aware of that could simply *write the wrong case* (either upper or lower). For example, a user could be interested in accepting all strings starting with ‘A’, while (s)he wrongly writes the regular expression `a[a-z]*` that only accepts words starting with ‘a’.

The operator **Case Change** (CC) mutates a regular expression 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).

In the example, CC produces the mutant `A[a-z]*` that only accepts strings starting with ‘A’. This warns the user of a possible fault in the original regular expression. The other mutants produced by CC are `a[A-Z]*` and `a[A-z]*`.

3.1.2. Missing upper/lower case A further mistake could be to *miss a case*. For example, a user could be interested in accepting all strings starting with ‘a’ or ‘A’, while (s)he wrongly writes the regular expression `a[a-z]*` that only accepts words starting with ‘a’.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

The operator **Case Addition** (CA) mutates a regular expression r by making both lower and upper cases possible when only one of the two is used in r: for each character of the regular expression 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.

Considering for example `a[a-z]*`, operator CA would mutate it in mutants `(a|A)[a-z]*` (able to generate the strings that the user had in mind) and `a[a-zA-Z]*`.

3.1.3. Char wrongly used as metachar In a regular expression, some characters can be interpreted as chars or metachars 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. Suppose a user writes the regular expression `[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 regular expression, '.' is a metachar and strings like "123A456" are accepted. Another example is when a user writes `[a-b]+` for matching strings as "a", "b", "-", "a-", "-b", ... However, the regular expression accepts only strings as "a", "b", "ab", "ba", ..., since, again, '-' is a metachar. In both cases, the user may want to use a char c, but (s)he wrongly uses c as metachar.

The operator **Metachar To Char** (M2C) transforms a regular expression r containing a char c interpreted as a metachar in a regular expression 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.

Considering for example `[0-9]{3}.[0-9]{3}`, M2C produces (among others) the mutant `[0-9]{3}\.[0-9]{3}` that correctly expresses what the user had in mind. Similarly, in case of `[a-b]+`, M2C produces the correct mutant `(a|-|b)+`.

3.1.4. Metachar wrongly used as char This is the opposite of the previous fault: a user wants to use a metachar c, but, because of the context, c is interpreted as a simple char. For example, the regular expression `\.{3}` is written with the intention of generating strings of three chars, but only string "..." is accepted since `\.` is interpreted as a normal char.

The operator **Char To Metachar** (C2M) transforms a regular expression r containing c interpreted as a char in a regular expression r' in which c is interpreted as a metachar.

Similarly to M2C, the way to mutate r depends on the metachar c. Applied to the example, C2M produces the mutant `.{3}` that correctly accepts all three char strings.

3.2. Character class faults

3.2.1. Missing char class specification Character classes are delimited by square brackets []. A user may want to use a character class and forgets (or ignores) the brackets. For example, (s)he could have wrongly written `(0-9)+` to accept all the sequences of digits, while strings like "0-9", "0-90-9" are accepted.

Given a regular expression $r = c_1 - c_2$, the operator **Character Class Creation** (CCC) mutates r in $[c_1 - c_2]$.

On the given example, the CCC operator produces the intended regular expression `([0-9])+`.

3.2.2. Missing specification of char interval The user could forget a given interval in a set of character classes. For example, (s)he writes a regular expression `[a-z]`, while (s)he means to include also uppercase characters.

Given a regular expression $r = [cc_1 \dots cc_n]$, the operator **Character Class Addition** (CCA) generates 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`.

On the above example, CCA creates mutants `[a-zA-Z]` and `[a-z0-9]`, the first one accepting the intended strings.

Copyright © 0000 John Wiley & Sons, Ltd.

Table III. Change of multiplicity in PA

Original quantifier	Mutated quantifier
* or {0,}	*
+ or {1,}	*
{n} or {n, n} (with $n > 1$)	{n-1}
{n,} (with $n > 1$)	{n-1,}
{0, m}	{0, m-1}
{n, m} (with $0 < n < m$)	{n-1, m-1}

3.2.3. Wrong specification of character class The user could misunderstand the meaning of the dash symbol $-$ in a character range of a character class; (s)he could, therefore, wrongly select only two characters in a character class (by forgetting the dash), or wrongly select an interval of characters (by misusing the dash).

Given a regular expression r = $[c_1c_2]$, with c_1 and c_2 chars, the operator **CharacterClassModification** (CCM) produces the mutant $[c_1 - c_2]$ (only if there is at least a character between c_1 and c_2); on the contrary, given the expression $[c_1 - c_2]$, CCM produces the mutant $[c_1c_2]$.

For example, given the regular expression r = $[az]$, the operator CCM produces the mutant $[a - z]$.

3.2.4. Wrong specification of char class limits The user could have specified a too tight or too broad interval. For example, in a given regular expression `[f-m]`, the bounds “f” and “m” could have been wrongly determined.

Given a regular expression r = $[c_1 - c_2]$, the operator **Range Modification** (RM) produces a mutant in which c_1 or c_2 is increased or decreased (if it is still a valid char).

On the example, RM creates mutants `[e-m]`, `[g-m]`, `[f-l]`, and `[f-n]` to slightly modify the interval.

3.2.5. Over-specification of the char set A common fault is writing a regular expression that is “too permissive”, i.e., it accepts characters that it should not [11]. For example, in a given regular expression `[a-zA-Z0-9]`, the uppercase characters could have been wrongly inserted.

Given a regular expression r = $[cc_1 \dots cc_n]$, the operator **Character Class Restriction** (CCR) 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).

On the regular expression `[a-zA-Z0-9]`, CCR creates mutants `[A-Z0-9]`, `[a-z0-9]`, and `[a-zA-Z]`.

3.2.6. Missing or wrong constraints on string initial characters 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.

Given a repeat regular expression r = $[cc_1, \dots, cc_n]$`m` (being `m` the multiplicity), the mutation operator **Prefix Addition** (PA) introduces a prefix that contains all the character classes but one (cc_i). The operator modifies the multiplicity `m` to `m`$'$ as shown in Table III. Formally, the i-th mutant is defined as follows: $[cc_1, \dots, cc_{i-1}cc_{i+1}, \dots, cc_n][cc_1, \dots, cc_n]$`m`$'$.

Suppose the regular expression `[a-zA-Z0-9]*` is given with the intention of generating only strings starting with an alphabetic char. The regular expression is not correct since it accepts also digits as first char. The PA mutant `[a-zA-Z][a-zA-Z0-9]*` satisfies the intended constraint on the string initial char; the other mutants are `[A-Z0-9][a-zA-Z0-9]*` and `[a-z0-9][a-zA-Z0-9]*`.

3.2.7. Missing negation char class Regular expression $[\hat{}cc_1 \dots cc_n]$ matches any character that is not listed in the character classes $cc_1, \dots, cc_n$. A user could have forgotten symbol ˆ and written $[cc_1 \dots cc_n]$.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table IV. Quantifier modification in QC

Original quantifier	Mutated quantifier
*	+, ?
+	*, ?
?	*, +
{n, m} (with $n < m$)	{n-1, m}, {n+1, m}, {n, m-1}, {n, m+1}
{n} or {n, n}	{n-1}, {n+1}, {n,}, {0, n}
{n,} (with $n > 1$)	{n-1,}, {n+1,}, {n}, {0, n}

The mutation operator **Character Class Negation** (CCN) introduces symbol ˆ at the beginning of a char class; it creates a mutant in which all the character classes are excluded (i.e, [$\hat{}cc_1 \ldots cc_n$]), and a mutant for each character class cc_i in which only cc_i is excluded (i.e, [cc_1] | . . . | [$\hat{}cc_i$] | . . . | [cc_n]).

Suppose to have the regular expression `[a-zA-Z]`. It is mutated in three alternative regular expressions: `[^a-zA-Z]`, `[^a-z]|[A-Z]`, and `[a-z]|[^A-Z]`.

3.2.8. Wrong negated char class Another common error regarding the use of a negated character class [ˆcc] is that it requires "to match a character that is not listed" and not "to not match what is listed" [18]. 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 of a word, (s)he could be interested in accepting also no character. Suppose, for example, a user wants to find all the words having a 'q' not followed by an 'u' [18]; (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.

The operator **Negated Character Class to Optional** (NCCO) makes ˆ optional, i.e., it mutates [ˆcc] in [ˆcc]?.

Applying NCCO to the regular expression `.*q[^u]` would produce `.*q[^u]?` that also accepts "Iraq".

3.3. Other faults

3.3.1. Missing negation In a regular expression, the user could forget a negation ˆ. For example, (s)he writes the regular expression `a` while (s)he wanted to write the regular expression `^a`.

The operator **Negation Addition** (NA) mutates r by adding a negation wherever possible in r, i.e., if $r = r_1 r_2 \ldots r_n$ then NA generates mutants $\hat{}r_1 r_2 \ldots r_n$, $r_1 \hat{}r_2 r_n$, $\ldots$, $r_1 r_2 \ldots \hat{}r_n$.

Given the regular expression `a`, NA creates mutant `^a`; instead, given the regular expression `[A-Z][a-z]`, the operator creates mutants `[^A-Z][a-z]` and `[A-Z][^a-z]`.

3.3.2. Wrong quantifier The user could have used the wrong cardinality. For example, (s)he could have wrongly written the regular expression `[0-9]*` to accept all the sequences of digits (that, however, also accepts the empty string), but (s)he uses the wrong quantifier (* instead of +).

The operator **Quantifier Change** (QC) mutates each simple repeat quantifier in another simple quantifier; moreover, for each user-defined quantifier `{n}` , it creates a mutant in which n (and also m if the quantifier is `{n, m}`) is increased and a mutant in which it is decreased. The complete list of modifications is described in Table IV.

On the given regular expression `[0-9]*`, the operator QC would produce the correct regular expression `[0-9]+`. On the regular expression `[a-z]{3}`, QC generates mutants `[a-z]{2}`, `[a-z]{4}`, `[a-z]{3,}`, and `[a-z]{0, 3}`.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

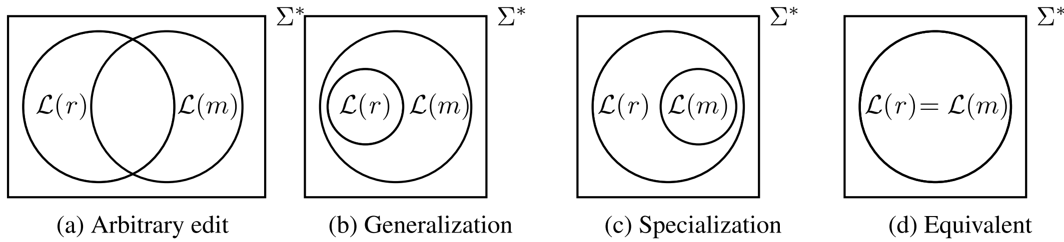


Figure 2. Classification of mutants

3.4. Classification of mutants

The user could be interested in knowing how a mutant m of a regular expression r modifies the language $\mathcal{L}(r)$ accepted by r. As shown in Fig. 2, there are four possible modifications:

- *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.

Such information could be useful to guide the test generation in case the user had some idea about the kind of possible faults contained in the regular expression under test. For example, if the user suspected that the developed regular expression is too permissive (i.e., it accepts strings that should not be accepted), for test generation (s)he could prefer to use mutants that more likely produce specializations.

Example 1 (Classification examples)
m = `[a-z][A-Z]*` is a CC mutant of r = `[a-z][a-z]*`; it is an arbitrary edit: for example, "aA" is accepted only by m, and "aa" is accepted only by r.

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

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

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

Note that the classification can be automatically performed by aptly combining the automata representations $\mathcal{R}$ and $\mathcal{M}$ of the original regular expression r and the mutant m; for example, if $\mathcal{R}^{\complement} \cap \mathcal{M} \neq \emptyset \wedge \mathcal{R} \cap \mathcal{M}^{\complement} = \emptyset$ holds, m is a generalization of r.

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 regular expression meaning and semantics. A simple alternative way to define mutation operators for regular expressions, would be ignoring fault classes and simply defining a unique syntactic variation by substituting a character c in a regular expression 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 our test generator algorithms.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Algorithm 1 Generation of a distinguishing set – Basic approach

```
Require: r: initial regular expression
 1: R ← toAutomaton(r)
 2: muts ← mutate(r)
 3: DSs ← ∅
 4: for m ∈ muts do                              ▷ Random iteration over the mutants
 5:     M ← toAutomaton(m)
 6:     U ← R ⊕ M
 7:     if U ≠ ∅ then                             ▷ r and m are not equivalent
 8:         DSs ← DSs ∪ {pickAword(U)}
 9:     end if
10: end for
11: return DSs
```

4. GENERATION OF FAULT DETECTING STRINGS

We here propose a sequence of labeled string generation algorithms (from a basic version to several improved ones) that exploit mutation over regular expressions to generate meaningful strings that are able to expose possible faults represented by mutants. Our goal is to build a set of strings that are *critical* for the regular expression under test. Such critical strings are those able to distinguish the given regular expression from its mutants, according to the following definition:

Definition 5 (Distinguishing string)
Given two regular expressions 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.

4.1. Basic approach

The *basic* approach for generating a set of distinguishing strings is shown in Alg. 1. The approach first generates a set of mutants $muts$ using the mutation operators defined in Sect. 3 (line 2); then, for each mutant m, it tries to generate a distinguishing string for r and m: it generates the symmetric difference $\mathcal{U}$ of the two automata representation of the regular expressions (line 6) and, if $\mathcal{U}$ is not empty (i.e., the two regular expressions are not equivalent), it randomly selects a distinguishing string ds from $\mathcal{U}$ using the function `pickAword`, and adds it to the set DSs (line 8). The mutant m is considered *covered* both when it is equivalent to r and when a test has been generated for it.

At the end of the algorithm, the set DSs, able to distinguish all the non-equivalent mutants, is returned.

The approach in Alg. 1 can produce big test suites, since it does not consider that a string could distinguish multiple mutants. Therefore, in the following we propose two improvements of the approach that permit to obtain smaller test suites, namely *monitoring* and *collecting*.

4.2. Monitoring approach

The monitoring technique is based on the observation that a string generated for distinguishing a mutant can accidentally also distinguish other mutants. Alg. 2 shows the modified algorithm. The technique keeps track of the previously generated distinguishing strings DSs and, after generating a mutant m, checks whether the mutant is distinguished by any $ds \in DSs$ (line 6); if this is the case, it does not compute a new distinguishing string (since m is already covered) and it continues from the next mutant.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Algorithm 2 Generation of a distinguishing set – Monitoring approach

Require: r: initial regular expression
1: $\mathcal{R} \leftarrow \texttt{toAutomaton}(r)$
2: $muts \leftarrow \texttt{mutate}(r)$
3: $DSs \leftarrow \emptyset$
4: **for** $m \in muts$ **do** ▷ Random iteration over the mutants
5: $\quad \mathcal{M} \leftarrow \texttt{toAutomaton}(m)$
6: $\quad$ **if** $\forall ds \in DSs\colon ds \notin \mathcal{L}(\mathcal{R} \oplus \mathcal{M})$ **then** ▷ monitoring
7: $\quad\quad \mathcal{U} \leftarrow \mathcal{R} \oplus \mathcal{M}$
8: $\quad\quad$ **if** $\mathcal{U} \neq \emptyset$ **then**
9: $\quad\quad\quad DSs \leftarrow DSs \cup \{\texttt{pickAword}(\mathcal{U})\}$
10: $\quad\quad$ **end if**
11: $\quad$ **end if**
12: **end for**
13: **return** DSs

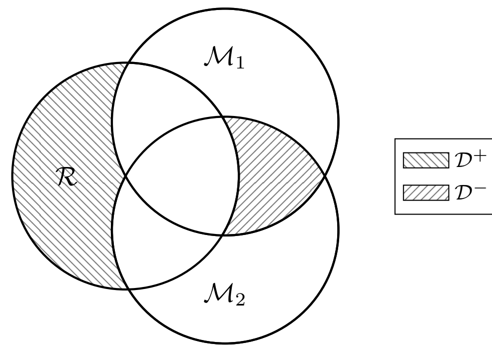


Figure 3. Positive and negative distinguishing automata

4.3. Collecting approach

Although the monitoring approach can obtain smaller test suites than the basic approach, it could still not obtain the best results, as the strings generated for some mutants could not be suitable to distinguish too many other mutants. Starting from this observation, the *collecting* approach aims at directly generating tests that can distinguish as many mutants as possible. 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:

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

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 *positive* and *negative distinguishing automata*. Fig. 3 shows an example of positive and negative distinguishing automata determined by two mutants m_1 and m_2 of a starting regular expression r. In this case, they can both be collected together in a positive or in a negative distinguishing automaton; note that it could be that only one kind of automaton is suitable for collecting them, or also none.

The collecting approach groups different mutants together in positive or negative distinguishing automata. 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.

The approach is shown in Alg. 3. The algorithm maintains a set of distinguishing automata DA (initially empty). For each mutant m, it tries to collect it, checking whether there exists a

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Algorithm 3 Generation of a distinguishing set – Collecting approach

```
Require: r: initial regular expression
 1: muts ← mutate(r)
 2: R ← toAutomaton(r)
 3: DA ← ∅
 4: for m ∈ muts do                                ▷ Random iteration over the mutants
 5:     M ← toAutomaton(m)
 6:     D′ ← ∅
 7:     for D ∈ DA do              ▷ Random iteration over the distinguishing automata
 8:         if isPos(D) then
 9:             M′ ← M^C
10:         else
11:             M′ ← M
12:         end if
13:         D′ ← D ∩ M′
14:         if D′ ≠ ∅ then
15:             D ← D′
16:             break
17:         end if
18:     end for
19:     if D′ = ∅ then
20:         for D ∈ {R ∩ M^C, R^C ∩ M} do     ▷ The two automata are considered randomly
21:             if D ≠ ∅ then
22:                 DA ← DA ∪ {D}
23:                 break
24:             end if
25:         end for
26:     end if
27: end for
28: DSs ← ∅
29: for D ∈ DA do
30:     DSs ← DSs ∪ {pickAword(D)}
31: end for
32: return DSs
```

distinguishing automaton $\mathcal{D}$ in DA able to distinguish $\mathcal{M}$ (the automaton of m); specifically, for each randomly selected automaton $\mathcal{D}$, the following instructions are executed:

- automaton $\mathcal{D}'$ is obtained as conjunction of $\mathcal{D}$ with $\mathcal{M}^{\complement}$ if $\mathcal{D}$ is positive, or with $\mathcal{M}$ otherwise (lines 8-13);
- if the conjunction is feasible (i.e., $\mathcal{D}'$ is not empty), $\mathcal{D}$ is updated with the conjunction and the search terminates (lines 14-16);

If neither $\mathcal{M}^{\complement}$ nor $\mathcal{M}$ can be conjuncted with any existing distinguishing automaton, the mutant must be collected in a new distinguishing automaton. The procedure tries to create a positive/negative distinguishing automaton (the polarity is randomly chosen) and, if not possible (i.e., the created automaton is empty), it tries to build the negative/positive automaton (lines 20 and 21). If both cannot be created, it means that r and m are equivalent; otherwise, the first automaton that can be created is added to DA (line 22).

After the iteration over all the mutants, each non-equivalent mutant has been added to a distinguish automaton. At the end, the algorithm builds the set of distinguishing strings DSs by taking a word from each distinguishing automaton (lines 29-30).

The collecting algorithm is computationally expensive. In the following, we propose two optimizations of the collecting process that should allow to reduce the computation time.

Algorithm 4 Generation of a distinguishing set – Collecting approach with QuitAfterN

Require: r: initial regular expression
Require: N: maximum number of collectable mutants

```
 4: for m ∈ muts do
 5:     M ← toAutomaton(m)
 6:     D′ ← ∅
 7:     for D ∈ DA do
 8:         if numOfMutsCollectedBy(D) = N then
 9:             continue
10:         end if
11:         ...
```

4.3.1. Parallel collecting This version of the collecting algorithm tries to optimize the computation time by parallelizing the collection of the mutants. The algorithm is similar to that shown in Alg. 3, except for the fact that the evaluation of mutants (i.e., the body of the for loop at line 4) is done in parallel. In order to avoid concurrency errors, accesses to set DA are synchronized and a distinguished automaton $\mathcal{D}$ is *locked* when the algorithm tries to add a mutant m to $\mathcal{D}$ (line 13). The number of mutants evaluated in parallel is determined by the number of available CPU cores.

4.3.2. Quit collecting after N (QuitAfterN) The collecting algorithm, in order to try to collect a mutant, always consider (in the worst case) all the previously generated distinguishing automata. However, as the number of mutants collected in a distinguishing automaton grows, the probability to collect new mutants in it decreases. The current version of the collecting algorithm tries to avoid checking too big automata (i.e., automata that have already collected several mutants). A distinguishing automaton is not considered any more when it has collected N mutants, being N a parameter of the algorithm. The modification of the collecting algorithm is shown in Alg. 4. At lines 8-9, if $\mathcal{D}$ collects N mutants, $\mathcal{D}$ is skipped.

5. COMPUTING THE MUTATION SCORE OF A SET OF STRINGS

The mutants introduced in Sect. 3 can also be used to assess the *quality* of a generic set of string S, in terms of its ability to expose possible faults contained in the regular expression under test. As in classical mutation testing, to measure the quality of a test suite, we use the notion of killed mutant and mutation score. We introduce the following definitions.

Definition 6 (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:

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

Example 2
If r is `[a-zA-Z]` and the CCR mutant m is `[a-z]`, then the string “A” distinguishes m, while the string “a” does not. The CCR mutant is killed by “A”.

Definition 7 (Mutation score)
The *mutation score* of a set of strings S for a regular expression r is defined as follows:

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

The mutation score of S w.r.t. to a regular expression r is the ratio of non-equivalent mutants that are killed by at least a string in S. We can compute this ratio by 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),

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Algorithm 5 Computation of the mutation score

Require: r: a regular expression
Require: S: set of strings
1: $muts \leftarrow \texttt{mutate}(r)$
2: $killedMuts \leftarrow \emptyset$
3: **for** $m \in muts$ **do**
4: **if** $\exists s \in S:\ killed(r, m, s)$ **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** $|killedMuts|/|muts|$

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 proposed test generation approaches (see Algs. 1, 2, and 3) guarantee to generate test suites having mutation score 1 (by definition of distinguishing string) if no timeout is imposed to the test generation.

The set of strings S for which we are interested in computing the mutation score could come from different sources:

- Although in the previous section we proposed two policies (QuitAfterN and parallel collecting) that should optimize the collecting process, this could still be computationally expensive. If the generation process is subjected to time constraints (e.g., a timeout), the final test suite could not cover all the mutants and, therefore, we could be interested in evaluating the mutation score of such set of strings.
- S could have been obtained using some tools for generating tests for regular expressions, as EGRET [6], EXREX [19], Generex [20], and regldg [21]. We will compare the mutation score of different tools to that of our approach (subjected to a fixed timeout) in Sect. 7.3.
- S could be the set of strings from which the user synthesized the regular expression r (for example, by the technique presented by Li et al. [11]). In this case, the mutation score of S w.r.t. r tells how well S represents r; a low value means that probably S is not sufficient to guess the right regular expression and more labeled strings should be added to S in order to synthesize a better regular expression.

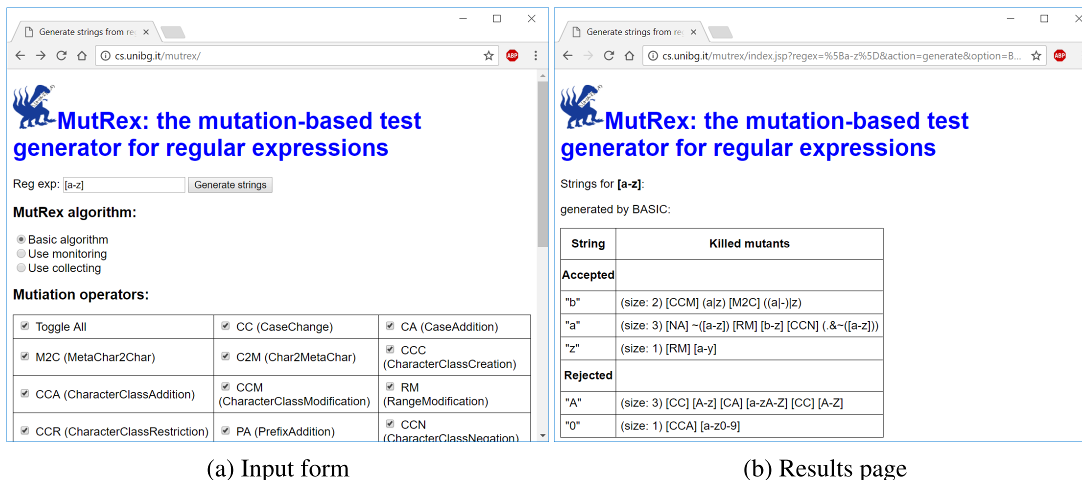
6. MUTREX

The test generation algorithms described in Sect. 4 have been implemented in the tool MUTREX[†] generating fault-detecting strings that can be used for different purposes as described in Sect. 1.

In addition, for easing the regular expression validation by the developer, a web service has been implemented[‡]. Fig. 4a shows the interface for submitting a regular expression to the web service. The user can:

- select one of the three generation approaches (basic, monitoring, and collecting);
- select mutation operators to be used during test generation (this could lead to smaller test suites and a shorter string generation time);
- indicate a preference towards strings that are either accepted or rejected by the regular expression under test. In many cases, there exist a distinguishing string that is accepted

[†]Code is available at `https://github.com/fmselab/mutrex/`.
[‡]The tool is available as web service at `http://foselab.unibg.it/mutrex/`.



(a) Input form (b) Results page

Figure 4. MUTREX web service

and one that is rejected. The user may prefer one type over the other and MUTREX tries to `pickAword` accordingly.

Fig. 4b shows the page reporting the generated strings. Strings are divided between those accepted by the original regular expression and those rejected; moreover, for each string s, the list of mutants killed by s is reported.

7. EXPERIMENTS

We here evaluate the effectiveness of the proposed approach. We first describe in Sect. 7.1 the benchmark set and analyze which kind of mutants are obtained using the proposed mutation operators. Then, in Sect. 7.2, by means of a series of research questions, we evaluate the performance in terms of test suite size and generation time when no time constraints are imposed. Finally, in Sect. 7.3, we evaluate how much the mutation score of our approach is affected by time constraints: in order to do this, we compare our approach with other test generation tools.

7.1. Benchmarks and mutants statistics

As benchmarks[§], we have collected 170 regular expressions of different kinds as, for example, those used to detect URLs, e-mail addresses, credit card numbers, phone numbers, car identification numbers, dates, or car plates. The benchmark set has been built has follows:

- 51 have been retrieved from different sources, as books [18], websites, and forums[¶];
- 119 have been selected among those reported by Chapman and Stolee [1] that surveyed regular expressions used in Python projects. We first identified those supported by our framework and we grouped them according to their length, since we want to focus on how the size affects MUTREX performances. We finally selected in each group the regular expression with the highest number of operators, because we want to maximize the coverage of operators.

Fig. 5 shows the size of each regular expression in terms of length and number of operators (as described in Table I). The benchmark set contains regular expressions of different complexity: both small regular expressions (long less than 100 characters and having less than 50 operators) and big

[§] All benchmarks can be downloaded from `http://foselab.unibg.it/mutrex/SImutation2017experiments.txt`

[¶] `http://www.regexlib.com`, `http://regexr.com/`, `http://www.regular-expressions.info`, `http://tusker.org/regex`

Copyright © 0000 John Wiley & Sons, Ltd.

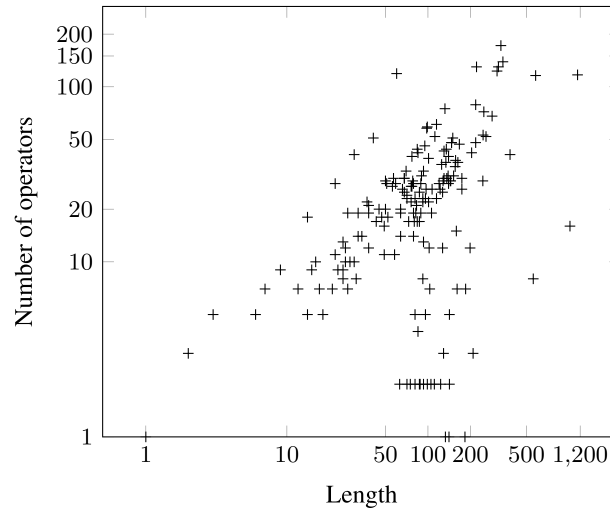
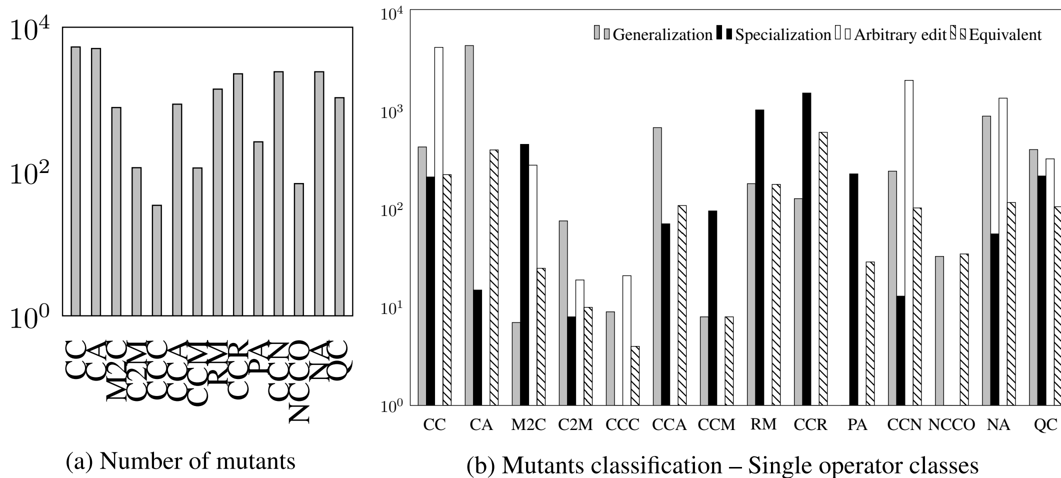


Figure 5. Benchmarks size



(a) Number of mutants

(b) Mutants classification – Single operator classes

Figure 6. Mutants

regular expressions (longer than 100 characters and with around 150-200 operators). We can see that the number of operators grows with the length of the regular expression; however, there are also long regular expressions with few operators.

For each regular expression of the benchmark set, we generated a set of mutants using the operators described in Sect. 3. We now analyze which kinds of mutants have been generated.

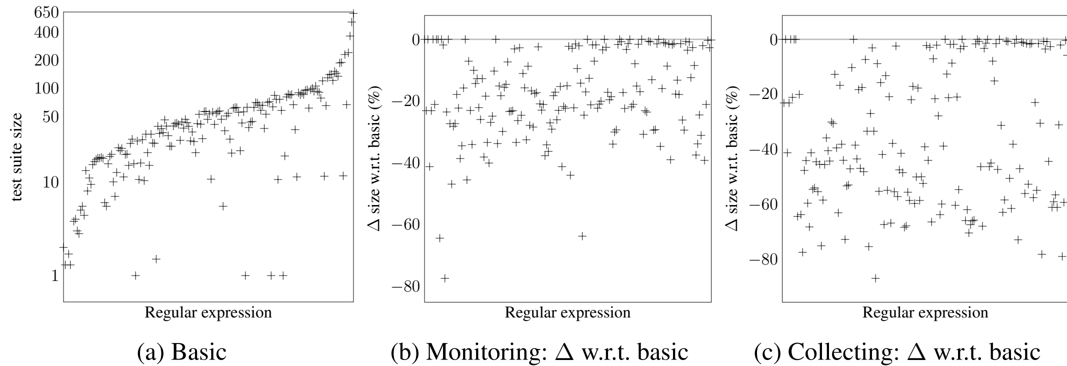
How many mutants do mutation operators generate? Fig. 6a shows, for each mutation operator, the number of mutants generated for all benchmarks. CC and CA (modifying the case in a character class), CCR (removing an interval from a character class), CCN (negating a character class), and NA (adding a negation where possible) are the operators that produce more mutants, since they modify elements (as character classes) that are widely used in regular expressions. Instead, other mutators affecting less used regular expression operators, produce few mutants; for example, CCC creates a character class when the user forgets to use square brackets, that is an uncommon situation. NCCO, instead, applies to negated character classes that are not frequently used by developers.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table V. Mutants classification – All operators

	Generalization	Specialization	Arbitrary edit	Equivalent	Total
Sum	7742	3931	8517	1960	22150
Ratio	34.95%	17.75%	38.45%	8.85%	



(a) Basic (b) Monitoring: Δ w.r.t. basic (c) Collecting: Δ w.r.t. basic

Figure 7. Test suite size (regular expressions are sorted in increasing order of number of mutants)

How do mutation operators modify the languages accepted by the regular expressions? Fig. 6b shows, for each mutation class, the classification of its mutants in generalizations, specializations, arbitrary edits, and equivalent mutants (see Sect. 3.4). As expected, PA only produces specializations or equivalent mutants: indeed, adding a prefix to a regular expression can not enlarge the set of accepted strings. In our experiments, NCCO (making a negated character class optional) only produces generalizations, when the mutant actually also accepts strings having no character in the place of the negated character class, or equivalent mutants, when the negated character class is already included in an outer optional block (e.g., it is part of a subexpression quantified with *). Table V reports the total number of mutant types, and their percentage over all the mutations. Most of the mutants are arbitrary edits, meaning that mutation operators tend to add constraints that both add and remove words from the language. More than one third of the mutants are generalization: indeed, several operators relax the constraints of the regular expression, such that the accepted language is enlarged. The number of specializations is lower (17.75%), as the proposed mutation operators do not restrict constraints so often. Equivalent mutants are not a problem for regular expression testing: first of all, detecting equivalent mutants is decidable in this context, and, secondly, their number (8.85%) is comparable to that measured in other contexts [22, 23].

7.2. *Evaluation of generation techniques*

We here assess the effectiveness of the proposed generation techniques.

All the experiments have been executed on a Linux PC with 2 Intel(R) Xeon CPUs (@2.3 Ghz and 12 cores), and 64 GB of RAM. Each experiment have been executed 10 times (unless stated otherwise); the reported values are the averages among the different runs.

RQ1 What is the size of the generated fault detecting test suites?

Fig. 7a reports, for each regular expression (sorted in increasing order of number of mutants), the sizes of the test suites generated by the *basic* approach shown in Alg. 1; the size of the test suite grows linearly with the number of mutants of the regular expression under test. Fig. 7b shows, for each regular expression, the percentage change $\Delta = \frac{M-B}{B}$ of the size (M) of the test suite generated by the *monitoring* approach (shown in Alg. 2) w.r.t. the size (B) of the test suite generated by the basic approach. Similarly, Fig. 7c show the percentage change of the *collecting* approach (shown in Alg. 3) w.r.t. the basic approach.

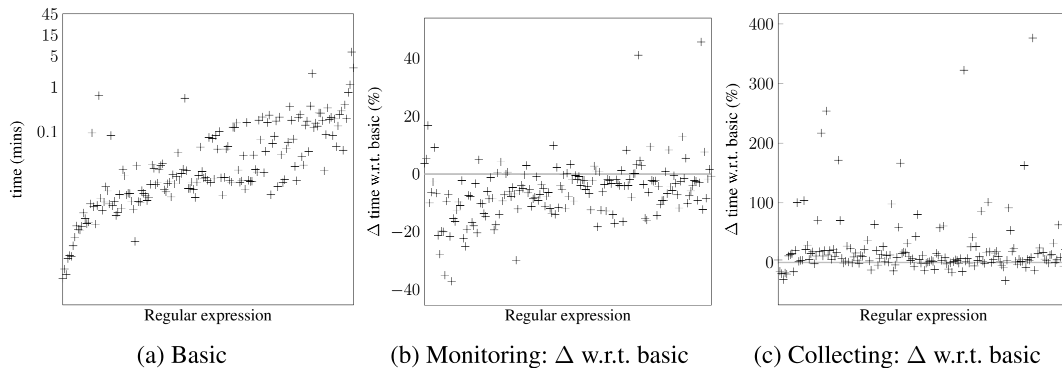
Table VI reports the total size of the test suites for all the benchmarks. The whole test suite

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table VI. Comparison of techniques

Technique	Size	Time (mins)
Basic	9296.2	22.81
Monitoring	8094.7	22.36
Collecting	6668.94	114.85
ParallelCollecting	7177.4	13.15
QuitAfterN ($N = 2$)	8665.8	32.42
QuitAfterN ($N = 3$)	7983.8	33.03
QuitAfterN ($N = 4$)	7583	31.1
QuitAfterN ($N = 5$)	7338.8	30.76
QuitAfterN ($N = 6$)	7177.4	30.18
QuitAfterN ($N = 7$)	7059	30.31
QuitAfterN ($N = 8$)	6981.4	32.34
QuitAfterN ($N = 9$)	6923.4	32.02
QuitAfterN ($N = 10$)	6875	30.17
QuitAfterN ($N = 15$)	6770.2	31.28
QuitAfterN ($N = 20$)	6731.6	35.84
QuitAfterN ($N = 25$)	6711	47.33
QuitAfterN ($N = 30$)	6697.4	66.06
QuitAfterN ($N = 35$)	6686	54.29
QuitAfterN ($N = 40$)	6684.6	74.55
QuitAfterN ($N = 45$)	6684.2	89.23
QuitAfterN ($N = 50$)	6679.8	96.73
QuitAfterN ($N = 55$)	6681	108.23
QuitAfterN ($N = 60$)	6676	106.14
Basic+MinTest	7748	42.36



(a) Basic (b) Monitoring: Δ w.r.t. basic (c) Collecting: Δ w.r.t. basic

Figure 8. Generation time (regular expressions are sorted in increasing order of number of mutants)

obtained with monitoring is 12.92% smaller than that obtained with basic, and that obtained with collecting 17.61% smaller than that obtained with monitoring (28.26% smaller w.r.t. basic): this demonstrates that collecting is very effective in building compact test suites. By Figs. 7b and 7c, we observe that monitoring and collecting are able to reduce the test suite size of almost all the regular expressions and never increase it.

RQ2 How long do the generation techniques take?

Fig. 8a reports, for each regular expression (sorted in increasing order of number of mutants), the time taken for generating test suites using the basic approach. As expected, the time grows with

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

the number of mutants of the regular expression. Fig. 8b and 8c show the percentage change of the generation time of the monitoring and collecting approaches w.r.t. the basic approach.

Table VI reports the total time over all the benchmarks; monitoring is slightly faster than basic (1.97% faster), while collecting is both slower than basic (403.53% slower) and than monitoring (413.67% slower). By Fig. 8b, we observe that monitoring reduces the time for most of the regular expressions, but it also increases it for few ones. By Fig. 8c, we notice that collecting increases the generation time for most of the regular expressions, although for some of them it is able to decrease it. Note that, for readability reasons, Fig. 8c does not show five expressions for which the percentage change is greater than 400%; the total collecting time (see Table VI) is greatly influenced by these five expressions; without them, collection would have a delta increase in time of around 32%.

RQ3 Is monitoring effective?

Monitoring is able to always produce smaller test suites in less time than basic, since checking whether a test distinguishes a new mutant is less expensive than generating a new test for it. Therefore, it makes sense to always apply monitoring.

However, w.r.t. collecting, monitoring produces bigger test suites (but in less time). Therefore, it should be preferred to collecting when generation time is a strict constraint.

RQ4 Is collecting effective?

Collecting is the most effective technique in terms of size, but it is also the slowest one. The bottleneck of the approach is the addition of a new mutant automaton to a distinguishing automaton (line 13 in Alg. 3). The two proposed collecting policies, *parallel collecting* and *QuitAfterN*, aim at reducing the computation time, as shown in the next research question.

RQ5 Which is the effect of the collecting policies (parallel collecting and QuitAfterN)?

Table VI reports the results obtained by the *parallel collecting* technique described in Sect. 4.3.1 that parallelizes the collection of the mutants. The technique is very effective, as it can complete the collecting process in around a tenth of the time required by normal collecting. Note that the size is bigger with parallel collecting than with collecting (but still smaller than that obtained with monitoring); this is due to the fact that parallel collecting, as it considers multiple mutants in parallel, may create some unnecessary distinguished automata for mutants that could be collected together with some mutants that are evaluated in parallel.

We now evaluate the effect of the collecting option *QuitAfterN* described in Sect. 4.3.2 that stops considering a distinguished automaton when it has already collected N mutants. Fig. 9 shows how the test suite size and the generation time change with the increase of the collection threshold (data for some values of N are also shown in Table VI). As expected, the test suite size decreases as N increases, while the generation time increases. For small values of N, the size rapidly decreases, but for N greater than 15 the reduction in test suite size is minimal. From Table VI, we can see that the size converges to the value obtained by collecting when N approaches 60, but the generation time still remains smaller. This is due to the fact that probably no distinguished automaton collects more than around 60 mutants and, therefore, the collecting approach wastes computation time in trying to collect mutants that cannot be collected in existing distinguished automata, instead of creating a new one.

RQ6 How far is the solution computed by collecting from the optimal one?

The collecting process allows to generate the most compact test suites and, in principle, should be able to obtain the minimum test suite [24]. However, the approach depends on the order in which the mutants are evaluated (line 4 in Alg. 3) and in which the distinguished automata are considered during collecting (line 7 in Alg. 3). Therefore, a run of the collecting algorithm may not find the *optimal* (i.e., minimum) test suite. We here want to assess how much the test suites computed by collecting are bigger than the optimal one. We have run the collecting algorithm 50 times for all the benchmarks. For each regular expression, we have identified, among the 50 runs, the smallest test suite size: such value is likely very close to the minimum value. The sum of all the smallest values for all the regular expressions is taken as *optimal* value of the global test suite size. Finally, we

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

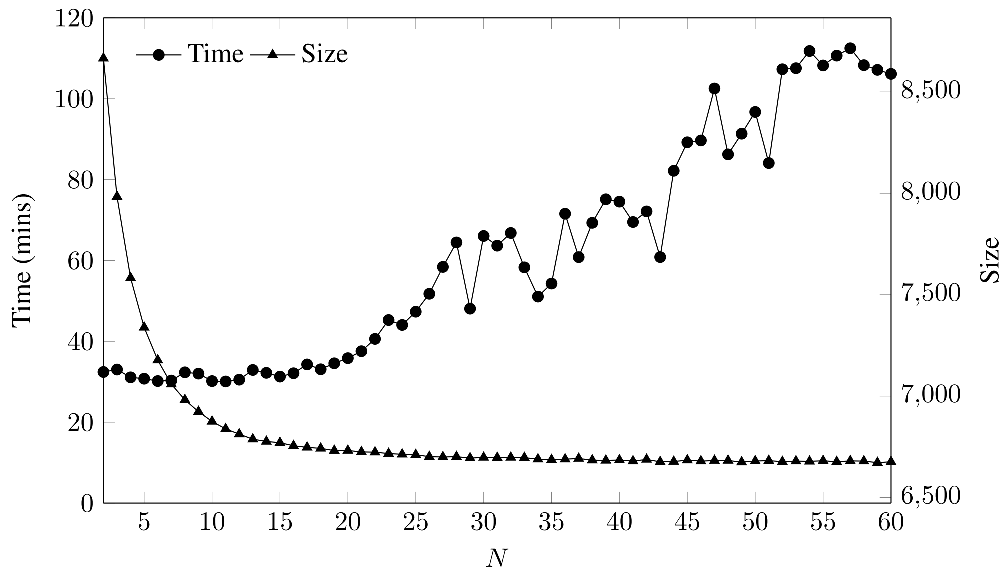


Figure 9. Quit collecting after N

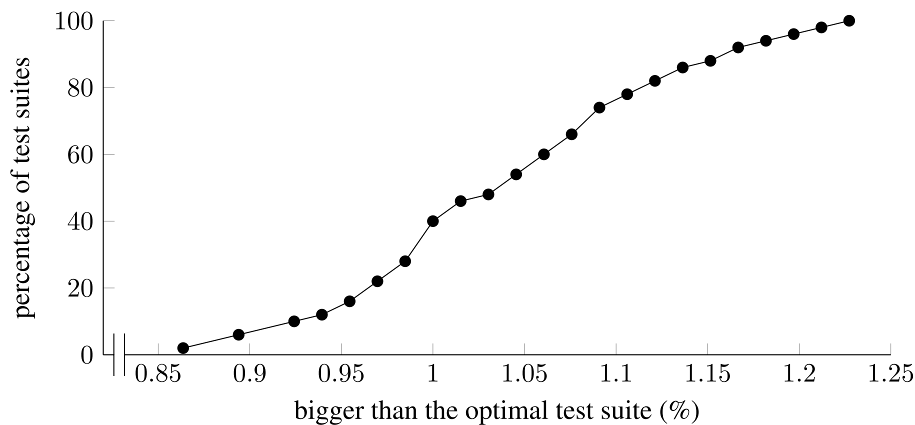


Figure 10. Optimality of collecting

have checked how much, in the different runs, the global test suite sizes are bigger than the optimal one. Fig. 10 shows the cumulative distribution of the percentage difference of the global test suite size, for every run, w.r.t. the optimal value. Each global test suite is at least 0.86% bigger than the optimal test suite; 40% of the test suites are at most 1% bigger; no test suite is greater than 1.23%. This means that, although the order in which mutants and distinguished automata are considered can affect the size of the final test suite, such effect is not so big.

RQ7 How do the proposed generation techniques compare with classical minimization techniques?

Monitoring and collecting aim at generating test suites more compact than those obtained with the basic approach, but still achieving the same mutation score. Other approaches as *MinTest* [24, 25], instead of trying to obtain compact test suites during generation, minimize a given test suite (i.e., after generation) without reducing its mutation score.

We are therefore interested in knowing whether our optimization techniques give any advantage (in terms of size and time) w.r.t. MinTest (as described by Ammann et al. [24]). We have executed the basic approach and applied the MinTest procedure to the obtained test suites (we call the approach

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

as *Basic+MinTest*); we run the experiment 10 times and computed the averages of the minimized test suite size and of the execution time (including the generation time and the minimization time): the average test suite size is 7748, and the average execution time is 42.36 mins. The Basic+MinTest approach can obtain smaller test suites than monitoring, but in almost the double of time (see Table VI); indeed, the computational complexity of MinTest is $\Theta(|DSs| \cdot |muts|)$ [24], while the computational complexity of the monitoring part (i.e., without generation) of the monitoring approach is $O(|DSs| \cdot |muts|)$. The collecting approach performs much better than Basic+MinTest in terms of size: this means that trying to generate a test covering multiple mutants is much more effective in terms of size than checking whether an already generated test covers new mutants (as done in monitoring and MinTest). As we already observed, collecting is computationally expensive and takes much more time than the Basic+MinTest approach; however, QuitAfterN, with $4 \leq N \leq 22$, can obtain smaller test suites in less time than Basic+MinTest.

7.3. Comparison with other test generation tools

We are interested in comparing MutRex with other string generator tools. As comparison, we have considered tools that are freely available (possibly open source), updated, well-documented, and running on Linux; we have selected the following tools:

1. **EGRET** (Evil Generation of Regular Expression Tests) which is a very recent tool [6] and it promises to generate the most "evil" strings. It has a web interface‖ and it can be compiled as a standalone C program.
2. **EXREX** [19] is a command line tool written in Python that systematically generates all matching strings to a given regular expression.
3. **Generex** [20] is a Java library for generating strings that match a given regular expression. It is based on the same library [15] of MutRex. It can generate strings systematically (**GenerexS**) or randomly (**GenerexR**).
4. **regldg** [21] is a fast program written in C which systematically generates strings matching a given regular expression.

Since EXREX, Generex, and regldg generate all the strings matching a regular expression and such number can be infinite in the presence of quantifiers, we have limited the number of generated strings to 250. Regarding MutRex, we have tested four versions: **MutRexB** for the basic approach, **MutRexM** for the version using monitoring, **MutRexC** for the collecting version, and **MutRexP** for the parallel collecting.

We have executed all the tools above with all the regular expressions in the benchmark set for 10 times with a timeout of 5 seconds. At the end, we have measured the **size** as the number of generated strings, the **time** as the time taken in milliseconds to generate the strings for a single regular expression, and the **mutation score** as defined in Sect. 5.

Note that in the presence of timeout, all the tools, including the MutRex versions, could not terminate, so the mutation score even for MutRex could be less than 100%. Moreover, a tool could be unable to generate any string. We measure this event by the **GI** (generation inability) which identifies the number of times a tool has generated no string over the number of times it has been called (e.g., GI $= 0$ means no failure in generating). In this way, we are able to discover how our approach compares (also in terms of mutation score) to other test generation tools when a fixed timeout is imposed on all the tools. The results of the experiments are shown in Fig. 11.

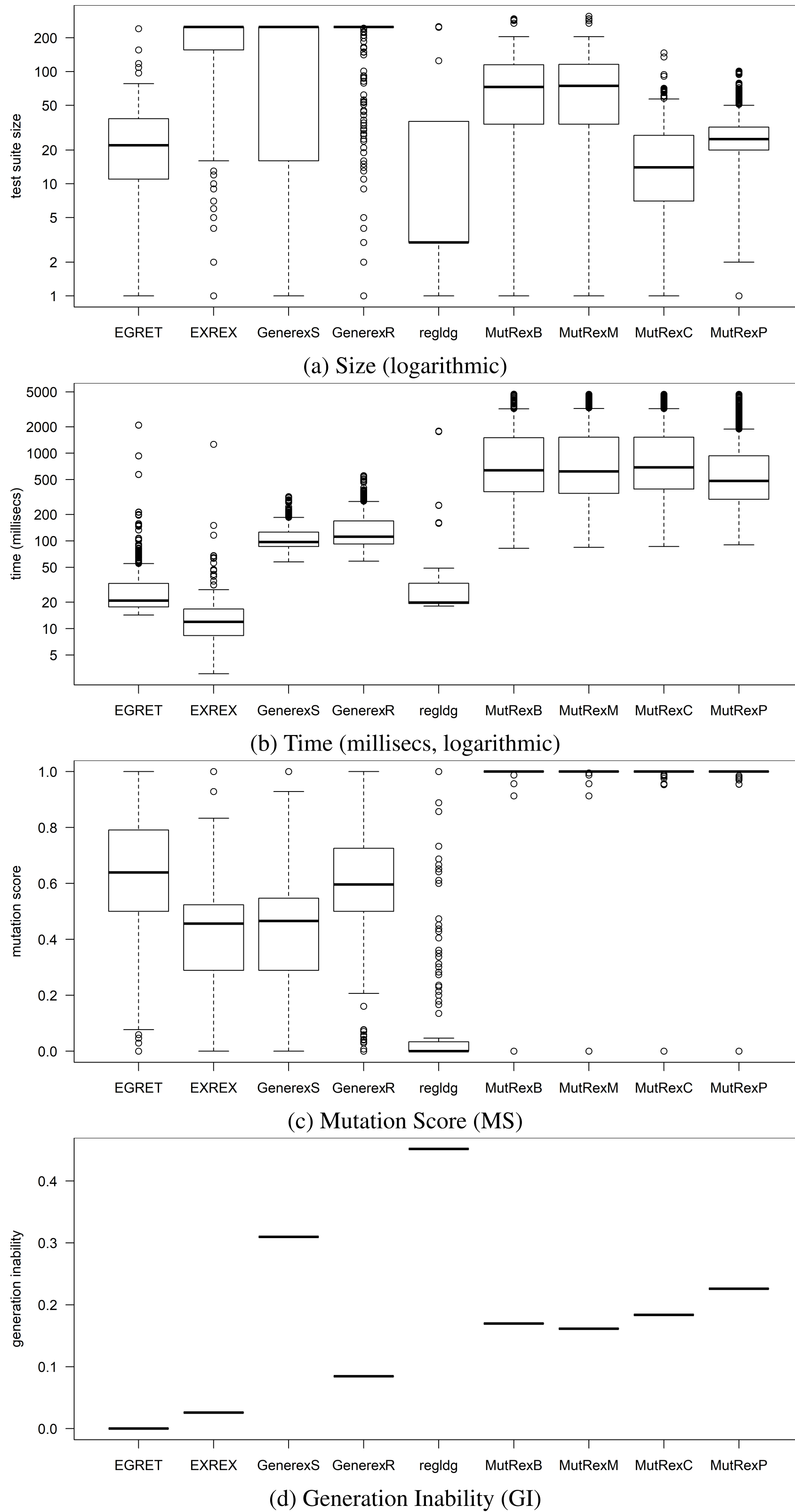
Regarding the size (Fig. 11a), regldg produces the smallest test suites, with MutRex in the collecting version the second best, followed by the EGRET and the other MutRex versions. The other tools produce very large test suites (very often as big as the limit of 250).

On the other hand, MutRex versions are the slowest of the group (Fig. 11b). The average, however, is around 1 second, which is reasonable, especially in case the consumer of the generated strings is a human. EGRET, EXREX, and regldg are the fastest ones.

‖`http://elarson.pythonanywhere.com/`

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



(a) Size (logarithmic)

(b) Time (millisecs, logarithmic)

(c) Mutation Score (MS)

(d) Generation Inability (GI)

Figure 11. Comparison with other test generation tools with timeout of 5 seconds

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Table VII. A qualitative comparison (+++ means best and --- means worst)

feature	EGRET	EXREX	GenerexS	GenerexR	regldg	MutRexB	MutRexM	MutRexC	MutRexP
size	++	---	---	---	+++	+	+	++	++
time	+++	+++	++	++	+++	--	--	---	-
MS	+	-	-	+	---	+++	+++	+++	+++
GI	+++	+	---	+	---	-	-	-	–

The mutation score of the MUTREX variants is almost always 100% (to be more precise the average is 99.2% for MutRexB, 99.2% for MutRexM, 99.1% for MutRexC, and 99.1% for MutRexP). The only two others with an average greater than 50% are EGRET (59.6%) and Generex in the random version (57.8%). Note that for Generex, the random version produces better tests than those obtained by systematic sampling of the regular expression. regldg has a very low mutation score (probably due to the smallness of the test suites).

Fig. 11d shows the generation inability of all the tools. regldg fails quite often (around 40%), while MUTREX fails around 19% of the times to produce a test set; MutRexM is the best of our generation algorithms (around 16% of failures). Only EGRET proved to be able to generate a test suite for each regular expression.

Table VII reports a brief qualitative analysis of the tools we have compared. All the tools except EGRET and MUTREX produce very big test suites or have other deficiencies (like high GI for regldg). EGRET is the fastest and produces quite compact string sets of sufficient quality (but many faults can pass undetected). MUTREX is much slower than EGRET. However, only MUTREX is able to generate actual fault detecting string sets, which in some cases (like for MutRexC) are rather compact and this makes MUTREX particularly suitable for string generation when the evaluation of the strings is performed manually. This proves, as expected, that explicitly targeting faults leads to test suites with greater mutation score, and that the faults we have identified in Sect. 3 are not already captured by existing tools.

8. THREATS TO VALIDITY

We have identified the following threats to the validity of the proposed approach.

Regarding external validity [26], it could be that the obtained results can not be generalized to all regular expressions. However, in the selection of the benchmarks, we have tried to select regular expressions from different sources (websites, forums, books, and benchmarks reported in other papers [1]) and of different sizes; therefore, we believe that the drawn conclusion can be applied to all regular expressions that use the same operators we consider (see Table I). Indeed, as said in Sect. 2, we support a limited set of regular expressions, i.e., those describing a regular language (that can be represented as automata using brics that, as reported by Chapman and Stolee [1], supports a limited set of regular expression operators); as future work, in order to support a wider set of regular expressions, we plan to use a different representation of regular expressions (e.g., a symbolic representation as in Rex [12]). It could be that, using a different representation, the results in terms of generation time and scalability would be different; the size of the obtained test suites, instead, should not be affected by the underling representation.

Regarding internal validity [26], we have carefully checked that the improvements of the optimization techniques depend only on the techniques themselves and not on some other factor. It could be that monitoring and collecting produce smaller test suites due to some implementation error; in order to avoid this situation, we checked that each produced test suite actually covered all the mutants.

Regarding construct validity [26], some actual faults of regular expressions could be not captured by our approach. First of all, proposed mutation operators could be not representative of real faults; however, we designed the proposed operators surveying websites, forums, and books reporting the more common faults made by developers. Moreover, the coupling effect may not hold for regular expressions and tests generated for capturing simple faults are not able to capture more complex

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

faults; in order to verify this, as future work, we plan to check whether tests generated for first order mutants are also able to kill higher order mutants.

9. 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 [17, 27, 28, 29, 30, 31]. In particular, Offutt et al. [32] argue that mutation analysis is a test generation method that creates 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 regular expressions perfectly reflects this concept of grammar-based testing.

Although no specific mutation operators have ever been defined for regular expressions, there exist approaches based on suitable (grammar) transformations, some on regular expressions, some on strings accepted by regular expressions, which resemble mutation. Among these approaches, Li et al. [11] introduce ReLIE, a learning algorithm that, starting from a plausible initial regular expression and a set of labeled strings, tries to learn the correct regular expression. ReLIE works on a set of regular expression transformations, a sort of regular expression mutation, to obtain a set of candidate regular expressions. Similarly, MUTREX allows to learn the intended regular expression by a set of strings accepted by the mutated regular expression.

Closer to our approach and based on string transformations, is the approach presented by Larson and Kirk [6] that promises to generate *evil* strings and implemented in EGRET. EGRET takes a regular expression as input and generates two lists of test strings: strings *accepted* by the regular expression, 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 regular expression 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 regular expressions into nondeterministic finite state automata, generating strings able to expose possible common mistakes, involving the user as oracle to decide regular expression correctness. Furthermore, EGRET generation of evil strings applies mutation on strings accepted by the regular expression (differently from MUTREX that mutates the regular expression): altering the number of iterations for each repeat operator and changing the character used for a character set. MUTREX performs better than EGRET in two directions. First, in terms of mutation score, MUTREX can guarantee the detection of faults, while by EGRET, especially for long regular expressions, faults can pass undetected if one relies on EGRET evaluation (see Sect. 7.3). Second, MUTREX can help in fault *localization*: when a user detects a faulty string, (s)he also knows the error made in writing the regular expression (that is the error induced by the mutation operator).

Several tools, like those presented in Sect. 7.3, have been developed for testing regular expressions. They are based on the exhaustive generation of strings that match a given regular expression. Some tools allow also random selection of strings, and many of them are available on line. However, as proved by our experiments, these tools tend to generate large string sets, especially in the presence of quantifiers and cannot guarantee any fault detection capability. MUTREX can be used in combination with these tools to evaluate the mutation score of the set of generated strings, as discussed in Sect. 7.3.

Other approaches have been defined for testing purposes, still based on strings generation, which exploit the symbolic automaton representation. Veanes et al. [12] have developed Rex, a general-purpose solver of regular expressions constraints. Rex translates a given regular expression 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/regular expression: the model represents an accepted string.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Rex is used in combination with Pex [10] for dealing with regular expression constraints in parameterized unit tests. Reggae [9] is a tool to generate string test inputs that are accepted by a given regular expression. The goal is to perform branch coverage for programs that use complex string operation including regular expression operations. It exploits dynamic symbolic execution based on path exploration. Kiezun et al. [33] propose HAMPI, a solver for string constraints over bounded string variables. Users of HAMPI specify constraints using regular expressions, 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.

10. CONCLUSIONS AND FUTURE WORK

We have presented a fault-based test generation approach for regular expressions. We have defined some fault classes (and their corresponding mutation operators) that represent mistakes that could be made when writing a regular expression; then, we have proposed a test generation approach for building fault detecting test suites, i.e., test suites able to expose all the possible faults contained in the regular expression under test. We have proposed three versions of the approach (*basic*, *monitoring*, and *collecting*) that provide different results in terms of test suite size and generation time. The *collecting* approach permits to obtain the most compact test suites, but it is computationally expensive: for this reason, we have also proposed two optimizations of this approach that allow to limit the computation time. Exploiting the definition of fault classes, we have also proposed a notion of fault coverage that can be used to evaluate test suites coming from different sources (for example, obtained through other test generation tools).

The proposed mutation operators are *semantic* ones, in the sense that they try to mimic the faults done by programmers. As future work, we plan to compare them with more simple *syntactic* operators (i.e., simply mutating single characters), in terms of generation time, test suite size, and mutation score.

We have compared MUTREX with all the other tools considering the fault classes all together; as future work, we plan to extend the comparison by considering fault classes singularly in order to discover the correlation between faults and test sets generated by different tools. For example, we could discover that an existing tool consistently detects faults of a given class. We found that MUTREX outperforms the other tools in terms of mutation score but it sometimes underperforms in terms of time and size. We could compare MUTREX with other tools on the same killed faults and we might find that, mutation score being equal, it performs better than other tools in terms of size and time.

Moreover, as future work we also plan to devise a technique that should be able not only to detect a fault, but also to repair the regular expression and remove the fault, or to improve evaluation performance [34]. Then, in order to further speed up the generation process, we plan to study whether approaches that consider mutants as part of a family [35], are applicable in our context. We also 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. Our approach relies on the representation of regular expressions as automata: this limits the class of regular expressions that we can support (only those describing a regular language). We plan to study whether our approach can be adapted to use a different regular expression representation (e.g., a symbolic representation [12]) that would allow us to represent a wider class of regular expressions.

ACKNOWLEDGEMENT

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

REFERENCES

Copyright © 0000 John Wiley & Sons, Ltd.

1. Chapman C, Stolee KT. Exploring regular expression usage and context in Python. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, ACM: New York, NY, USA, 2016; 282–293, doi:10.1145/2931037.2931073. URL http://doi.acm.org/10.1145/2931037.2931073.
2. The Bro network security monitor 2015. https://www.bro.org/.
3. Yeole AS, Meshram BB. Analysis of different technique for detection of SQL injection. *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, ACM: New York, NY, USA, 2011; 963–966, doi:10.1145/1980022.1980229. URL http://doi.acm.org/10.1145/1980022.1980229.
4. Arslan AN. Multiple sequence alignment containing a sequence of regular expressions. *2005 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, 2005; 1–7, doi:10.1109/CIBCB.2005.1594922.
5. Spishak E, Dietl W, Ernst MD. A type system for regular expressions. *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, ACM: New York, NY, USA, 2012; 20–26, doi:10.1145/2318202.2318207. URL http://doi.acm.org/10.1145/2318202.2318207.
6. Larson E, Kirk A. Generating evil test strings for regular expressions. *Software Testing, Verification and Validation (ICST), 2016 IEEE 9th International Conference on*, 2016.
7. Beck F, Gulan S, Biegel B, Baltes S, Weiskopf D. RegViz: Visual debugging of regular expressions. *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, ACM: New York, NY, USA, 2014; 504–507, doi:10.1145/2591062.2591111. URL http://doi.acm.org/10.1145/2591062.2591111.
8. Erwig M, Gopinath R. *Explanations for Regular Expressions*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2012; 394–408, doi:10.1007/978-3-642-28872-2_27. URL http://dx.doi.org/10.1007/978-3-642-28872-2_27.
9. Li N, Xie T, Tillmann N, Halleux Jd, Schulte W. Reggae: Automated test generation for programs using complex regular expressions. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, 2009; 515–519.
10. Tillmann N, de Halleux J. Pex–white box test generation for .NET. *Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, Beckert B, Hähnle R (eds.), Springer Berlin Heidelberg: Berlin, Heidelberg, 2008; 134–153, doi:10.1007/978-3-540-79124-9_10. URL https://doi.org/10.1007/978-3-540-79124-9_10.
11. Li Y, Krishnamurthy R, Raghavan S, Vaithyanathan S, Jagadish HV. Regular expression learning for information extraction. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, Association for Computational Linguistics: Stroudsburg, PA, USA, 2008; 21–30. URL http://dl.acm.org/citation.cfm?id=1613715.1613719.
12. Veanes M, Halleux Pd, Tillmann N. Rex: Symbolic regular expression explorer. *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, IEEE Computer Society: Washington, DC, USA, 2010; 498–507, doi:10.1109/ICST.2010.15. URL http://dx.doi.org/10.1109/ICST.2010.15.
13. Becchi M, Wiseman C, Crowley P. Evaluating regular expression matching engines on network and general purpose processors. *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ACM, 2009; 30–39.
14. Arcaini P, Gargantini A, Riccobene E. MutRex: A mutation-based generator of fault detecting strings for regular expressions. *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017; 87–96, doi:10.1109/ICSTW.2017.23.
15. Møller A. dk.brics.automaton – finite-state automata and regular expressions for Java 2010. http://www.brics.dk/automaton/.
16. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* Sep 2011; **37**(5):649–678, doi:10.1109/TSE.2010.62. URL http://dx.doi.org/10.1109/TSE.2010.62.
17. Woodward MR. Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal* Jul 1993; **8**(4):211–224.
18. Friedl JEF. *Mastering regular expressions (3. ed.)*. O'Reilly, 2006.
19. Tauber A. "exrex" 2017. URL https://github.com/asciimoo/exrex.
20. Mifrah Y. Generex 2017. URL https://github.com/mifmif/Generex.
21. Cronin P. "regldg" 2017. URL https://regldg.com/.
22. Schuler D, Zeller A. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability* 2013; **23**(5):353–374, doi:10.1002/stvr.1473. URL http://dx.doi.org/10.1002/stvr.1473.
23. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability* 1997; **7**(3):165–192, doi:10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U. URL http://dx.doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U.
24. Ammann P, Delamaro ME, Offutt J. Establishing theoretical minimal sets of mutants. *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, IEEE Computer Society: Washington, DC, USA, 2014; 21–30, doi:10.1109/ICST.2014.13. URL http://dx.doi.org/10.1109/ICST.2014.13.
25. Gopinath R, Alipour A, Ahmed I, Jensen C, Groce A. Measuring effectiveness of mutant sets. *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016; 132–141.
26. Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.
27. De Souza SDRS, Maldonado JC, Fabbri SCPF, De Souza WL. Mutation Testing Applied to Estelle Specifications. *Software Quality Control* December 1999; **8**:285–301, doi:10.1023/A:1008978021407.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

28. Arcaini P, Gargantini A, Riccobene E. Using mutation to assess fault detection capability of model review. *Software Testing, Verification and Reliability* 2015; **25**(5-7):629–652, doi:10.1002/stvr.1530.
29. Lee TC, Hsiung PA. Mutation Coverage Estimation for Model Checking. *Automated Technology for Verification and Analysis*, *Lecture Notes in Computer Science*, vol. 3299, Wang F (ed.). Springer Berlin / Heidelberg, 2004; 354–368.
30. Fabbri SCPF, Delamaro ME, Maldonado JC, Masiero PC. Mutation analysis testing for finite state machines. *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, 1994; 220 –229, doi: 10.1109/ISSRE.1994.341378.
31. Arcaini P, Gargantini A, Vavassori P. Generating tests for detecting faults in feature models. *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, IEEE, 2015; 1–10, doi:10.1109/ICST.2015.7102591.
32. Offutt J, Ammann P, Liu LL. Mutation testing implements grammar-based testing. *Proceedings of the Second Workshop on Mutation Analysis*, MUTATION '06, IEEE Computer Society: Washington, DC, USA, 2006; 12–, doi:10.1109/MUTATION.2006.11. URL `http://dx.doi.org/10.1109/MUTATION.2006.11`.
33. Kiezun A, Ganesh V, Artzi S, Guo P, Hooimeijer P, Ernst M. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering andMethodology-TOSEM* 2012; **21**(4).
34. Cody-Kenny B, Fenton M, Ronayne A, Considine E, McGuire T, O'Neill M. A search for improved performance in regular expressions. *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, ACM: New York, NY, USA, 2017; 1280–1287, doi:10.1145/3071178.3071196. URL `http://doi.acm.org/10.1145/3071178.3071196`.
35. Devroey X, Perrouin G, Papadakis M, Legay A, Schobbens PY, Heymans P. Featured model-based mutation analysis. *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, ACM: New York, NY, USA, 2016; 655–666, doi:10.1145/2884781.2884821. URL `http://doi.acm.org/10.1145/2884781.2884821`.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr