Interactive Testing and Repairing of Regular Expressions

Paolo Arcaini[1](✉), Angelo Gargantini[2], and Elvinia Riccobene[3]

[1] National Institute of Informatics, Tokyo, Japan
arcaini@nii.ac.jp
[2] University of Bergamo, Bergamo, Italy
angelo.gargantini@unibg.it
[3] Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy
elvinia.riccobene@unimi.it

Abstract. Writing a regular expression that exactly captures a set of desired strings is difficult, since regular expressions provide a compact syntax that makes it difficult to easily understand their meaning. Testing is widely used to validate regular expressions. Indeed, although a developer could have problems in writing the correct regular expression, (s)he can easily assess whether a string should be accepted or not. Starting from this observation, we propose an iterative mutation-based process that is able to *test* and *repair* a faulty regular expression. The approach consists in generating strings S that *distinguish* a regular expression r from its mutants, asking the user to assess the correct evaluation of S, and possibly substituting r with a mutant r' that evaluates S more correctly than r; we propose four variants of the approach which differ in the policy they employ to judge whether r' is better than r. Experiments show that the proposed approach is able to actually repair faulty regular expressions with a reasonable user's effort.

1 Introduction

Regular expressions (*regexp*) are used in different contexts [4] (e.g., MySQL injection prevention, DNA sequencing alignment, etc.) to validate data, perform lexical analysis, or do string matching in texts. A regexp characterizes a set of strings (i.e., a *language*). Several studies [6,12] show that regexps often contain *conformance faults*, i.e., they do not exactly describe the intended language: they either accept strings that should be refused, or refuse strings that should be accepted, or both. Indeed, specifying a *correct* regexp is usually difficult, since regexps provide a compact syntax that makes the understanding of their exact meaning difficult.

P. Arcaini—The author is supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. Funding Reference number: 10.13039/501100009024 ERATO.

I. Medina-Bulo et al. (Eds.): ICTSS 2018, LNCS 11146, pp. 1–16, 2018.
https://doi.org/10.1007/978-3-319-99927-2_1

Due to their tolerant syntax, most regexps are free of syntax errors and it is unlikely to find conformance faults by syntax checking or static analysis [12]. For this reason, the most used form of regexp validation is *testing*, i.e., generating a set of strings S and comparing the behavior of the regexp under test r over S w.r.t. an oracle. Often, the oracle is the tester, who has to check whether r correctly evaluates the strings in S. A common assumption done by many validation and testing techniques is that for a human evaluating if a given string should belong to the desired language $\mathcal{L}$ is much easier than writing the regexp that characterizes $\mathcal{L}$. Based on this assumption, several programs and web services allow to validate regexps by generating a set of strings [6] or allowing the user to enter strings and check whether they are correctly rejected or accepted by the regexp under test. However, two issues arise: (i) As checking all possible strings is not feasible, which strings are more critical to validate a regexp? (ii) Once a wrongly evaluated string is found, can a *repair* be suggested to the user?

In this paper, we address both issues. We start with the assumption that syntactical faults usually done by developers are those described by a set of fault classes identified in literature [2]. Based on this assumption, we can restrict the testing activity to the strings able to distinguish a regexp from its mutants, i.e., the strings demonstrating the presence of particular syntactic faults. Regarding what to do when a fault has been discovered (i.e., a string s which is rejected but it should be accepted or vice versa), we propose to repair the regexp under test by substituting it with its mutant which instead correctly evaluates s.

We devised an iterative process which starts with a developer who has written a regexp r and (s)he wants to check if r is *correct*, i.e., if it correctly evaluates the intended language. Then, the process proceeds through a sequence of steps in which each iteration consists in: (i) generating some mutants of the regexp r under test, (ii) computing a set S of *distinguishing strings* that permit to distinguish r from its mutants, (iii) checking the *quality* of the mutated regexps and the original regexp over S by asking the user to assess the correctness of the strings evaluation, (iv) and choosing the *best* regexp r' (i.e., the one evaluating correctly the majority of generated strings), and updating r with r'. The process iterates until no r' better than r is found. We propose four versions of the previous general process. The *greedy* one changes the regexp as soon as it finds a better candidate among the mutants. The *multiDSs* mitigates the greediness by evaluating several strings before changing the candidate. The *breadth* approach, before changing the candidate, evaluates all the mutants. The *collecting* approach aims at distinguishing multiple mutants at the same time. Experiments show that the approach is able to repair 85% of faulty regexps; on average, the user must evaluate around 40 strings and the whole process (also considering the user's effort) lasts, at most, 4.5 min.

Paper Structure. Section 2 gives background on regexps and mutation operators for them. Section 3 introduces the notions of conformance and mutation fault. Section 4 presents our approach for repairing a (possibly faulty) regexp. Section 5 describes the experiments we performed, Sect. 6 reviews related work, and Sect. 7 concludes the paper.

2 Background

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 (*regexp*) r is a sequence of* constant*s, defined over an alphabet Σ, and* operator *symbols. The regexp r* accepts *a set of* word*s $\mathcal{L}(r) \subseteq \Sigma^*$ (i.e., a* language*).*

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

As Σ we support the Unicode alphabet (UTF-16); the supported operators are union, intersection, concatenation, complement, character class, character range, and wildcards.

Normally, acceptance is computed by converting r into an automaton $\mathcal{R}$, and then checking whether $\mathcal{R}$ accepts the string. In this paper, we use the library dk.brics.automaton[1] to transform a regexp r into an automaton $\mathcal{R}$ (by means of function `toAutomaton`(r)) and perform standard operations (complement, intersection, and union) on it. Moreover, we use the operator `pickAword`($\mathcal{R}$) that returns a string s accepted by $\mathcal{R}$, i.e., $s \in \mathcal{L}(\mathcal{R})$.

For our purposes, we give the notion of a string distinguishing two languages.

Definition 2 (Distinguishing string). *Given two languages $\mathcal{L}_1$ and $\mathcal{L}_2$, we say that a string s is* distinguishing *if it is a word of the symmetric difference $\mathcal{L}_1 \oplus \mathcal{L}_2 = \mathcal{L}_1 \setminus \mathcal{L}_2 \cup \mathcal{L}_2 \setminus \mathcal{L}_1$ between $\mathcal{L}_1$ and $\mathcal{L}_2$.*

Fault Classes and Mutation Operators. Our approach is based on the idea that conformance faults (see Definition 4) can be discovered by means of mutation analysis. In mutation analysis for regexps, a *fault class* represents a possible kind of mistake done by a developer when writing a regexp; a *mutation operator* for a fault class is a function that given a regexp r, returns a list of regexps (called *mutants*) obtained by mutating r (possibly removing the fault); a mutation *slightly* modifies the regexp r under the assumption that the programmer has defined r close to the correct version (competent programmer hypothesis [10]).

We use the fault classes and the related mutation operators proposed in [2]. In the following, given a regexp r, we identify with `mut`(r) its set of mutants.

3 Conformance Faults and Mutation Faults

In this section, we describe how to compare the language characterized by a developed regexp with the intended language. Given a regexp r, we suppose to have a reference *oracle* that represents the intended meaning of r. Formally:

[1] http://www.brics.dk/automaton/.

Definition 3 (Oracle). *An* oracle o_r *is a predicate saying whether a string must be accepted or not by a regexp* r*, i.e.,* $o_r(s) = true$ *if* s *must be accepted by a correct regexp. We identify with* $\mathcal{L}(o_r)$ *the set of strings accepted by* o_r*.*

The oracle (a program or a human) can be queried and it must say whether a string belongs to $\mathcal{L}(o_r)$ or not, i.e., whether it must be rejected or accepted by r. Usually, the oracle is the developer who can correctly assess whether a string should be accepted or not (but (s)he may be not able to correctly write the correct regexp). A correct regexp is indistinguishable from its oracle (i.e., they share the same language) while a faulty one wrongly accepts or rejects some strings. We here define such strings.

Definition 4 (Conformance Fault). *A regexp* r *(supposed to behave as specified by* o_r*) contains a* conformance fault *if*

$$\exists s \in \Sigma^* \colon \mathit{distStr}(s, r, o_r)$$

where $\mathit{distStr}(s, r, o_r)$ *indicates that* s *is a distinguishing string between the languages of* r *and* o_r*, i.e.,* $s \in \mathcal{L}(r) \wedge s \notin \mathcal{L}(o_r)$ *or* $s \notin \mathcal{L}(r) \wedge s \in \mathcal{L}(o_r)$*. We name the first case as* inclusion fault*, and the second case as* exclusion fault*.*

A fault shows that the user specified the regexp wrongly, possibly misunderstanding the semantics of the regexp notation. Finding all conformance faults would require to compute $\mathcal{L}(r) \oplus \mathcal{L}(o_r)$; however, we do not know $\mathcal{L}(o_r)$ in advance, and we can only invoke the oracle o_r to check whether a string belongs to $\mathcal{L}(o_r)$ or not. Therefore, proving the absence of conformance faults in r would require to evaluate, for all the strings $s \in \Sigma^*$, $\mathit{distStr}(s, r, o_r)$ by checking whether s belongs to $\mathcal{L}(r)$ and not to $\mathcal{L}(o_r)$, or vice versa. Such exhaustive testing is infeasible and so we aim at finding a subset of strings of Σ^* that are more likely faulty. Following a fault-based approach, we claim that the syntactic faults that are more likely done by developers are those described by the fault classes we introduced in [2]. If such assumption holds, most conformance faults should be those strings that are evaluated differently by the regexp and its mutants. For this reason, we introduce a stronger type of fault defined in terms of mutations.

Definition 5 (Mutation Fault). *A regexp* r *(that is supposed to behave as specified by* o_r*) contains a* mutation fault *if*

$$\exists r' \in \mathtt{mut}(r), \exists s \in (\mathcal{L}(r) \oplus \mathcal{L}(r')) \colon \mathit{distStr}(s, r, o_r)$$

We call (r'*,*s*) failure couple.*

A regexp r contains a mutation fault if, given a mutant r', a distinguishing string s between r and r' shows that r is faulty. To find a mutation fault, we do not need to consider all the strings in Σ^*, since we know how to compute $\mathcal{L}(r) \oplus \mathcal{L}(r')$. Targeting mutation faults is beneficial thanks to this theorem.

Theorem 1. *Given a regexp* r*, if it contains a mutation fault with failure couple (*r'*,*s*), then* $\neg \mathit{distStr}(s, r', o_r)$*, i.e.,* r' *correctly evaluates* s*.*

Algorithm 1. Meta-algorithm for testing and repairing regular expressions

```
Require: $r$: initial regexp, $o_r$: oracle
Ensure: $result$: $r'$ repaired regexp
1: $A \leftarrow \emptyset \quad R \leftarrow \emptyset$        ▷ strings Accepted and Rejected by the oracle
2: loop
3:   $muts \leftarrow$ mutate($r$)
4:   $r' \leftarrow$ TESTANDREP($r$, $o_r$, $muts$, $A$, $R$)
5:   if $r \neq r'$ then $r \leftarrow r'$
6:   else return $r'$
7:   end if
8: end loop
```

Proof. Let us assume that $s \in \mathcal{L}(r) \oplus \mathcal{L}(r')$ and it is distinguishing w.r.t. the oracle, i.e., $distStr(s, r, o_r)$. This also means $s \in \mathcal{L}(r) \oplus \mathcal{L}(o_r)$. We can identify two cases. (1) s is accepted by r, i.e., $s \in \mathcal{L}(r)$, then s is rejected by both r' and o_r. (2) s is rejected by r, i.e., $s \notin \mathcal{L}(r)$, then s is accepted by both r' and o_r. In both cases, s is equally evaluated by r' and o_r, so $s \notin \mathcal{L}(r') \oplus \mathcal{L}(o_r)$. □

Theorem 1 is central in our approach that combines testing and repairing: if we *test* a regexp r with strings distinguishing r from one of its mutants r' (instead of other strings, like random ones), then in case of failure (the test distinguishes r from its oracle) we also know how to *repair* r to remove that particular fault, by taking r' as new regexp. However, note that r' may still contain other faults.

4 Testing and Repairing Regular Expressions

Our repairing approach exploits Theorem 1 in order to test and repair a regexp in an interactive way. It is based on the assumption that the mistakes done by a developer are those described by the fault classes proposed in [2]. However, if we showed to the developer a mutated regexp, (s)he could still have problems in understanding whether that is the correct regexp. On the other hand, if we showed her/him a string s, (s)he would have no difficulty in assessing the correct evaluation. We therefore propose an approach that generates strings distinguishing a regexp r from its mutants and that selects one of the mutants in case of mutation fault. The approach is formalized by the meta-algorithm presented in Algorithm 1. The algorithm takes in input the regexp r we want to repair, and an oracle o_r (usually the oracle is the user able to assess the correct evaluation of all the strings). Two sets A and R are created for memorizing the strings that are known to be accepted and rejected by the oracle. Then, the following instructions are repeatedly executed:

- r is mutated with the operators defined in [2] (mutants are stored in *muts*);
- function TESTANDREP checks whether in *muts* there exists a regexp r' *better* than r, asking the user to evaluate some distinguishing strings;
- if a better regexp is found, the process is iterated again using r' as new regexp under test (line 5), otherwise r' is returned as final regexp (line 6).

Algorithm 2. testAndRep – Greedy approach

```
1: function testAndRep($r$, $o_r$, $muts$, $A$, $R$)
2:   for all $m \in muts$ do
3:     $(ds, oEv) \leftarrow$ EvMut($m, r, o_r, muts, A, R$)
4:     if $ds \neq null \wedge m(ds) = oEv$ then
5:       return $m$
6:     end if
7:   end for
8:   return $r$
9: end function
10: function EvMut($m$, $r$, $o_r$, $muts$, $A$, $R$)
11:   if $(\exists s \in A\colon s \notin \mathcal{L}(m)) \vee (\exists s \in R\colon s \in \mathcal{L}(m))$ then return $(null, null)$
12:   end if
13:   $ds \leftarrow$ genDs($r, m$)
14:   if $ds = null$ then return $(null, null)$
15:   end if
16:   $oEv \leftarrow o_r(ds)$
17:   MarkDs($ds$, $oEv$, $A$, $R$)
18:   return $(ds, oEv)$
19: end function
20: procedure MarkDs($ds$, $oEv$, $A$, $R$)
21:   if $oEv$ then $A \leftarrow A \cup \{ds\}$
22:   else $R \leftarrow R \cup \{ds\}$
23:   end if
24: end procedure
```

We identify four possible ways to select the new repaired regexp (function testAndRep), described in the following sections. Note that Algorithm 1 does not guarantee to always terminate with any approach and, in any case, it could iterate through several regexps (asking the user to evaluate several distinguishing strings) before terminating. For this reason, the user should specify a maximum number of evaluations *MaxEval*, after which the process is interrupted[2].

4.1 Greedy Approach

Algorithm 2 shows the greedy version of testAndRep. The function, for each mutant m in $muts$, performs the following instructions:

- m is evaluated by function EvMut (line 3), working as follows:
  - if m evaluates wrongly a string in A or R, it returns $(null, null)$, meaning that m must not be considered (line 11);
  - it generates a distinguishing string ds between r and m, using function `genDs` (line 13); see [2] for the implementation of `genDs`;
  - if no string is generated, r and m are equivalent; in this case, it returns $(null, null)$, meaning that m must not be considered (line 14);
  - if a ds is generated, it stores the oracle evaluation of ds in oEv (line 16);
  - depending on the oracle evaluation, it adds ds either to A or R, using procedure MarkDs (line 17);
  - it returns the ds and its evaluation oEv (line 18);
- if a ds is returned, it checks whether the mutant m and the oracle assess the validity of ds in the same way (line 4). If this is the case, a mutation fault in r has been found, and m is returned as the new repaired regexp (line 5).

If no better mutant is found, r is returned (line 8). The approach guarantees to return a regexp that correctly accepts all strings in A and refuses those in R, while each of its mutants wrongly evaluates at least one of these strings.

[2] For the sake of simplicity, MaxEval is not shown in Algorithms 2, 3, 4, and 5.

Algorithm 3. testAndRep – `MultiDSs` approach (diff w.r.t. Alg. 2)

```
Require: N: number of strings to generate
 2: ...
 3: $DSsEvs \leftarrow$ EvMut($m$, $r$, $o_r$, $muts$, $A$, $R$)
 4: $numOK \leftarrow |\{(ds, oEv) \in DSsEvs : m(ds) = oEv\}|$
 5: $numNO \leftarrow |\{(ds, oEv) \in DSsEvs : r(ds) = oEv\}|$
 6: if $numOK > numNO$ then
 7:   return $m$
 8: end if
 9: ...
12: ...
13: $DSs \leftarrow$ genDs($r, m, N$)
14: if $DSs = \emptyset$ then return $\emptyset$
15: end if
16: $DSsEvs \leftarrow \emptyset$
17: for all $ds \in DSs$ do
18:   $oEv \leftarrow o_r(ds)$
19:   $DSsEvs \leftarrow DSsEvs \cup \{(ds, oEv)\}$
20:   MarkDs($ds$, $oEv$, $A$, $R$)
21: end for
22: return $DSsEvs$
```

4.2 MultiDSs Approach

The greedy approach returns the mutant m (line 5 of Algorithm 2) as new repaired regexp if it evaluates the generated string as the oracle. However, m may be distinguished by other strings and, so, changing the regexp could be a *too greedy* choice. To mitigate this problem, the `MultiDSs` approach (see Algorithm 3) generates N distinguishing strings (with $N \geq 2$) at line 13 (difference w.r.t. line 13 of Algorithm 2) and, at lines 6–7, takes m as the new repaired regexp only if it evaluates correctly more strings than r (difference w.r.t. lines 4–5 in Algorithm 2).

4.3 Breadth Approach

The approach in Sect. 4.1 and its extension in Sect. 4.2 are both greedy as they change the regexp under test as soon as a better regexp is found (it correctly evaluates the majority of generated strings). We here present a less greedy approach; Algorithm 4 shows the breadth search version of testAndRep. The algorithm evaluates all the mutants of a regexp under test and selects the *best* one (by function bestCand). If a better regexp is found, it is returned, otherwise r is returned.

In testAndRep, after the generation of the *ds* by EvMut (line 4), if the mutant m correctly evaluates *ds*, it is stored in a set of possible candidates *cands* (line 6). Then, all the previously generated candidates that do not correctly evaluate *ds* are removed from *cands* (line 8). When all the mutants have been evaluated, if there is no candidate, then the algorithm terminates and r is returned as final regexp (line 10). Otherwise, a candidate is selected by function bestCand as follows. As long as there is more than one candidate:

- two candidates c_1 and c_2 are randomly selected (line 18);
- a *ds* is generated for c_1 and c_2 (line 19);
- if *ds* is *null* (i.e., c_1 and c_2 are equivalent), c_1 is removed from *cands* (line 25); otherwise, the following instructions are executed;
- the candidates not evaluating *ds* correctly are removed from *cands* (line 22);
- *ds* is added either to A or R using procedure MarkDs (line 23).

Algorithm 4. testAndRep – Breadth approach

```
1: function testAndRep($r$, $o_r$, $muts$, $A$, $R$)
2:   $cands \leftarrow \emptyset$
3:   for all $m \in muts$ do
4:     $(ds, oEv) \leftarrow$ EvMut($m$,$r$,$o_r$,$muts$,$A$,$R$)
5:     if $ds \neq null \wedge m(ds) = oEv$ then
6:       $cands \leftarrow cands \cup \{m\}$
7:     end if
8:     $cands \leftarrow \{c \in cands | c(ds) = oEv\}$
9:   end for
10:  if $|cands| = 0$ then return $r$
11:  else
12:    return bestCand($cands$, $A$, $R$)
13:  end if
14: end function
16: function bestCand($cands$, $A$, $R$)
17:   while $|cands| > 1$ do
18:     $(c_1,c_2) \leftarrow$ pick2Cands($cands$)
19:     $ds \leftarrow$ genDs($c_1, c_2$)
20:     if $ds \neq null$ then
21:       $oEv \leftarrow o_r(ds)$
22:       $cands \leftarrow \{c \in cands | c(ds) = oEv\}$
23:       MarkDs($ds$, $oEv$, $A$, $R$)
24:     else
25:       $cands \leftarrow cands \setminus \{c_1\}$
26:     end if
27:   end while
28:   return $cands[0]$
29: end function
```

At the end of the while loop, the only survived candidate (it is guaranteed to exist) is selected as best candidate and returned as new repaired regexp (line 28).

4.4 Collecting Approach

The previous approaches always generate a string *ds* for distinguishing a regexp r from one of its mutants m; often *ds* does not distinguish r from other mutants and other strings must be generated for distinguishing them, so requiring more effort from the user who must evaluate more strings. The aim of the collecting approach is to generate strings that distinguish as many mutants as possible. A string *ds* distinguishes r from a set of mutants $M = \{m_1, \ldots, m_n\}$ if *ds* is accepted by r and not accepted by any mutant in M, or if *ds* is not accepted by r and accepted by all the mutants in M; the distinguishing string is a word of one of these two automata:

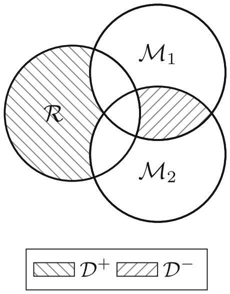

Fig. 1. Distinguishing automata

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

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*. Figure 1 shows a positive and a negative distinguishing automaton for two mutants m_1 and m_2 of a regexp r[3].

Algorithm 5 shows the collecting approach that exploits the definition of distinguishing automaton. It initially selects all the mutants *muts* as candidates in *cands* (line 2). Then, it performs the following actions as long as *cands* contains at least one regexp:

[3] In this case, they can be collected both in a positive and negative automaton; in some cases, only one kind of automaton is suitable for collecting them, or also none.

Algorithm 5. testAndRep – Collecting approach

```
Require: $CollTh$: collection limit
 1: function testAndRep($r$, $o_r$, $muts$, $A$, $R$)
 2:   $cands \leftarrow muts$
 3:   while $|cands| > 0$ do
 4:     $\mathcal{D} \leftarrow$ collect($r$, $cands$)
 5:     if $\mathcal{D} = \emptyset$ then
 6:       return $r$
 7:     end if
 8:     $ds \leftarrow$ pickAword($\mathcal{D}$)
 9:     $oEv \leftarrow o(ds)$
10:     $cands \leftarrow \{m \in cands | m(ds) = oEv\}$
11:     if $oEv \neq r(ds)$ then
12:       $r \leftarrow$ pickAregexp($\mathcal{D}$)
13:     end if
14:   end while
15:   return $r$
16: end function
17: function collect($r$, $cands$)
18:   $\mathcal{R} \leftarrow$ toAutomaton($r$)
19:   for $\mathcal{D} \in \{\mathcal{R}, \mathcal{R}^{\mathsf{C}}\}$ do
20:     for $m \in cands$ do
21:       $\mathcal{M} \leftarrow$ toAutomaton($m$)
22:       if $isPos(\mathcal{D})$ then $\mathcal{D}' \leftarrow \mathcal{D} \cap \mathcal{M}^{\mathsf{C}}$
23:       else $\mathcal{D}' \leftarrow \mathcal{D} \cap \mathcal{M}$
24:       end if
25:       if $\mathcal{D}' \neq \emptyset$ then
26:         $\mathcal{D} \leftarrow \mathcal{D}'$
27:         if numColl($\mathcal{D}$) $= CollTh$ then return $\mathcal{D}$
28:         end if
29:       end if
30:     end for
31:     if $\mathcal{D} \neq \emptyset$ then return $\mathcal{D}$
32:     end if
33:   end for
34:   return $\emptyset$
35: end function
```

- it collects as many regexp as possible using function collect:
  - it randomly initializes a positive or a negative distinguishing automaton $\mathcal{D}$ (line 19);
  - for each mutant m, it tries to add it to $\mathcal{D}$ considering the polarity (lines 21–23); $\mathcal{D}$ is modified only if the addition does not produce an empty automaton (lines 25–26); if the addition has been performed and *CollTh* regexps have been collected (being *CollTh* a parameter of the approach), it returns $\mathcal{D}$ (line 27).
  - if after trying all the candidates, at least one regexp has been collected, it returns $\mathcal{D}$ (line 31); otherwise, it tries the distinguishing automaton with the opposite polarity (if any);
  - if no regexp can be collected in any distinguishing automaton of any polarity, the empty automaton is returned (line 34).
- if the returned automaton $\mathcal{D}$ is empty (meaning that all the candidates are equivalent to r), r is returned as selected regexp (line 6); otherwise, a word is randomly selected from $\mathcal{D}$ (line 8) and *cands* is updated with only the regexps that evaluate *ds* correctly (lines 9–10);
- if r does not evaluate *ds* correctly, it is changed with one mutant randomly selected among those collected in $\mathcal{D}$ (lines 11–12).

When there are no more candidates, r is returned as selected regexp (line 15).

TearRex. We implemented the approach in the tool TearRex (TEst And Repair Regular EXpressions)[4]. Figure 2 shows an interaction with the tool. At the beginning, the user inserts the regexp [a-z]* for accepting all the strings

[4] The tool and all benchmarks are available at http://foselab.unibg.it/tearrex/.

```
Insert regexp: [a-z]*
Do you accept "a"? Y for yes, N for no Y
Do you accept "A"? Y for yes, N for no Y
The mutant is the new regexp: [A-z]*
Do you accept "0"? Y for yes, N for no N
```

```
Do you accept "z"? Y for yes, N for no Y
Do you accept "AA"? Y for yes, N for no Y
Do you accept ""? Y for yes, N for no N
The mutant is the new regexp: [A-z]+
The mutant is the final regexp: [A-z]+
```

Fig. 2. TearRex– Testing and repairing of `[a-z]*` with oracle `[a-zA-Z]+`

containing only Latin letters; however, since (s)he misunderstood the semantics of operator * and of character classes, the developed regexp also accepts the empty string and does not accept upper-case Latin letters. The tool asks her/him to evaluate two strings, “a” and “A”, and (s)he assesses that both strings should be accepted. Since the developed regexp does not accept “A”, the tool modifies the regexp in `[A-z]*` that also accepts “A”. The process continues by evaluating strings “0”, “z”, and “AA”, that are all correctly evaluated. The empty string “”, instead, is wrongly evaluated and the tool changes the regexp in the more correct version `[A-z]+` that does not accept it. Since no possible mutation fault exists between `[A-z]+` and its mutants, it is returned as final regexp.

5 Experiments

Websites http://www.regular-expressions.info/ and http://www.regexlib.com report, for different matching tasks (e.g., email addresses, credit cards, social security numbers, etc.), some regexps implementing them. To build the benchmark set `Bench`, we considered 20 tasks and, for each task, we selected two regexps, one acting as initial (possibly faulty) regexp and the other one as oracle; for 19 couples the initial regexp is indeed faulty, while in one couple the initial regexp is correct[5]. The initial regexps are between 17 and 279 chars long (60 on average) and have between 7 and 112 operators (24.45 on average).

Note that, in our approach, the oracle should be the user able to assess the evaluation of strings; in the experiments, we use another regexp as oracle for the sake of automation. However, in RQ2 we estimate which is the burden required to the user in evaluating the generated strings during the repair process.

We run the approaches `Greedy`, `MultiDSs3` (MultiDSs with $N = 3$), `Breadth`, and `Coll5` (collecting with $CollTh = 5$) on the selected regexps and fixing the maximum number of evaluations MaxEval to 10, 30, 100, and 200 strings.

Experiments have been executed on a Linux machine, Intel(R) Core(TM) i7, 4 GB RAM. All the reported data are the averages of 10 runs of the experiments.

Tables 1a and b report experimental results aggregated by type of repair *process* and maximum number of evaluations *MaxEval*. They both report the average number of strings generated and shown to the user for *evaluation* `E`, the average process *time* `T`, the percentage `R` of regexps that have been *repaired* (i.e., the final regexp has lower failure index than the initial one), and the percentage

[5] Note that the process guarantees to not worsen correct regexps; we introduced this correct regexp to double-check that this is indeed the case.

Table 1. Experiment results for `Bench` (E: avg # evaluations, T: avg time (secs), R: repaired (%), CR: completely repaired (%), R′: avg # r')

Process	E	R′	T	R	CR
`Greedy`	47.82	19.60	0.50	77.5	3.75
`MultiDSs3`	53.39	7.63	1.04	72.5	2.92
`Breadth`	41.41	3.33	1.74	85.0	3.75
`Coll5`	42.27	4.57	2.79	76.67	4.58

(a) Process type results

MaxEval	E	T	R	CR
10	9.95	0.29	76.67	0.42
30	27.55	0.57	75.83	3.75
100	61.6	2.21	78.75	4.58
200	85.44	2.99	80.42	6.25

(b) MaxEval results

`CR` of regexps that have been *completely repaired* (i.e., all the faults have been removed). Table 1a also reports the average number `R′` of regexps r' that are changed during the process. The cells in gray highlight the best results. We evaluate our approach with a series of research questions.

RQ1. How many distinguishing strings are generated?

Since the correct evaluation of the generated strings must be assessed by the user, we want to keep their number low. Table 1a (column `E`) shows that the four approaches generate, on average, between 41 and 53 strings; the limited number of strings allows the manual evaluation by the user. `MultiDSs3` generates more strings than `Greedy` as three strings are generated for each mutant (if possible). `Breadth` generates slightly fewer strings than `Greedy` because, on average, the third changed regexp is the final expression (see column `R′`); `Greedy` changes on average 20 regexps, so producing more strings. `Coll5` also produces few strings, as it directly generates strings for set of mutants.

Table 1b (column `E`) shows that a higher maximum number of evaluations allows to generate more strings. However, with MaxEval equal to 10 and 30, the number of generated strings is close to MaxEval itself, while with MaxEval equal to 100 and 200 the number of strings is much lower than the limit: this means that, on average, the process requires less than 100 strings to terminate.

RQ2. How long does the repair process take and how much is the estimated user's effort?

Table 1a (column `T`) shows that `MultiDSs3` is twice slower than `Greedy`, as it generates more strings. `Breadth` is more than three times slower than `Greedy`; a possible reason is that although `Greedy` generates more strings than `Breadth`, it creates much fewer mutants (that is a costly operation). `Coll5` is the slowest one, as automata intersection (see lines 22 and 23 in Algorithm 5) is a costly operation. Table 1b shows that increasing the limit of evaluated strings also increments the time (as the process can continue the search).

All the experiments show that, regardless of the process configuration, the process is rather fast. However, the final time should be computed as $pTime + |A \cup R| \times eTime$, being $pTime$ the time of the generation and repair process (the one we measured), and *eTime* the time taken by a user to evaluate a string[6]. For example, considering 5 s per string as *eTime*, the average times would be: 4 min for `Greedy`, 4.5 min for `MultiDSs3`, 3.5 min for `Breadth`, and 3.6 min for `Coll5`. This means that the whole process involving the user's evaluations is feasible. Note that `Greedy` and `MultiDSs3` would be equivalent from the time point of view (break-even) to `Breadth` and `Coll5` if the user could evaluate the strings in less than 0.2 s, which seems impossible; so, `Breadth` and `Coll5` are always advantageous in terms of time if the user's effort is taken into account.

RQ3. Is the proposed process able to repair regular expressions?

We are here interested in assessing whether the proposed approach is actually able to repair faulty regexps. We are able to assess whether a starting regexp r has been completely repaired by checking the equivalence between the final regexp r' and the oracle o_r. For not completely repaired regexps, we introduce the measure F_x counting the number of strings that a regexp x wrongly accepts or rejects, i.e., that are misclassified. As the number of misclassified strings is possibly infinite, we need to restrict the length of the considered strings to n. Being $\mathcal{L}^n(x) = \mathcal{L}(x) \cap \Sigma^n$, F_x^n is defined as follows:

$$F_x^n = |\mathcal{L}^n(x) \oplus \mathcal{L}^n(o_x)|$$

In order to know if the repaired regexp r' is better than r, we can compute $\Delta F = F_{r'}^n - F_r^n$ with a fixed n. In the experiments, we set n to 20, to count the strings of length n up to 20. If $\Delta F < 0$ or the final regexp is completely repaired, the regexp under test is considered *repaired*, otherwise, it means that the process did not remove any fault (or removed and introduced faults equally).

By Table 1a, we can see that the processes can repair (column `R`) between 72% and 85% of regexps, but they can completely repair (column `CR`) a small amount of them (between 2.92% and 4.58%). Thus, the proposed techniques are not reliable in finding the completely correct regexp, but they are very efficient in removing some faults. However, some regexps are not repaired. This can be due to two reasons. First, the new changed regexp r' behaves better than the original r on the strings that are generated and tested, but it fails to correctly accept/reject other strings that are not tested, so ΔF is actually greater than 0. Secondly, we are trying to repair a regexp r that is *too far* from the oracle, so each mutation of r is not better than it; indeed, when selecting the benchmarks, we did not assume the competent programmer hypothesis, so a regexp and its oracle may be very different. In RQ5, we will evaluate how the results change if the competent programmer hypothesis holds.

Table 1b (columns `R` and `CR`) shows that increasing the number of maximum number of evaluations permits to (completely) repair more regexps.

RQ4. Which is the best approach?

[6] Note that we could also specify two different evaluation times for accepted and rejected strings (supposing that one of the two tasks is easier).

Table 2. Experiment results for `MutBench` (E: avg # evaluations, T: avg time (secs), R: repaired (%), CR: completely repaired (%), R′: avg # r')

Process	E	R′	T	R	CR
`Greedy`	37.05	10.07	0.40	87.5	8.75
`MultiDSs3`	48.33	7.36	2.03	92.08	7.5
`Breadth`	29.48	1.64	1.06	85.0	11.67
`Coll5`	27.24	2.01	5.11	85.83	12.92

(a) Process type results

MaxEval	E	T	R	CR
10	9.87	0.85	84.58	7.92
30	24.99	1.21	87.50	9.58
100	47.57	4.12	89.17	11.67
200	59.67	2.42	89.17	11.67

(b) MaxEval results

From Table 1a, we observe that `MultiDSs3` under-performs in all the measures. `Breadth` and `Coll5`, instead, are better approaches. `Breadth` repairs 7.5% more regexps than `Greedy` using fewer strings. Although `Breadth` is more than three times slower than `Greedy`, we saw in RQ2 that the program time (without string evaluation by the user) is negligible w.r.t. the evaluation time of the user: therefore, in order to contain the total time of the process, it makes sense to keep the number of generated strings limited. `Coll5` is the approach that completely repairs more regexps, although it does not behave so well in the repaired ones: this probably means that the strings generated over the collection are good to drive towards correct candidates, but also that `Coll5` is not incremental enough and it tends not to choose regexps that are only *slightly better*.

RQ5. How are the results if the competent programmer hypothesis hold?

When building `Bench`, we did not make any assumption and most of the selected couples of regexps are very different. However, the common assumption that is done in fault-based approaches is the *competent programmer hypothesis*, i.e., the programmer defined the regexp close to the correct one (only with one or few of the syntactic faults defined by the fault classes [2]). We here evaluate the approach performances when the competent programmer hypothesis holds. We built a second benchmark set, `MutBench`, as follows. We took the oracles of `Bench` and we randomly modified them introducing n faults (with $n = 1, \ldots, 3$), obtaining three faulty versions of the oracle; we therefore added to `MutBench` 60 couples of regexps. The regexps are between 43 and 302 characters long (108.15 on average) and contain between 11 and 63 operators (25.52 on average). We then applied our approach to `MutBench`. Table 2 reports the aggregated results for the experiments performed over the regexps in `MutBench`. Most of the observations we did for `Bench` are still valid for `MutBench`, although there are some interesting differences. First of all, by Table 2a, we observe that `Greedy` now behaves as good as `Breadth` in terms of repaired regexps (actually slightly better): this means that if the syntactic faults of the regexp under test are those identified by our fault classes, applying a greedy approach that changes the regexp as soon as a fault is found pays off. Moreover, for both approaches, the numbers of evaluated strings and of changed regexps are lower w.r.t. Bench: this means that the approaches converge faster to the final solution as they are able to apply the correct mutations to remove the syntactic faults. MultiDSs3 is now the best approach in terms of repaired regexps: this probably means that generating more strings w.r.t. Greedy for reinforcing the decision makes sense only if there are few faults, otherwise additional strings are useless.

Remark. A threat to external validity is that the obtained results could be not generalizable to all regexps. In order to mitigate this threat, we tried to select the most diverse regexps performing different tasks; in order to evaluate the approach on the worst case scenario, we also did not assume the competent programmer hypothesis. We saw in RQ5 that, if the hypothesis holds (as it is assumed by fault-based approaches), the performance of the approach improves.

6 Related Work

As far as we know, no approach exists to repair regexps. The approaches proposed in literature mainly focus on regexps testing or on their synthesization.

Regarding test generation of labeled strings, MutRex [2] is an open source tool able to generate fault-detecting strings. We exploit its mutation operators and its string generation facility (i.e., function genDs).

Another test generator is EGRET [6] that generates *evil* strings that should be able to expose errors commonly made by programmers. EGRET takes a regexp as input and generates both accepted and rejected strings. The user can estimate the regexp correctness by evaluating these strings and identifying those that are wrongly classified. As in our approach, the user is the oracle. Also EGRET applies mutation, but on the strings accepted by the regexp under test, and not on the regexp itself as in our case. The advantage of the strings we use in our approach is that, once we detect a failure, we also know how to remove the corresponding syntactic fault in the regexp; instead, using the strings generated by EGRET would leave open the problem to localize the syntactic fault.

Another tool that can be used for labeled string generation is Rex [13], a solver of regexps constraints. Rex translates a given regexp into a symbolic finite automaton; the Z3 SMT solver is used for satisfiability checking. Since Z3 is able to generate a *model* as witness of the satisfiability check, Rex can be used to build strings accepted by the regexp.

Reggae [7] is a tool based on dynamic symbolic execution that generates string inputs that are accepted by a regexp. Reggae aims at achieving branch coverage of programs containing complex string operations.

Several other tools for testing regexps exist, as EXREX, Generex, and regldg[7]. However, they are based on exhaustive or random generation of strings matching a given regexp, and the strings they generate are not useful for repairing regexps.

[7] https://github.com/asciimoo/exrex, https://github.com/mifmif/Generex, https://regldg.com/.

A different use of labeled strings is the synthesization of regexps. ReLIE [8] is a learning algorithm that, starting from an initial regexp and a set of labeled strings, tries to learn the correct regexp. It performs some regexp transformations (a kind of regexp mutation); however, no definition of fault class is given. Our approach could be adapted for regexps synthesis as well.

Our approach has some similarities with *automatic software repair*. The automatic repair of software requires an oracle: in our approach, the oracle is the user, while in software repair the oracle is usually specified using test suites [3,14], pre- and post-conditions [11], etc. Moreover, such approaches also identify techniques to repair the software when a fault is detected: in [9], for example, some *repair actions* are proposed, that are similar to our mutation operators.

Automatic repair has also been proposed for specifications, as, for example, automatic repair of feature models describing software product lines. The approach in [5] applies a cycle of *test-and-fix* to a feature model in order to remove its wrong constraints; the approach uses configurations derived both from the model and from the real system and checks whether these are correctly evaluated by the feature model. The approach has similarity with ours in alternating testing and fixing (similar to our repair phase); however, since the evaluation of configurations is done automatically on the system, the approach can produce several configurations, while in our approach we need to keep the number of generated labeled strings limited, as these must be assessed by the user.

The aim of our work is similar to that in [1] in which a *student* tries to learn an unknown regular set S by posing two types of queries to a *teacher*. In a *membership query*, the student gives a string t and the teacher tells whether it belongs to S or not. In a *conjecture query*, the student provides a regular set S' and the teacher answers *yes* if S' corresponds to S, or with a *wrong* string t (as our distinguishing string) otherwise. In our approach, the user plays the role of the teacher only for the first kind of query, but not for the second kind (if (s)he could, (s)he would also be able to write the correct regexp). Our tool, instead, plays the role of the student by providing membership queries.

7 Conclusions

The paper presents an approach able to detect and remove conformance faults in a regular expression, i.e., faults that make a regular expression to wrongly accept or reject some strings. The approach consists in an iterative process composed of a *testing* phase in which the user who wrote the regular expression is asked to assess the correct evaluation of some strings that are able to distinguish a regular expression from its mutants, and in a *repair* phase in which the mutant evaluating correctly all the generated strings is taken as new version of the regular expression. The approach terminates when it is no more possible to repair the regular expression under test. Experiments show that the approach is indeed able to remove conformance faults from regular expressions, in particular if the competent programmer hypothesis holds, i.e., the user did some small syntactical faults as those described by the fault classes proposed in literature.

In the experiments, we performed different evaluations regarding the effect of the process configuration on the final results, without assessing their statistical significance. As future work, we plan to perform experiments on a wider set of regular expressions and to conduct some statistical tests to assess the statistical significance of the drawn conclusions.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
2. Arcaini, P., Gargantini, A., Riccobene, E.: Fault-based test generation for regular expressions by mutation. Softw. Test. Verif. Reliab. e1664 (2018)
3. Arcuri, A.: Evolutionary repair of faulty software. Appl. Soft Comput. **11**(4), 3494–3514 (2011)
4. Chapman, C., Stolee, K. T.: Exploring regular expression usage and context in Python. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, pp. 282–293. ACM, New York (2016)
5. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Le Traon, Y.: Towards automated testing and fixing of re-engineered feature models. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 1245–1248. IEEE Press, Piscataway (2013)
6. Larson, E., Kirk, A.: Generating evil test strings for regular expressions. In: 2016 IEEE 9th International Conference on Software Testing, Verification and Validation (ICST), pp. 309–319, April 2016
7. Li, N., Xie, T., Tillmann, N., Halleux, J.D., Schulte, W.: Reggae: automated test generation for programs using complex regular expressions. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 515–519. IEEE Computer Society (2009)
8. Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., Jagadish, H. V.: Regular expression learning for information extraction. In: Proceedings of EMNLP 2008, pp. 21–30. Association for Computational Linguistics (2008)
9. Martinez, M., Monperrus, M.: Mining software repair models for reasoning on the search space of automated program fixing. Empir. Softw. Eng. **20**(1), 176–205 (2015)
10. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: an analysis and survey. Adv. Comput. (2018). Elsevier
11. Pei, Y., Furia, C.A., Nordio, M., Wei, Y., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. IEEE Trans. Softw. Eng. **40**(5), 427–449 (2014)
12. Spishak, E., Dietl, W., Ernst. M.D.: A type system for regular expressions. In: Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, pp. 20–26. ACM, New York (2012)
13. Veanes, M., Halleux, P.D., Tillmann, N.: Rex: symbolic regular expression explorer. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, ICST 2010, pp. 498–507. IEEE Computer Society (2010)
14. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Commun. ACM **53**(5), 109–116 (2010)