

Regular Expression Learning with Evolutionary Testing and Repair

Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene

▶ **To cite this version:**

Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene. Regular Expression Learning with Evolutionary Testing and Repair. 31th IFIP International Conference on Testing Software and Systems (ICTSS), Oct 2019, Paris, France. pp.22-40, 10.1007/978-3-030-31280-0_2 . hal-02526352

HAL Id: hal-02526352

https://inria.hal.science/hal-02526352

Submitted on 31 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Regular expression learning with evolutionary testing and repair

Paolo Arcaini[1][0000−0002−6253−4062]*, Angelo Gargantini[2][0000−0002−4035−0131], and Elvinia Riccobene[3][0000−0002−1400−1026]

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

Abstract. Regular expressions are widely used to describe and document regular languages, and to identify a set of (valid) strings. Often they are not available or known, and they must be learned or inferred. Classical approaches like L* make strong assumptions that normally do not hold. More feasible are testing approaches in which it is possible only to generate strings and check them with the underlying system acting as *oracle*. In this paper, we devise a method that starting from an initial guess of the regular expression, it repeatedly generates and feeds strings to the system to check whether they are accepted or not, and it tries to repair consistently the alleged solution. Our approach is based on an evolutionary algorithm in which both the population of possible solutions and the set of strings co-evolve. Mutation is used for the population evolution in order to produce the offspring. We run a set of experiments showing that the string generation policy is effective and that the evolutionary approach outperforms existing techniques for regular expression repair.

Keywords: regular expression · mutation testing · software repair · evolutionary approach

1 Introduction

Regular expression (regexps) are widely used to describe regular languages and to identify a set of (valid) strings. However, in some cases they are not available. Consider, for example, a method that accepts strings in an unknown format or a service that validates some inputs according to some rules only partially documented. In such cases, the user may have a good candidate regexp C for these inputs but still (s)he wants to check if C is correct and, in case it is not,

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

(s)he wants to *learn* or *infer* the correct regexp representing the language of the System Under Learning (*SUL*) starting from C. Having a formal representation or model of the *SUL* input format, enables several engineering activities like verification and validation, test-case generation, and documentation.

Consider, for example, a web service that accepts a time t presumably in HH:MM format and converts t into different time zones (if t is in the correct format). The user formulates an initial guess about the time format, e.g., `[0-2][0-3]:[0-5][0-9]`, but still one wants to test if the initial guess is correct and, in case some faults are found, to repair it and learn the correct regexp.

Classical approaches like the L* algorithm [2], solve this problem by querying the *SUL* for checking string acceptance and equivalence of the alleged solution. However, L* is based on several strong assumptions (e.g., the possibility to check equivalence of regexps) that rarely hold in practice. Indeed, in general, there is no way for a tester to know whether a regexp is equivalent to the language accepted by the *SUL*, and the only information (s)he can gather is the acceptance of a given string. In the example above, the server code may not be accessible, or may not be using any regexp internally. The only available action is calling the time server with a string at a time and see whether it is accepted or not. The general problem we tackle here is to build a model of some unknown regular language based only on sets of positive and negative classifications [9].

Under the weaker assumption of performing only acceptance queries, synthesizing or repairing a regexp with a symbolic method like L* becomes unpractical. For this reason, evolutionary algorithms have been widely used in this ambit [6,16]. They take a given data set *DS* (strings and their evaluation) and build the regexp that gives the largest number of right evaluations of the strings in *DS*. Genetic programming and operations like crossover and genetic mutations are used in [6], while in [16] mutation operators similar to those we use in this paper are employed. These mutation operators try to mimic possible faults done by programmers [4].

Another assumption regards strings construction. In *passive* learning methods [7], the set of strings together with their labels is given before the inference method is started; in *active* learning [8,10], strings are assumed to be continuously generated, and most of these methods like [8] aim at synthesizing a regexp from scratch.

We presented an approach based on active testing and mutation in [3]. In that work, we assumed that a human user is used as oracle, so we severely limited the number of strings to be evaluated. In case of an automatic oracle, this limitation can be partially relaxed, as calling the oracle to properly label strings is not very expensive (however, a constraint over the maximum number of queries one can execute may still exist).

In this work, we propose an evolutionary approach that aims at learning a regexp modeling the set of strings accepted by the *SUL*. The following assumptions make this active learning approach different from existing methods:

– The user has an initial guess of the regexp (s)he wants to learn. In this way, (s)he can provide the learning algorithm with some domain knowledge

about the structure of the language of the *SUL*. Furthermore, this allows our technique to consider more possibly good solutions, as it does not have to consider very wrong solutions at the beginning. The more accurate the guess, the more likely should our process be to converge to a right solution.
- Strings are continuously generated and fed to the *SUL* that acts as oracle and gives an almost immediate answer (no user is involved, differently from [3]). In this way, the population of possible solutions co-evolves with the test suite.

Upon building the initial population on the base of an initially guessed regexp, the process iterates through a sequence of steps, and at each iteration:

- it generates *meaningful strings*, that show the differences among the solutions in the population. Specifically, it generates strings able to *distinguish* pairs of members of the population and invokes the *SUL* for assessing the classifications of these strings;
- for each member of the population, a *fitness* value is computed in terms of generated strings it evaluates correctly;
- some members of the current population are selected as parents of the population of the next iteration;
- the new population is obtained by mutation of the parents, using suitable operators.

As the fitness is subject to change over time and may be very unstable (at least at the beginning), it is not suitable to be used as termination condition. Therefore, we decided to stop the process after a given timeout decided by the user.

Experiments show that, under the assumption that the initial guess of the user is reasonable, the approach, on average, is able to completely repair (i.e., find a regexp capturing the *SUL* language) 81.3% of the initial regexps and improve 99.8% of them. Also in the case in which the initial guess of the user is not very good, 25.4% of initial regexps are totally repaired, and 83.5% improved.

Paper structure Sect. 2 provides basic definitions on regexps and their mutation operators. Sect. 3 presents our evolutionary method for learning a regexp modeling the language of the *SUL*. Sect. 4 describes the experiments we did to evaluate the approach, both on an existing web service and on some benchmarks. Finally, Sect. 5 reviews some related work, and Sect. 6 concludes the paper.

2 Background

We focus on 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 automaton accepting the language described by a regular expression.

Definition 1 (Regular expression). *A regular expression (*regexp*), defined over an alphabet Σ, can be recursively defined as follows. (i) The empty string*

ϵ and each symbol in Σ is a regular expression; (ii) if $r_1 \dots r_n$ are regular expressions, then $op(r_1, \dots, r_n)$ is a regular expression where op is a valid operator (see below). The regexp r accepts *a set of* words *$\mathcal{L}(r) \subseteq \Sigma^*$ (i.e., a* language*).*

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

As Σ we support the Unicode alphabet (UTF-16); the supported operators *op* are union, concatenation, intersection, repeat, complement, character range, and wildcards (e.g., any char or any string).

Acceptance can be computed by converting r into an automaton $\mathcal{R}$, and checking whether $\mathcal{R}$ accepts the string. In this paper, we use the library dk.brics.automaton [19] to transform a regexp r into an automaton $\mathcal{R}$ and perform standard operations on it.

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

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

We define a function `genDs`(r_1, r_2) that returns a distinguishing string between the languages of two regexps r_1 and r_2. `genDs` builds two automata for r_1 and r_2, makes the symmetric difference (using union, intersection, and complement of automata) and returns a string accepted by the difference. If r_1 and r_2 are equivalent, it returns `null`.

2.1 Conformance faults

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

Definition 3 (Oracle). *An* oracle *sul is a predicate saying whether a string is accepted or not. We identify with $\mathcal{L}(SUL)$ the set of strings accepted by the SUL.*

The oracle can be queried and it must say whether a string belongs to $\mathcal{L}(SUL)$ or not, i.e., whether it must be accepted or rejected. Usually, the oracle is a software procedure that can be called and that checks whether a given string should be accepted or not. The oracle may use a regexp and some string matching code in order to evaluate strings, but this internal mechanism is unknown during the process.

A correct regexp is indistinguishable from its oracle (i.e., they share the same language), while a faulty one wrongly accepts or wrongly rejects some strings.

Definition 4 (Conformance Fault). *A regexp r (allegedly representing the language of SUL) contains a* conformance fault *if and only if $\exists s \in \Sigma^* \colon r(s) \neq sul(s)$, i.e., there is a string s which is wrongly evaluated by r with respect to its reference oracle sul.*

A fault shows that the user specified the regexp r wrongly, maybe misunderstanding the semantics of the regexp notation or the *SUL* documentation. In this case, r must be repaired in order to learn the correct regexp describing the language accepted by *SUL*.

Finding all conformance faults would require to test all the strings in Σ^*, which is unfeasible. Therefore, the first problem is how to select strings to be used as tests. Randomly is an option that we will explore in the experiments, but we propose an alternative method based on distinguishing strings. The second problem is what to do when a fault has been found and how to repair a faulty regexp. In the following, we try to answer these two questions by proposing a *testing and repair* approach.

Theorem 1. *Let r_1 and r_2 be two regexps that are not equivalent, and s a distinguishing string returned by* $\texttt{genDs}(r_1, r_2)$*, then either r_1 or r_2 will contain a conformance fault.*

To check which of the two regexps contains a fault, we can test s with the oracle and see which one wrongly evaluates s. If we find that r_1 is faulty, we can say that r_2 fits better to the oracle than r_1 with the string s, and r_2 should be preferred to r_1.

Note that even if r_2 fits better than r_1 with s, on other strings r_1 may fit better than r_2 and r_2 may still contain other conformance faults. For this reason, it is important to keep track of the generated strings in the whole test suite.

How to choose suitable r_1 and r_2? In this paper, we assume that r_1 and r_2 belong to the population of possible solutions, which evolves over time. Moreover, following a fault-based approach, we claim that the syntactical faults that are more likely done by developers are those described by fault classes. If such assumption holds, possible solutions should be generated using a mutation approach as explained in the following.

2.2 Mutation

Our approach is based on the idea that regexps can be repaired by slightly modifying them. In the process, we try to remove faults by applying *mutation operators*. A mutation operator 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 [20]).

We use the fault classes and related mutation operators proposed in [4]; for example, some operators change metachars in chars (and the other way round), change quantifiers, modify/add character classes, etc. In the following, given a regexp r, we identify with $\texttt{mut}(r)$ the set of its mutants.

3 Proposed approach

We here explain the technique that users can apply in order to discover the regexp describing the language accepted by a *SUL*.

In our approach, they specify a regexp r_i representing their initial guess. Starting from this, they want to validate r_i and possibly automatically modify it in order to find a regexp r_f that correctly captures the language of the *SUL*. In order to learn the correct regexp, we propose an approach based on evolutionary algorithms [11]: it can be understood as an optimization problem in which different solutions (*population*) are modified by random changes (by *mutation*) and their quality is checked by an objective (*fitness*) function. The process consists in the following phases:

1. **Initial population**. Initially, starting from the user's regexp r_i, a *population* P of candidate solutions is created.
2. Then, the following steps are repeatedly executed:
   (a) **Evaluation**. Each member of the population P is evaluated using a given *fitness function*.
   (b) **Termination**. A termination condition is checked to decide whether to start the next iteration or to terminate. If the termination condition holds, the candidate with the best *fitness value* is returned as final regexp r_f.
   (c) **Selection**. Some members of P are selected as parents (collected in the multi-set PAR) of the next generation.
   (d) **Evolution**. Parents PAR are mutated to obtain the *offspring* to be used as population in the next iteration.

   In the following, we describe each step in details.

Initial population The size M of the population is a parameter of the evolutionary algorithm. In our approach, we set M to $H \cdot |operators(r_i)|$, where *operators* is a function returning the operators of a regexp, and H a parameter of our approach. In the experiments, we set H=100. The initial population contains r_i and M-1 mutants randomly taken from $\texttt{mut}(r_i)$.

Evaluation In this step, the quality of the candidates in P is evaluated using a *fitness function* being the objective of our search. Usually, the fitness function is *complete* as it precisely identifies when a candidate is totally correct. In our approach, instead, the fitness function is only *partial*; indeed, to learn the regular language accepted by the *SUL*, we can only rely on the knowledge of the *SUL* evaluation of a finite set of strings. We therefore define the fitness function as follows.

Definition 5 (Fitness function). *Given a regexp r and two sets A and R of strings accepted and rejected by the SUL, the* fitness *value of r is defined as:*

$$fitness(r, A, R) = \frac{|\{s \in A \mid r(s)\}| + |\{s \in R \mid \neg r(s)\}|}{|A \cup R|}$$

Algorithm 1 Evaluation step: Test generation and fitness update

Require: P: population
Require: *sul*: oracle of the *SUL*
1: $A \leftarrow \emptyset$: strings **A**ccepted by the oracle
2: $R \leftarrow \emptyset$: strings **R**ejected by the oracle
3: startGenTimeout()
4: **for each** $p_i, p_j \in P$ **do**
5: **if** *genTimeout* **then**
6: **return**
7: **end if**
8: **if** $\exists s \in (A \cup R) : p_i(s) \neq p_j(s)$ **then**
9: **continue**
10: **end if**
11: $ds \leftarrow$ `genDs`(p_i, p_j)
12: **if** $ds \neq$ `null` **then**
13: $sulEv \leftarrow sul(ds)$
14: **if** *sulEv* **then**
15: $A \leftarrow A \cup \{ds\}$
16: **else**
17: $R \leftarrow R \cup \{ds\}$
18: **end if**
19: updateFitness(P, *ds*, *sulEv*)
20: **end if**
21: **end for**

Note that the fitness value depends on strings that are known to be accepted and rejected by the *SUL* (contained in A and R). Since the possible strings are infinite, finding meaningful strings to be used for fitness computation is extremely important. Since the fitness function is used to evaluate and rank regexps contained in P, we try to generate strings able to show the differences among the different candidates. Thanks to Thm. 1, if such strings are generated as those distinguishing regexps of the population P, the fitness value can better differentiate the elements in P. Moreover, A and R should not be too small, in order to avoid a too partial fitness evaluation. At each iteration, we therefore enlarge the sets A and R by generating new strings, using the procedure described in Alg. 1. The approach tries to generate a distinguishing string for each pair of members p_i and p_j of the population, as follows:

1. Since generating strings for each pair of candidates could be computationally too expensive, we fix a timeout *genTimeout* after which the test generation for the current generation terminates (line 6).
2. If the timeout does not occur, the algorithm first checks whether there is already a string able to distinguish the two regexps (line 8) and, if so, it continues with the next pair of regexps (line 9).
3. Otherwise, a string *ds* is generated using function `genDs` that picks a string from the symmetric difference of the languages of two regexps [4] (line 11).
4. If the two regexps are equivalent, no string is produced; otherwise, *ds* is submitted to the oracle *sul* of the *SUL* and its evaluation recorded in *sulEv* (line 13).
5. According to the string evaluation, *ds* is added either to A or R (lines 15 and 17).
6. Finally, the fitness of each member of the population is updated by considering the new string *ds* with its correct evaluation *sulEv* (line 19).

Note that, as new strings are generated in each iteration, the fitness value of a candidate may change in different evolutions.

This phase corresponds to the *testing* part of our approach, in which strings are generated to be used as tests, and collected in A and R representing the test suite.

Termination condition In this step, the process decides whether to terminate or continue with another iteration. For this, different termination conditions can be specified. A classical termination condition is usually related to the fitness function (when it reaches 1 or a given threshold). In our context, however, the fitness function is partial as it only considers a finite subset of all the possible strings that can be evaluated; it is therefore not particularly suitable as termination condition. The only reasonable termination condition that we can impose is related to the computation time: after a given *timeout*, we terminate the process. The timeout represents the effort that the user can spend in learning the language accepted by *SUL*.

When the process terminates, the member of the population P with the highest fitness is returned as final regexp r_f.[4]

Selection In this step, starting from population P, a multiset of *parents* PAR of size p is built, being p a parameter of the evolutionary process. Different selection strategies have been proposed in literature, as *truncation*, *roulette wheel*, and *rank* [11]. We here use the *truncation* selection that selects the first $n = \lceil K \cdot |P| \rceil$ members of the population with the highest fitness value, where K is a parameter specifying a percentage of the population; in the experiments, we use K=5%. Then, the first n elements are added to PAR as many times as necessary to reach the size p.

Evolution In this step, the population (called *offspring*) for the next generation is built starting from the parents in PAR. In order to build the offspring, we mutate all the regexps r in PAR using function `mut`, as defined in Sect. 2.2. We set an upper bound M to the new population size. If the mutation operators generate at most M mutants, we take all of them as the new population; otherwise, we randomly select M of them.

The mutation operators applied by `mut` [4] resemble the edit operations a user would make on the initial regexp r_i to repair it. If r_i is not too faulty, the mutation operators should be sufficient to repair it; the assumption that the user has defined a software artefact (in this case, a regexp) close to the correct one is know in literature as *competent programmer hypothesis* [20].

This is the *repair* part of the approach, that possibly removes syntactical faults.

4 Experiments

To validate our approach, we need to select some *SULs* that we can query and a set of regexps to be used as initial guesses r_i. We have performed two types

[4] If there is more than one regexp with the highest fitness, the process randomly selects one.

Table 1: regexps used as r_i in the experiment with the real web service

id	expression	length	# oper.
r_i^1	[0-9]{4}-(1[0-2]—0[1-9])-(3[01]—0[1-9]—[12][0-9])T(2[0-3]—[01][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?(Z)?	109	45
r_i^2	(-?([1-9][0-9]*)?[0-9]{4}-(1[0-2]—0[1-9])-(3[01]—0[1-9]—[12][0-9])T(2[0-3]—[01][0-9]):([0-5][0-9]):([0-5][0-9])(.[0-9]+)?(Z)?	126	54
r_i^3	([0-9]{4}-?(1[0-2]—0[1-9])-?(3[01]—0[1-9]—[12][0-9])(2[0-3]—[01][0-9]):?([0-5][0-9]):?([0-5][0-9])	99	41
r_i^4	(2[0-3]—[01][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?(Z—[+-](2[0-3]—[01][0-9]):[0-5][0-9])?	93	37
r_i^5	([0-9]{4}(((\-)?(00[1-9]—0[1-9][0-9]—[1-2][0-9][0-9]—3[0-5][0-9]—36[0-6]))?—((\-)?(1[0-2]—0[1-9]))?—((\-)?(1[0-2]—0[1-9])(\-)?(0[1-9]—[12][0-9]—3[01]))?—((\-)?W(0[1-9]—[1-4][0-9]5[0-3]))?—((\-)?W(0[1-9]—[1-4][0-9]5[0-3])(\-)?[1-7])?)?)	235	90

of experiments. In the first one, the *SUL* is a real web service for which we want to learn a regexp representing the language the *SUL* accepts. This type of experiment is useful to assess the viability of our approach, but suffers from several limitations discussed below. In the second type of experiments, we have simulated a *SUL* in a controlled environment and this has allowed us to better estimate the effectiveness of our method.

The process has been implemented in Java using Watchmaker [26] as evolutionary framework. Experiments have been executed on a Linux machine, Intel(R) Core(TM) i7, 16 GB RAM. Code, benchmarks, and results are available online at `https://github.com/fmselab/evo_regexp_learn`.

4.1 Learning the language of a real web service

In this experiment, we want to discover the format of dates and times accepted by the API services of the timeanddate.com web site. That site offers, among other services, a time API service that does time zone conversions. When the user calls that service, (s)he must provide the date and time in ISO format (as it is claimed on the web site) and the service will convert from one time zone to another one. The conversion can be obtained by calling a web service like
`https://api.xmltime.com/converttime?fromid=750&toid=146`
`&iso=DATETIME;version=2&accesskey=KEY&expires=TIMESTAMP`
`&signature=SIGNATURE`.
If {`DATETIME`} is in the wrong format (or the query is incorrect), the web service returns an error code, otherwise it returns a json message with the conversion of {`DATETIME`}.

We have performed the experiment with 5 initial regexps (see Table 1) we found on the Internet representing possible ISO dates and times[5] and called our approach to test and repair them w.r.t. the web service. We executed the approach with 7 combinations of its two parameters, the total *timeout* (used in

[5] The ISO 8601 format does not come with an official regexp and it is regularly updated, so several versions exist.

Table 2: Results of the experiment with the real web service

(a) Aggregated by approach setting

TO (s) total	TO (s) gen	avg # str	avg # gen	avg fitness of r_f
180	10	835.4	7.2	0.99
180	30	1272.8	3.6	0.99
300	30	1478.6	6.4	0.99
300	60	2114	3.8	0.99
600	30	1831.2	11.6	1
600	60	2769.6	7	0.99
600	120	3445.6	3.8	1

(b) Aggregated by r_i (average)

r_i id	# str	# gen	# failures in r_i	fitness of r_f
r_i^1	1914.0	7	115.4	0.99
r_i^2	1975.7	6.4	195.7	0.98
r_i^3	2450.6	6.1	219.9	1
r_i^4	2662.9	7.4	161.2	1
r_i^5	816.3	4	106.2	0.99

the termination phase) and the timeout *genTimeout* on string generation (see Alg. 1 and Sect. 3). Table 2a reports, for each setting, the results in terms of average number of generated strings, average number of generations, and average final fitness. As expected, increasing the total timeout allows testing more strings against the *SUL*. Moreover, for a given total timeout, increasing *genTimeout* allows generating more strings, but decreases the number of generations. The final fitness of the returned final regexp r_f is always almost 1. However, in this case, even when the fitness is 1, the only assurance that we have is that r_f evaluates correctly all the generated strings; we cannot be sure that the final regexp r_f actually represents the language accepted by the service since we do not have access to the server internals.

Table 2b reports the results aggregated by r_i (average across the approach settings) with also the average number of strings wrongly evaluated by r_i. We observe that the highest number of strings is generated for r_i^3 and r_i^4, while the lowest number is generated for r_i^5. This is probably related to the size of r_i (length and number of operators in Table 1): bigger regexps as r_5 produce big populations requiring more time in the evolution phase (during mutation) and in the fitness update during the evaluation phase. We observe that all the initial guesses of regexps were wrong, as proved by the number of generated strings that are evaluated differently by the initial regexp r_i and by the *SUL*.

When starting from r_i^3 and r_i^4, the process always terminates with a regexp r_f having final fitness 1. In these cases, the final regexp evaluates correctly all the generated tests. However, as said before, since we do not know exactly the language of the *SUL*, it is impossible to check if the final regexps are completely correct (even when their fitness is 1). For this reason, we perform the experiments described in Sect 4.2 with a known *SUL* language.

4.2 Controlled experiment

In this experiment, we assume that the *SUL* itself is defined by a regexp r_{SUL}, unknown to the user, but available to us. In this way, we can measure the quality

of the final regexp r_f w.r.t. the initial regexp r_i, by comparing both of them to the correct regexp r_{SUL} representing the *SUL*.

We have taken as *SULs* 20 different matching tasks (e.g., email addresses, credit cards, social security numbers, zip codes, Roman numbers, etc.) from two websites [25,24]. For each task, we have taken a regexp as r_{SUL} and another regexp developed for the same matching task as initial regexp r_i. We have randomly chosen the two regexps (r_i and r_{SUL}) in a way that they are never equivalent (i.e., a repair is necessary) and moreover, they are also syntactically quite different (i.e., the repair is not trivial and the competent programmer hypothesis may not hold).

The initial regexps r_i are between 17 and 279 chars long (60.65 on average) and have between 7 and 112 operators (26.25 on average). All the regexps operators are considered, as shown in the results reported online.

In order to evaluate our approach, we introduce a measure of the reduction of conformance faults in the final regexp r_f. In order to check whether the final regexp r_f captures the regular language described by the *SUL*, we check the equivalence between r_f and r_{SUL}. For non-totally repaired regexps, we need to count the number of conformance faults. We use the measure F_r that counts the number of strings that a regexp r does not evaluate correctly (i.e., wrongly accepts or wrongly rejects). Since the number of such strings is possibly infinite, we have to consider only strings of length up to n. The number of faults of a regexp r is defined as follows, where $\mathcal{L}^n(r_x)=\mathcal{L}(r_x) \cap (\bigcup_{i=0}^{n} \Sigma^i)$:

$$F_r^n = |\mathcal{L}^n(r) \oplus \mathcal{L}^n(r_{SUL})|$$

In order to know whether the repaired regexp r_f is better than the initial regexp r_i, we compute $\Delta F = F_{r_f}^n - F_{r_i}^n$ with a fixed n. In the experiments, we set n=100 to restrict the evaluation to the strings of length n up to 100. If $\Delta F < 0$ or the final regexp r_f is totally repaired, the process has *improved* the original regexp; if ΔF=0, the process did not remove any conformance fault (or removed and introduced faults equally); otherwise, ΔF>0 means that the process has *worsened* the regexp by introducing more faults.

To better evaluate the approach, we measure the fault reduction (*FR*) as percentage change of the number of faults between the final and initial regexps r_f and r_i:

$$FR = \frac{F_{r_f}^n - F_{r_i}^n}{F_{r_i}^n} * 100$$

As done in the experiment with the real *SUL* (see Sect. 4.1), we performed a series of experiments by varying the two main parameters of the proposed approach, the total *timeout* and the timeout *genTimeout* on string generation; in total, we experimented with 7 combinations of these two parameters. We executed 10 runs of each experiment, and all the reported data are the averages of the 10 runs. In all the experiments, we fix the parameters of the evolutionary algorithm as follows: in the selection phase, we use truncation with K=5%, and as multiplicative factor H for the population size we use 100. Table 3a

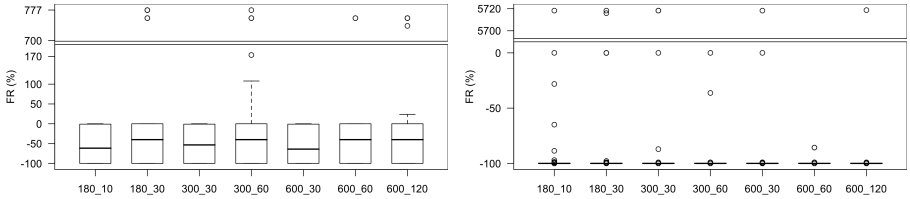
Table 3: `Bench`

(a) Results

TO (s) total	TO (s) gen	Rep (%)	Impr (%)	Wor (%)	avg *FR*	avg # str	avg # gen
180	10	25	85	0	-53.9	1075.2	14.9
180	30	23.5	81.5	3.5	-26	1202.8	8.1
300	30	25.5	85	0	-53.6	1343.8	12.4
300	60	24	82	3	-37	1415	10.1
600	30	28.5	85	0	-54	1471.7	21.2
600	60	28	83.5	1.5	-44.3	1577.7	15.9
600	120	23	82.5	2.5	-35.8	1603.1	13.5
AVG		25.4	83.5	1.5	-43.5	1384.2	13.7

(b) Results using random strings

TO (s) total	TO (s) gen	Rep (%)	Impr (%)	Wor (%)	avg *FR*	avg # str	avg # gen
180	10	0.5	16.5	82.5	1.9e+296	49964.2	6.6
180	30	0	22.5	73	9.6e+288	81408.9	3.1
300	30	0	16.5	79.5	3.8e+296	114560.6	4.5
300	60	0	24	72	9.6e+288	162314.8	2.8
600	30	0	16.5	82.5	1.9e+296	133290.6	6.8
600	60	0	26	71.5	1.3e+294	176726.8	4.3
600	120	0	18.5	77	3.8e+296	250812.2	2.9
AVG		0.1	20.2	76.9	1.6e+296	138439.7	4.4



(a) Fault reduction – `Bench` (b) Fault reduction – `MutBench`

Fig. 1: Fault reduction

reports the experimental results in terms of percentage of completely repaired (*Rep*) regexps, percentage of improved (*Imp*) regexps (i.e., for which the number of faults decreased), percentage of worsened (*Wor*) regexps (i.e., for which the number of faults increased), average fault reduction (*FR*) among the regexps, average number of generated strings, and average number of generations of the evolutionary algorithm. The execution time corresponds to the total timeout (*TO total*).

In the following, we analyse the results using different research questions.

RQ1 Is the proposed process able to learn the regular language of the *SUL*? What is the improvement in terms of fault reduction?

First of all, we are interested in checking whether the proposed approach is able to learn the regular language of the *SUL*. From Table 3a, we observe that, with any setting, we are always able to completely repair around 25.4% of the regexps, and improve around 83.5%. With some settings, the process can also worsen the regexp (maximum 3.5%). However, on average, the number of faults is always decreased; Fig. 1a shows the distribution of the fault reduction among the regexps for each setting. We notice that the process worsens some regexps when the timeout for string generation (*TO gen*) is higher; it seems that generating too many strings for one generation is not beneficial and it is more productive to spend time in evaluating new generations.

RQ2 Is using distinguishing strings effective?

We are here interested in investigating if using distinguishing strings generated among the mutants is effective. In order to do this, in Alg. 1, instead of generating distinguishing strings, we generate random strings. The maximum number of strings that can be generated (if the timeout does not occur) is still $C_2^{|P|}$. Table 3b reports the results by using random strings. We observe that the approach is almost never able to totally repair the regexp: it at most improves 26% of the benchmarks, and it always worsens at least 71.5% of them. We can also observe that, since the test generation is fast, many more strings are generated (from 1 to 2 orders of magnitude more); however, their fault-detecting capability is much lower than that of the distinguishing strings. This confirms that targeting conformance faults among mutants in the search is effective.

RQ3 How many distinguishing strings are generated? How long does the *SUL* take to evaluate them?

Differently from [3], we assume that calling the *SUL* (acting as oracle) is not very expensive and, so, there is not urgency to limit the number of string evaluations. However, in some settings (e.g., database query), calling the *SUL* may have a non-negligible cost. We here try to assess such cost. In the experiments (see Table 3a), we at most generate 1603 strings with one setting having the highest timeout of 600 secs (10 mins). If we assume that invoking the *SUL* for a single input is expensive and takes 1 sec (we used such delay between two queries in the experiment of Sect. 4.1 to wait for the service response and to avoid queries were interpreted as DoS attacks), the evaluation of the strings takes 26.7 mins; therefore, in this case, the total time would be 36.7 mins.

RQ4 How are the results if the competent programmer hypothesis hold?

Benchmark set `Bench` has been built by taking regexps pairs r_i and r_{SUL} that are very different from each other (the average Levenshtein distance between the initial regexp r_i and the oracle r_{SUL} is 36.65). However, fault-based approaches usually assume the *competent programmer hypothesis*, stating that the user defined a regexp close to the correct one (different for one or few syntactic faults).

Therefore, we want to assess the process effectiveness under the assumption of the competent programmer hypothesis. We built another benchmark set, `MutBench`, as follows. We took each oracle regexp r_{SUL} of `Bench` and we randomly mutated it introducing n faults (with $n = 1, \ldots, 3$), obtaining three faulty versions of the oracle. In this way, `MutBench` contains 60 pairs of regexps. The regexps are between 38 and 277 characters long (111.7 on average) and contain between 5 and 59 operators (26.82 on average). Then, we have experimented with our approach using `MutBench`; Table 4a reports the aggregated results for the experiments. Regarding RQ1, we observe that the results are much better than the benchmark `Bench`. Any timeout configuration is able to totally repair at least 78.5% of the regexps, and all of them improve more than 99.5% of the regexps. Fig. 1b reports detailed results of fault reduction.

Table 4: `MutBench`

(a) Results

TO (s) total	TO (s) gen	Rep (%)	Impr (%)	Wor (%)	avg *FR*	avg # str	avg # gen
180	10	79.8	99.7	0.3	-87.5	797.5	1010.3
180	30	78.7	99.7	0.3	-80.3	871.9	4177.6
300	30	79.8	99.5	0.5	-79.7	974.5	1446
300	60	78.5	100	0	-99.7	1042.6	3373.9
600	30	86	99.8	0.2	-90.1	1130.9	10843.4
600	60	83.7	100	0	-99.9	1128.7	7667.9
600	120	82.3	99.8	0.2	-90.3	1191.8	7578.8
AVG		81.3	99.8	0.2	-89.6	1019.7	5156.8

(b) Results using random strings

TO (s) total	TO (s) gen	Rep (%)	Impr (%)	Wor (%)	avg *FR*	avg # str	avg # gen
180	10	0.3	35.3	62.5	1.9e+294	41721	6.4
180	30	0.2	31.8	62.8	9.4e+293	70417	3.1
300	30	0.5	34.2	64.3	6.3e+293	92685.4	4.6
300	60	0.2	33.8	61.5	3.1e+293	126599	2.9
600	30	0.8	35.5	63.7	1.6e+294	121126	7
600	60	0.2	33.7	64	7.1e+292	158346	4.5
600	120	0.5	31.5	63.7	6.2e+293	191531.7	2.9
AVG		0.4	33.7	63.2	8.6e+293	114632.3	4.5

Table 5: Results of a state-of-the-art approach IA [3]

Benchmark	Rep (%)	Impr (%)	Wor (%)	avg *FR* (%)	avg # str	avg time (s)
`Bench`	19	65.5	3	6e+283	636.8	61.5
`MutBench`	77	97.3	0	-96	447.2	140.6

We also evaluated the approach using randomly generated strings on `MutBench`, and the results are reported in Table 4b. Results are slightly better with benchmark `MutBench` than with `Bench` (compare the results in Table 3b), but still almost no regexp is repaired and at most only 35.5% of the regexps are improved.

RQ5 Does the proposed approach improve the state of the art?

A related approach that just relies on *interactive* string evaluation of the *SUL* is the one we presented in [3] (from now on, called IA). IA is quite different from the current *evolutionary* approach (from now on, called EA), as it does not have a population of candidates, but a single candidate; when a mutant *better* than the current candidate is found, the candidate is changed; the repairing process stops when no better mutant is found. Four policies are proposed for choosing the new candidate (*Greedy*, *MultiDSs*, *Breadth*, and *Collecting*). The only similarity with EA is that candidates are generated by mutation and fault-detecting strings are used as tests.

We applied IA to the benchmarks `Bench` and `MutBench`, using the setting *Breadth* that in [3] showed, on average, the best performance in terms of fault reduction. Table 5 shows the results for the two benchmarks. We observe that for `MutBench` (i.e., if the competent programmer hypothesis holds), IA has a good performance, similar to the one of EA (cfr. with Table 4a); however, EA with the best setting can completely repair 9% more regexps. For `Bench`, EA has an even better performance (cfr. with Table 3a); it totally repairs from 4% to 9.5% more regexps, improves from 16% to 19.5% more regexps, and worsens (in the worst case) almost the same number of regexps. IA is meant for interactive

testing with a user and, therefore, it tries to generate not too many strings; for this reason, it always produces fewer strings and takes less time than EA.

We checked if the results are statistically significant by performing the Wilcoxon signed-rank test[6] between the results of EA and IA. Regarding totally repaired (Rep) and improved (Impr) regexps, EA is significantly better, while there is no significant difference for worsen regexps (Wor) and fault reduction (*FR*). As expected, EA is worse in terms of number of generated strings (# str), as IA is designed to generate few strings.

We conclude that a conservative approach as IA should be used if there are constraints on time and number of strings (usually, when the oracle is the user), while our approach EA should be used when the oracle can be called several times (as our *SUL*).

RQ6 How does our approach compare with L*?

The classical algorithm for learning regexps is L*. L* starts from the strong assumption of having the possibility to check the equivalence of a regexp w.r.t. the *SUL*. This assumption is unpractical so there exist several attempts to use L* in combination with approximate equivalent checkers. These methods are generally based on conformance testing methods: some tests are generated from the current hypothesis and executed over the *SUL* and, if they are all equally accepted or rejected by using membership queries, then the hypothesis is considered equivalent to the *SUL*. Conformance checking can be expensive though, since methods like the W-method, Wp-method, or UIO-method require an exponential number of tests, although they can provide some guarantees (e.g., assuming the target system has at most N additional states). Other approximate methods randomly generate a maximum number of tests but they cannot guarantee correctness. To study the feasibility of using L* with an approximate equivalent checker, we have executed L* using LearnLib [23,14] over the regexps of `Bench`. For each oracle regexp r_{SUL} in `Bench`, we have tried to learn it using L* with different equivalence checkers: different versions of the random equivalence checker (between 10 and 10^6 tests of length between 10 and 10^2) and three versions of the W-method (with exploration depth from 1 to 3); we restrict the alphabet to the readable range of ASCII, because using Unicode would make L* unusable. Table 6 reports results for 6 versions of the random method and all the 3 versions of the W-method. It shows the percentage of times that L* is better/worst than the best setting of EA (600-30 in Table 3a) in terms of final number of faults in the final regexp r_f; moreover, it reports the average number of strings (i.e., tests) and average execution time across the benchmarks. We observe that increasing the number of tests and their length in the random method does not improve the learning capability of L*; the W-method does improve the learning capability by increasing the exploration depth, but at the expenses of an exponential number of tests.

[6] We checked that the distributions are not normal with the non-parametric Shapiro-Wilk test.

Table 6: L* results (R-n-m: random method with m tests of length up to n. W-n: W-method with exploration depth n)

Eq. checker	Comparison with EA better (%)	worse (%)	avg # str	avg time (s)
R-10-10^2	35	61.75	200.75	0.01
R-10-10^4	35	61.75	10100.75	0.02
R-10-10^6	34.75	62.25	1042574.6	1.93
R-10^2-10^2	35	61.75	200.75	0
R-10^2-10^4	35	61.75	10100.75	0.16
R-10^2-10^6	34.75	62.25	1076471.6	16.72
W-1	35	61.75	53491.45	0.17
W-2	37	58.50	6.47×10^6	15.55
W-3	38	56.25	1.93×10^8	2668.81
EA 600–30			1471.7	600

5 Related work

A first set of related works regards the use of the L* algorithm [2] and its variants for specification mining which shares many similarities with regexp learning. In the L* algorithm, a *Student* tries to learn an unknown regular automaton 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 is accepted by S or not. In a *conjecture query*, the student provides a regular automaton S' and the teacher answers *yes* if S' corresponds to S, or with a *wrong* string t (as our distinguishing string) otherwise. L* can be used also to learn a black-box system *SUL* by replacing the ability of the teacher to test conjectures by a random sampling. This works only under the strong assumptions that the Student *gets an approximately correct hypothesis with high probability and the problem of testing a conjecture against randomly drawn strings is sufficiently tractable* [2]. Lee and Yannakakis [15] showed how to use L* for conformance testing, to check if an FSM I implements a specification S with n states, and to repair I if a fault has been found. Nevertheless, serious practical limitations (like the number of states and the types of faults) are associated with such experiments (see [15] for a complete discussion). A more recent survey on automata learning can be found in [13], and a survey on the usage of model learning for testing can be found in [1].

For these reasons, genetic and evolutionary algorithms for regular expression learning are preferred instead. They are mainly used for automatically synthesizing a regular expression from sets of accepted/rejected strings. Bartoli and al. [7] propose a passive approach for synthesizing a regular expression automatically, based only on examples of the desired behavior, i.e., strings described by the language and of strings not described by the language. The approach is improved in [8] by proposing an active learning approach aimed at minimizing the user annotation effort: the user annotates only one desired extraction and then merely answers extraction queries generated by the system. Differently from them, we do not start from scratch by generating the initial population, and we do not use predefined set of accepted/rejected strings.

Another approach for regexp synthesis from a set of labeled strings is ReLIE [16]. It is a passive learning algorithm that, given an initial regexp and a set of labeled strings, tries to learn the correct regexp. It performs some regexp

transformations (a kind of mutation); however, it is not an evolutionary algorithm, as it exploits a greedy hill climbing search that chooses, at every iteration, the regexp with the highest fitness value.

We share some ideas with our previous work [3]. Mutation operators and use of automata is in common with that paper. However, in [3] we did not use an evolutionary algorithm, but a greedy approach very similar to [16]. Moreover, we assumed that the oracle is the user and the approach was meant to be interactive. For this reason, we tried to generate very few strings. In RQ5, we demonstrate that the current evolutionary approach is much more efficient in repairing regexps. We believe that a population-based evolutionary algorithm is able to reduce the risk of being stuck in local optimum since possible solutions are continuously evaluated and generated.

The idea of co-evolving the population of solutions together with the tests is not new; e.g., the approach in [28] evolves software artefacts and test cases in tandem.

Our approach has some similarities with *automatic software repair* [27,5,21,18,22] and automatic repair of specifications [12].

6 Conclusions and Future Work

In the paper, we proposed an evolutionary approach to learn the regexp modeling the unknown language accepted by a given system. The user gives an initial guess of a regexp, and the approach tries to learn the correct regexp by evolving a population of possible solutions obtained by mutation. The fitness function is based on the strings a candidate evaluates correctly. The testing strings evolve together with the population: at each evolution step, they are built as those strings able to distinguish the population members. Experiments have been done on a case study of a real web service, and on some benchmarks. Results show that the approach, on average, is able to completely repair 81.3% of the initial regexps and improve 99.8% of them. Even if the initial guess of the user is not accurate, 25.4% of initial regexps are totally repaired, and 83.5% improved.

In the evaluation, we used a given setting for the parameters of the evolutionary approach. As future work, we plan to use a tool as irace [17] to find the best setting.

Our approach could be adapted for regexp synthesis (but not passive) as well, e.g., by taking as r_i the first random string that is accepted by the *SUL*. Although the experiments suggest that starting from a very faulty regexp makes finding the correct solution very hard, we are interested to experiment also with this use of our technique.

We plan to improve the fitness in several directions: considering readability, weighting differently false negative and false positive, and making the fitness more sensitive to minor improvements by introducing a measure of *partial acceptance*.

References

1. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model Learning and Model-Based Testing, pp. 74–100. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_3
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
3. Arcaini, P., Gargantini, A., Riccobene, E.: Interactive testing and repairing of regular expressions. In: Medina-Bulo, I., Merayo, M.G., Hierons, R. (eds.) Testing Software and Systems. pp. 1–16. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-99927-2_1
4. Arcaini, P., Gargantini, A., Riccobene, E.: Fault-based test generation for regular expressions by mutation. Software Testing, Verification and Reliability **29**(1-2), e1664 (2019). https://doi.org/10.1002/stvr.1664
5. Arcuri, A.: Evolutionary repair of faulty software. Applied Soft Computing **11**(4), 3494 – 3514 (2011). https://doi.org/10.1016/j.asoc.2011.01.023
6. Bartoli, A., Davanzo, G., De Lorenzo, A., Medvet, E., Sorio, E.: Automatic synthesis of regular expressions from examples. Computer **47**(12), 72–80 (Dec 2014). https://doi.org/10.1109/MC.2014.344
7. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Inference of regular expressions for text extraction from examples. IEEE Trans. on Knowl. and Data Eng. **28**(5), 1217–1230 (May 2016). https://doi.org/10.1109/TKDE.2016.2515587
8. Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Active learning of regular expressions for entity extraction. IEEE Transactions on Cybernetics **48**(3), 1067–1080 (mar 2018). https://doi.org/10.1109/tcyb.2017.2680466
9. Bergadano, F., Gunetti, D.: Inductive Logic Programming: From Machine Learning to Software Engineering. MIT Press, Cambridge, MA, USA (1995)
10. Bongard, J., Lipson, H.: Active coevolutionary learning of deterministic finite automata. J. Mach. Learn. Res. p. 28 (2005), 00076
11. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer Verlag (2003)
12. 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. pp. 1245–1248. ICSE '13, IEEE Press, Piscataway, NJ, USA (2013). https://doi.org/10.1109/ICSE.2013.6606689
13. Howar, F., Steffen, B.: Active Automata Learning in Practice, pp. 123–148. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_5
14. Isberner, M., Howar, F., Steffen, B.: The Open-Source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 487–495. Springer International Publishing, Cham (2015)
15. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines-a survey. Proceedings of the IEEE **84**(8), 1090–1123 (1996). https://doi.org/10.1109/5.533956
16. Li, Y., Krishnamurthy, R., Raghavan, S., Vaithyanathan, S., Jagadish, H.V.: Regular expression learning for information extraction. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing. pp. 21–30. EMNLP '08, Association for Computational Linguistics, Stroudsburg, PA, USA (2008)

17. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016). https://doi.org/10.1016/j.orp.2016.09.002
18. Martinez, M., Monperrus, M.: Mining software repair models for reasoning on the search space of automated program fixing. Empirical Softw. Engg. **20**(1), 176–205 (Feb 2015). https://doi.org/10.1007/s10664-013-9282-8
19. Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2010), `http://www.brics.dk/automaton/`
20. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: An analysis and survey. In: Advances in Computers. Advances in Computers, Elsevier (2018). https://doi.org/10.1016/bs.adcom.2018.03.015
21. Pei, Y., Furia, C.A., Nordio, M., Wei, Y., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. IEEE Transactions on Software Engineering **40**(5), 427–449 (May 2014). https://doi.org/10.1109/TSE.2014.2312918
22. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: A comprehensive survey. IEEE Transactions on Evolutionary Computation **22**(3), 415–432 (jun 2018). https://doi.org/10.1109/tevc.2017.2693219
23. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: A framework for extrapolating behavioral models. Int. J. Softw. Tools Technol. Transf. **11**(5), 393–407 (Oct 2009). https://doi.org/10.1007/s10009-009-0111-8
24. Regexlib.com. `http://www.regexlib.com`, accessed: 2019-05-24
25. Regular-expressions.info. `http://www.regular-expressions.info/`, accessed: 2019-05-24
26. Watchmaker. `https://watchmaker.uncommons.org/`, accessed: 2019-05-24
27. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Commun. ACM **53**(5), 109–116 (May 2010). https://doi.org/10.1145/1735223.1735249
28. Wilkerson, J.L., Tauritz, D.: Coevolutionary automated software correction. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO'10. ACM Press (2010). https://doi.org/10.1145/1830483.1830739