String Generation for Testing Regular Expressions

Lixiao Zheng[1], Shuai Ma[2], Yuanyang Wang[1] and Gang Lin[1]

[1]*College of Computer Science and Technology, Huaqiao University, Xiamen, China*
[2]*SKLSDE Lab, Beihang University & Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China*
Email: mashuai@buaa.edu.cn

Regular expressions have been widely studied due to their expressiveness and flexibility for various applications. A common yet challenging way to ensure the quality of regular expressions is regular expression testing. In this work, we study coverage criteria based string generation for testing regular expressions. Firstly, we propose a notion of pairwise coverage criterion for regular expressions, and analyze the subsumption relationships with existing coverage criteria for both regular grammars and finite automata. Secondly, we design an algorithm that given as input a regular expression, outputs a small set of strings that satisfies the pairwise coverage criterion. Thirdly, we extend the coverage criterion and the generation algorithm to further deal with regular operators counting and interleaving. Fourthly, we experimentally demonstrate the effectiveness and efficiency of our algorithms by testing element type definitions of real-world XML schemas. Finally, we identify more applications of pairwise coverage and its corresponding generation algorithm, and show that they can be used to generate characteristic samples for certain regular expression learning algorithms that follow Gold's learning paradigm of learning (identification) in the limit. These results are not only theoretically meaningful, but are also useful for practical applications involved with regular expressions.

Keywords: String generation; testing; regular expressions; finite automata; regular grammars; coverage criterion.

Received 04 January 2018; revised 21 July 2018

1. INTRODUCTION

Regular expressions, due to their expressiveness and flexibility, have long been used in a large variety of application domains such as text processors, programming languages, query processing, and XML schema languages [1, 2]. A faulty regular expression may cause unwanted consequences. Consider for example an XML database storing information of a list of books. Each book should have a title, followed by one or more authors and one optional publisher. This requirement can be specified by regular expression $title, author^{+}, publisher?$. If we incautiously write the expression as $title, author^{*}, publisher?$, then books with no authors may also be accepted by the database, which obviously does not conform to the expectation. Actually, previous studies [3, 4] show that regular expressions are quite error-prone, i.e., the defined languages do not agree exactly with the intended languages. Therefore, ensuring the (semantic) correctness of regular expressions is a vital prerequisite for their use in practical applications.

Testing is a common way to ensure the quality of regular expressions. The purpose of regular expression testing is to check whether the defined language meets the specification of users. One straightforward way to achieve this purpose is to automatically generate a number of strings from the regular expression under testing, and to check whether they comply with the intended language [3, 4, 5]. Recall the example given before. If a meaningful set of strings are generated from the incorrect expression $title, author^{*}, publisher?$, and contains strings with no $author$, then the fault of using improper regular operator ($*$ instead of $+$) can be detected. Negative strings, i.e., strings that are rejected by the regular expression, are also useful for the testing purpose. However, the generation of negative strings is typically based on the generation of positive ones. For example, negative strings can be obtained by treating ? as repetition of twice and + as repetition of zero times during the generation process [3]. Therefore, we focus on the generation of positive strings for testing regular expressions in this study.

In formal language theory, two problems concerning the generation of regular languages have been studied:

(1) *sampling* that generates a uniformly random string of length n of a regular language [6, 7, 8, 9] so that strings of length n in that language have all the same probability to be returned, and (2) *enumerating* that tries to enumerate all the distinct strings or all strings of length n of a regular language in lexicographical order [10, 11, 12, 13]. However, most existing sampling and enumerating algorithms take finite automata or regular grammars as their inputs. To our knowledge, there is only one sampling algorithm [14] taking regular expressions directly as its input, which accepts regular expressions without Kleene star only, and is a purely theoretical analysis without experimental validations. Besides, sampling and enumerating are not suitable for testing purposes. For example, sampling algorithms randomly produce one string at a time, which makes it difficult for testers to decide when the testing activities should be terminated. Enumerating algorithms usually produce a large number of strings, i.e., all strings or all strings of a certain length in the regular language, which imposes a considerable overhead on the testing task. That is, there is a need to automatically generate strings for regular expressions that are capable of detecting the potential errors with a reasonable overhead.

In software testing community, coverage criterion is a common measure to justify the sufficiency of a particular test set, and provides a basis for test data generation algorithms [15]. Coverage criteria are usually defined with respect to programs. For instance, the most well-known structural coverage criteria for programs include statement coverage, branch coverage and path coverage, which require that each statement, branch and path in the program's code is executed by the test set. Researchers from formal language community have adopted this concept to grammars, and proposed several coverage criteria for different grammar formalisms. Purdom [16] introduced rule coverage for context-free grammars, which requires that each grammar rule is used at least once in deriving the test strings. Based on rule coverage, more precise criteria for context-free grammars were proposed such as context-dependent rule coverage [17] and length-k successor coverage [18]. A notion of two-dimensional approximation coverage was proposed for attribute grammars [19]. String generation algorithms with respect to some of these coverage criteria for grammars have also been devised and implemented [16, 20, 21, 22].

For regular expressions, however, there has been little work devoted to the coverage criteria and the related string generation problems. Although one can first convert regular expressions to regular grammars, a subclass of context-free grammars, and then use coverage based generation algorithms for context-free grammars to generate strings for regular expressions, this indirect approach is not very suitable. The generated strings only fulfill some coverage criterion for the translated grammars, and it is hard to measure whether they are sufficient for testing the regular expressions. Finite automata are another formalism equivalent to regular expressions, and can be viewed as control flow graphs of programs and several coverage criteria for them such as edge coverage and path coverage exist. However, using finite automata and their existing coverage to generate strings for regular expressions has the same disadvantages as regular grammars. Therefore, a more suitable approach is to define coverage criteria for regular expressions directly, and to develop algorithms generating strings directly from regular expressions.

Apart from standard regular expressions that only use standard regular operators such as concatenation, alternation and Kleene star, many applications allow the use of additional operators. For instance, egrep [23], Perl patterns [24] and XML schema language XSD [25] allow the use of counting operator that defines the minimal and maximal number of times a regular construct can be repeated. The XML schema language Relax NG [26] supports the use of interleaving operator that specifies unordered concatenations. Although these operators can be equivalently expressed by standard operators, they have a drastic impact on succinctness. For instance, it has been shown that the complexity of translation from expressions extended with counting and interleaving into standard expressions is exponential [27] and double exponential [28], respectively. Hence, it is also necessary to study the coverage criteria and corresponding generation algorithms to handle such operators, which has been ignored by the community.

Contributions. To this end, we study coverage criteria for both standard and extended regular expressions, and string generation algorithms for regular expressions based on different coverage criteria.

(1) We propose a notion of pairwise coverage for regular expressions, inspired by pairwise testing from traditional software testing techniques (Section 3). For comparison, we also consider a much more rigorous but trivial criterion, namely combination coverage, such that the Kleene star (*zero-or-more repetitions*) is restricted to a fixed number k. We also extend these coverage criteria to deal with non-standard regular operators: counting and interleaving.

(2) We give an analysis of our coverage criteria for regular expressions and four existing criteria for regular grammars and finite automata (Section 4). We study the subsumption relationships among the criteria defined with respect to the three different formalisms for regular languages, and show that pairwise coverage, although weaker than combination and path coverage, is stronger than all the other existing practical criteria for grammars and automata.

(3) We develop an algorithm that takes as input a regular expression and outputs a small set of strings that satisfies pairwise coverage (Section 5). The

generations are conducted by performing a bottom-up traversal along the abstract syntax trees of regular expressions. For the purpose of comparison, we develop an algorithm for combination coverage as well. Both algorithms accept not only standard regular expressions but also expressions extended with counting and interleaving operators.

(4) We experimentally verify the effectiveness and efficiency of our algorithms by applying them to the testing of real-world regular expressions, specifically, element type definitions of real-world XML schemas (Section 6). We use mutation testing technique to assess the quality of the generated test data. We find that test data produced by coverage criteria based generation algorithms has in general a higher fault detection ability than the randomly generated one. Comparing pairwise coverage with the stronger combination coverage, we find that the former requires a much smaller test set, but has almost the same fault detection ability compared with the latter. Moreover, our algorithms can help to reveal faults ignored by other existing string generation tools.

(5) We discuss more applications of pairwise coverage and the related string generation algorithm (Section 7). In particular, we show that they are very useful for generating characteristic samples for some regular expression learning algorithms that are based on Gold's learning model of learning (identification) in the limit.

2. PRELIMINARIES

In this section, we introduce basic notions.

2.1. Regular Expressions

Following the notations in [29], we define the following. Consider an alphabet Σ of symbols.

A Σ-symbol (or simply symbol) is an element of Σ, a Σ-string (or simply string) is a finite sequence $w = a_1...a_n$ of Σ-symbols, and the empty string is denoted by ϵ.

The set of all strings over alphabet Σ is denoted by Σ^*, and a string language is a subset of Σ^*.

The concatenation of two strings w_1 and w_2 is denoted by $w_1 \cdot w_2$ or simply w_1w_2. The concatenation $L \cdot L'$ (or simply LL') of two string languages $L, L' \subseteq \Sigma^*$ is $\{ww' | w \in L, w' \in L'\}$. The concatenation $L \cdot \ldots \cdot L$ (i times) is also abbreviated by L^i, where $L^0 = \{\epsilon\}$.

A *regular expression* (or expression for short) over Σ is defined recursively as follows. (1) The empty string ϵ, the empty set $\emptyset$ and each symbol $a \in \Sigma$ is a regular expression; (2) If E_1 and E_2 are regular expressions, then $E_1 \cdot E_2$, $E_1|E_2$ and E_1^* are regular expressions. Here, operators $\cdot$, $|$ and $*$ denote concatenation, alternation and zero-or-more repetitions (i.e., Kleene star), respectively.

For convenience, we sometimes omit the concatenation operator, simply write E_1E_2, instead of $E_1 \cdot E_2$. To eliminate any ambiguity, parentheses have highest precedence, followed by Kleene star, concatenation, and alternation. Note that the above definition does not contain Kleene plus $+$ (one-or-more repetitions) and optional operator ?. This is not a restriction since they can be equivalently expressed as follows: $E^+ = E \cdot E^*$, and $E? = \epsilon|E$.

The language defined by a regular expression E, denoted by $L(E)$, is a subset of Σ^*, and is defined as follows. $L(\epsilon) = \{\epsilon\}$, $L(\emptyset) = \emptyset$ and $L(a) = \{a\}$ for some $a \in \Sigma$. If $E = E_1 \cdot E_2$, then $L(E) = L(E_1) \cdot L(E_2)$; If $E = E_1|E_2$, then $L(E) = L(E_1) \cup L(E_2)$; and finally if $E = E_1^*$, then $L(E) = \bigcup_{i=0}^{\infty} L(E_1)^i$.

What we have defined above is about standard regular expressions. We next give the formal definition of regular expressions extended with two additional operators counting $[l, r]$ and interleaving &: when E_1 is a regular expression, then $E_1^{[l,r]}$ is also a regular expression with $l, r \in \mathbb{N}$ and $0 \leq l \leq r$; when E_1, E_2 are regular expressions, then $E_1 \& E_2$ is also a regular expression. Such expressions are called extended regular expressions.

For an extended regular expression, the defined language is as follows: $L(E_1^{[l,r]}) = \bigcup_{i=l}^{r} L(E_1)^i$; $L(E_1 \& E_2) = \{w_1 \& w_2 | w_1 \in L(E_1) \text{ and } w_2 \in L(E_2)\}$ where by $w_1 \& w_2$ we denote the set of strings that is obtained by interleaving or shuffling w_1 and w_2 in every possible way. That is, $w_1 \& \epsilon = \epsilon \& w_1 = w_1, w_2 \& \epsilon = \epsilon \& w_2 = w_2$, and if both w_1 and w_2 are non-empty and $w_1 = aw_1', w_2 = bw_2'$, where a and b are single symbols, then $w_1 \& w_2 = a\{w_1' \& bw_2'\} \cup b\{aw_1' \& w_2'\}$. One can verify that, the same as binary operators $|$ and $\cdot$, & also obeys the associative law. For example, $L((ab)^{[2,4]}) = \{abab, ababab, abababab\}$, $L(a\&(b\&c)) = L((a\&b)\&c) = \{abc, bac, bca, cba, cab, acb\}$, and $L((ab)\&(cd)) = \{acbd, acdb, cabd, cadb, abcd, cdab\}$, i.e., $(ab)\&(cd)$ accepts all strings that have a occurring before b and c occurring before d.

In the sequel, unless otherwise stated, regular expressions mean both standard and extended expressions.

2.2. Regular Grammars and Finite Automata

Formally, a *context-free grammar* (or *grammar* for short) is a 4-tuple $G = (N, T, P, S)$, in which N and T are disjoint sets of *nonterminals* and *terminals* respectively, P is a set of *rules* of the form $X \rightarrow \alpha$ with $X \in N$ and $\alpha \in (N \cup T)^*$, and $S \in N$ is the *start symbol*. A *derivation step* is an element of the form $\beta X \gamma \Rightarrow \beta \alpha \gamma$ with $\beta, \gamma \in (N \cup T)^*$ and $X \rightarrow \alpha \in P$. A *derivation*, denoted as $\overset{*}{\Rightarrow}$, is a sequence of derivation steps. The language defined by G, denoted as $L(G)$, is the set of strings $L(G) = \{w \in T^* | S \overset{*}{\Rightarrow} w\}$.

A *regular grammar* is a grammar such that all the rules are of one of the following forms: $X \rightarrow \epsilon$, $X \rightarrow a$ or $X \rightarrow aY$ where X and Y are nonterminals and a

p_1: $S \rightarrow a\,A$	p_1: $S \rightarrow a\,A$
p_2: $S \rightarrow b\,A$	p_2: $S \rightarrow b\,A$
p_3: $A \rightarrow c\,A$	p_3: $A \rightarrow c\,B$
p_4: $A \rightarrow \varepsilon$	p_4: $A \rightarrow \varepsilon$
	p_5: $B \rightarrow c\,B$
	p_6: $B \rightarrow \varepsilon$
G_1	G_2

FIGURE 1. Two equivalent regular grammars that define the same language as regular expression $(a|b)c^*$. We use $p_1, ..., p_n$ to denote rules and S to denote the start symbol. The upper-case letters are used to distinguish nonterminals from terminals.

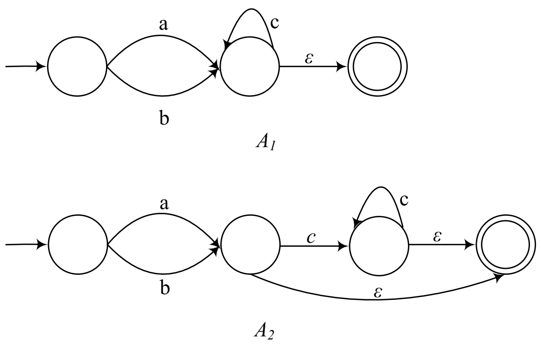

FIGURE 2. Two equivalent finite automata that define the same language as regular expression $(a|b)c^*$. The node with an edge coming in from nowhere denotes the start state, and the node with a double circle denotes a final state, respectively.

is a terminal. For any regular expression E, there is a regular grammar G such that $L(G) = L(E)$. Note that different regular grammars may define the same language. For instance, regular grammars G_1 and G_2 in Figure 1 define the same language as expression $(a|b)c^*$.

A *finite automaton* (or automaton for short) is a tuple $A = (\Sigma, Q, q_0, F, \delta)$, in which Q is a finite set of states, q_0 is the initial state, F is the set of final states, and δ is the transition function $\delta : Q \times \Sigma \rightarrow 2^Q$ that maps each pair of a state and symbol to a set of states. A run ρ of A on some string $w = a_1...a_n$ is a sequence of states $q_0, ..., q_n$, such that $q_i \in \delta(q_{i-1}, a_i)$ for each $i \in [1, n]$. Furthermore, when q_n is a member of F, we say that the run is accepting. The string language accepted by A is denoted by $L(A)$ and is defined as the set of strings w for which there exists an accepting run of A on w.

For any regular expression E, there is a finite automaton A such that $L(A) = L(E)$. Note that, similarly to regular grammars, different automata may define the same language. For instance, the two automata illustrated in Figure 2 are equivalent, defining the same language as expression $(a|b)c^*$.

3. COVERAGE CRITERIA FOR REGULAR EXPRESSIONS

In this section, we propose two coverage criteria for regular expressions, namely combination coverage and pairwise coverage. Combination coverage is simple and strong, but may introduce the combinatorial explosion problem. Hence, we introduce pairwise coverage, a loose but more practical coverage. We first define these criteria for standard expressions, and then extend them for expressions with counting and interleaving.

3.1. Coverage Criteria for Standard Expressions

In software testing, the strongest coverage is full coverage. That is, for a regular expression E, all strings of $L(E)$ are generated. However, $L(E)$ may be infinite, which is typically caused by Kleene star operators. One simple approach to eliminating this infiniteness is to restrict the repetitions within a limited range. We shall first define a trivial but relatively strong coverage, called combination coverage, by restricting the repetitions.

Combination coverage. Let E be a regular expression. A subset $S \subseteq L(E)$ achieves combination coverage for E if it contains the finite set $C(E)$ defined below by induction on the structure of E.

(1) If E is ϵ, $\emptyset$ or a for some $a \in \Sigma$, then $C(E) = L(E)$;

(2) If $E = E_1|E_2$, then $C(E) = C(E_1) \cup C(E_2)$;

(3) If $E = E_1 \cdot E_2$, then $C(E) = C(E_1) \cdot C(E_2)$;

(4) If $E = E_1^*$, then $C(E) = \{\epsilon\} \cup C(E_1) \cup C(E_1)^k$ for some $k > 1$.

A string set achieving combination coverage for expression E contains each possible combination of characteristic substrings generated by the subexpressions of E. Actually, the finite set $C(E)$ in the above definition is just the language of E when we restrict Kleene star $*$ to be three possibilities: zero, one and more-than-one repetitions. For example, the set $\{a, b, ac, bc, accc, bccc\}$ achieves combination coverage for expression $(a|b)c^*$ while for expression $(a|b)c^*(d|e)$ the set would be $\{ad, ae, bd, be, ace, acd, bce, bcd, accce, acccd, bccce, bcccd\}$. Note that if we fix the more-than-one repetition number, for example, as exactly 3, the minimal set achieving combination coverage for expression E is unique.

However, this coverage may introduce combinatorial explosion problem. Take expression $a_1^* a_2^* ... a_n^*$ for instance, there are at least 3^n strings required for achieving combination coverage. This obviously makes combination coverage impractical for use.

In light of this, we introduce a loose but practical coverage. The idea is inspired by pairwise testing [30]. Pairwise testing is a combinatorial method of software testing that, for each pair of input parameters to a system, tests all possible combinations of these two parameters. Pairwise testing is a good trade-off between test effort and test coverage. Using carefully chosen test cases, this can be done much faster than an exhaustive search of all combinations of all parameters. Empirical results have shown that pairwise testing is practical and effective for various types of software systems [31].

We adopt pairwise testing strategy and propose a notion of pairwise coverage for regular expressions. Let w_1, w_2 be two strings. We say that the ordered pair $\langle w_1, w_2\rangle$ is *covered* by string w if both w_1 and w_2 are substrings of w and the last symbol of w_1 appears before the first symbol of w_2. Especially, if w_1 and w_2 are adjacent to each other, we say that the ordered pair $\langle w_1, w_2\rangle$ is *covered adjacently* by string w. For example, $\langle a, cd\rangle$ is covered by $abcd$, and $\langle ab, c\rangle$ is covered adjacently by $abcd$.

Pairwise coverage. For the alternation operator $|$ and the atomic constructors $\epsilon, \emptyset$ and a single symbol a, pairwise coverage is the same as combination coverage. The main difference lies in operators concatenation and Kleene star. Let E be a regular expression. A subset $S \subseteq L(E)$ achieves pairwise coverage for E if it contains the finite set $C(E)$ satisfying conditions defined below by induction on the structure of E.

(1) If E is ϵ, $\emptyset$ or a for some $a \in \Sigma$, then $C(E) = L(E)$;

(2) If $E = E_1|E_2$, then $C(E) = C(E_1) \cup C(E_2)$;

(3) If $E = E_1 \cdot ... \cdot E_n$, then for any two sets $C(E_i)$ and $C(E_j)$ where $i, j \in [1, n]$, $i < j$ and any two strings $s \in C(E_i), s' \in C(E_j)$, there exists a string $w \in C(E)$ such that the ordered pair $\langle s, s'\rangle$ is *covered* by w, and furthermore if for each E_k $(i < k < j)$, $\epsilon \in C(E_k)$, then there exists a string $w \in C(E)$ such that the ordered pair $\langle s, s'\rangle$ is *covered adjacently* by w;

(4) If $E = E_1^*$, then $\{\epsilon\} \cup C(E_1) \subseteq C(E)$, and for any two strings $s, s' \in C(E_1)$ there exists a string $w \in C(E)$ such that the ordered pair $\langle s, s'\rangle$ is covered adjacently by w.

We give a detailed explanation of the definition for concatenation and Kleene star below.

Concatenation. Suppose that a regular expression is of the form $E_1 \cdot ... \cdot E_n$. Combination coverage requires taking all the possible combinations of characteristic substrings defined by those subexpressions E_k $(k \in [1, n])$, while pairwise coverage only requires covering all combinations of any two subexpressions E_i and E_j $(i \neq j$ and $i, j \in [1, n])$. Table 1 shows an example. The expression E is a concatenation of four subexpressions E_i $(i \in [1, 4])$. For each subexpression, there are three substrings. Testing all possible combinations of those substrings requires $3^4 = 81$ test strings. As shown by the lower part of the table, while using pairwise testing, only 9 test strings are needed to cover all combinations of any two subexpressions. Furthermore, taking the very special empty string ϵ into account, we require that if all the subexpressions between the two subexpressions E_i and E_j define languages containing empty string, then all the combinations of E_i and E_j must be adjacently covered. Consider $(a|\epsilon)(b|\epsilon)(c|\epsilon)$ for instance.

$C(E_1)$	$a_1,\ b_1,\ c_1$
$C(E_2)$	$a_2,\ b_2,\ c_2$
$C(E_3)$	$a_3,\ b_3,\ c_3$
$C(E_4)$	$a_4,\ b_4,\ c_4$
$C(E)$	$a_1a_2a_3a_4$, $a_1c_2c_3b_4$, $a_1b_2b_3c_4$, $b_1a_2c_3c_4$, $b_1b_2a_3b_4$, $b_1c_2b_3a_4$, $c_1a_2b_3b_4$, $c_1c_2a_3c_4$, $c_1b_2c_3a_4$

TABLE 1. Example pairwise coverage for regular expression $E = E_1E_2E_3E_4$, where $E_1 = a_1|b_1|c_1$, $E_2 = a_2|b_2|c_2$, $E_3 = a_3|b_3|c_3$ and $E_4 = a_4|b_4|c_4$, respectively.

Pairs	$\langle a,a\rangle$	$\langle a,b\rangle$	$\langle a,c\rangle$	$\langle b,a\rangle$	$\langle b,b\rangle$	$\langle b,c\rangle$	$\langle c,a\rangle$	$\langle c,b\rangle$	$\langle c,c\rangle$
aabbcc	✓	✓			✓	✓			✓
bacacb			✓	✓			✓	✓	

TABLE 2. Example pairwise coverage for regular expression $E = E_1^*$, where $E_1 = a|b|c$ and, hence, $C(E_1) = \{a, b, c\}$. There are 9 ordered pairs covered adjacently by the two strings shown in the table.

Since the second subexpression $(b|\epsilon)$ produces an empty string, the string set fulfilling pairwise coverage should at least contain strings ϵ, a, c and ac which adjacently cover the four combinations $\langle\epsilon, \epsilon\rangle$, $\langle a, \epsilon\rangle$, $\langle\epsilon, c\rangle$ and $\langle a, c\rangle$ of subexpressions $(a|\epsilon)$ and $(c|\epsilon)$.

Kleene star. Suppose that a regular expression is of the form E_1^*. Similarly to combination coverage, we choose three typical possibilities for $*$: zero, one and more-than-one repetitions. By original definition of $*$, the language defined by E_1^* is the set of all strings obtained by concatenating any finite number of strings from $L(E_1)$. That is, strings in $L(E_1)$ can be concatenated in any order. Following the pairwise testing idea, we consider any two strings s, s' and require that $s \cdot s'$ occurs as a substring in some string $w \in C(E_1^*)$, i.e., the ordered pair $\langle s, s'\rangle$ is covered adjacently by w. Take $E = E_1^*$ where $E_1 = a|b|c$ for example. We have $C(E_1) = \{a, b, c\}$, and there are 9 ordered pairs that can be covered adjacently by $aabbcc$ and $bacacb$ as shown in Table 2. Thus set $\{\epsilon, a, b, c, aabbcc, bacacb\}$ achieves pairwise coverage for E.

Remarks. (1) Note that the string set achieving pairwise coverage for a regular expression E is not necessarily unique even if we fix the repetition number k. Consider the expression $E = (a|b|c)^*$ for example. Both sets $\{\epsilon, a, b, c, aabbcc, bacacb\}$ and $\{\epsilon, a, b, c, aabbca, bacccb\}$ fulfill pairwise coverage for this expression.

(2) Clearly, combination coverage is stronger than pairwise coverage. That is, if a set satisfies combination

coverage then it also satisfies pairwise coverage. The converse is not necessarily true.

3.2. Coverage Criteria for Extended Expressions

We then extend the coverage criteria for standard regular expressions to deal with counting and interleaving.

Consider combination coverage first. For counting $[l,r]$, we adapt the boundary-value analysis technique from software testing and require that the number of times of repetitions must contains l, r and a number k between l and r, i.e., the representatives of boundary values in range $[l,r]$. For interleaving &, we simply take all the possible shuffling results of its operands.

Combination coverage. Continuing with combination coverage, a subset of $L(E)$ is said to achieve combination coverage criterion for E extended with counting and interleaving if it further considers two cases.

(1) If $E = E_1^{[l,r]}$ then $C(E) = C(E_1)^l \cup C(E_1)^r \cup C(E_1)^k$, where $l < k < r$.

(2) If $E = E_1 \& E_2$ then $C(E) = \{w_1 \& w_2 | w_1 \in C(E_1)$ and $w_2 \in C(E_2)\}$.

For example, both sets $\{aa, aaa, aaaaa\}$ and $\{aa, aaaa, aaaaa\}$ achieve combination coverage for $a^{[2,5]}$, while the set achieving combination coverage for $(a|b|c)^{[2,5]}$ contains $3^2+3^3+3^5$ strings for $k=3$ and $3^2+3^4+3^5$ strings for $k=4$, respectively. The set satisfying combination coverage for $a\&(b\&c)$ is exactly the language $L(a\&(b\&c)) = \{abc, bac, bca, cba, cab, acb\}$.

Pairwise coverage. For counting $[l,r]$, we also take the boundary-value l, r and a number k between l and r. For Kleene star, we follow the same idea as combination coverage that the ordered pair of any two representative strings of the counting operator's operand constitutes a substring in the final string set. For interleaving &, we view interleaving as concatenations without orders for simplicity. Suppose that $E = E_1 \& ... \& E_n$. Similar to the concatenations of pairwise coverage, we require that for any two operands E_i and E_j, this unordered feature must be reflected at least once. More specifically, for any two representative strings s, s' of E_i and E_j respectively, s appears at least once before s', and at least once after s'. Note that s and s' are not necessarily adjacent.

Continuing with pairwise coverage for standard expressions, a subset of $L(E)$ is said to achieve pairwise coverage for extended expression E if it contains the finite set $C(E)$ satisfying two more conditions.

(1) If $E = E_1^{[l,r]}$, then there exist strings $w, w', w'' \in C(E)$ such that $w \in C(E_1)^l, w' \in C(E_1)^r, w'' \in C(E_1)^k$ for some $l < k < r$, and for any ordered pair $\langle s, s' \rangle$ where $s, s' \in C(E_1)$ there exists a string $v \in C(E)$ such that $\langle s, s' \rangle$ is covered adjacently by v.

(2) If $E = E_1 \& ... \& E_n$, then for any two sets $C(E_i)$ and $C(E_j)$ with $i, j \in [1, n]$, $i \neq j$ and any two strings $s \in C(E_i), s' \in C(E_j)$, there exist two strings $w, w' \in C(E)$ such that ordered pairs $\langle s, s' \rangle$ and $\langle s', s \rangle$ are covered by w and w', respectively.

For example, the set $\{aa, cc, bbcb, abaca\}$ achieves pairwise coverage for $(a|b|c)^{[2,5]}$. Here aa and cc represent repetitions of 2 times, $abaca$ of 5 times and $bbcb$ of 4 times, a valid value k between the boundary-values 2 and 5. The ordered pairs are the same as those shown in Table 2 and are all covered adjacently by these strings. For expression $a\&b\&c$, a set achieving pairwise coverage could be $\{abc, cba\}$ where abc covers ordered pair $\langle a, b \rangle$, $\langle a, c \rangle$, $\langle b, c \rangle$ and cba covers ordered pair $\langle c, b \rangle$, $\langle c, a \rangle$, $\langle b, a \rangle$. One can verify that for expression $(a|b)\&(c|d)\&(e|f)$, a set achieving pairwise coverage could be $\{ace, bde, adf, bcf, eca, edb, fda, fcb\}$.

4. ANALYSIS OF COVERAGE CRITERIA

It is well known that regular expressions, regular grammars and finite automata have an equivalent expressive power [29]. There have been coverage criteria proposed for context-free grammars which subsume regular grammars. Finite automata can be seen as control flow graphs of programs and for them several coverage criteria exist. In this section, we compare the criteria for regular expressions with those for regular grammars and finite automata, and analyze their subsumption relationships.

4.1. Coverage Criteria for Grammars

In the seminal paper [16] for generating strings from context-free grammars, Purdom proposed *rule coverage* for context-free grammars. In a follow-up work [17], Lämmel proposed a much stronger criterion called *context-dependent rule coverage*. We introduce these two criteria which are formally defined below.

Rule coverage. Let $G = \langle N, T, P, S \rangle$ be a context-free grammar. A string $w \in L(G)$ is said to cover a rule $p = X \to \alpha \in P$ if there is a derivation $S \overset{*}{\Rightarrow} \beta X \gamma \overset{p}{\Rightarrow} \beta \alpha \gamma \overset{*}{\Rightarrow} w$, where $X \in N$ and $\alpha, \beta, \gamma \in (N \cup T)^*$. A string set $S \subseteq L(G)$ is said to achieve rule coverage for G, if for each $p \in P$ there is a $w \in S$ that covers p.

Take the grammars in Figure 1 for example. String set $\{ac, b\}$ satisfies rule coverage for G_1, and string set $\{acc, b\}$ satisfies rule coverage for G_2.

Rule coverage explores a grammar structure by considering rules independently. To achieve a better accuracy, Lämmel proposed a generalization such that the context in which a rule is covered is also taken into consideration, known as *context-dependent rule coverage* [17].

Context-dependent rule coverage. Let $G = \langle N, T, P, S \rangle$ be a context-free grammar. If $Y \to \beta X \gamma \in P$, where $X, Y \in N, \beta, \gamma \in (N \cup T)^*$, then $Y \to \beta \boxed{X} \gamma$ is called a direct occurrence of X in G. A string $w \in L(G)$ is said to cover a rule $p = X \to \alpha \in P$

for the occurrence $Y \to \beta\boxed{X}\gamma$ if there is a derivation $S \overset{*}{\Rightarrow} \delta Y \eta \overset{q}{\Rightarrow} \delta\beta X \gamma\eta \overset{p}{\Rightarrow} \delta\beta\alpha\gamma\eta \overset{*}{\Rightarrow} w$ with $q = Y \to \beta X \gamma \in P$. A string set $S \subseteq L(G)$ is said to achieve context-dependent rule coverage for G, if all $p \in P$ for all occurrences are covered.

The above definition is for general context-free grammars. In this study, we concentrate on regular grammars in which, by definition, the right-hand side of each rule contains at most only one nonterminal. This means that for each rule p, there is at most one direct occurrence of a nonterminal at the right-hand side of p. Therefore, a derivation can be represented as a sequence of rules. Consider the regular grammar G_1 in Figure 1. Derivation $S \Rightarrow aA \Rightarrow acA \Rightarrow ac$ can be represented as sequence $p_1p_3p_4$, denoting the unique rules used in derivation steps.

We can now interpret context-dependent rule coverage for regular grammars as follows. A string set $S \subseteq L(G)$ is said to achieve context-dependent rule coverage for regular grammar G, if for each possible rule-pair p_ip_j of grammar G, there exists a string $w \in S$ such that p_ip_j is a subsequence appearing in the derivation sequence of w. Here *possible* rule-pair means rules used in a possible derivation of 2 steps. For G_1 in Figure 1, the possible rule-pairs are p_1p_3, p_1p_4, p_2p_3, p_2p_4, p_3p_3 and p_3p_4. The string ac whose derivation sequence is $p_1p_3p_4$ only covers p_1p_3 and p_3p_4. So to achieve context-dependent rule coverage for G_1, we need to add more strings such as a, b, bcc. For G_2 in Figure 1, the possible rule-pairs are p_1p_3, p_1p_4, p_2p_3, p_2p_4, p_3p_5, p_3p_6, p_5p_5 and p_5p_6. A possible string set satisfying context-dependent rule coverage for G_2 is $\{a, b, accc, bc\}$.

4.2. Coverage Criteria for Automata

We now consider coverage for finite automata. A finite automaton can be viewed as a directed graph where states and transitions of the automaton are nodes and edges of the graph. An accepting run of the automaton is actually a path from the initial state to one of the final states. Automata can be viewed as the control flow graphs for programs and for them several coverage criteria exist. Here we introduce two commonly used coverage criteria: edge coverage and path coverage.

Edge coverage. Let A be a finite automaton and S be a subset of $L(A)$. If for each edge e of A there exists a string $w \in S$ such that the accepting run of w contains e, then we say that S achieves edge coverage for A.

Take the automata in Figure 2 for example. The string set $\{ac, b\}$ satisfies edge coverage for A_1 and string set $\{acc, b\}$ satisfies edge coverage for A_2.

Path coverage. Let A be a finite automaton and S be a subset of $L(A)$. If for each path p of A that starts at the initial state and ends at a final state, there exists a string $w \in S$ such that the accepting run of w corresponds exactly to path p, then we say that S achieves path coverage for A.

Since loops introduce an unbounded number of paths, we consider only a limited number of looping possibilities. Similarly to the boundary-interior path coverage in software testing field, we consider three possibilities for loops: zero, once, and multiple repetitions [32]. Consider automata A_1 and A_2 in Figure 2, string sets $\{a, b, ac, bc, acc, bcc\}$ and $\{a, b, ac, bc, acc, bcc, accc, bccc\}$ fulfill path coverage for A_1 and A_2, respectively.

4.3. Subsumption and Transformation

When talking about different test criteria, we usually need to compare them and to see whether one criterion is stronger than another one. One of the commonly used comparison method is to analyze the subsumption relationship defined as follows [33].

Subsumption. Let C_1 and C_2 be two coverage criteria. C_1 is said to subsume C_2 if every test set S that satisfies C_1 also satisfies C_2.

We next compare the criteria for regular expressions with those for regular grammars and finite automata, and analyze the subsumption relationships among them. Our criteria can handle not only standard but also extended expressions. Note that the translation from extended expressions to standard ones and thus to regular grammars and finite automata have a drastic impact on succinctness. It may not be suitable to compare criteria defined on compact representation, i.e., extended expressions, with that defined on the translated representation, i.e., grammars or automata, which may have a size of high complexity. Thus in the analysis we only consider standard expressions.

For criteria defined with respect to the same formalism, the subsumption relationship is clear. Specifically, context-dependent rule coverage subsumes rule coverage for regular grammars, path coverage subsumes edge coverage for finite automata and combination coverage subsumes pairwise coverage for regular expressions.

For criteria between different formalisms, the analysis is more involved. It is known that these different formalisms, i.e., regular grammars, finite automata and regular expressions are equivalently transferable to each other. However, the order and translations algorithms are very important. Therefor, we first fix the setting for our analysis.

We start with a particular regular grammar G, then G is transformed to an equivalent finite automaton A_G and finally A_G is transformed to an equivalent regular expression E_{A_G}. We shall study that, given any string set S satisfying coverage C for formalism Γ, whether it also satisfies coverage C' for another formalism Γ' where Γ and Γ' range over grammar G, automaton A_G and expression E_{A_G}, and C ranges over the criteria of Γ, C' ranges over the criteria of Γ'.

Two transforming algorithms are needed: one from

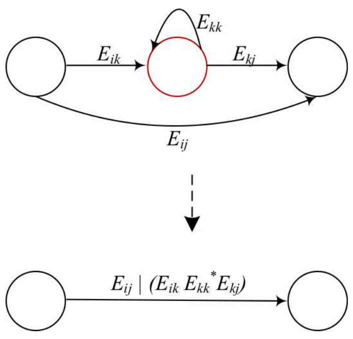

FIGURE 3. Illustration of the state elimination method for converting finite automata to regular expressions.

regular grammars to finite automata and the other from finite automata to regular expressions. We briefly describe two classical transforming algorithms, and details can be found in any textbook on formal languages, e.g., [29].

Regular Grammars into Finite Automata. Given a regular grammar G, an equivalent finite automaton A_G can be obtained as follows. The number of states in the automaton is equal to the number of nonterminals plus one. Each state in the automaton represents each nonterminal in the regular grammar. The additional state is the final state of the automaton. The state corresponding to the start symbol of the grammar is the initial state. The transitions to automaton are obtained as follows.

- For each rule in the form of $X \to aY$ in G, add a transition from state X to the state Y labeled with symbol a;
- For each rule in the form of $X \to a$ in G, add a transition from state X to the final state labeled with symbol a;
- For each rule in the form of $X \to \epsilon$ in G, add a transition from state X to the final state labeled with empty string ϵ.

As an example, the above algorithm converts regular grammars G_1 and G_2 in Figure 1 to finite automata A_1 and A_2 in Figure 2, respectively.

Finite Automata into Regular Expressions. We use the state elimination method to transform a finite automaton into an equivalent regular expression. This method extends finite automata with generalized transitions that are allowed to be labeled by regular expressions instead of just symbols from Σ. We briefly describe the basic idea of this method (more details can be found in [34]). First, replace multiple transitions between the same two states by one transition using the alternation operator. Then, repeatedly remove states and change transitions accordingly until only the start state and final state remain. For each removed state, a regular expression is generated accordingly. The resulting regular expression is just the label on the single transition from the start state to the final state. If there are n accepting states, union of different regular expressions are taken.

The state elimination strategy is briefly illustrated in Figure 3. Suppose we want to eliminate state q_k that is neither starting nor final, and q_i and q_j are two of the remaining states ($i = j$ is possible) that have q_k as their middle node. To safely delete q_k, we have to replace the paths between q_i and q_j by a single edge.

- Suppose that the transitions from q_i to q_k, q_k to q_j are labeled by expressions E_{ik} and E_{kj}, and the loop on q_k is labeled by E_{kk}. If there is no loop on q_k, we treat E_{kk} as ϵ. Then we replace the path via q_k with one edge labeled by $E_{ik}E^*_{kk}E_{kj}$.
- If there is already an edge from q_i to q_j labeled with E_{ij}, then we add E_{ij} by using the alternation operator and obtain the expression $E_{ij}|(E_{ik}E^*_{kk}E_{kj})$.

For instance, using state elimination method, we can transform finite automaton A_1 in Figure 2 to regular expression $(a|b)c^*\epsilon$, and finite automaton A_2 to regular expression $(a|b)(\epsilon|cc^*\epsilon)$. Both expressions are equivalent and define the same language as expression $(a|b)c^*$.

4.4. Subsumption Analysis

We are now ready to analyze the relationships between different coverage criteria. Figure 4 summarizes our analysis. The subsumption relationship between pairwise coverage and combination coverage for regular expressions is quite obvious. So are the relationships between edge coverage and path coverage for finite automata, and between rule coverage and context-dependent rule coverage for regular grammars, respectively. We next explain the reasons behind the other relationships.

Criteria between Grammars and Automata. We first compare coverage for regular grammars with those for finite automata. Recall that a string set achieves rule coverage for a given regular grammar G if it covers all the rules of G, and a string set achieves context-dependent rule coverage for G if it covers all possible rule-pairs p_ip_j of grammar G. Note that during the transformation from regular grammar to finite automaton, a rule of the grammar is translated to an edge of the automaton. Correspondingly, we can then interpret rule coverage for grammars as edge coverage for automata, and context-dependent rule coverage for grammars as edge-pair coverage for automata. Here edge-pair means each reachable subpath of length 2. One can then readily infer that path coverage subsumes context-dependent rule coverage, and edge coverage and rule coverage subsume each other.

Criteria between Automata and Expressions. We then compare coverage for finite automata with those for regular expressions. Note that during the state elimination construction from automata to expressions, loops are represented by Kleene stars, and multiple edges between two nodes are represented by alternations. According to the requirements of pairwise coverage, the operand of Kleene star must repeat at least zero, once and more than once, while each operand of alternation must appear at least once. Thus for an automaton A and its constructed expression E_A, if string set S achieves pairwise coverage for E_A, it must cover all the edges of A. The converse however does not necessarily hold. Therefore we conclude that pairwise coverage for expressions subsumes edge coverage for automata. Along the same lines, one can verify that combination coverage for expressions and path coverage for automata subsume each other.

Criteria between Expressions and Grammars. We now compare coverage for regular expressions with those for regular grammars. As we have stated, combination coverage for expressions subsumes path coverage for automata, and the latter again subsumes context-dependent rule coverage for grammars. Consequently, we have that combination coverage is stronger than context-dependent rule coverage. Similarly, we have that pairwise coverage is stronger than rule coverage. In fact, pairwise coverage is even stronger than context-dependent rule coverage. To show this, we introduce the notion of *2-grams*. A 2-gram of a language L is a string of length 2 that occurs as a substring in some $w \in L$. For instance, the set of 2-grams for language $L((a|b)^*c)$ is $\{aa, ab, ba, bb, ac, bc\}$.

We next verify that if a string set achieves pairwise coverage for an expression, then the strings must contain all the 2-grams of the language defined by that expression.

Proposition 4.1. *Let E be a standard regular expression. If a string set $S \subseteq L(E)$ achieves pairwise coverage for E, then S contains all the 2-grams of language $L(E)$.*

Proof. It suffices to show that the set $C(E)$ included by S as defined by pairwise coverage contains all the 2-grams of $L(E)$. We show this by induction on the structure of E.

- If E is ϵ, $\emptyset$ or a for some $a \in \Sigma$, then $C(E) = L(E)$ clearly contains all the 2-grams of $L(E)$.
- If $E = E_1|E_2$ and suppose that $C(E_1)$ and $C(E_2)$ contain all the 2-grams of $L(E_1)$ and $L(E_2)$ respectively, then $C(E) = C(E_1) \cup C(E_2)$ contains all the 2-grams of $L(E)$.
- If $E = E_1 \cdot \ldots \cdot E_n$ and suppose that for each E_k ($k \in [1, n]$), $C(E_k)$ contains all the 2-grams of $L(E_k)$. We show that $C(E)$ contains all the 2-grams of $L(E)$. Let ab be a 2-gram of $L(E)$. Clearly $C(E)$ contains it if ab is a 2-gram of some $L(E_k)$. Otherwise, there must exist a substring w_a from some set $C(E_i)$ ending with symbol a and a substring w_b from some set $C(E_j)$ starting with symbol b. If subexpressions E_i and E_j are adjacent then according to the requirement of pairwise coverage, the ordered pair $\langle w_a, w_b\rangle$ must be adjacently covered by a string in $C(E)$. If subexpression E_i and E_j are not adjacent, then the existence of a 2-gram ab implies that all the subexpressions between E_i and E_j contain empty strings. Again according to the requirement of pairwise coverage, the ordered pair $\langle w_a, w_b\rangle$ must be adjacently covered by a string in $C(E)$. Thus we can assure that for any 2-gram ab of $L(E)$, ab is also a 2-gram of $C(E)$.
- If $E = E_1^*$ and suppose that $C(E_1)$ contains all the 2-grams of $L(E_1)$. Let ab be a 2-gram of $L(E)$. Clearly $C(E)$ contains it if ab is a 2-gram of $L(E_1)$. Otherwise, there must exist a substring $w_a \in C(E_1)$ ending with symbol a and a substring $w_b \in C(E_1)$ starting with symbol b. According to the definition of pairwise coverage, the ordered pair $\langle w_a, w_b\rangle$ must be adjacently covered by a string in $C(E)$ which ensures that the 2-gram ab of $L(E)$ is also a 2-gram of $C(E)$.

□

Clearly, if language L is described by an automaton, then each 2-gram of L corresponds to a subpath of length 2, i.e., an edge-pair of the automaton. Since context-dependent rule coverage can be interpreted as edge-pair coverage for automata, along with Proposition 4.1, we can then infer that if a string set achieves pairwise coverage for expression E it also achieves context-dependent rule coverage for the grammar from which E is translated. Thus pairwise coverage subsumes context-dependent rule coverage. The converse does not always hold, as shown by the following. Consider the grammar G_1 in Figure 1 and the expression $(a|b)c^*\epsilon$ translated from G_1 (G_1 is first translated to automaton A_1 in Figure 2 and then A_1 translated to expression $(a|b)c^*\epsilon$). One can verify that string set $\{a, b, acc, bcc\}$ satisfies the context-dependent rule coverage for G_1, but not the pairwise coverage for the expression.

The reason why context-dependent rule coverage does not necessarily subsume pairwise coverage lies mainly in the Kleene star operators, which correspond to recursions in grammars. Here recursions refer to rules whose left-hand sides appear exactly in the right-hand sides. Such rules introduce self loops in the converted automata. Context-dependent rule coverage, which is interpreted as edge-pair coverage on automata, requires that such self loops repeat at least zero and twice. While pairwise coverage requires that the operands of Kleene star must repeat at least zero, once and more than once. Hence, a string set satisfies context-dependent rule coverage does not necessarily satisfy pairwise coverage.

We emphasize again that the order and algorithms used for transforming between different formalisms are

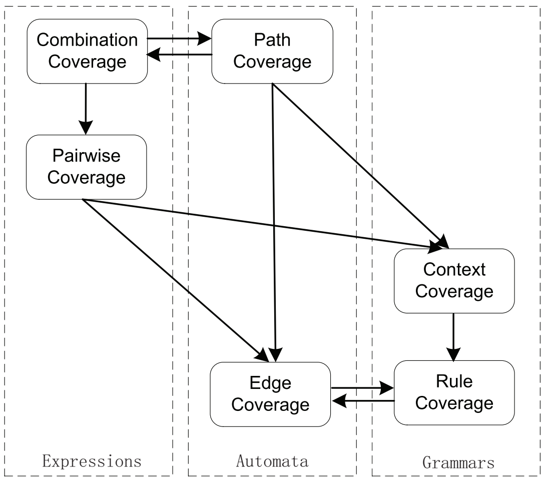

FIGURE 4. Subsumption relationships of coverage criteria. Here, *Context Coverage* is abbreviated for *Context-Dependent Rule Coverage*.

very important for the subsumption analysis. For instance, if we consider grammar G_1 in Figure 1 and automaton A_2 in Figure 2, which is not converted from the former using the algorithm discussed in our article, one can verify that there is a string set $\{ac, b\}$ that achieves rule coverage for the former while does not achieve edge coverage for the later. Another example, consider the expression $(a|b)c^*\epsilon$ and grammar G_2 in Figure 1. The string set $\{a, b, ac, bc, acc, bcc\}$ achieves combination coverage for the expression but does not satisfy context-dependent rule coverage for grammar G_2. If we change the expression to $(a|b)(\epsilon|cc^*\epsilon)$ which is converted from G_2 (G_2 is first transformed to A_2 in Figure 2 and then A_2 to the above expression), one can verify that the unique string set $\{a, b, ac, bc, acc, bcc, accc, bccc\}$ (if we fix the multiple repetitions as exactly twice) achieving combination coverage definitely satisfies context-dependent rule coverage for G_2.

Remarks. (1) Our analysis shows that pairwise coverage as proposed in this study for regular expressions is stronger than rule coverage, and the more accurate context-dependent rule coverage for grammars, and is also stronger than edge coverage for automata.

(2) Combination coverage is as rigorous as path coverage for automata. However, these coverage criteria are not practical due to the high complexity of sufficient test sets.

5. STRING GENERATION ALGORITHMS

In this section, we develop two algorithms for automatically generating string sets from regular expressions, one for pairwise coverage and the other for combination coverage, for both standard and extended regular expressions. The generations utilize the *abstract syntax trees* of regular expressions. Hence, we first introduce abstract syntax trees, and then present the algorithms.

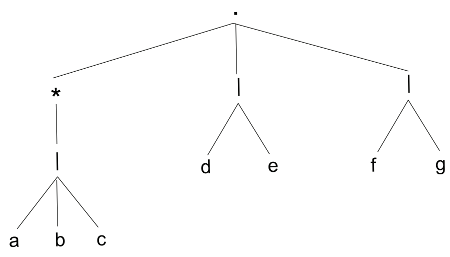

FIGURE 5. The abstract syntax tree for regular expression $(a|b|c)^*(d|e)(f|g)$.

An abstract syntax tree (or simply syntax tree) of a regular expression is a labeled and ordered tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the internal operators. That is, the leaves have nullary operators, i.e., symbols, empty string ϵ or empty set $\emptyset$. The structure of the syntax tree of an expression shows the precedence of operators occurred in the expression. Parentheses are implicit in the tree structure. Note that we do not restrict the syntax trees to be binary. This is without loss of generality since all the binary regular operators are associative. Thus for each node labeled with unary operator $*$ or $[l, r]$, there is only one child while for each node labeled with binary operator $|$, $\cdot$ or &, there are two or more children. As an example, Figure 5 illustrates the syntax tree of regular expression $(a|b|c)^*(d|e)(f|g)$.

Given an expression E, we first construct its syntax tree, and then perform a bottom-up traversal along this tree. For each node of the tree, we compute a set of substrings corresponding to the subexpression described by that node. Starting from the leaves, we go up and combine these substrings into longer substrings. As a leaf is simply a single symbol a or ϵ or $\emptyset$, thus the substring set associated with each leaf consists of the leaf itself or just simply $\emptyset$. The computation strategy of the substring set for each internal node depends on the labeled operator and the target criterion. Figure 6 shows the bottom-up process of generating strings along with the abstract syntax tree for an example regular expression, in which the final set generated based on combination coverage contains 52 strings, while the set based on pairwise coverage contains 14 strings only.

We describe in detail the generation of strings from syntax trees in Algorithm 1 and Algorithm 2, which are defined recursively. In the following, we assume that λ is an internal node of an abstract syntax tree, S_i is the set computed for its ith child if λ is labeled by binary operator *concatenation*, *alternation* or *interleaving*, S_C is the set computed for its only child if λ is labeled by unary operators *Kleene star* or *counting*. For analyzing the size complexity of the generated string set, we use

Algorithm 1 ComGen

```
Input: Abstract syntax tree $T$ of regular expression $E$;
Output: A set achieving combination coverage for $E$;
 1: Let $\lambda$ be the root of $T$
 2: if $\lambda$ is a leaf then
 3:     return $\{l\}$ or $\emptyset$ where $l$ is the label of $\lambda$
 4: else if $\lambda$ is labeled by $|$ then
 5:     $S \leftarrow \emptyset$
 6:     for each child $\lambda_i$ of $\lambda$ do
 7:         $S \leftarrow S \cup$ ComGen($\lambda_i$)
 8:     end for
 9: else if $\lambda$ is labeled by . then
10:     for each child $\lambda_i$ of $\lambda$ do
11:         $S_i \leftarrow$ ComGen($\lambda_i$)
12:     end for
13:     $Prod \leftarrow$ CartesianProduct($S_1, ..., S_n$)
14:     $S \leftarrow$ Product2String($Prod$)
15: else if $\lambda$ is labeled by $*$ then
16:     $S_C \leftarrow$ ComGen($\lambda.child$)
17:     $Prod \leftarrow$ CartesianProduct($S_C, S_C$)
18:     $S \leftarrow \{\epsilon\} \cup S_C \cup$ Product2String($Prod$)
19: else if $\lambda$ is labeled by $[l, r]$ then
20:     $S_C \leftarrow$ ComGen($\lambda.child$), $S \leftarrow \emptyset$
21:     for $k \leftarrow l, r$ and Random($l, r$) do
22:         $Prod \leftarrow$ CartesianProduct($S_C, ..., S_C$) with $k$
            sets $S_C$
23:         $S \leftarrow S \cup$ Product2String($Prod$)
24:     end for
25: else if $\lambda$ is labeled by & then
26:     $S \leftarrow \{\epsilon\}$
27:     for each child $\lambda_i$ of $\lambda$ do
28:         $S_i \leftarrow$ ComGen($\lambda_i$)
29:         $S \leftarrow$ Interleave($S, S_i$)
30:     end for
31: end if
32: return DeleteDuplicate($S$)
```

n to denote the number of λ's children if λ is not unary, and $|S|$ to denote the number of elements in a string set S, respectively.

5.1. Combination Coverage as Target Criterion

The string generation process when taking combination coverage as target is described in Algorithm 1. Consider an internal node λ of the syntax tree of a regular expression. We discuss the following cases.

(1) λ is labeled by *alternation* (lines 4-8). We just take the union of all those string sets computed for its children, which produces $\sum_{i=1}^{n} |S_i|$ strings.

(2) λ is labeled by *concatenation* (lines 9-14). We take all possible discrete combinations of substrings generated for its children. That is, we first compute the Cartesian product of those S_i sets, then convert the product to a set of strings by concatenating for each tuple of the product all the members in the order they appear in that tuple (line 14). This produces exactly $\prod_{i=1}^{n} |S_i|$ strings that increase exponentially with n. Consider the left syntax tree in Figure 6. There are 52 substrings generated for the root node which is labeled with $\cdot$.

(3) λ is labeled by *Kleene star* (lines 15-18). We merge the three sets $\{\epsilon\}$, S_C and the set containing all the possible combinations of S_C with itself, denoting repetitions of zero, once and more-than-once respectively. To control the size of generated sets, here we fix more-than-once as exactly twice, and $1 + |S_C| + |S_C|^2$ strings are generated. Consider the left syntax tree in Figure 6 for example. There are 13 substrings generated for the node labeled with $*$.

(4) λ is labeled by $[l, r]$ (lines 19-24). We combine the substrings of S_C with themselves for l times, r times and then k times if there exists an integer $l < k < r$. We take all possible discrete combinations. This produces exactly $|S_C|^l + |S_C|^r + |S_C|^k$ strings.

(5) λ is labeled by & (lines 25-30). We get the interleavings of all the string sets computed for λ's children. This step essentially reduces to computing the interleaving of two strings. Recall that for any two strings w_1 and w_2, if one is ϵ then the interleaving result is the other string; otherwise, let $w_1 = aw_1', w_2 = bw_2'$ where a and b are single symbols, then $w_1 \& w_2 = a\{w_1' \& bw_2'\} \cup b\{aw_1' \& w_2'\}$. Following this definition, we use a recursive function to implement it. We omit the details due to space limitations. We next give a brief analysis of the number of strings generated in this case. First, note that the interleaving of two strings w_1 and w_2 with lengths m and n produces $\binom{m+n}{m}$ strings. Second, note that there are $|S_1| \times |S_2|$ interleaving of strings when λ has two children. Suppose that λ has n children and the average length of strings in each child set S_i is l. Then interleaving S_1 and S_2 results in $|S_1| \times |S_2| \times \binom{2l}{l}$ strings, and then interleaving those strings with S_3 results in $|S_1| \times |S_2| \times \binom{2l}{l} \times |S_3| \times \binom{3l}{l}$ strings. In this way, we can estimate that the total number of strings computed for the node λ is $\prod_{i=1}^{n} |S_i| \times \prod_{i=1}^{n} \binom{i \times l}{l}$. Especially, when $l = 1$, the total number is $\prod_{i=1}^{n} |S_i| \times n!$ that increases factorially with n. For instance, for $E = a\&b\&c$, the algorithm generates $1 \times 1 \times 1 \times 3! = 6$ strings, and for $E = (a|b)\&(c|d)\&(e|f)$, the algorithm generates $2 \times 2 \times 2 \times 3! = 48$ strings.

5.2. Pairwise Coverage as Target Criterion

The generation of strings taking pairwise coverage as the target criterion as described in Algorithm 2 is more involved than combination coverage based generation. We explain in detail below. Again, suppose that λ is an internal node of the syntax tree of a regular expression.

(1) λ is labeled by *alternation* (lines 4-8). We just need to merge the sets computed for its children, which generates $\sum_{i=1}^{n} |S_i|$ strings.

(2) λ is labeled by *concatenation* (lines 9-16). We need to find a pairwise test set from the sets generated for λ's children. The problem of generating a minimum

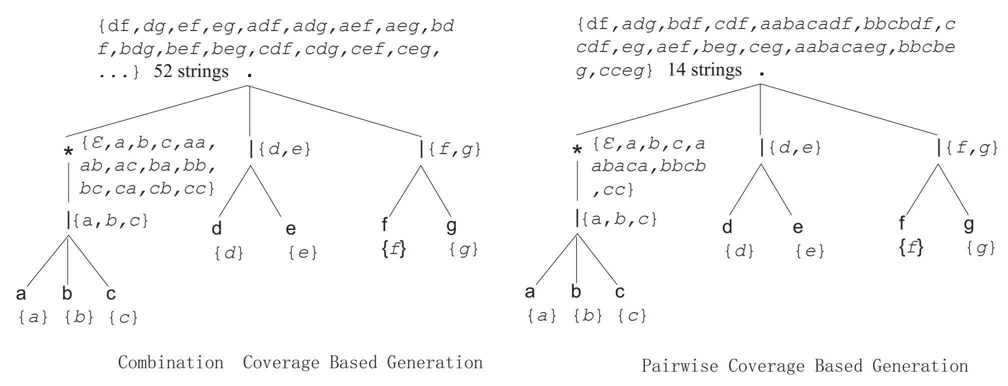
{df,dg,ef,eg,adf,adg,aef,aeg,bdf,bdg,bef,beg,cdf,cdg,cef,ceg, ...} 52 strings
{ℰ,a,b,c,aa, ab,ac,ba,bb, bc,ca,cb,cc}
{a,b,c}
a b c
{a} {b} {c}
{d,e}
d e
{d} {e}
{f,g}
f g
{f} {g}
Combination Coverage Based Generation
{df,adg,bdf,cdf,aabacadf,bbcbdf,ccdf,eg,aef,beg,ceg,aabacaeg,bbcbeg,cceg} 14 strings
{ℰ,a,b,c,aabaca,bbcb,cc}
{a,b,c}
a b c
{a} {b} {c}
{d,e}
d e
{d} {e}
{f,g}
f g
{f} {g}
Pairwise Coverage Based Generation

FIGURE 6. The bottom-up generation of strings achieving combination coverage and pairwise coverage, respectively.

pairwise test set is known to be NP-complete [35]. Many heuristic strategies to generate near-minimum pairwise test set have been proposed and tools have been developed [31]. Our algorithm utilizes the Pairwise Independent Combinatorial Tool (PICT for short, available at https://github.com/microsoft/pict) to generate strings achieving pairwise coverage when the regular operator is concatenation. More specifically, for λ with n children, the algorithm calls PICT by providing n substring sets, and gets a set of n-tuples. Those n-tuples are then translated to a set of strings by concatenating for each tuple all the members in the order they appear in that tuple (see the root node in the right syntax tree of Figure 6 for example). Finally, we need to compute all the ordered pairs that are required to be adjacently covered and find which of them already adjacently covered by the strings generated by PICT. For those uncovered ones, we construct new strings to adjacently cover them. The construction follows a greedy strategy similar to the generation for Kleene star (lines 20-27), which will be explained immediately.

The number of strings generated in this case depends on the output of the PICT tool and the output of greedy construction to adjacently cover some necessary ordered pairs. To our knowledge, there is no accurate analysis on the former problem. A trivial lower bound for the minimal number of strings output by PICT is $max\{|S_i| \times |S_j|\}$, where $i, j \in [1, n]$ and $i \neq j$. Furthermore, it has been shown that the minimal number of test cases for pairwise testing grows only logarithmically with n [36]. For the latter problem, we consider the worst case in which each S_i contains the empty string ϵ. In such a case, there are $\sum_{k=3}^{n}(|S_1| \times |S_k|)$ ordered pairs beginning with strings from S_1 that need to be adjacently covered, and $\sum_{k=4}^{n}(|S_2| \times |S_k|)$ ordered pairs beginning with strings from S_2 that need to be adjacently covered, and so on. Assume that the average number of strings in each S_i is $|\bar{S}|$. Then in total there are $\binom{n-1}{2} \times |\bar{S}|^2$ ordered pairs. A string can at least adjacently cover one such ordered pair and at most about $(n-1)/2$ such ordered pairs (one example is $s_1s_3s_5...s_n$, where each $s_i \in S_i$ for $i = 1, 3, 5, ..., n$.) Therefore, we obtain a lower bound for the number of generated strings to adjacently cover all those pairs is about $n \times |\bar{S}|^2$ and an upper bound $n^2 \times |\bar{S}|^2$.

(3) λ is labeled by *Kleene star* (lines 17-27). The key step is to generate a set of strings that adjacently cover all the ordered pairs $\langle s, s'\rangle$ where s, s' are any two strings of S_C, the child set computed for λ's only child. In our algorithm, we adopt a kind of greedy strategy. First compute all the ordered pairs of S_C. Then pick an uncovered pair $\langle s, s'\rangle$, start from string s and extend it as long as possible whenever it can cover more uncovered pairs. Repeat the above process until all pairs are covered. For instance, suppose that the child set S_C contains 3 strings a, b and c. The algorithm produces a set $\{aabaca, bbcb, cc\}$, which covers the $|S_C| \times |S_C| = 9$ ordered pairs as below, where ordered pairs above symbol $\Longrightarrow$ are to be covered during the process of string extension.

① $a \overset{\langle a,a\rangle}{\Longrightarrow} \underline{aa} \overset{\langle a,b\rangle}{\Longrightarrow} a\underline{ab} \overset{\langle b,a\rangle}{\Longrightarrow} aa\underline{ba} \overset{\langle a,c\rangle}{\Longrightarrow} aab\underline{ac} \overset{\langle c,a\rangle}{\Longrightarrow} aaba\underline{ca}$

② $b \overset{\langle b,b\rangle}{\Longrightarrow} \underline{bb} \overset{\langle b,c\rangle}{\Longrightarrow} b\underline{bc} \overset{\langle c,b\rangle}{\Longrightarrow} bb\underline{cb}$

③ $c \overset{\langle c,c\rangle}{\Longrightarrow} \underline{cc}$

At last, we add the empty string ϵ and all strings in S_C itself, denoting repetitions of zero and once, respectively. For the above example, we finally obtain a string set $\{\epsilon, a, b, c, aabaca, bbcb, cc\}$ for the node λ labeled by Kleene star (see the $*$ node in the right syntax tree of Figure 6).

We next estimate the number of generated strings in this case. In the greedy construction, we try to cover as many uncovered pairs as possible. Suppose that s_1 is an element of S_C. We can generate only one string to cover all the pairs beginning with s_1 and all the pairs ending with s_1, i.e., pairs $\langle s_1, s_k\rangle$ and $\langle s_k, s_1\rangle$ for all $s_k \in S_C$. Since there are $|S_C|$ elements in S_C, our greedy construction produces at most $|S_C|$ strings to cover all the $|S_C| \times |S_C|$ pairs. Thus, in total $2|S_C| + 1$ strings are generated in this case. Note that the analysis of the number of strings produced by greedy construction is a bit different from the case for concatenation because in the case of Kleene star we can extend a string as long as possible, while the extension

Algorithm 2 PairGen

```
Input: Abstract syntax tree $T$ of regular expression $E$;
Output: A set achieving pairwise coverage for $E$;
 1: Let $\lambda$ be the root of $T$
 2: if $\lambda$ is a leaf then
 3:     return {$l$} or $\emptyset$ where $l$ is the label of $\lambda$
 4: else if $\lambda$ is labeled by | then
 5:     $S \leftarrow \emptyset$
 6:     for each child $\lambda_i$ of $\lambda$ do
 7:         $S \leftarrow S \cup$ PairGen($\lambda_i$)
 8:     end for
 9: else if $\lambda$ is labeled by . then
10:     for each child $\lambda_i$ of $\lambda$ do
11:         $S_i \leftarrow$ PairGen($\lambda_i$)
12:     end for
13:     $Prod \leftarrow$ PICT($S_1, ..., S_n$)
14:     $S \leftarrow$ Product2String($Prod$)
15:     Compute pairs that need to be adjacently covered but
        not yet covered by $S$
16:     Similar codes to lines 20-27
17: else if $\lambda$ is labeled by $*$ then
18:     $S_C \leftarrow$ PairGen($\lambda.child$), $S \leftarrow \{\epsilon\} \cup S_C$
19:     Mark all ordered pairs of $S_C$ uncovered
20:     for each uncovered pair $\langle s, s' \rangle$ do
21:         $\bar{s} \leftarrow s; w \leftarrow \bar{s}$
22:         while there are uncovered pair $\langle \bar{s}, \bar{s}' \rangle$ do
23:             $w \leftarrow w \cdot \bar{s}'$ and mark $\langle \bar{s}, \bar{s}' \rangle$ covered
24:             $\bar{s} \leftarrow \bar{s}'$
25:         end while
26:         $S \leftarrow S \cup \{w\}$
27:     end for
28: else if $\lambda$ is labeled by $[l, r]$ then
29:     $S_C \leftarrow$ PairGen($\lambda.child$), $S \leftarrow \emptyset$
30:     Mark all ordered pairs of $S_C$ uncovered
31:     $T_r, T_l, T_k \leftarrow False$
32:     for each uncovered pair $\langle s, s' \rangle$ do
33:         $\bar{s} \leftarrow s; w \leftarrow \bar{s}; T \leftarrow 1$
34:         while there are uncovered pair $\langle \bar{s}, \bar{s}' \rangle$ do
35:             $w \leftarrow w \cdot \bar{s}'$ and mark $\langle \bar{s}, \bar{s}' \rangle$ covered
36:             $\bar{s} \leftarrow \bar{s}'; T++;$
37:             if $T \equiv r$ then
38:                 $T_r \leftarrow True$; break
39:             end if
40:         end while
41:         if $T \equiv l$ or $l < T < r$ then
42:             $T_l \leftarrow True$ or $T_k \leftarrow True$ respectively
43:         end if
44:         $S \leftarrow S \cup \{w\}$
45:     end for
46:     if $T_r \equiv False$ then
47:         pick a string $s \in S_C$, add $s^r$ to $S$
48:     end if
49:     Similarly if $T_l \equiv False$ or $T_k \equiv False$
50: else if $\lambda$ is labeled by & then
51:     Same codes as lines 10-16
52:     $S \leftarrow S \cup$ Reverse($S$)
53: end if
54: return DeleteDuplicate($S$)
```

is restricted in the case of concatenation. For example, if $E = E_1 \cdot ... \cdot E_n$, then the extension can only be done at most $n-1$ times.

(4) λ is labeled by $[l, r]$ (lines 28-49). The generation is similar to that for Kleene star. We take the same greedy strategy to generate strings adjacently covering all the ordered pairs of S_C, and at the same time incorporate a length control mechanism to ensure that the repetition of strings in S_C must include l times, r times and k times for some integer k with $l < k < r$. For instance, suppose that the child set S_C contains 3 strings a, b, c, and λ is labeled with $[2, 5]$. The following process illustrates the generation of a set $\{aabac, bbca, cb, cc\}$ that adjacently covers the $|S_C| \times |S_C| = 9$ ordered pairs, and satisfies the repetition requirements.

① $r = 5 : a \overset{\langle a,a \rangle}{\Longrightarrow} \underline{aa} \overset{\langle a,b \rangle}{\Longrightarrow} a\underline{ab} \overset{\langle b,a \rangle}{\Longrightarrow} aa\underline{ba} \overset{\langle a,c \rangle}{\Longrightarrow} aab\underline{ac}$

② $k = 4 : b \overset{\langle b,b \rangle}{\Longrightarrow} \underline{bb} \overset{\langle b,c \rangle}{\Longrightarrow} b\underline{bc} \overset{\langle c,a \rangle}{\Longrightarrow} bb\underline{ca}$

③ $l = 2 : c \overset{\langle c,b \rangle}{\Longrightarrow} \underline{cb}$

④ $l = 2 : c \overset{\langle c,c \rangle}{\Longrightarrow} \underline{cc}$

The number of generated strings in this case has the same order of magnitude as the case of Kleene star.

(5) λ is labeled by & (lines 50-53). In this case, pairwise coverage requires that for any two children sets S_i and S_j of node λ ($i \neq j$) and any two strings $s \in S_i, s' \in S_j$, both the ordered pairs $\langle s, s' \rangle$ and $\langle s', s \rangle$ are covered by the generated strings for λ. Clearly, if a string w covers the pair $\langle s, s' \rangle$, then the reverse of w will definitely cover the pair $\langle s', s \rangle$. Based on this observation, we reuse the same codes that handle *concatenation* to get a string set, then reverse all the strings of this set, and finally take the union of the original set with the reversed set. Thus, the number of generated strings in this case has the same order of magnitude as the case of concatenation.

It should be pointed out that duplicated strings may appear during combinations, which is mainly caused by the concatenation of empty strings. Consider the expression ϵ^* for example. Algorithm 1 produces three empty strings. Thus, we need to eliminate duplications before returning the final result, as indicated in the last lines of Algorithm 1 and Algorithm 2.

6. EXPERIMENTAL STUDY

In this section, we experimentally evaluate our algorithms. We have analyzed the subsumption relationships among coverage criteria for regular expressions and for the other two regular language definition formalisms, i.e., regular grammars and finite automata in Section 4. Different coverage based string generation algorithms may take different formalisms as inputs. For fairness, we only compare our two algorithms (PairGen and ComGen) and other two existing string generation tools that take regular expressions as input. We compare the number of generated strings and assess the quality of the strings.

All experiments were performed on a 2.6Ghz Intel i5 machine with 8GB of RAM.

6.1. Background

We take XML schema testing as the application background to evaluate our algorithms. XML (eXtensible Markup Language) is a popular method for data exchange on the Internet [37]. XML documents are usually accompanied with a schema that describes their structures. The structures can be formally specified in a *schema* written in a schema language such as the Document Type Definitions (DTDs) or the XML Schema Definitions (XSDs). Regular expressions are an important component of many commonly used XML schema languages. In fact, for DTDs, element type declarations are described precisely by regular expressions. An example DTD is shown below.

```
<!ELEMENT addrbook (person+)>
<!ELEMENT person (name,tel?,email*)>
<!ELEMENT name #PCDATA>
<!ELEMENT tel #PCDATA>
<!ELEMENT email #PCDATA>
```

As shown above, a DTD is essentially a mapping ρ from element names to regular expressions over element names. An XML document is valid with respect to ρ if for every occurrence of an element name e in the document, the string formed by its children belongs to the language of the corresponding regular expression $\rho(e)$. For instance, the above DTD requires each `addrbook` element to have one or more `person` children, and each `person` must have a `name` child, which must be followed by an optional `tel` and zero or more `email` elements. Therefore, testing whether a DTD is correctly defined essentially reduces to testing whether each element name is mapped to a correctly defined regular expression. XSDs can be seen as an extension of DTDs with a restricted form of specialization. Although XSDs are more expressive than DTDs, regular expressions remain one of the main building blocks. For instance, the complex element type definitions of XSDs are actually specified by regular expressions. In particular, XSDs support additional regular operators including counting and interleaving. In light of this, we choose regular expressions occurring in real-world DTDs and XSDs to evaluate our algorithms.

6.2. Real-World Expressions

Our experiments are conducted on the data set discussed in [38] (`http://lcs.ios.ac.cn/~zhangxl/project.html`). This data set contains a large corpus of about 7000 DTDs and XSDs gathered from different websites covering various fields such as education, agriculture, science, economics, engineering, sports and so on. We extract from these DTDs and XSDs a set of non-trivial regular expressions of different types and forms. Specifically, we classify regular expressions into different categories according to the contained regular operators; we further distinguish regular expressions of the same category, i.e., containing the same set of regular operators, by their star heights, nesting depths, parenthesis depths and sizes. These measures are defined as follows.

Star Height [39]. The star height of a regular expression E over alphabet Σ, denoted by $h(E)$, is a nonnegative integer defined recursively as follows. If E is ϵ, $\emptyset$ or a for some $a \in \Sigma$, then $h(E) = 0$; If $E = E_1|E_2$ or $E = E_1 \cdot E_2$, then $h(E)$ =max$\{h(E_1), h(E_2)\}$; If $E = E_1^*$, then $h(E) = h(E_1) + 1$. Star height is an illustration of iteration depth of regular expressions. Kleene plus and counting are treated similarly as Kleene star. Optional is treated similarly as alternation and interleaving is treated similarly as concatenation.

Nesting Depth [38]. The nesting depth of a regular expression E over alphabet Σ, denoted by $d(E)$, is a nonnegative integer defined recursively as follows. If E is ϵ, $\emptyset$ or a for some $a \in \Sigma$, then $d(E) = 0$; If $E = E_1 \lambda E_2$ where λ is a binary operator, then $d(E)$ =max$\{d(E_1), d(E_2)\}$; If $E = E_1^\lambda$ where λ is a unary operator, then $d(E) = d(E_1) + 1$. Intuitively, nesting depth is the maximum nesting depth of unary operators occurring in a regular expression. Compared with star height, it can reflect the syntactic structural complexity of a regular expression. For example, the star height of $((a?|b|c)^*|d)?$ and a^* are both 1 while the nesting depths are 3 and 1 respectively.

Parenthesis Depth and **Size** are other two measures. The former is the maximum nesting depth of parenthesis occurring in that expression. The latter is the number of operator occurrences in that expression with parentheses not counted.

In total, we extracted 58 non-trivial regular expressions which are classified into 25 categories as illustrated in the left part of Table 3. It should be pointed out that even two regular expressions have almost the same properties, their forms may differ quite much. We try to cover as more forms as possible. For example, r_{22} and r_{26} have the same operators, star heights, nesting depths and parenthesis depths but they are of different forms: $a_1^*(a_2^*|a_3^*|a_4^*)a_1^*$ and $(a_1|a_2|a_3)^*(a_4|...|a_{47})^*$. We choose both of them in our experiments.

6.3. Experimental Design

We apply mutation testing technique to assess the quality of the test data generated by our algorithms. Mutation testing is a fault-based technique to design new test data or evaluate the quality of existing test data [40]. It involves modifying the software artifact under test in small ways. Each mutated version is called a *mutant*, and test sets detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called *killing* the mutant. Test sets are measured by the percentage of mutants that they kill. The higher the percentage is, the stronger the fault detection ability is.

We compare our two algorithms PairGen and ComGen with other two string generators. One is Xeger [41], an open-source Java library that generates strings for regular expressions in a random way. Another is Egret [3], a recent tool that generates both positive and negative strings for regular expressions, i.e., strings accepted and strings rejected.

The experimental process is depicted in Figure 7. For an original expression E, a set S of strings are generated. The generated strings are undoubtedly accepted by E. We then obtain a set of mutants of E by applying mutation operators. For each mutant M, we check whether M can recognize all the strings in S. If there exists a string $s \in S$ not acceptable by the mutant, then this indicates that M behaves differently with the original expression E, and thus we mark that M is killed by S, otherwise not killed. Note that the string set S generated by Egret for an expression contains two subsets: positive strings S_p and negative strings S_n. For S_p we follow the same strategy as above to check the killing of mutants. For rejected strings we follow the opposite strategy. That is, if there exists a string $s \in S_n$ acceptable by mutant M, then we mark M killed. We mark that M is killed by S if M is killed by either S_p or S_n or both.

An important step in our experiments is to choose suitable mutation operators for regular expressions. Li and Miller [42] proposed a set of mutation operators for XSDs, which are based on the most common faults that may occur in developing XML schemas. We select four operators (the first four listed below) that are suitable for standard and extended regular expressions. Arcaini et al. [4] introduced several mutation operators for regular expressions that are used in string pattern matching. Such regular expressions have a quite different syntax and thus their mutation operators cannot applied directly in our experiments. We adopt ideas of the mutation operators in [4], such as *Negation Addition* and *Character Class Negation*, and define another four operators (the last four listed below) that are suitable for expressions used in our experiments.

- Replace one of the binary/unary regular operators by each of the other binary/unary operators;
- Change the order of elements one by one, if there is more than one element connected by the *concatenation* operator;
- Delete elements one by one if there is more than one element in the type declaration;
- Change the value of an element occurrence constraint (particularly applied to the *counting* operator);
- Change one of the element names to a new one or another one in the type declaration;
- Add a unary regular operator whenever possible in a regular expression;
- If a unary operator has more than one operand, then distribute it to all or one of its operands;

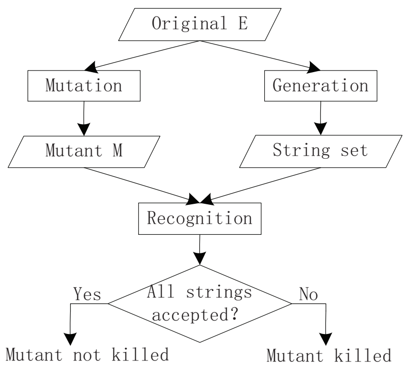

FIGURE 7. Process of mutation testing

- If a unary operator has more than one operand, then take one of its operands out.

For each expression, we obtain a set of mutants by applying these mutation operators. We then delete all equivalent mutants by using the dk.brics.automaton package [43]. The number of non-equivalent mutants for each expression is shown in column *Num.OfMutants* of Table 3. For each mutant, we conduct mutation testing as illustrated by Figure 7, and check whether the mutant is killed or not by a specific string set. The checking can be automatically done by using a regular expression pattern matching tool. In our experiments, we use the java.util.regex package for pattern matching with standard regular expressions and expressions extended with counting. This package does not support interleaving, so for expressions extended with interleaving, we first translate them into standard ones, and then utilize java.util.regex package for validation.

6.4. Experimental Results

6.4.1. Generated Strings

For each expression, we record the numbers of strings produced by our algorithms and Egret, as illustrated in the middle part of Table 3. For the random tool Xeger, the number of strings is controlled to be the same with algorithm PairGen for comparison purpose and is omitted. For Egret, we record the number of positive strings (column *Egret*$^+$) and the total number of both positive ones and negative ones (column *Egret*). Since Xeger does not support the interleaving operator, so the numbers for the last two expressions are omitted and similarly for Egret.

(1) PairGen vs. ComGen. The number of strings generated by ComGen is much larger than the number of those generated by PairGen especially for expressions with large sizes. The average number of strings generated by ComGen for those 58 expressions is about 200,000, while the average number is reduced to only about 70 by PairGen. This also confirms our theoretical analysis in Section 5 that the number of strings generated by ComGen increases exponentially with the size of expression E and even factorially if E

contains interleaving, while by PairGen the increasing rate is at most quadratic.

(2) PairGen itself. PairGen produces a relatively small number of strings for most of the expressions. About 75% of the expressions generate no more than 50 strings, 88% generate fewer than 100 strings. Five expressions generate more than 200 strings and only one generate more than 1000 strings. Roughly speaking, expressions with large sizes and complex structures generate more strings. For expressions with almost the same sizes and structural complexities, those that contain $|$ mixed with $*$ or $+$ generate more strings than those that do not contain $|$.

(3) PairGen vs. Egret. For simple expressions, the numbers of strings generated by PairGen and Egret are comparable. For complex expressions especially those containing $|$ mixed with $*$ or $+$, PairGen produces more positive strings than Egret. This is because that for such kinds of expressions, PairGen requires that all the adjacency cases between the operands of $*$ or $+$ are covered while Egret only pick one case. Take $(c_1 \;|...|\; c_{10})^*$ for instance, Egret produces set $\{\epsilon,\ c_1, \ldots, c_{10}, c_1c_1\}$ containing only one string c_1c_1 which only covers the adjacent pair $\langle c_1, c_1\rangle$ while PairGen produces tens of strings covering all the adjacent pairs $\langle c_i, c_j\rangle$ $(i, j \in [1, 10])$. We also observe that a majority of the strings generated by Egret are positive ones: about 40% of the expressions generate zero negative strings; 28% generate only one negative string; 98% generate fewer than six negative strings; the most negative string generated is thirteen. In fact, Egret's generation strategy of negative strings is simple. For example, $(a|b)?$ generates one negative string aa, $(a|b)^+$ generates one negative string ϵ, and $(a|b)^*$ generates no negative strings.

6.4.2. Quality of Generated Strings

We perform mutation testing to assess the quality of the generated strings. We compute the mutation score of a string set S for an expression E using the following formula

$$MutationScore(E, S) = \frac{\#KilledMTs(E, S)}{\#TotalNonEqMTs(E)} \times 100\%$$

where $\#KilledMTs(E, S)$ denotes the number of mutants of expression E killed by set S and $\#TotalNonEqMTs(E)$ denotes the total number of non-equivalent mutants of expression E, respectively. We compare strings generated by PairGen, ComGen, random algorithm Xeger and Egret. Since Egret generates also negative strings, we consider two cases of Egret: only positive strings (denoted by $Egret^+$) and total strings including both positive and negative. The number of strings generated by random algorithm is controlled to be the same with PairGen. The experiments of mutation testing for the random algorithm were repeated over 50 times and the average is reported here. The comparison results are shown in the last five columns of Table 3. Note that since Xeger and Egret do not support the interleaving operator, so the comparison for the last two expressions are omitted. We next report our findings.

(1) PairGen vs. ComGen. The mutation scores of strings generated by those two algorithms are the same for all the expressions. This implies that although pairwise coverage is weaker than combination coverage, its faults revealing ability does not decrease greatly. Hence, considering both the fault detection ability and the string set size, we conclude that pairwise coverage is much more suitable in practice. We analyze why PairGen even ComGen cannot always earn a 100% mutation score. Basically we can classify a non-equivalent mutant M of an expression E into three types: generalized $(L(M) \supset L(E))$, specialized $(L(M) \subset L(E))$ and arbitrary edit $(L(M) \not\supset L(E)$ and $L(M) \not\subset L(E))$. Clearly, generalized mutants can never be killed by only positive strings of the original expression. Indeed, we find out that in the experiments the mutants that PairGen and ComGen fail to kill are exactly of this type. Mutants of other two types are all successfully killed.

(2) PairGen vs. Random. As shown by the data, PairGen outperforms random generation in almost all the cases. On average, the mutation score of strings generated by the former is about 12% higher than the later. This shows that strings generated based on pairwise coverage in general have higher fault detection ability than randomly generated ones. For $r_{26}, r_{38}, r_{47}, r_{52}, r_{53}, r_{54}$, the scores are equal or very close. We analyze that the reason lies mainly in the number of the generated strings. In our experiments, when comparing PairGen and random generation, we use Xeger randomly generating the same number of strings with PairGen. As can be seen from Table 3, for these expressions, PairGen produces relatively large numbers of strings compared with the other expressions. Thus the numbers of randomly generated strings are also relatively large which helps to earn higher mutation scores.

(3) PairGen vs. Egret$^+$. For expressions containing nearly no $|$ operator, i.e., categories $C_1 - C_{11}$ and C_{24}, PairGen and Egret$^+$ kill the same number of mutants. However, for the other categories especially those that contain various kinds of regular operators and have complex expression structures (C_{23} for example), PairGen works better than Egret$^+$ in most cases. Among those categories, there are also a few expressions for which PairGen and Egret$^+$ earn a same mutation score. However, we find out that such expressions are either of small sizes $(r_{22}, r_{27}, r_{45}, r_{46}, r_{48})$ or of simple forms $(r_{33} : a_1^*|...|a_{11}^*|a_{12}?|a_{13}?)$. This indicates that PairGen can help to reveal some of the faults that may pass undetected by using Egret$^+$ especially when the expression is complex. In fact, the

RegularExpression							Num.OfStrings				Num.Of	MutationScore(%)				
Category	RE	OPs	Sheight	Ndepth	Pdepth	Size	Pair	Com	Egret$^+$	Egret	Mutants	Pair	Com	Rand	Egret$^+$	Egret
C_1	r_1	$\{\cdot\}$	0	0	0	5	1	1	1	1	49	**57**	**57**	**57**	**57**	**57**
	r_2	$\{\cdot *\}$	2	2	0	6	19	757	9	9	17	**100**	**100**	88	**100**	**100**
C_2	r_3	$\{\cdot *\}$	1	1	0	9	10	27	8	8	54	**78**	**78**	70	**78**	**78**
C_3	r_4	$\{\cdot ?\}$	0	1	0	10	4	8	5	8	62	71	71	60	71	**76**
C_4	r_5	$\{\cdot +\}$	1	1	0	7	2	2	2	3	54	65	65	61	65	**67**
	r_6	$\{\cdot * ?\}$	1	2	1	5	12	12	7	8	25	84	84	64	84	**88**
C_5	r_7	$\{\cdot * ?\}$	1	1	0	11	9	72	9	12	54	83	83	70	83	**89**
	r_8	$\{\cdot + ?\}$	1	2	1	6	6	6	5	7	29	69	69	62	69	**90**
	r_9	$\{\cdot + ?\}$	1	2	1	7	6	12	6	10	33	79	79	67	79	**91**
C_6	r_{10}	$\{\cdot + ?\}$	1	1	0	10	6	16	6	10	54	76	76	72	76	**83**
	r_{11}	$\{\cdot * +\}$	2	2	1	5	15	21	7	8	23	83	83	82	83	**91**
C_7	r_{12}	$\{\cdot * +\}$	1	1	0	15	10	108	10	12	86	77	77	74	77	**80**
	r_{13}	$\{\cdot * + ?\}$	1	1	0	9	7	24	7	10	46	80	80	76	80	**87**
	r_{14}	$\{\cdot * + ?\}$	2	2	1	9	27	126	9	12	40	80	80	78	80	**93**
C_8	r_{15}	$\{\cdot * + ?\}$	1	2	1	12	21	78	10	16	52	83	83	75	83	**94**
C_9	r_{16}	$\{\vert\}$	0	0	0	8	9	9	9	9	53	**49**	**49**	44	**49**	**49**
C_{10}	r_{17}	$\{\cdot \vert\}$	0	0	1	5	4	4	3	3	62	**52**	**52**	40	**52**	**52**
	r_{18}	$\{\cdot \vert ?\}$	0	1	1	8	10	20	12	14	58	66	66	53	66	**69**
C_{11}	r_{19}	$\{\cdot \vert ?\}$	0	2	2	10	12	12	7	11	55	71	71	65	71	**80**
	r_{20}	$\{\vert *\}$	1	1	1	13	17	51	12	12	140	**67**	**67**	61	29	29
C_{12}	r_{21}	$\{\vert *\}$	1	1	1	14	29	211	16	16	115	**100**	**100**	47	62	62
	r_{22}	$\{\cdot \vert *\}$	1	1	1	9	22	63	12	12	45	**86**	**86**	82	**86**	**86**
	r_{23}	$\{\cdot \vert *\}$	2	2	2	9	75	1281	11	11	69	**87**	**87**	83	67	67
	r_{24}	$\{\cdot \vert *\}$	2	2	1	12	49	2366	15	15	72	**94**	**94**	90	74	74
	r_{25}	$\{\cdot \vert *\}$	1	1	2	16	99	9765	19	19	137	**100**	**100**	85	72	72
C_{13}	r_{26}	$\{\cdot \vert *\}$	1	1	1	45	581	22399	48	48	359	**100**	**100**	**100**	62	62
	r_{27}	$\{\vert +\}$	1	1	0	5	6	6	6	7	20	**85**	**85**	60	**85**	**85**
	r_{28}	$\{\vert +\}$	2	2	1	5	9	21	5	6	28	61	61	54	50	86
C_{14}	r_{29}	$\{\vert +\}$	1	1	1	47	94	2256	48	48	472	**80**	**80**	79	70	70
	r_{30}	$\{\cdot \vert +\}$	2	2	1	5	8	20	4	5	31	**74**	**74**	55	55	**74**
C_{15}	r_{31}	$\{\cdot \vert +\}$	1	1	1	7	10	22	7	8	55	**91**	**91**	48	67	82
	r_{32}	$\{\vert * ?\}$	1	2	1	9	15	57	11	11	48	**100**	**100**	65	60	60
C_{16}	r_{33}	$\{\vert * ?\}$	1	1	0	25	24	24	24	25	73	81	81	50	81	**84**
	r_{34}	$\{\cdot \vert * ?\}$	1	1	1	12	42	222	14	15	124	**99**	**99**	96	57	58
	r_{35}	$\{\cdot \vert * ?\}$	1	2	2	21	88	3472	19	19	78	**88**	**88**	73	60	60
C_{17}	r_{36}	$\{\cdot \vert * ?\}$	1	2	1	22	145	2110	21	25	163	**96**	**96**	92	69	71
	r_{37}	$\{\vert * +\}$	2	2	1	8	18	90	9	9	31	**100**	**100**	71	39	39
C_{18}	r_{38}	$\{\vert * +\}$	2	2	1	77	155	6162	78	78	314	**100**	**100**	98	50	50
	r_{39}	$\{\cdot \vert * +\}$	1	1	1	9	37	93	11	12	57	**95**	**95**	89	68	74
	r_{40}	$\{\cdot \vert * +\}$	1	1	1	9	66	372	12	13	93	**92**	**92**	91	68	76
	r_{41}	$\{\cdot \vert * +\}$	2	2	1	11	66	1116	13	14	67	**97**	**97**	96	89	93
C_{19}	r_{42}	$\{\cdot \vert * +\}$	1	1	2	19	24	24	13	15	104	73	73	69	70	**74**
	r_{43}	$\{\vert + ?\}$	1	1	1	13	24	134	14	15	108	**99**	**99**	76	78	88
C_{20}	r_{44}	$\{\vert + ?\}$	1	2	1	16	20	110	18	18	68	**100**	**100**	71	62	62
	r_{45}	$\{\cdot \vert + ?\}$	1	2	1	6	10	10	7	8	26	85	85	77	85	**88**
	r_{46}	$\{\cdot \vert + ?\}$	2	2	1	6	10	10	7	8	28	79	79	64	79	**82**
C_{21}	r_{47}	$\{\cdot \vert + ?\}$	1	1	1	59	220	12320	113	116	662	**83**	**83**	**83**	66	75
	r_{48}	$\{\vert * + ?\}$	1	1	0	13	12	12	12	15	44	80	80	50	80	**93**
	r_{49}	$\{\vert * + ?\}$	2	2	1	23	38	380	25	25	118	**100**	**100**	88	64	64
C_{22}	r_{50}	$\{\vert * + ?\}$	2	2	1	28	37	343	30	30	120	**100**	**100**	68	60	60
	r_{51}	$\{\cdot \vert * + ?\}$	3	3	3	22	100	6007	35	39	143	**94**	**94**	90	83	89
	r_{52}	$\{\cdot \vert * + ?\}$	2	3	3	39	1022	3664128	43	44	179	**95**	**95**	**95**	85	89
	r_{53}	$\{\cdot \vert * + ?\}$	1	2	1	49	249	20676	48	52	368	**98**	**98**	**98**	72	73
C_{23}	r_{54}	$\{\cdot \vert * + ?\}$	1	1	3	67	676	76624	121	125	518	**91**	**91**	**91**	80	86
	r_{55}	$\{\cdot ? []\}$	0	1	0	12	19	96	9	15	54	83	83	74	83	**92**
C_{24}	r_{56}	$\{\cdot \vert ? []\}$	0	1	1	24	26	3840	16	29	110	79	79	73	79	**90**
	r_{57}	$\{? \&\}$	0	1	0	11	28	1957	-	-	46	**87**	**87**	-	-	-
C_{25}	r_{58}	$\{? \&\}$	0	1	0	19	61	9864101	-	-	78	**87**	**87**	-	-	-
Aver.							76	236280	18	20		**85**	**85**	73	72	77

TABLE 3. Real-world expressions used in the experiments, numbers of strings generated by different algorithms and the mutation testing results.

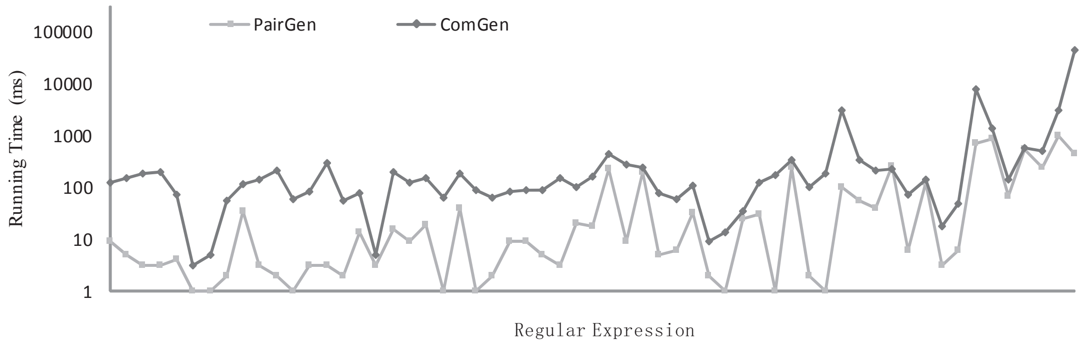

FIGURE 8. Running time of string generation algorithms ComGen and PairGen.

generation strategy adopted by Egret is simpler than ours especially when dealing with | (see the example given in Section 6.4.1 (3)). Therefore, the generated strings may be not sufficient to detect some complex errors. For instance, consider expression $(a|b)^*$ and one of its specialized mutant $(a^*|b^*)$, Egret produces positive strings $\{\epsilon, a, b, aa\}$, Clearly, they cannot kill this mutant.

(4) PairGen vs. Egret. Theoretically, negative strings may probably be able to kill some of the generalized mutants that can never be killed by positive ones. The experimental results verify this analysis. The mutation score of Egret is increased when negative strings are used. Thus it is higher than PairGen. for expressions where PairGen and Egret$^+$ earn a same mutation score. However, as we have mentioned in Section 6.4.1, Egret generates only a small number of negative strings. However, our experimental results show that there are still generalized mutants not killed. Take a very simple expression $a?|b?$ for instance, Egret generates two negative strings $\{aa, bb\}$. Consider the mutant $a?b?$, which is a generalization of $a?|b?$. Since $a?b?$ rejects both aa and bb, this mutant cannot be killed. Actually, the average mutation score of Egret even with negative strings is still lower than PairGen which generates no negative strings (77% vs. 85%).

6.4.3. Running Time

Figure 8 shows, for each regular expression used in our experiments, the time taken for generating strings using algorithms PairGen and ComGen. Regular expressions are sorted by their sizes. The tests were repeated 100 times and the total running time was recorded. Since the running time of ComGen for certain regular expressions are two large, we use a logarithmic scale to show the results in figures.

We can observe that, in general, the running time grows with the sizes of the expressions for both PairGen and ComGen; however, there are also long regular expressions consuming less time than shorter expressions. Indeed, the time required for string generation depends not only on the size but also on the structure and form of the regular expression. Take $(a|b|c)?$ and $(a|b|c)^*$ for instances, the two expressions have the same sizes; but the former consumes less time because the generation strategy for ? is much simpler than $*$. Comparing PairGen and ComGen, we observe that, for a same expression, PairGen takes less time than ComGen. This is not surprising because ComGen generates much more strings which consequently requires much more running time.

Summary. We find the following from these experiments. (1) Test data generated based on the coverage criteria proposed in this study is more capable of detecting faults than randomly generated test data.

(2) Considering both the size and the fault detection ability of test data, pairwise coverage based generation is much more useful than combination coverage based generation in practical applications.

(3) Compared with Egret, pairwise coverage based generation can reveal more faults for testing complex regular expressions while Egret is more suitable when dealing with simple regular expressions.

7. MORE APPLICATIONS

The generation algorithms proposed in this work are not only useful to test regular expressions themselves but also useful to test systems that require structured data as inputs and use regular expressions to describe the input structures. One application example is in computational biology. It has been long proposed [44] that DNA sequences can be modeled as strings in regular languages or in languages higher up in the Chomsky hierarchy such as context-free languages. Automatic generation of strings in these languages provides a good set of test data for algorithms that process DNA sequences. Another application example is in semi-structured data management. As introduced in Section 6, regular expressions are used in XML schema languages to specify the structural constraints of XML documents. Automated generation of XML documents conforming to a given schema can be used for black-box testing of applications accepting as input XML documents and for benchmarking of XML database management systems. Since regular expressions consist of an important part of XML schemas, generation of XML documents largely relies on algorithms for generating strings from regular expressions. Finally, being able to generate strings from regular expressions allows us to generate test samples to test regular expression learning algorithms. We discuss this application in the following.

Over the past years, many algorithms have been proposed that are capable of learning a regular expression, given a finite number of samples of the language defined by the target expression. Those algorithms are useful in various fields such as information extraction [45], pattern recognition [46], etc. Because it is not always possible to formally prove optimality of the learning algorithms, their effectiveness is usually validated by experiments. One experimental method is to generate samples from the target expression and to see how closely the learned expression resembles the target one. Thus the string generation algorithms proposed in this work can be used for generating test samples for regular expression learning algorithms. In particular, we show that they are especially useful for some algorithms that are based on Gold's learning paradigm of *learning (identification) in the limit* [47], which is explained as follows.

A language (target) class $\mathbb{L}$ (defined via a class of language describing formalisms Γ as, e.g., expressions, automata or grammars) is said to be *learnable* or *identifiable* if there is an algorithm M mapping samples to objects in Γ such that: (1) $S \subseteq L(M(S))$ for every sample S, and (2) to every object $d \in \Gamma$ we can associate a so-called *characteristic sample* $S_c \subseteq L(d)$ such that, for each sample S with $S_c \subseteq S \subseteq L(d)$, $L(M(S)) = L(d)$. Intuitively, the first condition says that algorithm M must be sound; the second condition says that M must be complete, i.e., M should convergence when the sample contains enough data. By the above definition, *enough data* is specified by the so-called characteristic sample. Clearly, characteristic samples are very useful and important in testing such learning algorithms. If such a sample is the input data for an expression E, then according to Gold's *learning in the limit* model, the learning algorithm should return exactly E or an expression equivalent to E.

It was shown by Gold [47] that the class of all regular languages is not learnable from positive data. Researchers have identified several subclasses of regular languages that can be learnable. We next show that for some of the learnable subclasses, we can automatically generate the characteristic samples by using the generation algorithms proposed in this study. The regular expressions mentioned below contain only standard operators.

Single Occurrence Regular Expressions. Bex et al. [48] identified a subclass described by a kind of restricted regular expressions, called single occurrence regular expressions (SOREs for short) and proposed a novel algorithm for learning SOREs from positive data. A follow-up work [49] made further improvements. A regular expression is single occurrence if every symbol occurs at most once in it. For example, $(a|b)c^*$ is a SORE while $(a|b)c^*(a|d)$ is not. Bex et al. pointed out that the characteristic sample for the language defined by a given SORE E is specified by using the notion of *2-grams*. Recall that a 2-gram of a set of strings W is a string of length 2 that occurs as a substring in some $w \in W$. A sample $S \subseteq L(E)$ is characteristic for an SORE E if the following holds:

(1) For every $a \in \Sigma$ starting a string in $L(E)$ there is a string in S that starts with a;

(2) For every $b \in \Sigma$ ending a string in $L(E)$ there is a string in S that ends with b;

(3) Every 2-gram of $L(E)$ is a 2-gram of S.

Clearly, a set fulfilling combination coverage satisfies the above three conditions. Actually, we show that pairwise coverage is already sufficient for this purpose.

Proposition 7.1. *Given an SORE E, the set generated by algorithm* PairGen *is a characteristic sample for E.*

Proof. By Proposition 4.1, if a set S satisfies pairwise coverage for E, then S contains all the 2-grams of $L(E)$. We next show that conditions (1)(2) also hold by induction on the structure of expression E. We only need to show that the set $C(E)$ defined by pairwise coverage satisfies conditions (1)(2). One can easily verify this for the case that E is ϵ, $\emptyset$ or a for some $a \in \Sigma$, or E is of the form $E = E_1|E_2$. We next prove the other two cases.

- If $E = E_1 \cdot ... \cdot E_n$ and suppose that for each E_k ($k \in [1, n]$), $C(E_k)$ satisfies conditions (1)(2). Let a be a starting symbol in $L(E)$. Then a must be a starting symbol in $L(E_k)$ for some $k \in [1, n]$. If $k = 1$, by supposition, there is a string $s \in C(E_1)$ that starts with a and by the definition of pairwise coverage there exists a string $w \in C(E)$ that starts with s and thus also starts with a. If $k > 1$, this implies that for each E_i with $i < k$, $L(E_i)$ contains empty string ϵ. One can easily verify by the definition of pairwise coverage that if $L(E_i)$ contains ϵ then $C(E_i)$ is also required to contain ϵ. This means that $\epsilon \in C(E_1)$ and furthermore for each E_j between E_1 and E_k, $\epsilon \in C(E_j)$. Again by definition of pairwise coverage, there exists a string $w \in C(E)$ that adjacently covers the ordered pair $\langle \epsilon, s \rangle$ (Note that ϵ in this pair comes from $C(E_1)$). This indicates that w starts with s and thus with symbol a. Therefore we conclude that in both of the two cases, $C(E)$ contains a string starting with a. That is, condition (1) holds. Along the same lines, one can verify that condition (2) also holds.

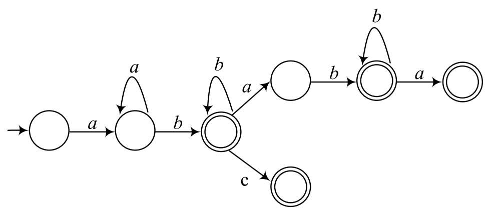

FIGURE 9. A simple looping automaton.

- If $E = E_1^*$ and suppose that $C(E_1)$ satisfies conditions (1)(2). Let a be a starting symbol in $L(E)$. Then a must be a starting symbol in $L(E_1)$. By supposition, there is a string $s \in C(E_1)$ that starts with a. Notice that to capture the feature of the $*$ operator and meanwhile to avoid infinite generation, pairwise coverage restricts the $*$ repetitions to be three possibilities: zero, once and more than once. We hence have $C(E_1) \subseteq C(E)$ which implies that s also appeared in $C(E)$. Thus condition (1) holds. The verification for condition (2) is similar.

□

Simple Looping Regular Expressions. Fernau [50] identified another subclass of learnable regular languages, namely, simple looping regular languages.

This languages are defined by simple looping regular expressions. A regular expression is called simple looping if it is a finite union of pairwisely left-aligned union-free expressions α such that α is ϵ or $a_1^{k_1}a_1^{x_1}a_2^{k_2}a_2^{x_2}...a_n^{k_n}a_n^{x_n}$, where each a_i is a single symbol from the basic alphabet Σ, each k_i is some positive integer, each x_i equals to either 0 or $*$, and for all $1 \le i < n$, $a_i \ne a_{i+1}$. Two such union-free expressions β, γ are left-aligned if $\beta \ne \gamma$ or if they share a common longest prefix subexpression $\bar{E}$, i.e., $\beta = \bar{E}E_\beta$, $\gamma = \bar{E}E_\gamma$, then E_β and E_γ begin with symbols different from the ending symbol of $\bar{E}$. For example, aa^*b and aa^*c are left-aligned while a and aa are not.

Simple looping languages can also be characterized by simple looping automata. A finite automaton A is called simple looping iff when deleting all the loops in the automaton graph, the resulting (arc-labeled) skeleton graph would be a directed tree without multiple arcs such that for any node of outdegree larger than one, its emanating arcs carry pairwisely different labels, and the label a carried by some loop arc (at some state q) in the automaton graph is likewise carried by the necessarily existing other arc pointing to q. For example, the automaton displayed in Figure 9 is simple looping. (This example is taken from [50].)

A simple looping automaton can be equivalently translated to a simple looping expression and vice versa. Simply speaking, given an automaton A, we can decompose it into a set of paths such that each path contains only one final state, namely the last one on the path. Each path can be turned into a union-free expression whose collection yields the required simple looping regular expression. Conversely, given a simple looping regular expression consisting of pairwisely left-aligned union-free subexpressions E_i, we can turn each E_i into a simple looping automaton A_i, and overlay the different A_is to produce a total simple looping automaton. For instance, the simple looping expression corresponds to the automaton in Figure 9 is

$$aa^*bb^*|aa^*bb^*c|aa^*bb^*abb^*|aa^*bb^*abb^*a$$

Given some sample strings, Fernau's learning algorithm first constructs a simple looping automaton and then translates the automaton into a simple looping expression. The construction of a simple looping automaton A is divided into two steps. (1) Create a start tree. First transform strings into block strings where each block $[x]$ represents the repetition of symbol x for any times. For instance, $\{ababb, aabb, ababa, abc\}$ is transformed into $\{[a][b][a][b], [a][b], [a][b][a][b][a], [a][b][c]\}$. Then create a classical prefix tree acceptor [51] from the block strings where arcs are labeled by block letters. For example, the underlying tree structure (deleting loops) of the automaton shown in Figure 9 is the tree constructed from the above sample strings. (2) Introduce loops. Consider an arc that is labeled with a block letter $[a]$. If the finite set of strings represented by $[a]$ during step (1) contains more than one string, say $a^{n_1}, ..., a^{n_m}$, then only keep the shortest among these words as a label of the arc and enable the generation of the longer words by introducing a loop labeled a at the node that arc points to. For example, the automaton illustrated in Figure 9 is the final simple looping automaton constructed from the given sample $\{ababb, aabb, ababa, abc\}$.

Fernau described how to obtain a characteristic sample from a simple looping automaton. Actually, from the algorithm described above, one can infer that if a sample can provide the information to construct the prefix block tree and then to provide enough information for introducing the loops, then this sample is a characteristic sample. We next show that using our pairwise coverage based generation algorithm, we can automatically obtain a characteristic sample for a simple looping language L when L is described by a simple looping expression.

Proposition 7.2. *Given a simple looping regular expression E, the set generated by algorithm* PairGen *is a characteristic sample for E.*

Proof. By definition, regular expression E is of the form $E = E_1|...|E_m$ where each E_i ($i \in [1, m]$) is a concatenation of subexpressions of the form a^ka^* or a^k with k a positive integer. Let A be the simple looping automaton translated from E. By translation, each E_i corresponds to a path from the initial state to a final state in the prefix block tree (deleting loops) of A and each Kleene star corresponds to a loop in A. Due to the restricted form of each E_i, one can verify that from any string $w \in L(E_i)$ we can create the corresponding path of E_i in the prefix block tree of automaton A. Let S be the string set generated by PairGen. On one hand, pairwise coverage ensures that for each E_i there exists at least one string of $L(E_i)$ contained in S. Thus S provides enough information to construct the prefix block tree. On the other hand, pairwise coverage ensures that for each Kleene star a^*, a repeats at least more than once. Thus S provides enough information to introduce the loops. In total, we conclude that S is a characteristic sample for E.

□

8. THREATS TO VALIDITY

In this study, we investigate regular expressions from the perspective of formal language theory and practice. We consider the formal definition of regular expressions and support basic regular operators and some commonly used extended regular operators. We take XML schema testing as the application background to evaluate our algorithms. We select regular expressions from a large data set of real-world XML schemas. In our selection, we try to cover various kinds of regular expressions occurring in these schemas, that is, expressions containing different regular operators, of different forms, different structures, different sizes and different complexities. Therefore, we believe that the experimental results could be at least generalized to all expressions that are used in the same application, i.e., XML schemas. In other applications, for example, string pattern matching, regular expressions have a very different syntax. Operators such as *negation*, *character class* are allowed. Some string generation tools including Egret are developed specially for testing such expressions. In theory, this kind of expressions can be translated to the standard expressions and thus can use our algorithms for string generation. Therefore, in such application, our algorithm can be used as a complement to other specialized algorithms.

We next discuss the faults that may not be revealed by our algorithms. First of all, since our algorithms generate only positive strings, generalized mutants can never be killed. In our experiments, we have tried to add the negative strings produced by Egret to ours and find that the average mutation score is indeed increased. However, as we have analyzed, since Egret's negative generation strategy is relatively simple, there are still a large number of generalized mutants not killed. How to derive meaningful negative strings is a problem that needs further investigation. Secondly, similar to pairwise testing, pairwise coverage has its own limitations. For instance, since it covers only two (pairwise) combinations, errors related to more than two combinations cannot be captured. Although combination coverage can help revealing such kind of errors, the generated test set is too large. We always need to balance the test size with the fault detection capability. One solution might be to define more coverage criteria that are stronger than pairwise coverage and weaker than combination coverage, for example, n-wise coverage ($n \geq 2$), and let the user to decide which coverage is used for generating test strings.

9. RELATED WORK

In software testing community, the objects under test are usually software systems, or more specifically, programs or applications. Test techniques include the process of executing the program or application with the intent of finding software bugs (errors or other defects), and verifying that the software system is fit for use. As stated in [52], *structural descriptions* or *descriptions of structures* are indispensable in some software systems such as parsers and XML document processors. These descriptions are usually specified by different kinds of grammar formalisms including context-free grammars, tree/graph grammars, regular expressions and so on. In such *grammarwares*, ensuring the completeness and correctness of structural descriptions is a vital prerequisite for their uses. Some efforts have been devoted to this issue. For instance, [17, 53, 54] studied testing approaches and metrics definitions for context-free grammars, [22, 55] investigated testing approaches and metrics definitions for attribute grammars, [56, 42, 57] explored testing approaches for XML schemas (tree grammars) and [58] studied approaches for developing high-quality generative design grammars (graph grammars). However, little work has been found concerning the quality of regular expressions.

Coverage Criteria are an important topic in software testing, which are usually sets of rules to help determine whether a test set has adequately tested a software and provide a basis for test data generation. A number of coverage criteria have been proposed and applied such as statement coverage, branch coverage, path coverage and so on. These criteria are defined for programs, i.e., defined based on the codes of programs under test. Coverage criteria for different grammar formalisms have also been studied. The notion of rule coverage used as a criterion for testing context-free grammars was first introduced by Purdom [16]. A test string is said to cover a grammar rule if that rule is used at least once in deriving that string. Purdom described an algorithm for generating a minimal set of strings that uses all the rules of a grammar, i.e., 100% rule coverage. Following up on Purdom's work, several string generation algorithms based on rule coverage have been developed. [59] extended Purdom's algorithm by modifying the rule choose strategy and [20] by integrating a length control mechanism in string generation process. Lämmel [17] proposed a generalization of rule coverage, where the context in which a rule is covered is taken into account. This criterion is called context-dependent rule coverage and a string generation algorithm for this metric was worked out in [21]. Based on Purdom and Lämmel's work, [18] introduced more test criteria for context-free grammars including length-k successor coverage that takes not just one or two derivation steps but $k > 2$ derivation steps of the context in which a grammar rule is covered into account, but did not provide corresponding string generation algorithms. [19] developed the notion of two-dimensional approximation coverage for attribute grammars, which involves a syntactic dimension corresponding to the underlying context-free grammar and a semantic dimension corresponding to the attributes, conditions and computations, and designed a generator algorithm for test sets achieving this coverage. These work on coverage criteria and string generation algorithms focuses mainly on grammars, while our work is devoted specifically to regular expressions.

Also close to our work is automatic generation of strings for regular languages. This is a fundamental issue in formal language theory, and sampling and enumerating have been extensively studied. (1) Sampling, i.e., randomly and uniformly generating strings from a regular language, was first addressed by Hickey and Cohen [6], as a particular case of context-free languages, and a recursive sampling algorithm was proposed to take a deterministic finite automaton of a regular language, and generate uniformly at random a string of length n. Kannan et al. [7] studied the problem of counting and random generation of strings in regular languages that are described by nondeterministic finite automata, and provided randomized algorithms for approximate counting and almost uniform generation. Recently, Bernardi and Giménez [8] developed a new divide-and-conquer approach for sampling strings of regular languages, which improves the average complexity of generation by using floating point arithmetic. Also based on a divide-and-conquer approach, Oudinet et al. [9] devised the so-called dichopile method which offers an excellent compromise in terms of space and time requirements. (2) Enumerating was first discussed in [10] where the author presented an algorithm that given a regular grammar returns all strings of length n derived by that grammar in lexicographical order. Then Ackerman and Shallit [11, 12] modified the algorithm to obtain an algorithm that takes a nondeterministic finite automaton as input. In [13], the authors presented new and more efficient algorithms with better practical running times for regular language enumeration problems, including finding the lexicographically minimal string of length n, listing all strings of length n in lexicographical order, and listing the first m strings according to length-lexicographic order. This work is different to ours in that we consider taking a regular expression as direct input and emphasize on producing string sets achieving certain coverage criterion for that regular expression.

There are several tools developed for generating strings from regular expressions, such as Xeger [41], Exrex [60], Generex [61], Egret [3]. These tools focus on regular expression *patterns*, i.e., expressions used in manipulating text strings, while in this paper we study general-purpose regular expressions from the perspective of formal language theory. Moreover, the generation strategies of these tools are random [41, 60, 61] or based on the underlying automata [3]. Our generation, on the contrast, is based on coverage criteria defined directly on expressions themselves. The experimental results show that coverage based generation outperforms random generation in testing regular expressions. Egret first converts a regular expression into a specialized automaton, then derives a set of basis paths of the resulting automaton and finally creates strings from the basis paths. In one sense, we may say that the generated strings can cover the basis paths of the specialized automaton. Roughly speaking, this coverage is not as strict as pairwise coverage proposed in this paper. Our experiments show that some of the faults captured by our algorithm could not be exposed by Egret even if Egret can produce negative test strings. Moreover, our algorithms can be used to generate characteristic samples for validating regular expression learning algorithms. Egret, however, cannot guarantee the generation of characteristic samples.

10. CONCLUSIONS

Coverage criteria are used to measure the quality of a particular test set, and to provide strategies for test data generation algorithms. In this study, we have proposed a novel coverage for both standard regular expressions and regular expressions extended with counting and interleaving. We have compared the criterion for regular expressions with existing criteria for both regular grammars and finite automata, and analyzed their subsumption relationships. We have also developed the string generation algorithm, and experimentally verified the effectiveness of our algorithms. Finally, we have identified more applications of our coverage and generation algorithm, especially in the application of generating characteristic samples for some regular expression learning algorithms. These results are not only theoretically meaningful in formal language research, but are also practically useful for applications involved with regular expressions.

In the future, we plan to investigate more practical applications of the coverage and string generation algorithms, such as DNA sequences. Another work is to extend our algorithms for generating not only positive but also useful negative strings.

ACKNOWLEDGEMENTS

Zheng is supported in part by NSFC (No. 61502184 & 61472405) and Natural Science Foundation of Fujian

Province (No. 2015J01259). Ma is supported in part by NSFC (No. U1636210 & 61421003) and 973 Program (No. 2014CB340300). We also thank the research group of Professor Haiming Chen for preparing the extended regular expressions used in our experiments.

REFERENCES

[1] Murata, M., Lee, D., Mani, M., and Kawaguchi, K. (2005) Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, **5**, 660–704.

[2] Fan, W., Li, J, Ma, S., Tang, N. and Wu, Y. (2012) Adding regular expressions to graph reachability and pattern queries. *Frontiers of Computer Science*, **6(3)**, 313–338.

[3] Larson, E. and Kirk, A. (2016) Generating evil test strings for regular expressions. *Proceedings of International Conference on Software Testing, Verification and Validation (ICST),* , Chicago, IL, USA, 11-15 April, pp. 309–319. IEEE CS, Washington.

[4] Arcaini, P., Gargantini, A., and Riccobene, E. (2018) Fault-based test generation for regular expressions by mutation. *Software Testing, Verification and Reliability* , e1664.

[5] Li, N., Xie, T., Tillmann, N., de Halleux, J., and Schulte, W. (2009) Reggae: Automated test generation for programs using complex regular expressions. *Proceedings of International Conference on Automated Software Engineering (ASE)* , Auckland, New Zealand, 16-20 November, pp. 515–519. IEEE CS, Washington.

[6] Hickey, T. and Cohen, J. (1983) Uniform random generation of strings in a context-free language. *SIAM Journal on Computing*, **12**, 645–655.

[7] Kannan, S., Sweedyk, Z., and Mahaney, S. (1995) Counting and random generation of strings in regular languages. *Proceedings of annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, San Francisco, California, USA, 22-24 January pp. 551–557. SIAM, Philadelphia.

[8] Bernardi, O. and Giménez, O. (2012) A linear algorithm for the random sampling from regular languages. *Algorithmica*, **62**, 130–145.

[9] Oudinet, J., Denise, A., and Gaudel, M. C. (2013) A new dichotomic algorithm for the uniform random generation of words in regular languages. *Theoretical Computer Science*, **502**, 165–176.

[10] Mäkinen, E. (1997) On lexicographic enumeration of regular and context-free languages. *Acta Cybernetica*, **13**, 55–61.

[11] Ackerman, M. and Shallit, J. (2007) Efficient enumeration of regular languages. *Proceedings of International Conference on Implementation and Application of Automata (CIAA)*, Prague, Czech Republic, 16-18 July, pp. 226–242. Springer, Berlin.

[12] Ackerman, M. and Shallit, J. (2009) Efficient enumeration of words in regular languages. *Theoretical Computer Science*, **410**, 3461–3470.

[13] Ackerman, M. and Mäkinen, E. (2009) Three new algorithms for regular language enumeration. *Proceedings of International Computing and Combinatorics Conference (COCOON)*, Niagara Falls, NY, USA, 13-15 July, pp. 178–191. Springer, Berlin.

[14] Gore, V., Jerrum, M., Kannan, S., Sweedyk, Z., and Mahaney, S. (1997) A quasi-polynomial-time algorithm for sampling words from a context-free language. *Information and Computation*, **134**, 59–74.

[15] Myers, G. J., Sandler, C., and Badgett, T. (2011) *The art of software testing.* John Wiley & Sons, Hoboken, New Jersey.

[16] Purdom, P. (1972) A sentence generator for testing parsers. *BIT*, **12**, 366–375.

[17] Lämmel, R. (2001) Grammar testing. *Proceeding of International Conference on Fundamental Approaches to Software Engineering (FASE)*, Genova, Italy, 2-6 April, pp. 201–216. Springer, Berlin.

[18] Li, H., Jin, M., Liu, C., and Gao, Z. (2004) Test criteria for context-free grammars. *Proceedings of International Computer Software and Applications Conference (COMPSAC)*, Hong Kong, China, 27-30 September, pp. 300–305. IEEE CS, Washington.

[19] Harm, J. and Lämmel, R. (2000) Two-dimensional approximation coverage. *Informatica*, **24**, 355–369.

[20] Zheng, L. and Wu, D. (2009) A sentence generation algorithm for testing grammars. *Proceedings of International Computer Software and Applications Conference (COMPSAC)*, Seattle, Washington, USA, 20-24 July, pp. 130–135. IEEE CS, Washington.

[21] Xu, Z., Zheng, L., and Chen, H. (2011) A toolkit for generating sentences from context-free grammars. *International Journal of Software and Informatics*, **5**, 659–676.

[22] Harm, J. and Lämmel, R. (2000) Testing attribute grammars. *Proceedings of third workshop on attribute grammars and their applications (WAGA)*, Ponte de Lima, Portugal, 7 July, pp. 79–98. Springer, Berlin.

[23] Hume, A. (1988) A tale of two greps. *Software: Practice and Experience*, **18**, 1063–1072.

[24] Wall, L., Christiansen, T., and Orwant, J. (2000) *Programming perl.* O'Reilly Media, Inc, Sebastopol, California.

[25] Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (2004). XML Schema part 1: structures. Second edition.

[26] Clark, J. and Makoto, M. (2001). Relax NG Tutorial.

[27] Kilpeläinen, P. and Tuhkanen, R. (2003) Regular expressions with numerical occurrence indicators-preliminary results. *Proceeding of Symposium on Programming Languages and Software Tools (SPLST)*, Kuopio, Finland, 17-18 June, pp. 163–173. CEUR-WS.org, Aachen, Germany.

[28] Gelade, W. (2010) Succinctness of regular expressions with interleaving, intersection and counting. *Theoretical Computer Science*, **411**, 2987–2998.

[29] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001) *Introduction to Automata Theory, Languages, and Computation (Second Edition).* Addison-Wesley, Boston, MA.

[30] Nie, C. and Leung, H. (2011) A survey of combinatorial testing. *ACM Computing Surveys*, **43**, 11:1–11:29.

[31] Pairwise testing. http://www.pairwise.org/.

[32] Ntafos, S. C. (1988) A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, **14**, 868–874.

[33] Zhu, H., Hall, P. A. V., and May, J. H. R. (1997) Software unit test coverage and adequacy. *ACM Computing Surveys*, **29**, 366–427.

[34] Han, Y. and Wood, D. (2007) Obtaining shorter regular expressions from finite-state automata. *Theoretical Computer Science*, **370**, 110–120.

[35] Lei, Y. and Tai, K.-C. (1998) In-parameter-order: A test generation strategy for pairwise testing. *Proceedings of International High-Assurance Systems Engineering Symposium (HASE)*, Washington, D.C, USA, 13-14 November, pp. 254–261. IEEE CS, Washington.

[36] Anant P. Godbole, D. E. S. and Sunley, R. A. (1996) t-covering arrays: upper bounds and poisson approximations. *Combinatorics, Probability and Computing*, **5**, 105–117.

[37] Abiteboul, S., Buneman, P., and Suciu, D. (2000) *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann, San Francisco.

[38] Li, Y., Zhang, X., Peng, F., and Chen, H. (2016) Practical study of subclasses of regular expressions in DTD and XML Schema. *Proceedings of Asia-Pacific Web Conference (APWeb)*, Suzhou, China, 23-25 September, pp. 368–382. Springer, Berlin.

[39] Bala, S. (2002) Intersection of regular languages and star hierarchy. *Proceedings of International Colloquium on Automata, Languages, and Programming (ICALP)*, Malaga, Spain, 8-13 July, pp. 159–169. Springer, Berlin.

[40] Demillo, R. A., Lipton, R. J., and Sayward, F. G. (1978) Hints on test data selection: Help for the practicing programmer. *Computer*, **11**, 34–41.

[41] Xeger. https://code.google.com/archive/p/xeger/.

[42] Li, J. B. and Miller, J. (2005) Testing the semantics of W3C XML Schema. *Proceedings of International Computer Software and Applications Conference (COMPSAC)*, Edinburgh, Scotland, UK, 25-28 July, pp. 443–448. IEEE CS, Washington.

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

[44] Searls, D. B. (1993) The computational linguistics of biological sequences. In Hunter, L. (eds), *Artificial intelligence and molecular biology*. American Association for Artificial Intelligence, Menlo Park, CA, USA.

[45] Brauer, F., Rieger, R., Mocan, A., and Barczynski, W. M. (2011) Enabling information extraction by inference of regular expressions from sample entities. *Proceedings of International Conference on Information and Knowledge Management (CIKM)*, Glasgow, United Kingdom, 24-28 October, pp. 1285–1294. ACM, New York.

[46] Garcia, P. and Vidal, E. (1990) Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **12**, 920–925.

[47] Gold, E. M. (1967) Language identification in the limit. *Information and Control*, **10**, 447 – 474.

[48] Bex, G. J., Neven, F., Schwentick, T., and Vansummeren, S. (2010) Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems*, **35**, 1–47.

[49] Freydenberger, D. D. and Kötzing, T. (2015) Fast learning of restricted regular expressions and DTDs. *Theory of Computing Systems*, **57**, 1114–1158.

[50] Fernau, H. (2009) Algorithms for learning regular expressions from positive data. *Information and Computation*, **207**, 521–541.

[51] Angluin, D. (1982) Inference of reversible languages. *Journal of the ACM*, **29**, 741–765.

[52] P.Klint, R.Lämmel, and C.Verhoef (2005) Towards an engineering discipline for grammarware. *ACM Transaction on Software Engineering and Methodology*, **14**, 331–380.

[53] Power, J. F. and Malloy, B. A. (2004) A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, **16**, 405–426.

[54] Crepinsek, M., Kosar, T., Mernik, M., Cervelle, J., Forax, R., and Roussel, G. (2010) On automata and language based grammar metrics. *Computer Science and Information Systems*, **7**, 309–329.

[55] Cruz, J., Henriques, P. R., and da Cruz, D. (2015) Assessing attribute grammars quality: metrics and a tool. *Proceedingd of International Symposium on Languages, Applications and Technologies (SLATE)*, Madrid, Spain, 18-19 June, pp. 137–144. Springer, Berlin.

[56] Emer, M. C. F. P., Vergilio, S. R., and Jino, M. (2005) A testing approach for XML schemas. *Proceedings of International Computer Software and Applications Conference (COMPSAC)*, Edinburgh, Scotland, UK, 25-28 July, pp. 57–62. IEEE CS, Washington.

[57] Emer, M. C. F. P., Nazar, I. F., Vergilio, S. R., and Jino, M. (2012) Fault-based test of XML schemas. *Computing and Informatics*, **30**, 531–557.

[58] Königseder, C. and Shea, K. (2014) Systematic rule analysis of generative design grammars. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **28**, 227–238.

[59] Celentano, A., Crespi-Reghizzo, S., Della-Vigna, P., and Ghezzi, C. (1980) Compiler testing using a sentence generator. *Software Practice and Experience*, **10**, 897–918.

[60] Exrex. https://github.com/asciimoo/exrex.

[61] Generex. https://github.com/mifmif/generex.