Ensuring the Correctness of Regular Expressions: A Review

Lixiao Zheng[1] Shuai Ma[2] Zuxi Chen[1] Xiangyu Luo[1]

[1] College of Computer Science and Technology, Huaqiao University, Xiamen 361021, China

[2] SKLSDE Lab, Beihang University, Beijing 100191, China

Abstract: Regular expressions, due to their expressiveness and flexibility, are widely used within and even outside of computer science. However, regular expressions have a quite compact and rather tolerant syntax that makes them hard to understand, hard to compose and error-prone. Faulty regular expressions may cause failures of the applications that use them. Therefore, ensuring the correctness of regular expressions is a vital prerequisite for their use in practical applications. The importance and necessity of ensuring correct definitions of regular expressions have attracted extensive attentions from both researchers and practitioners especially in recent years. In this study, we provide a review on the recent works for ensuring the correct usage of regular expressions. We classify those works into different categories, including the empirical study, test string generation, automatic synthesis and learning, static checking and verification, visual representation and explanation, and repairing. For each category, we review the main results, compare different approaches, and discuss their advantages and disadvantages. We also discuss some potential future research directions.

Keywords: Regular expressions, correctness, string generation, learning, static checking, verification, visualization, repairing.

1 Introduction

Regular expressions, due to their expressiveness and flexibility, are widely used within and even outside of computer science. The importance of regular expressions for constructing the scanners of compilers is well known[1]. Nowadays, their applications extend to more areas such as network protocol analysis[2], MySQL injection prevention[3], network intrusion detection[4], XML data specification[5] and database querying[6], or more diverse applications like DNA sequence alignment[7]. Regular expressions are commonly used in computer programs for pattern searching and string matching, are a core component of almost all modern programming languages, and frequently appear in software source codes. Studies have shown that more than a third of JavaScript and Python projects contain at least one regular expression[8, 9].

However, recent researches have found that regular expressions are hard to understand, hard to compose and error-prone[10, 11]. Indeed, regular expressions have a quite compact and rather tolerant syntax that makes them hard to understand even for very short regular expressions. For example, it is not easy for users to capture immediately what strings the regular expression "\[([^\]]+)\]|\(([^\)]+)\)" specifies. It becomes much more difficult for complex regular expressions that may contain more than 100 characters or may have more than ten nested levels[12]. This is a real situation for software developers. For example, on the popular website stackoverflow.com, where developers learn and share their programming knowledge, there are more that 235,000 questions tagged with "regex".

Faulty regular expressions may cause failures in the corresponding applications that use them. Therefore, ensuring the correctness of regular expressions is a vital prerequisite for their use in practical applications. Actually, the importance of ensuring the correctness of regular expressions or other structural description models has already been recognized by some researchers. Klint, Lämmel and Verhoef [13] used the term "grammarware" to refer to all software that involves the grammar knowledge in an essential manner. Here grammar is meant in the sense of all structural descriptions or descriptions of structures used in software systems including regular expressions, grammars and so on. They noted that "in reality, grammarware is treated, to a large extent, in an ad-hoc manner with regard to design, implementation, transformation, recovery, testing, etc." Take the testing of regular expressions as an example, a survey of professional developers reveals that developers test their regular expressions less than the rest of their codes[8]. Indeed, an empirical study shows that about 80% of the regular expressions used in practical projects are not tested and among those tested, about half use only one test string that is far from sufficient[14]. Hence, sound and systematical methods and techniques are necessary to improve the quality of such software components.

The importance and necessity of checking the correctness and thus improving the quality of regular expressions have attracted extensive attentions from both researchers and practitioners especially in recent years. In this article, we provide a review on the recent works related to this issue. We classify the related works into different categories including empirical study, test string generation, automatic synthesis and learning, static checking and verification, visual representation and explanation, and

Research Article
Manuscript received date; accepted date; published online date
Recommended by Associate Editor xxxx

repairing. For each category, we review the main results, compare different approaches, and discuss their advantages and disadvantages.

The rest of this article is organized as follows: Section 2 introduces the preliminary knowledge on regular expressions, their different dialects, the meaning of correctness and finite automata. Section 3 to Section 8 review the relevant works on the correctness assurance of regular expressions according to different categories, respectively. Section 9 concludes with a summary and a discussion for further work.

2 Regular expressions

2.1 Formal definition

Regular expressions arose in the context of the formal language theory[15]. Simply speaking, a regular expression is a sequence of characters that defines a (possibly infinite) set of strings, which is called the language described by the regular expression. Formally, a regular expression defined on an alphabet Σ of symbols is described recursively as follows. The empty set $\emptyset$, the empty string ε and each symbol $a \in \Sigma$ are regular expressions, denoting the empty set $\emptyset$, the set containing only the "empty" string, the set containing only the character a, respectively. Suppose that R and S are two regular expressions, then the concatenation RS, the alternation R|S, and the Kleene star R^* are regular expressions, denoting the set of strings that can be obtained by concatenating a string described by R and a string described by S (in that order), the union of sets of strings described by R and S, and the set of all strings that can be obtained by concatenating any finite number (including zero) of strings from the set described by R, respectively. A string w belonging to the language defined by regular expression R is called positive or accepted by R, otherwise called negative or refused by R. For a given language, there exist many corresponding regular expressions that can describe the language.

2.2 Different dialects

Apart from the above standard regular operators, extensions have been added to regular expressions to enhance their ability to specify string patterns. For example, operators +, ?, {m,n} and && are used for specifying one or more repetition, one or zero repetition, repetition for at least m times and at most n times, interleaving or shuffling[16] of strings, respectively. Different applications may support different additional operators. For instance, XML schema language DTD permits only additional operators + and ?, while XSD[17] further supports operator {m,n}. The XML schema language Relax NG[18] even further allows the use of interleaving operator that specifies unordered concatenations of strings.

In particular, in the field of string pattern matching, more additional notions and more compact syntax are used to describe string patterns, such as the character class, character range, complement, and wildcard. Different regular expression pattern matching engines are not fully compatible with one another. The syntax and behavior of a particular regular expression engine may differ from the others. Although Perl Compatible Regular Expressions (PCRE)[19] and POSIX Regular Expressions[20] have greatly influenced the features of most regular expression engines, they have not been standardized yet. Thus a lot of regular expression dialects exist in practical uses. Some regular expression engines even contain non-regular operators, such as backreferences or lookaround assertions. For convenience, Table 1 illustrates the syntax and operators supported in most regular expression dialects.

Among the existing works on ensuring the correctness of regular expressions, some take the formal definition of regular expressions into account, while some consider various dialects. In this survey, we do not distinguish the differences of the target regular expressions. We classify and compare those works according to the topics on which they focus.

TABLE 1 Syntax of regular expressions

Name		Rule	Description
regexp	::=	unionexp	
unionexp	::=	interexp | unionexp	union
	|	interexp	
interexp	::=	concatexp & interexp	intersection
	|	concatexp	
concatexp	::=	repeatexp concatexp	concatenation
	|	repeatexp	
repeatexp	::=	repeatexp ?	zero or one occurrence
	|	repeatexp *	zero or more occurrences
	|	repeatexp +	one or more occurrences
	|	repeatexp {n}	n occurrences
	|	repeatexp {n,}	n or more occurrences
	|	repeatexp {n,m}	n to m occurrences
	|	complexp	
complexp	::=	~ complexp	complement
	|	charclassexp	
charclassexp	::=	[charclasses]	character class
	|	[^charclasses]	negated character class
	|	simpleexp	
charclasses	::=	charclass charclasses	
	|	charclass	
charclass	::=	charexp - charexp	character range
	|	charexp	
simpleexp	::=	charexp	
	|	.	any single character
	|	#	the empty language
	|	@	any string
	|	“<Unicode string without double-quotes>”	a string

	|	( )	the empty string
	|	(unionexp)	precedence override
charexp	::=	<Unicode character>	a single non-reserved character
	|	\w	word character
	|	\d	digit character
	|	\s	whitespace character
	|	\ <Unicode character>	a single character

2.3 Correctness

Simply speaking, a regular expression is correct if it does exactly what its designers and users intend it to do - no more and no less. The correctness involves two levels: syntax correctness and semantic correctness. Syntax errors can be checked by compilers. However, when regular expressions are embedded in source programs, the compilers usually treat them as literal strings and syntactical errors are reported only by throwing exceptions at run time. Thus some tools are developed to help developers to compose syntactically correct regular expressions. For examples, almost all regular expression editors provide syntax highlighting or syntax colouring.

The semantic correctness of a regular expression means that the language defined complies with the intended language, i.e., it meets the users' requirements. More specifically, a regular expression is semantically correct if it defines all the strings intended to be accepted and does not define any strings intended to be rejected. Ensuring the semantic correctness is difficult since the intended language is not easy to be formally specified. Different methods and approaches have been proposed to check semantic correctness of regular expressions or to assist the developers to write semantically correct regular expressions. For example, generating a set of strings from the regular expression to validate whether they conform to users' intention, or automatically learn a regular expression from a set of intended strings given by users. Almost all the works reviewed in this paper are devoted to ensuring the semantic correctness of regular expressions.

2.4 Automata

Formally, a non-deterministic finite automaton (NFA) is a 5-tuple $A = (\Sigma, Q, q_0, F, \delta)$, in which Σ is the alphabet, 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 that maps each pair of a state and a symbol to a set of states. A string $w = a_1a_2...a_n$ is accepted by an automaton A if and only if there exists a sequence of states $q_0q_1,...,q_n$ such that q_n is one of the final states, and $q_i \in \delta(q_{i-1},a_i)$ for each $i \in [1,n]$. A deterministic finite automaton (DFA) is a special case of NFA, in which the transition function δ maps each pair of a state and a symbol to a singleton set or the empty set Ø. If a regular expression contains only regular operators, then it can be equivalently converted to a NFA representation or a DFA representation. Some regular expressions support non-regular features such as backreferences and they may require more complex automaton representations.

3 Empirical study

To ensure the correctness and improve the quality of regular expressions, it is quite important and necessary to first investigate the practical issues which developers face with in regular expression usage, including, for instance, what kind of bugs mostly addressed by developers, what kind of operators/features mostly used in practical applications, what kind of representations are more understandable than others, and so on. Those works are usually conducted by empirical studies. In this section, we review the recent results on the empirical researches of regular expressions.

3.1 Testing status and bugs classification

Developers have reported that they test their regular expressions less than the rest of their code[8]. In [14], the authors further investigated how thoroughly tested regular expressions are by examining open source projects. They used test metrics for graph-based coverage[21] over the DFA representation of regular expressions, including node coverage, edge coverage, and edge-pair coverage. Their evaluation shows that only 16.8% of the regular expressions are executed by the test suites and among the tested regular expressions, half (41.9%) contain only one test string, and a majority (72.7%) use only positive input strings or only negative input strings. Edge coverage and edge-pair coverage are both very low (less than 30% on average). These results reveal that regular expressions are not well tested as expected in practice, which thus may be responsible for many software faults.

In a follow-up work[22], the authors presented an empirical study on regular expression bugs in real-world open-source projects. By analyzing a sample of merged pull request bugs related to regular expressions, they show that about half of the bugs are caused by incorrect regular expression behavior, and the remaining are caused by incorrect API usage and other code issues that require regular expression changes in the fix. Among the incorrect regular expression behaviors, about two third fall into the category of rejecting valid strings, one fifth accept invalid strings, and one tenth accept invalid strings and rejecting valid strings. These indicate that developers are more likely to write incorrect regular expressions that are too constrained than the regular expressions intended to use. The authors also observe that fixing regular expression bugs is nontrivial as it takes more time and more lines of codes to fix them compared with the general pull requests.

3.2 Feature usage and evolution

The authors of [8] studied how regular expressions are used in practice and what features of regular expressions are most commonly used in practice. Their findings show that regular expressions are frequently used to locate the content within a file, capture parts of strings, and parse user inputs. They have identified eight regular expression operators/features most commonly used in Python projects. These features include one-or-more repetition, group

(parenthesis), zero-or-more repetition, character range, and so on. In a follow-up work[23], the authors further explored how different features impact the readability, comprehension and understandability of regular expressions. Their study reveals that some features make regular expressions more readable and understandable, while some do not. For example, "\d" is semantically equivalent to "[0-9]". While "\d" is more succinct, "[0-9]" may be easier for developers to read and thus may help to avoid potential errors.

The work of [24] is devoted to regular expression evolution, studying the syntactic and semantic differences and feature changes of regular expressions over time. The main results include: most edited expressions have a syntactic distance of 4-6 characters from their old versions; over half of the edits tend to expand the scope of the expression, indicating that the old versions define languages smaller than the intended ones. These results can help to design mutation operators and repair operators to assist with the testing and fixing of regular expressions.

3.3 Composition, re-use and risks

Researchers of [25] conducted an exploratory case study to find how developers compose their regular expressions during the development of their projects. They find that a large majority of developers search online resources such as Q&A sites (e.g., stackoverflow.com, online forums), repositories or libraries (e.g., RegExLib[26]) during their problem solving tasks. This verifies that writing a correct regular expression is usually difficult, and developers may prefer to re-use or modify an existing expression rather than write it from scratch. Actually, an earlier study[27] has reported that the use of regular expressions is becoming highly repetitive although they are being used more and more often, and on the most popular websites gathered in their study only 4% of the regular expressions are unique. However, as noted in [25], syntactical errors often arise when participants copy/paste expressions from another language to their own projects.

Reference [28] pointed out that even no syntactical errors appear during regular expression re-using, the semantic and performance characteristic of expressions may no longer retain. The authors surveyed a number of professional developers to understand their regular expression re-use practices, and empirically measured the semantic and performance portability problem introduced by re-using of regular expressions. Their results show 15% exhibit semantic differences across languages and 10% exhibit performance differences across languages, although most regular expressions compile across language boundaries without syntactical errors. This implies that the re-use of regular expressions may introduce bugs. Semantics inconsistency means that the same regular expression may match different sets of strings across programming languages, resulting in logical errors. Performance inconsistency means that the same regular expression may have different worst-case performance behavior, resulting in Regular expression Denial of Service (ReDoS) vulnerabilities[9, 29]. Actually, the authors state that they have identified hundreds of source modules containing potential semantic bugs or potential security vulnerabilities.

To find out how developers actually work with regular expressions and what difficulties they face with, the authors of [30] provided a study of regular expression development cycle by interviewing a number of professional developers. In general, developers say that regular expressions are hard to read, hard to search for and hard to validate. Besides, most developers are also unaware of critical security risks that can occur when using regular expressions.

3.4 Determinism

A regular expressions is said deterministic if we always know definitely the next symbol we will match in the expression without looking ahead in the string, when we match a string from left to right against this expression[31]. For instance, "$(a|b)^*a$" is not deterministic as the first symbol in the string "aaa" could be matched by either the first or the second a in the expression. Without looking ahead, it is impossible to know which one to choose. The equivalent expression "$b^*a(b^*a)^*$", on the other hand, is deterministic. Deterministic regular expressions allow pattern matching more efficient than the general ones. Several decision problems also behave better for deterministic expressions. For example, language inclusion for general expressions is PSPACE-complete, but is tractable when the expressions are deterministic. Some applications require the determinism of regular expressions, while some do not. For example, W3C specification requires that the content models of XML schema language DTDs and XSDs[17] must be deterministic regular expressions, while there is no determinism restrictions on Relax NG[18] and regular expressions used for string pattern matching. The work of [32] is devoted to finding how deterministic real regular expressions are. The authors found that more than 98% of regular expressions in Relax NG are deterministic, although Relax NG does not have the determinism constraint for its content models. And more than half regular expressions from RegExLib are deterministic. These results indicate that deterministic regular expressions are commonly used in practice. Therefore exploring effective methods to ensure the quality of such expressions is worthy of attention.

3.6 Others

The authors of [34] performed an empirical study by comparing two different notations (textual and graphical) of regular expressions and by considering different factors such as the length of regular expressions to find how these factors affect the readability of regular expressions. They used the time required for finding shortest strings as the primary measurement in their experiments. Their findings show that the graphical notation of regular expressions is much more readable than the textual notation, and that the length has a strong effect on the regular expression's readability while the participants' background shows no measurable effect.

Different empirical researches may follow different methodologies to extract regular expressions, e.g., from different sources and written in only one or two programming languages. In [33], the authors tried to find whether existing empirical research results can be generalized. They report that significant differences exist in some characteristics by programming languages and suggest that empirical methodologies should consider the programming languages, as the generalizability is not always be assured for regular expressions supported in different programming languages.

4 Test string generation

Testing is a common way to ensure the correctness 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. The generated test strings can be positive or negative. If users find that a generated positive string should be rejected or a generated negative string should be accepted, then this indicates that the regular expression is incorrectly defined.

4.1 Coverage based generation

Coverage criteria are used to measure the quality of a particular test set and to provide strategies for test data generation algorithms. Coverage criteria are usually defined with respect to programs. In [34], a notion of pairwise coverage is proposed that is defined with respect to regular expressions. The idea adopts from pairwise testing[35], a combinatorial method for testing software systems. For each pair of input parameters to a software system, pairwise testing tries to test all possible combinations of these two parameters. Similarly, pairwise coverage for regular expressions tests all possible combinations of any two sub-expressions concatenated in the regular expression under testing. Furthermore, to avoid generate infinite strings, pairwise coverage restricts only three typical possibilities for Kleene star *: zero, one and more-than-one repetitions. Consider the following regular expression "$a^*b^*c^*$" for example. The string sets {ε, a, aaa}, {ε, b, bb} and {ε, c, cccc} cover the three typical repetitions for these sub-expressions "a^*", "b^*" and "c^*" respectively. Pairwise coverage criterion requires that all combinations of any two of these three string sets should be covered. For instance, the string "aaabbcccc" covers the combinations ("aaa", "bb"), ("aaa", "cccc") and ("bb", "cccc"). The string set achieving pairwise coverage for a regular expression is not necessarily unique. A string generation algorithm that given as an input a regular expression, outputs a small set of strings that satisfies the pairwise coverage criterion is implemented. The algorithm generates only positive strings. Besides, it considers the formal definition of regular expressions and supports only basic operators concatenation, alternation, Kleene star and two extended regular operators counting and interleaving. The algorithm can be used for testing content models of XML schemas, but may be not suitable to be used directly for testing regular expressions in other specialized applications such as string pattern matching where regular expressions have a very different syntax.

Egret[11] generates strings from regular expressions based on the underlying automata. It 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 this sense, we may say that the generated strings cover the basis paths of the specialized automaton. Egret focuses on regular expression patterns, that is, expressions used in manipulating text strings. Compared with the pairwise coverage based generation, it allows more regular operators such as the character class. We next use the regular expression "a?[2-9](b|c)\d{3}(e|f)" as an example to explain Egret's generation process. The generation is divided into three steps.

Step 1. Egret first coverts the expression into a specialized automaton, as shown in Fig. 1. In this automaton, transitions can be labeled by an individual symbol, a character set, or an epsilon. Special "begin repeat" and "end repeat" states are added to the beginning and end of each repeating operator, respectively.

Step 2. The specialized automaton is then traversed to obtain a set of basis paths, and for each basis path, an initial test string is generated. For the automaton in Fig. 1, we have the following basis paths and initial sets of strings.

basis paths	*initial strings*
path1: 0-...-6-7-8-11-...-16-17-18-21	a2bddde
Path2: 0-...-6-7-8-11-...-16-19-20-21	a2bdddf
Path31: 0-...-6-9-10-11-...-16-17-18-21	a2cddde

Step 3. This step creates additional strings from the initial set of strings. Two strategies are used for creating additional strings: (a) altering the number of iterations for each repeat operator and (b) changing the character used for a character set. For instance, consider the initial string "a2bddde" derived from path1. By replacing "a" with ε and "aa" (0 and 2 repetitions for "a?"), it obtains one positive string "2bddde" and one negative string "aa2bddde". By replacing "2" with "0" (one digit outside [2-9]), it obtains one negative string "a0bddde". By replacing "ddd" with "dd" and "dddd" (2 and 4 repetitions for {3}), it obtains two negative strings "a2bdde" and "a2bdddde".

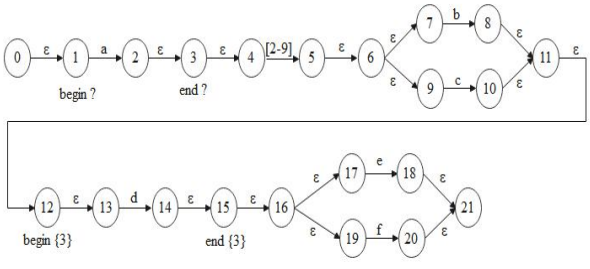

Fig. 1 Specialized automaton converted from the regular expression "a?[2-9](b|c)\d{3}(e|f)" during Egret's generation.

Due to different creation strategies of additional strings in step 3, Egret can produce not only positive test strings but also negative test strings. However, Egret's generation strategy especially for negative strings is relatively simple. For example, "$(a|b)^{+}$" generates only one negative string ε, and "$(a|b)^{*}$" generates no negative strings. Some faults may not be revealed for complex regular expressions.

4.2 Mutation based generation

Mutation testing is a fault-based technique to design new test data or to evaluate the quality of existing test data[36]. It involves introducing into the software artifact under test small changes, called mutations, which represent typical mistakes that developers could make. Each mutated version is called a mutant. 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. In [37], the authors applied mutation technique to regular expressions for test string generation. They identified a family of possible faults on regular expression definitions that represent the common mistakes programmers make when writing regular expressions. A set of mutants is created according to those faults, and then a set of critical strings that are able to distinguish the given regular expression from its mutants are generated. We next briefly describe this generation approach with an example.

One possible fault in defining regular expressions is that a programmer could have used the wrong repetition operator. Suppose that a programmer wants to define a regular expression accepting all the non-empty sequences of digits, (s)he could have wrongly written it as "$[0\text{-}9]^{*}$" instead of the correct one "$[0\text{-}9]^{+}$". According to this fault, the authors proposed the mutation operator Quantifier Change (QC) that mutates each simple repetition operator into another simple repetition operator, and for each user-defined operator {m}, creates a mutant in which m (and also n if the operator is {m,n}) is increased and a mutant in which it is decreased. For example, for regular expression "$[0\text{-}9]^{*}$", the mutation operator QC produces two mutants "$[0\text{-}9]^{+}$" and "[0-9]?". For regular expression "[0-9]{3}", QC generates mutants "[0-9]{2}", "[0-9]{4}", "[0-9]{3,}" and "[0-9]{0,3}".

Given a regular expression R and one of its mutant M, a string s is said to be able to distinguish R from M if s is accepted by R and not by M, or vice versa. That is, s is a string of the symmetric difference between R and M. For example, the empty string ε is distinguishing for regular expression "$[0\text{-}9]^{*}$" and its mutant "$[0\text{-}9]^{+}$". As long as the mutant is not equivalent to the original expression, one can always finds certain distinguishing strings. The string generation algorithm developed in [37] uses 14 mutation operators. For a regular expression R, the algorithm first mutates it to obtain a set of mutants, then for each non-equivalent mutant, it computes the symmetric difference of the automaton representation of the original regular expression and the mutant. If the symmetric difference is not empty (which means that the mutant is not equivalent), it randomly selects a distinguishing string from the symmetric difference set.

This mutation based generation could produce both the positive strings and negative strings. However, since the number of mutants generated after using the 14 mutation operators is very large, the generation process usually takes a lot of time and the generated string set may be big especially for long and complex regular expressions.

Some researchers adopted the technique of mutation testing to validate XML schemas[38, 39]. Since the content models of XML data are described by regular expressions, schema testing reduces to some extent the testing of expressions. Mutation operators applied to different XML schema components are proposed and mutation based generation of XML data are implemented. The generation algorithm usually produces only positive XML data.

4.3 Sampling and enumeration

In formal language theory, sampling and enumerating are two fundamental problems concerning the generation of regular languages. Sampling focuses on generating a uniformly random string of length n of a regular language so that strings of length n in that language all have the same probability to be generated[40, 41]. Enumerating tries to enumerate all the distinct strings or all strings of length n of a regular language in lexicographical order[42, 43]. However, most of the existing sampling and enumerating algorithms take finite automata or regular grammars as their inputs. In [44], the authors presented an enumeration based algorithm that takes regular expressions as direct input. These regular expressions can be extended with intersection and complement operators. The algorithm generates both positive and negative strings, which can be used to test regular expression parsers as well as to test regular expressions themselves.

There are many practical tools online available for generating strings from regular expressions, such as Xeger[45], Exrex[46], Generex[47] and Regldg[48]. These tools generate strings either randomly or systematically. For example, Xeger randomly generates test strings for a regular expression, and allows the users to specify how many strings to be generated. Exrex produces all matched strings for a given regular expression, and in case that the matched strings are infinite, users are asked to restrict the number of repetition of Kleene star operator or restrict the number of generated strings. All these tools generate only positive strings. The fault detection ability of the generated strings are not quite satisfactory compared with coverage based generation or mutation based generation methods as illustrated in [34] [11] and [49].

5 Learning

Since writing a regular expression for a specific task can be time consuming and error-prone, and require special skills and the familiarity with the formalism involved in constructing regular expressions, some researchers have working on synthesizing or learning regular expressions automatically, either from a set of sample strings or from a

natural language specification described by a human being.

5.1 Learning from examples

The problem of synthesizing regular languages from examples is a traditional topic in formal language theory. Prior works concentrated mostly on learning deterministic finite automata[50, 51]. Recent emerging researches begin to consider regular expressions as the learning target. The examples can contain positive strings or both positive and negative strings.

Reference [52] devised an automaton-based approach that learns regular expressions for information extraction from only positive samples. Reference [53] developed a system that automatically creates a regular expression from a set of input strings annotated by users, which parts of strings to be matched (positive) and which parts to be not matched (negative). The learned regular expressions are expected to generalize the matching behavior represented by examples. The authors have demonstrated that their system has similar performance in both time and accuracy with an experienced developer[54]. The system adopts the idea of genetic programming in the generation of regular expressions. Genetic programming is a heuristic technique searching for an optimal or at least suitable solution for a target problem. It starts with an initial set of candidate solutions built usually at random, and then repeatedly evolves by trying to build new candidate solutions from existing ones using genetic operators and meanwhile to discard worst candidate solutions. A problem-dependent function called *fitness* is defined in order to quantify the ability of each candidate solution to solve the target problem. The evolving procedure is repeated for a predefined number of times or until a satisfying solution is found, for example, a solution with a perfect fitness is found. The system developed in [53] adapted this framework to the specific problem of regular expression learning from examples. Each candidate regular expression is represented by an abstract syntax tree and two fitness functions (one concerning the length of the candidate expression and the other concerning the matching results of the candidate and the given examples) are defined to measure the quality of each candidate expression.

Along the same lines as [53], subsequent regular expression learning algorithms have been proposed for entity extraction[55] or text extraction[56]. Those genetic programming based learning techniques, although powerful, usually execute slowly. They may take a lot of minutes to obtain the synthesized results. Reference [57] proposed to rapidly infer a simplest regular expression over a binary alphabet from a set of positive and negative examples, which can then be interactively used by students to assist them studying and understanding regular expressions.

Another class of regular expression learning algorithms follow Gold's framework of learning (identification) in the limit[58], which is explained as follows. Let Γ be a subclass of regular expressions. Γ is said to be learnable or identifiable if there is an algorithm ρ mapping sets of example strings to expressions in Γ such that (1) S is a subset of $\rho(S)$ for every example set S and (2) to every regular expression R of Γ, we can associate a so-called characteristic sample S_c such that, for each example set S with $S_c \subseteq S$, $\rho(S)$ is equivalent to R. Intuitively, the first condition says that algorithm ρ must be sound; the second condition says that it must be complete, i.e. ρ should convergence when the sample contains enough data.

It was shown by Gold[58] that the class of all regular expressions is not learnable from only positive data. Therefore researchers have turned to identify subclasses of regular expressions that can be learnable, such as single occurrence regular expressions and chain regular expressions[59, 60], simple looping regular expressions[61]. The learning of such special expressions usually contains two steps: first constructs an automaton from the given example strings and then derives a target regular expression from the automaton. We next take the learning of single occurrence and chain regular expressions as example to explain the two steps.

A regular expression is called single occurrence if every alphabet symbol occurs at most once in it. If a single occurrence regular expression is of the form $f_1...f_n$ (n>1) where each f_i is a chain factor: an expression of the form $(a_1|..|.a_k)$, $(a_1|...|a_k)?$, $(a_1|...|a_k)^+$, or $(a_1|...|a_k)^+?$ with k>=1 and every a_i is an alphabet symbol, then this expression is called a chain regular expression. For example, "$(a|b)^+?c?(d|e|f)^+$" is a chain regular expression while "$(ab^+|c)?(d?|e|f^+)^+$" is not. Given a set of examples, the learning algorithm first constructs a single occurrence automaton using the classical 2T-INF algorithm[59]. A single occurrence automaton is a specific kind of DFA in which no edges are labeled and all states, except for the initial and final state, are symbol names. Fig. 2 shows the single occurrence automaton constructed from examples {aab, cdd}. The learning algorithm then transforms the automaton into a single occurrence regular expression using a set of graph based rewriting rules. If the target expression is a chain regular expression, the transformation adopts a different strategy. For example, the single occurrence regular expression derived from the automaton in Fig. 2 is "$(a^+b)|(cd^+)$", while the chain regular expression derived from this automaton is "$a^+?c?b?d^+?$".

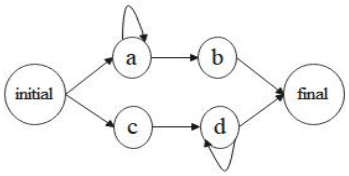

Fig. 2 Example of single occurrence automaton.

The algorithms following Gold's learning model can be theoretically proved sound and complete. Since the learned regular expressions are restricted, such subclasses learning algorithms are usually used in specific applications, such as the automatic inference of XML schemas.

5.2 Learning from specifications

Some researchers stated that although writing regular expressions is time consuming and error-prone, it is often much easier for users to specify their tasks or requirements in natural language. Therefore, automatic learning of regular expressions from natural language specifications is necessary and meaningful to help reducing possible errors caused by incorrect regular expressions. Approaches have been recently proposed for automatic generation of regular expression from specifications, by training a probabilistic parsing model from natural language sentences and then generating regular expressions from the model[62], or by using a sequence-to-sequence learning model to directly translate natural language sentences to regular expressions[63, 64]. For example, the authors of [63] created a corpus of regular expression and natural language pairs using the grammar rules such as "[0-9] → a number", "$(x)^*$ → x zero or more times", "$.^*x$ → ends with x" and "$.^*x.^*$ → contains x". They then applied the LSTM model to train this corpus to accomplish the natural text descriptions to regular expressions translation. However, as pointed out in [65], these approaches use only synthetic data in their training datasets and validation/test datasets, they may not be effective to handle real-world situations.

Taking into account that natural language specifications alone is often ambiguous and that examples in isolation are often not sufficient for conveying the user intent, reference [66] proposed a multi-modal synthesis technique for creating regular expressions from a combination of examples (including both positive and negative) and natural language specifications. The implemented tool produces top-k results that satisfy the examples and natural language descriptions, but it is still up to the user to check the results, and to provide more information if needed.

6 Static checking and verification

6.1 Syntax checking

Formal methods and verification are important to improve the quality of software. However, most of the static analysis tools only consider program codes, and do not check regular expressions. Moreover, compilers usually treat regular expressions in the source program as literal strings and syntactic errors of regular expressions are reported by throwing exceptions at run time. In [10], the authors designed a system that validates the regular expression syntax at compile time. In addition, the system checks the use of incorrect capturing group numbers that results in an exception error at run time. For example, the regular expression "$(a^*)([0\text{-}9]^+)$" contains two groups delimited by parentheses. If a program accesses the result matched by the third group, then an error occurs because there is no group three in this regular expression.

6.2 Semantic checking

Some researchers[67] focused on regular expressions used to specify the structures of elements and attributes in XML documents. The type system designed for such regular expressions aims at determining program's semantic errors, such as subtype checking[68, 69]. Subtype checking means that given a function for which the input and the output are specified by regular expressions R_1 and R_2 respectively, the type system statically verifies that for any input that matches R_1, the output of this function always matches R_2. This checking can detect type related errors in the function. If subtype checking fails, it might also indicate that the regular expressions specifying input and output types are not correctly defined.

ACRE[70] attempts to statically detect semantic errors of regular expressions used mainly in the area of string pattern matching. It performs eleven checks on regular expressions based on common mistakes for developing regular expressions. For example, it checks invalid ranges within a character set such as [A-z]. It is more likely that the correct range should be [A-Z] or [a-z]. For another example, it checks whether "braces" are balanced in the matching strings. This helps to detect the cases where strings with unbalanced braces are incorrectly accepted. Consider the syntactically correct regular expression "[{\(]?[0-9]{3}[\)}]?" for example. The tools report that a braces-unbalanced string "(000}" is accepted, which might not be expected.

6.3 Verification

A verification framework for regular expressions is proposed in [71]. The framework does not adopt the testing techniques to validate regular expressions. Instead, it allows users to express their expression requirements using natural language, which are then compiled into a kind of domain-specific formal specifications. It then checks the consistency between the formal specifications and the verified regular expressions using equivalence checking.

We use the motivating example from [71] to briefly explain how the verification framework works. Suppose that the requirement of the regular expression to be verified is informally stated using natural language, the valid string should contain a sub-string X such that before X there are only c's, after X there are only d's, in X, b's and a's alternate, and the first symbol in X is a and the last symbol is b. This requirement description is then further translated into the following formal specifications:

```
my@spec = (
  let X ← [a|b]*, // in X, b's and a's alternate
  X ← (? = a.*b) in e; // in X, begin in a end in b
  let e ← (? = c*X.*), // there are only c's before X
  e ← (? = .*Xd*) in S; // there are only d's after X
);
```

Finally, the framework converts this formal specification into an automaton, and conducts the equivalence checking on this specification automaton and the target regular expression converted automaton. If they are equivalent, then the target regular expression is considered correct; otherwise, the framework returns the "failed" information with counterexamples. For instance, if the verified expression is "$c^*(ab)^*d^*$" for the above example, the

framework reports "failed" with a counterexample "caabd", which is matched with the specifications but not the target regular expression. In such a way, incorrect regular expressions are detected. However, as pointed out in [71], the translation of formal specifications from natural language descriptions is not always accurate and still needs the users' interaction.

7 Visualization and abstraction

It is a common agreement that regular expressions, due to their complexity and compactness, introduce large challenges to the composition and comprehension for developers. Visual representations or explanations of the underlying structures of regular expressions can help to improve the readability and understandability and thus the quality of regular expressions. A lot of efforts have been put into this direction.

7.1 Highlighting

Highlighting the syntax of regular expressions is a traditional way of visualization that is supported in almost all text editors. Apart from this, some tools[72-74] provide debugging environments that can explain string matching results by highlighting the parts of regular expressions matching a certain string or highlighting the strings matched by regular expressions. This helps to check whether the matching results are expected.

7.2 Graphical representation

A regular expression can be transformed into an equivalent automaton, which in turn can be visualized as an diagram. Several tools[75-77] such as RegExpert[78] and Regexper[79] are developed to visualize regular expressions as their corresponding automaton-like graph representations. There are also some tools[74, 80] that provide additional tree views that show the hierarchical structures of the regular expression components. This enables developers to easily understand and track the structural and functional relationships among the sub-expressions or components contained in the regular expression. Reference [81] described a visual interface and a development environment for developing regular expressions that allows the users to construct complex regular expressions via a drag and drop visual interface. Reference [82] proposed to show the structures of regular expressions by augmenting the original textual representations with visual elements instead of using automaton-like graph representations or tree-view representations. The developed tool highlights the structure of a regular expression through horizontal lines at the top and bottom of the expression and discerns special-purpose tokens by color. For example, it highlights those repetition operators (*, ?, {m,n} and so on) in yellow color and highlights the union operator (|) in blue color. Groups in the regular expression are highlighted through horizontal lines attached to the bottom of the expression. The stacking of these lines reflects the nesting hierarchy of groups. Feedback from users show that such a visualization approach is "intuitive", "clear", "self explaining", "easy to understand", "helpful" and "useful".

7.3 Abstraction

The authors of [12] tried to establish an abstraction mechanism to serve as explanations of regular expressions. They identified and abstracted the common sub-expressions or components occurred in a regular expression, then introduced names for those common sub-expressions, and finally obtained a representation that directly reveals the overall structure of the original regular expression. As an example considering the regular expression (taken from [12]) "<\s*[aA]\s+[hH][rR][eE][fF]=f\s*>\s*\s*[iI][mM][gG]\s+[sS][rR][cC]=f\s*>[^<>]*<\s*/[iI][mM][gG]\s*>\s*<\s*/[aA]\s*>". It is abstracted as the following representation: "< a href =f > < img src=f >[^<>]*< /img >< /a >", which makes it much more clearer what kind of strings the original regular expression describes.

8 Repairing

Repair of regular expressions are usually done by the synthesis from examples. That is, if a regular expression is incorrect, then a new set of examples is provided from the correct expression that is consistent with the given examples is synthesized or learned. The work of [83] aimed at repairing regular expressions that define languages larger than the intended ones. Therefore, to repair a faulty regular expression, a new set of negative examples are required and the goal is to modify the original expression so that it rejects the new examples. On the contrary, the work of [84] focused on repairing regular expressions that define languages smaller than the intended ones. Thus a new set of positive examples is provided, and the goal is to modify the original expression so that it accepts the new examples. The repair processes of these two works use a set of heuristics to transform an initial regular expression into a modified one that accepts/rejects the new examples. The transformation rules include: remove a disjunct from a union, for example, "a|b|c" to "a|c", or restrict the repetition range of a repetition operator, for example, "a{1,3}" to "a{1,2}", and so on. However, these repairs do not provide any minimality guarantees and may produce regular expressions that are very different from the original ones.

Reference [85] repaired regular expressions with both positive and negative examples, and it guaranteed to find the syntactically smallest repair of the original regular expression. Here, "smallest" is measured by the edit distance between the abstract syntax trees of the initial regular expression and the target regular expression. The repair algorithm first generates a set of initial templates based on the initial regular expression and the given positive and negative examples. It then processes the templates by discarding templates that can not result in a correct repair, generating new templates based on the current ones, until an optimal regular expression is obtained. Some works have been conducted to study how to repair the ReDoS vulnerable regular expressions[86, 87].

Reference [88] devised an evolutionary approach to testing and repairing regular expressions. The approach starts from an initial guess of the regular expression, then it repeatedly generates meaningful strings to check whether they are accepted or not, and tries to repair consistently the desired solution. Reference [89] proposed a genetic programming approach to repairing regular expressions. It uses genetic programming operators over the DFA representation of the regular expression, and then the obtained DFA is converted back to a regular expression. The conversion from DFAs could yield expressions that are completely different from the original ones.

Reference [90] presented an iterative mutation-based process for testing and repairing regular expressions. For a regular expression, the approach generates a set of strings that distinguishes the regular expression from its mutants, and asks the users to assess the correct evaluation of these strings. If a mutant evaluates these strings more correctly than the original regular expression, then it substitutes the faulty expression with this mutant. This process iterates until no mutants better than the original expression are found. The repair process, however, requires a lot of users' efforts as they are frequently asked to assess the correctness of the strings evaluation or to judge whether a mutant is better than the original expression.

9 Conclusions

Regular expressions are widely used in different fields within and even outside of computer science. Ensuring the correctness of regular expressions is a vital prerequisite for their usage in practical applications. Efforts have been made in recent years to assist users or programmers to write correct regular expressions or to validate the correctness of regular expressions. In this paper, we have conducted a review around this topic. In particular, we have classified existing relevant works into six categories, including empirical study on various problems in the development of regular expressions, test string generation from regular expressions, automatic learning or synthesis of regular expressions from example strings or specifications, statically checking the syntax or specific semantic errors in regular expressions or verifying regular expressions with specifications, visual representations or explanations of the underlying structures of regular expressions, and repairing of faulty regular expressions. For each category, we have reviewed different approaches and discussed their advantages and disadvantages. Table 2 provides an overview of our classification.

The importance of ensuring the correctness of regular expressions is just beginning to be addressed, and the current research progresses on this topic are far from enough. There are still a few research problems waiting for new solutions and tools, and we list some in the following.

(1) Generation of negative strings. Generating test strings is a common yet effective way to discover errors in regular expressions. Empirical studies show that a majority of faulty regular expressions define languages that are too constrained, i.e., they reject valid strings. Such kind of faults cannot be detected by positive strings generated from the incorrect expressions. Therefore, how to generate meaningful negative strings that are outside of the regular expression languages is an importance problem to study.

(2) Refactoring of regular expressions. The lack of readability is usually a pain point for composing and re-using high quality regular expressions. Thus, refactoring transformations are needed to enhance the readability or comprehension of regular expressions. For example, \d is semantically equivalent to [0123456789] and [0-9]. While \d is more succinct, [0-9] may be easier to read.

(3) Fault location and automatic repair. Fault detection and diagnose is in general a challenging problem[91]. Yet, even though a regular expression is detected incorrectly defined, locating the faulty areas and further repairing them are even more involved, as practical expressions are usually large and have complex structures. Existing works on fault location and automatic repair of regular expressions are relatively insufficient. More attentions are needed to be paid to these issues.

Table 2 Classification of recent works on ensuring correctness of regular expressions

Empirical Study	Test String Generation	Learning and Systhesis	Static Checking and Verification	Visualization and Abstraction	Repairing
testing status[14]	RE coverage based[34]	from positive examples[52]	syntax check[10]	highlight[72-74]	based on positive examples [83]
bug classification[22]	DFA coverage based[11]	from both examples[53-57]	semantic check[68,69,70]	automaton graph[75-79]	based on negative examples[84]
feature usage[8]	mutation based[49]	Gold's learning[59-61]	verification[71]	tree view[74,80]	based on both examples[85]
evolution[24]	sampling[40, 41]	from specifications[62-64]		visual IDE[81]	based on genetic programming[88, 89]
comprehension[23]	enumerating[91]	from specifications and examples[66]		visual structure[82]	Interactive repair[90]
composition[25]	random[45, 47]			abstraction[12]	repair vulnerable[86, 87]
portability[28]	systematic[46-48]				

development cycle[30]
determinism[32]
readability[92]

Acknowledgements

This work was supported by the National Natural Science Foundation of China (61872339 & 61502184 & 61925203).

References

[1] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman. *Compilers: principles, techniques and tools, 2nd edition*: Pearson Addison Wesley, 2007.

[2] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda and S. S. S. Anna. Automatic Network Protocol Analysis. In *Proceedings of the Network and Distributed System Security Symposium*, pp. 1-14, 2008.

[3] A. Yeole and B. Meshram. Analysis of different technique for detection of SQL injection. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, pp. 963-966, 2011.

[4] The bro network security monitor, 2015.

[5] M. Murata, D. Lee, M. Mani and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, vol. 5, no. 4, pp. 660-704, 2005.

[6] F. Alkhateeb, J. F. Baget and J. Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *Journal of Web Semantics*, vol. 7, no. 2, pp. 57-73, 2009.

[7] A. N. Arslan. Multiple sequence alignment containing a sequence of regular expressions. In *Proceedings of the IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, pp. 1-7, 2005.

[8] C. Chapman and K. T. Stolee. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 282-293, 2016.

[9] J. C. Davis, C. A. Coghlan, F. Servant and D. Lee. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 26th ACM joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 246-256, 2018.

[10] E. Spishak, W. Dietl and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pp. 20-26, 2012.

[11] E. Larson and A. Kirk. Generating evil test strings for regular expressions. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pp. 309-319, 2016.

[12] M. Erwig and R. Gopinath. Explanations for regular expressions. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pp. 394-408, 2012.

[13] P. Klint, R. Lammel and C. Verhoef. Towards an engineering discipline for grammarware. *ACM Transaction on Software Engineering and Methodology*, vol. 14, no. 3, pp. 331-380, 2005.

[14] P. Wang and K. T. Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 668-678, 2018.

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

[16] W. Gelade. Succinctness of regular expressions with interleaving, intersection and counting. *Theoretical Computer Science*, vol. 411, no. 31-33, pp. 2987-2998, 2010.

[17] H. S. Thompson, D. Beech, M. Maloney and N. Mendelsohn. *XML Schema part 1: structures. Second edition*.

[18] J. Clark and M. Makoto. *Relax NG Tutorial*.

[19] P. Hazel. Pcre-perl compatible regular expressions, 2005.

[20] IEEE and The Open Group. The open group base specifications issue 7, 2018.

[21] P. Ammann and J. Offutt. *Introduction to software testing*: Cambridge University Press, 2016.

[22] P. Wang, C. Brown, J. Jennings and S. Kathryn. An empirical study on regular expression bugs. In *Proceedings of the International Conference on Mining S*oftware Repositories, pp. 103-113, 2020.

[23] C. Chapman, P. Wang and K. T. Stolee. Exploring regular expression comprehension. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pp. 405-416, 2017.

[24] P. Wang, G. R. Bai and K. T. Stolee. Exploring regular expression evolution. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, pp. 502-513, 2019.

[25] G. R. Bai, B. Clee, N. Shrestha, C. Chapman, C. Wright and K. T. Stolee. Exploring tools and strategies used during regular expression composition tasks. In *Proceedings of the 27th International Conference on Program Comprehension*, pp. 197-208, 2019.

[26] Regular Expression Library. https://www.regexlib.com.

[27] R. Hodován, Z. Herczeg and Á. Kiss. Regular expressions on the web. In *Proceedings of the International Symposium on Web Systems Evolution*, pp. 29-32, 2010.

[28] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant and D. Lee. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 443-454, 2019.

[29] J. Kirrage, A. Rathnayake and H. Thielecke. Static analysis for regular expression denial-of-service attacks. In *Proceedings of the International Conference on Network and System Security*, pp. 135-148, 2013.

[30] L. G. Michael, J. Donohue, J. C. Davis, D. Lee and F. Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *Proceedings of the International Conference on Automated Software Engineering*, pp. 415-426, 2019.

[31] A. Brüggemann-Klein, D. Wood. One-unambiguous regular languages. *Information and Computation*, vol. 140, no. 2, pp. 229-253, 1998.

[32] Y. Li, X. Chu, X. Mou, C. Dong and H. Chen. Practical study of deterministic regular expressions from large-scale XML and schema data. In *Proceedings of the 22nd International Database Engineering & Applications Symposium*, pp. 45-53, 2018.

[33] J. C. Davis, D. Moyer, A. M. Kazerouni and D. Lee. Testing regex generalizability and its implications: A large-scale many-language measurement study. In *Proceedings of the 34th IEEE International Conference on Automated Software Engineering*, pp. 427-439, 2019.

[34] L. Zheng, S. Ma, Y. Wang and G. Lin. String generation for testing regular expressions. *The Computer Journal,* vol. 63, no. 1, pp. 41-65, 2020.

[35] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, vol. 43, no. 2, pp. 11:1-11:29, 2011.

[36] R. A. Demillo, R. J. Lipton and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, vol. 11, no. 4, pp. 34-41, 1978.

[37] P. Arcaini, A. Gargantini and E. Riccobene. Mutrex: A mutation-based generator of fault detecting strings for regular expressions. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 87-96, 2017.

[38] M. C. F. P. Emer, I. F. Nazar, S. R. Vergilio and M. Jino. Fault-based test of XML schemas. *Computing and Informatics*, vol. 30, no. 3, pp. 531-557, 2012.

[39] J. B. Li and J. Miller. Testing the Semantics of W3C XML Schema. In *Proceedings of the 29th Annual International Coference on Computer Software and Applications* pp. 443-448, 2005.

[40] S. Kannan, Z. Sweedyk and S. Mahaney. Counting and random generation of strings in regular languages. In *Proceedings of the 6th annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 551-557, 1995.

[41] O. Bernardi and O. Giménez. A linear algorithm for the random sampling from regular languages. *Algorithmica*, vol. 62, no. 1-2, pp. 130-145, 2012.

[42] M. Ackerman and J. Shallit. Efficient enumeration of regular languages. In *Proceedings of the International Conference on Implementation and Application of Automata*, pp. 226-242, 2007.

[43] M. Ackerman and J. Shallit. Efficient enumeration of words in regular languages. *Theoretical Computer Science*, vol. 410, no. 37, pp. 3461-3470, 2009.

[44] G. Radanne and P. Thiemann. Regenerate: a language generator for extended regular expressions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 202-214, 2018.

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

[46] https://github.com/asciimoo/exrex.

[47] https://github.com/mifmif/generex.

[48] https://regldg.com/.

[49] P. Arcaini, A. Gargantini and E. Riccobene. Fault-based test generation for regular expressions by mutation. *Software Testing, Verification and Reliability*, vol. 29, no. 1-2, pp. e1664, 2019.

[50] E. M. Gold. Complexity of automaton identification from given data. *Information and Control,* vol. 37, no. 3, pp. 302-320, 1978.

[51] J. Oncina and P. Garcia. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition*: *World Scientific*, pp. 99-108, 1993.

[52] F. Brauer, R. Rieger, A. Mocan and W. M. Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, pp. 1285-1294, 2011.

[53] A. Bartoli, G. Davanzo, A. De Lorenzo, E. Medvet and E. Sorio. Automatic synthesis of regular expressions from examples. *IEEE Computer*, vol. 47, no. 12, pp. 72-80, 2014.

[54] A. Bartoli, A. De Lorenzo, E. Medvet and F. Tarlao. Can a machine replace humans in building regular expressions? A case study. *IEEE Intelligent Systems*, vol. 31, no. 6, pp. 15-21, 2016.

[55] A. Bartoli, A. De Lorenzo, E. Medvet and F. Tarlao. Active learning of regular expressions for entity extraction. *IEEE Transactions on Cybernetics*, vol. 48, no. 3, pp. 1067-1080, 2017.

[56] A. Bartoli, A. De Lorenzo, E. Medvet and F. Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge Data Engineering*, vol. 28, no. 5, pp. 1217-1230, 2016.

[57] M. Lee, S. So and H. Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 70-80, 2016.

[58] E. M. Gold. Language identification in the limit. *Information and Control,* vol. 10, no. 5, pp. 447 - 474, 1967.

[59] G. J. Bex, F. Neven, T. Schwentick and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems*, vol. 35, no. 2, pp. 1-47, 2010.

[60] D. D. Freydenberger and T. Kötzing. Fast learning of restricted regular expressions and DTDs. *Theory of Computing Systems*, vol. 57, no. 4, pp. 1114-1158, 2015.

[61] H. Fernau. Algorithms for learning regular expressions from positive data. *Information and Computation*, vol. 207, no. 4, pp. 521-541, 2009.

[62] N. Kushman and R. Barzilay. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 826-836, 2013.

[63] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman and R. Barzilay. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *Proceedings of the Conference on Empirical Methods in Natural Language*, pp. 1918-1923, 2016.

[64] Z. Zhong et al. SemRegex: A semantics-based approach for generating regular expressions from natural language specifications. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 1608–1618, 2018.

[65] Z. Zhong et al. Generating regular expressions from natural language specifications: Are we there yet? In *Proceedings of the AAAI Workshops*, pp. 791-794, 2018.

[66] Q. Chen, X. Wang, X. Ye, G. Durrett and I. Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 487-502, 2020.

[67] H. Hosoya and B. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, vol. 3, no. 2, pp. 117-148, 2003.

[68] G. Castagna, D. Colazzo and A. Frisch. Error mining for regular expression patterns. In *Proceedings of the Italian Conference on Theoretical Computer Science*, pp. 160-172, 2005.

[69] V. Benzaken, G. Castagna and A. Frisch. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices,* vol. 38, no. 9, pp. 51-63, 2003.

[70] E. Larson. Automatic Checking of Regular Expressions. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 225-234, 2018.

[71] X. Liu, Y. Jiang and D. Wu. A lightweight framework for regular expression verification. In *Proceedings of the IEEE International Symposium on High Assurance Systems Engineering*, pp. 1-8, 2019.

[72] https://regexr.com/.

[73] https://regex101.com/.

[74] http://www.regexbuddy.com/.

[75] I. Budiselic, S. Srbljic and M. Popovic. RegExpert: A tool for visualization of regular expressions. In *Proceedings of the International Conference on "Computer as a Tool"*, pp. 2387-2389, 2007.

[76] https://regexper.com/.

[77] https://rise4fun.com/rex/.

[78] B. Braune, S. Diehl, A. Kerren and R. Wilhelm. Animation of the Generation and Computation of Finite Automata for Learning Software. In *Proceedings of the 4th International Workshop on Implementing Automata*, pp. 39-47, 2001.

[79] T. Hung and S. H. Rodger. Rodger. Increasing visualization and interaction in the automata theory course. *ACM SIGCSE Bulletin*, vol. 32, no. 1, pp. 6-10, 2000.

[80] A. Blackwell. See what you need: Helping end-users to build abstractions. *Journal of Visual Languages Computing*, vol. 12, no. 5, pp. 475-499, 2001.

[81] K. Oflazer and Y. Yılmaz. Vi-xfst: a visual regular expression development environment for Xerox finite state tool. In *Proceedings of the 7th Meeting of the ACL Special Interest Group in Computational Phonology: Current Themes in Computational Phonology and Morphology*, pp. 86-93, 2004.

[82] F. Beck, S. Gulan, B. Biegel, S. Baltes and D. Weiskopf. Regviz: Visual debugging of regular expressions. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 504-507, 2014.

[83] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan and H. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 21-30, 2008.

[84] T. Rebele, K. Tzompanaki and F. M. Suchanek. Adding missing words to regular expressions. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 67-79, 2018.

[85] R. Pan, Q. Hu, G. Xu and L. D'Antoni. Automatic repair of regular expressions. *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 139:1-139:29, 2019.

[86] Y. Li et al. FlashRegex: Deducing Anti-ReDoS Regexes from Examples. In *Proceedings of the IEEE International Conference on Automated Software Engineering,* pp. 659-671, 2020.

[87] N. Chida and T. Terauchi. Automatic Repair of Vulnerable Regular Expressions. *preprint arXiv:.12450*, 2020.

[88] P. Arcaini, A. Gargantini and E. Riccobene. Regular Expression Learning with Evolutionary Testing and Repair. In *Proceedings of the IFIP International Conference on Testing Software and Systems*, pp. 22-40, 2019.

[89] R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar and M. Veanes. Program boosting: Program synthesis via crowd-sourcing. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 677-688, 2015.

[90] P. Arcaini, A. Gargantini and E. Riccobene. Interactive testing and repairing of regular expressions. In *Proceedings of the IFIP International Conference on Testing Software and Systems*, pp. 1-16, 2018.

[91] X. Qiu, Y. Hu and B. Li. Sequential Fault diagnosis using an inertial velocity differential evolution algorithm. *International Journal of Automation and Computing*, vol.16, no. 3, pp. 389-397, 2019.

Lixiao Zheng received the Ph.D. degree from Institute of Software, Chinese Academy of Sciences in 2012, and B.Sc. degree from Jilin University in 2006, respectively. Currently, she is an associate professor at the College of Computer Science and Technology, Huaqiao University, China. Her research interests include formal language theory and software testing.

E-mail: zhenglx@hqu.edu.cn

ORCID iD: 0000-0002-9146-5465

Shuai Ma received his Ph.D. degrees from University of Edinburgh in 2010 and from Peking University in 2004, respectively. Currently, he is a professor at the School of Computer Science and Engineering, Beihang University, China. He obtained He is a recipient of the best paper award for VLDB 2010 and best paper candidate for ICDM 2019. He is an Associate Editor of VLDB Journal since 2017, IEEE Transactions On Big Data Since 2020 and Knowledge and Information Systems since 2020. His current research interests include database theory and systems, and big data.

E-mail: mashuai@buaa.edu.cn (Corresponding author)

ORCID iD: 0000-0002-4050-0443

Zuxi Chen received the Ph.D. degree from Tongji University in 2016. Currently, he is a lecturer at the College of Computer Science an d Technology, Huaqiao University, China. His research interests include formal methods and software safety.

E-mail: zuxichen@hqu.edu.cn

Xiangyu Luo received the Ph.D. degree in computer science from Sun Yat-sen University, in 2006. He is currently an Associate Professor with the College of Computer Science and Technology, Huaqiao University. His main research interests include model checking, multi-agent systems, temporal logics, and epistemic logics. He has been involved in the national project of model checking for multi-agent systems.

E-mail: luoxy@hqu.edu.cn