Deducing Matching Strings for Real-World Regular Expressions

Yixuan Yan[1,2], Weihao Su[1,2], Lixiao Zheng[3], Mengxi Wang[1,2], Haiming Chen[1,2(✉)], Chengyao Peng[1,2], Rongchen Li[1,2], and Zixuan Chen[1,2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{yanyx,suwh,wangmx,chm,pengcy,lirc,chenzx}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing 101400, China
[3] College of Computer Science and Technology, Huaqiao University, Xiamen, China

Abstract. Real-world regular expressions (regexes for short) have a wide range of applications in software. However, the support for regexes in test generation is insufficient. For example, existing works lack support for some important features such as lookbehind, are not resilient to subtle semantic differences (such as partial/full matching), fall short of Unicode support, leading to loss of test coverage or missed bugs. To address these challenges, in this paper, we propose a novel semantic model for comprehensively modeling the extended features in regexes, with an awareness of different matching semantics (i.e. partial/full matching) and matching precedence (i.e. greedy/lazy matching). To the best of our knowledge, this is the first attempt to consider partial/full matching semantics in modeling and to support lookbehind. Leveraging this model we then develop `PowerGen`, a tool for deducing matching strings for regexes, which randomly generates matching strings from the input regex effectively. We evaluate `PowerGen` against nine related state-of-the-art tools. The evaluation results show the high effectiveness and efficiency of `PowerGen`.

Keywords: regex · semantics · modeling · generation · matching string

1 Introduction

As a versatile mechanism for pattern matching, searching, substituting, and so on, real-world regular expressions (regexes for short) have become an integral part of modern programming languages and software development, with numerous applications across various fields [3,13,18,19,31,43]. Previous research [12,18,53] has reported that regexes are utilized in 30–40% of Java, JavaScript, and Python software.

Though popular, regexes can be difficult to comprehend and construct even for proficient programmers, and error-prone, due to the intrinsic complexities

Y. Yan and W. Su—These authors contributed equally.
Zixuan Chen is currently employed at Kuaishou Technology.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
H. Hermanns et al. (Eds.): SETTA 2023, LNCS 14464, pp. 331–350, 2024.
https://doi.org/10.1007/978-981-99-8664-4_19

of the syntax and semantics involved, resulting in tens of thousands of bug reports [13,27,30,31,44]. Therefore, it is crucial to offer automated techniques for test generation and bug finding within regexes. Producing matching strings for regexes is essential for many tasks, such as automated testing, verifying, and validating programs that utilize regexes. There have been numerous studies related to this problem using various techniques [14,25,26,28,29,42,50,52]. However, there are crucial concerns that have been either overlooked or inadequately addressed in the existing literature, limiting their utility. We classify these issues as follows.

Features Support. All existing works lack support for some important features. For example, none of the existing works support lookbehind, and only one work supports lookahead but with soundness errors (see Sect. 2.3). Regexes are featured with various extended features (or simply called features) such as lookarounds, capturing groups, and backreferences. An instance of a regex using a backreference is $(.^*)\backslash 1$, which defines a context-sensitive language $\{ww|w\in\Sigma^*\}$ where $\backslash 1$ is a backreference. In addition, if such an expression is also included in lookarounds, then those lookarounds effectively encode the intersection of context-sensitive languages [15]. These show that regexes are not limited to representing regular languages [1], and as a result, generating strings for regexes becomes more involved. For instance, in [15], the authors demonstrated that the emptiness problem of regular expressions with lookaheads and backreferences is undecidable. Thus, in many works, these features are often disregarded or imprecisely estimated. This lack of support can lead to, for instance, poor coverage or unrevealed bugs. Furthermore, based on our analysis of open-source projects for six programming languages (Java, Python, JavaScript, C#, PHP, Perl) which yielded 955,184 unique regexes, the average ratio of capturing group usage exceeds 38%, while the average percentage of lookarounds and backreferences is over 4%, while it approaches 10% in C#, thus those features are non-negligible. Similar observations for JavaScript were also reported by [29].

Subtle Semantic Differences. Regexes have different semantics which can result in different matching results. For example, there are partial and full matching functions for the regexes in most programming languages, which can lead to different matching results. For instance, the regex $node_modules(\texttt{?=}paradigm.^*)$ from practical project [19] matches $node_modulesparadigm$ under a partial matching call, but is unsatisfiable under a full matching call. None of the existing works addressed the different matching semantics of regexes (such as partial/full matching), thus may lead to wrong results. As another example, backreference has different semantics in different programming languages. For instance, the regex $(\texttt{?:}(a)b)^?\backslash 1c$ can match c in JavaScript, but does not match c in Java, Python, PHP and C#. See more examples in Sect. 2.3.

Unicode Support. Supporting the Unicode Standard can be useful in the internationalization and localization of practical software. PCRE and POSIX standards for regexes defined several operators such as $\backslash uFFFF$, $[\texttt{:}word\texttt{:}]$ and $\backslash p\{L\}$ to represent Unicode code points, improving the usability of regexes. In

modern mainstream regex engines and string constraint solvers [17,24,45], those operators are common. However, we found the existing tools show incomplete support for those features.

Various Kinds of Soundness Errors. We also found incorrect outputs generated by existing works, reflecting logic errors in their implementation which may be due to the intricacy of the syntax and semantics of regexes. See Sect. 2.3 for details.

To achieve the end, this paper proposes a novel semantic model for comprehensively modeling the extended features in regexes with the awareness of different matching semantics (i.e. partial/full matching) and matching precedence (i.e. greedy/lazy matching). Leveraging this model we then develop `PowerGen`, a tool for deducing matching strings for regexes. Specifically, `PowerGen` first rewrites the input regex by selecting the appropriate optimization rule based on the input programming language and rewrites the input regex based on the information of partial/full matching function. Then it uses Unicode automata to support a vast class of extended Unicode-related features. Next `PowerGen` selects the appropriate induction rules based on the input programming language to perform the top-down induction of the sub-expressions of the rewritten regex. Finally, `PowerGen` randomly generates matching strings according to the induction rules and stack compiled from the rewritten regex, which effectively identifies unsatisfiable cases.

We evaluate `PowerGen` by comparing `PowerGen` against nine state-of-the-art tools on publicly available datasets. Our evaluation demonstrates the high effectiveness and efficiency of `PowerGen`.

The contributions of this paper are listed as follows.

- We propose a novel semantic model for regexes, which comprehensively models the extended features, with the awareness of different matching semantics and matching precedence. To the best of our knowledge, it is the first one to consider partial/full matching semantics in modeling and supporting lookbehind.
- Based on our model, we develop `PowerGen`, a tool for deducing matching strings for regexes. To this end, we devise novel algorithms that randomly generate matching strings according to the input regex, which effectively identifies unsatisfiable cases.
- Evaluation shows the high effectiveness and efficiency of `PowerGen`.

2 Background

2.1 Regex

Let Σ be a finite alphabet of symbols and Σ^* be the set of all possible words (i.e. strings) over Σ, ε denotes the empty word and the empty set is denoted by $\varnothing$. For the definition of standard regular expressions we refer to [55].

Table 1. The Results from Each Tool for Examples in Practical Projects

No.	Regex	Egret	dk.brics	Mutrex	Generex	Exrex	Xeger (O'Connor)	Randexp.js	ExpoSE	Ostrich
#1	$\hat{}<(\backslash S^{+?}@@)>$	<@@>(✓)	–	–	–	<@>	<QrpwE^bd@>(✓)	–	$<U+00E7@>$(✓)	<@>
#2	$\backslash boldgnu\backslash b.^{*}\backslash bformat\backslash b$	oldgnu format(✓)	–	–	–	...8Uformat	oldgnuX...	oldgnuB...	*error*	oldgnuformat
#3	$(a)(b)(c)(d)(e)(f)(g)(h)(i)(j)(k)\backslash 11$	abcdefghijkk(✓)	–	–	–	abcdefghijkk(✓)	abcdefghijkk(✓)	abcdefghijk	*unsat*	–
#4	$(?<foo>xyz)(?<bar>\backslash d^{+})abc\backslash k<bar>$	xyz0abc0(✓)	–	–	–	xyz2375022abc2375022(✓)	xyz14369195abc14369195(✓)	*error*	*error*	–
#5	$\hat{}(?:(a\backslash 2)(b))^{+}\$$	–	–	–	–	–	–	ababbabbab...	ab(✓)	–
#6	$(?:()^{\{0\}}(?:(?:)^{\{0\}})\backslash 1)\vert(?:()(?:(?:))\backslash 2)$	*error*	–	–	–	*error*	*error*	ϵ(✓)	*unsat*	–
#7	$\hat{}(?:(d)(a\backslash 2))^{+}\$$	*error*	–	–	–	*error*	*error*	*dadaadaaa...*	*error*	–
#8	$(a)(?=\backslash 1)\backslash w^{+}$	–	–	–	–	–	–	–	*unsat*	–

In practice, real-world regular expressions (regexes) are pervasive in diverse application scenarios. A regex over Σ is a well-formed parenthesized formula constructed by, besides using the constructs for standard regular expressions and character classes, as well as using the following operators (i) capturing group (E); (ii) named capturing group $(<name>E)$; (iii) non-capturing group $(?:E)$; (iv) lookarounds: positive lookahead $(?=E_1)E_2$, negative lookahead $(?!E_1)E_2$, positive lookbehind $E_1(?<=E_2)$, and negative lookbehind $E_1(?<!E_2)$; (v) anchors: word boundary $\backslash b$, non-word boundary $\backslash B$, start-of-line anchor ^, and end-of-line anchor \$; (vi) greedy quantifiers: $E^?$, E^*, E^+, and $E^{\{m,n\}}$; (vi) lazy quantifiers: $E^{??}$, $E^{*?}$, $E^{+?}$, and $E^{\{m,n\}?}$; (vii) backreference $\backslash i$ and (viii) named backreference $\backslash k<name>$, etc.

In addition, $E?$, E^*, E^+ and $E^{\{i\}}$ where $i \in \mathbb{N}$ are abbreviations of $E^{\{0,1\}}$, $E^{\{0,\infty\}}$, $E^{\{1,\infty\}}$ and $E^{\{i,i\}}$, respectively. $E_1^{\{m,\infty\}}$ is often simplified as $E_1^{\{m,\}}$. Following symbols (,), {, }, [,], ^, \$, |, \, ., ?, * and + are escaped with the escape character \ in Σ. The *language* $L(E)$ of a regex E is the set of all strings accepted by E.

2.2 Research Problem

In this paper, we focus on the research problem of finding matching strings which depends on the partial/full matching semantics in regexes. We present related concepts below.

In most programming languages there are partial and full matching functions for the regexes (e.g. the `matches` and `find` functions in Java for full matching respectively partial matching). For a regex E, if it is used with the full matching function, then a string w is a **matching** string of E if $w \in L(E)$. If E is used with the partial matching function, then a string w is a **matching** string of E if $w \in L(.^*E.^*)$.

2.3 The Current Status of Existing String Generation Tools

We identified 9 state-of-the-art string generation tools for comparison which can be categorized into three groups: (1) string generation based on automata, including Egret [25], dk.brics [35], Mutrex [2] and Generex [54]; (2) string generation based on AST (Abstract Syntax Tree), including Exrex [49], Xeger (O'Connor) [37] and Randexp.js [22]; (3) string generation based on SMT (Satisfiability Modulo Theories) solvers, including ExpoSE [29], and Ostrich [14]. It

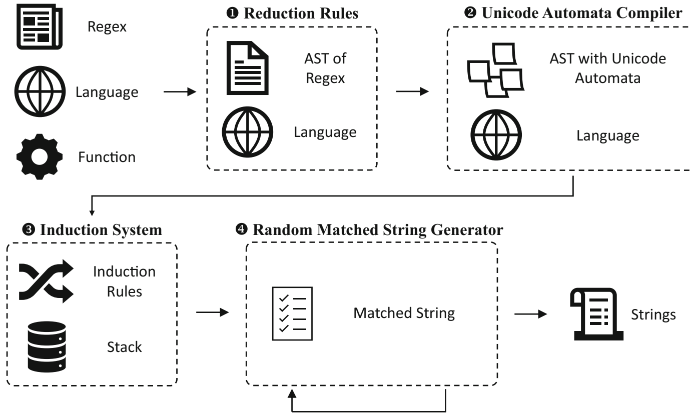


Fig. 1. The Framework of PowerGen

should be noticed that string constraint solvers do more work than string generators: they handle word equations and other more complicated string constraints like `ReplaceAll`.

We notice that, even under the features they claim to support, errors exist and are predominantly on features like lazy quantifier, word boundary, backreference. Examples[1] from practical projects [19] are listed in Table 1 with corresponding strings generated by each tool mentioned above, where the correct results are marked with "(✓)". In addition, "–" indicates that the tool does not support one or more features in the regex, *error* indicates run-time errors, and *unsat* indicates the tool determines that the regex cannot be satisfied with any input. It is evident that certain tools exhibit flawed handling of lazy quantifier, word boundary, backreference and lookaround. In the second example, the `.*` should be constrained to favor $\backslash b$, but Exrex, Xeger (O'Connor), Randexp.js and Ostrich do not take that into account and gives wrong results, and Expose returns an error. For the third regex, Randexp.js and ExpoSE, yielding *abcdefghijk* and *unsat* respectively, fail to support more than 9 backreferences. In the case of the expression `^(?:`$(a\backslash 2)(b)$`)+$` semantic differences arise between languages. JavaScript is capable of supporting backreferences preceding the corresponding capture group and generating the correct output, such as *ab*, *abab*. Nevertheless, Randexp.js fails to return the correct string. For the given example 6, the right node of the logical OR operator can accept an empty string. However, Egret, Exrex and Xeger('Connor) return an error and ExpoSE returns unsat. Moreover, none of them are capable of correctly handling a self-referenced backreference like example 7, or combining a backreference with lookaround as shown in example 8.

[1] To facilitate error identification, we simplify lengthy regexes by isolating the problematic fragment.

3 Overview

In this section, we provide an overview of our approach. Our method, depicted in Fig. 1, encompasses four main components: reduction rules, Unicode automata compiler, induction system, and random matching string generator. Initially, the **Reduction Rules** module takes the regex, the language, and a matching function to form an AST, addressing the semantic divergence between partial and full match calls. This simplified regex AST and language are then forwarded to the **Unicode Automata Compiler**, which develops automata to integrate the Unicode 15.0.0 standard into UTF-8 and compiles the AST leaves into the Unicode Automaton. The **Induction System** component, chosen based on the input language, converts the AST into induction rules and stack. Lastly, the **Random Matching String Generator** uses these induction rules and stack with capture information to generate matching strings. By iteratively executing the generation function, multiple matching strings are produced.

In the following section, we exemplify our approach by highlighting the intractable part of a regex Ψ = `(?:=(")?[^;"\s]*\1)` from Node.js version 18.16.0. The original regex is `^(?:<[^>]*>)(?:\s*;\s*[^;"\s]+(?:=(")?[^;"\s]*\1)?)*$`, which is used to validate the Web Linking header within HTML documents. This validation process ensures that the input value is free from any syntactically invalid Uniform Resource Identifiers (URIs). A legal input of Ψ is `="`*style*`"` and the backreference referred to the first capturing group `(")`$^{?}$ ensures the quotation marks `"` are matching, e.g. `= "`*style* should be rejected by Ψ, but `=` *style* is acceptable.

4 Modeling and String Generation Algorithms

In this section, we present the details of our model for regex semantics. First, we describe our extension of functions as a foundation for our model in Sect. 4.1. For optimizing the efficiency of our tool, we implemented some reduction rules in Sect. 4.2. Then, we introduce a new automaton model for effective representation and Boolean operations for Unicode character classes in Sect. 4.3. We provide the induction rules from the AST in Sect. 4.4. Finally, we give a brief account of the random generation algorithms in Sect. 4.5.

4.1 Extension of Functions to Regex

Among the basic functions to synthesize position automata [23], according to the nomenclature in [11], the output of *first, follow, last* functions are called *position sets*. Here we generalize the *nullable, first* and *last* functions to regexes by giving computation rules for operators of regexes, which is necessary for modeling regexes. Due to space limitation, the details of computation rules are shown in our complete version.[2] We also deploy these functions on tasks such as processing semantic of anchors and identification of unsatisfiable cases, as heuristics to avoid the algorithms that require exponential time [46].

[2] https://cdn.jsdelivr.net/npm/dataset2023/.

$$c_1|\ldots|c_n \Longrightarrow [c_1\text{-}c_n] \qquad [c_1c_2]|[c_1c_3] \Longrightarrow [c_1c_2c_3]$$
$$(\texttt{?:}E_1^{\{0,l\}})^{\{m,n\}} \Longrightarrow E_1^{\{0,l\times n\}} \qquad [[cc_1][\hat{}cc_1]] \Longrightarrow .$$
$$(\texttt{?=}[cc_1])[cc_2] \Longrightarrow [[cc_1]\&\&[cc_2]] \qquad (\texttt{?!}[cc_1])[cc_2] \Longrightarrow [[cc_1]\text{-}[cc_2]]$$
$$[cc_1](\texttt{?<=}[cc_2]) \Longrightarrow [[cc_1]\&\&[cc_2]] \qquad [cc_1](\texttt{?<!}[cc_2]) \Longrightarrow [[cc_1]\text{-}[cc_2]]$$
$$E_1|...|E_n| \Longrightarrow (\texttt{?:}E_1|...|E_n)^{\{0,1\}} \qquad \underbrace{E_1\ldots E_1}_{k \text{ times}} \Longrightarrow E_1^{\{k\}}$$

Fig. 2. Reduction Rules for Regex

Definition 1. *For a regex E over Σ, we define the following functions:*

$$first(E) = \{a \mid aw \in L(E), a \in \Sigma, w \in \Sigma^*\} \tag{1}$$
$$last(E) = \{a \mid wa \in L(E), a \in \Sigma, w \in \Sigma^*\} \tag{2}$$
$$nullable(E) = \begin{cases} true, & if\ \varepsilon \in L(E) \\ false, & otherwise \end{cases} \tag{3}$$

The definitions effectively compute the possible prefix/suffix of length one from a regex, without a full traversal on the AST. For example, for $E = \hat{}\backslash b[\hat{}\backslash d]^{\{2,4\}}(\texttt{?<!}\backslash w)$, $first(E) = \{[a\text{-}zA\text{-}Z]\}$, $last(E) = \{[\hat{}\backslash w]\}$, $nullable(E) = false$. We also notice those functions for backreferences depends on the capturing information during the generation, which can not be soundly computed statically.

4.2 Reduction Rules for Regex

We implemented several reduction rules shown in Fig. 2 to optimize the efficiency of our tool. Some of these reduction rules above are derived from existing regex engines and practical tools. For instance, mechanic of $[c_1c_2]|[c_1c_3] \Longrightarrow [c_1c_2c_3]$ is also found in the C# regex library. The others provide significant help in terms of efficiency and precision. The reduction rules we show here are common among our language-dependent reduction rules according to their original engine implementation.

We handle the semantic differences of partial/full match calls in the reduction system. For tight relationships between function calls and anchors, we consider semantic equivalence reduction by appending $.^*$ in an unanchored prefix/suffix, and none when anchored under partial match call. We also consider the full matching semantic by appending `^` and `$`. Inside our algorithms, anchors are processed as empty characters with constraints and for the sake of succinctness, we support start-of-line/end-of-line anchors implicitly. For our running example Ψ, the $^?$ and * operators are rewritten into $^{\{0,1\}}$ and $^{\{0,\infty\}}$, thus the output is $(\texttt{?:=(")}^{\{0,1\}}[\hat{}\texttt{;"}\backslash s]^{\{0,\infty\}}\backslash 1)$. However $E\hat{}$ and $\$E$ when E is not nullable will be considered *unsat* in the reduction system and directly return $\varnothing$.

4.3 Effective Representation of Unicode Character Classes

We build automata in a top-down manner to encode the Unicode 15.0.0 [51] standard into UTF-8 in Algorithm 1 and traverse the automata to generate

acceptable strings in UTF-8 byte-by-byte. This structure makes string generation feasible in acceptable time. In situations when ASCII flags (e.g. re.A in Python) are enabled, our representation is simplified into ASCII ranges.

In Algorithm 1, each Unicode character sets c_i when transformed into UTF-8 encoding composed with several runes (ranges) c_i^j defined on a byte. After initialization, Algorithm 1 first checks whether the character class is $\varnothing$. If not, the algorithm iterates the Unicode ranges $c_1, c_2, .., c_r$ in the character class cc, initializes the current state $\mathcal{A}$ as the initial state *init* and j as the height of each c_i.

Then Algorithm 1 checks whether $j \geq 0$ and whether there has been a transition with c_i^j from the current state $\mathcal{A}$ to a non-null state, if so make this state as the current state and substract j. If there is no such non-null state and $j \geq 0$, we build a new state $\mathcal{S}$, and mark a transition $\delta(\mathcal{A}, c_i^j)$ to $\mathcal{S}$, then take $\mathcal{S}$ as the current state and substract j. Finally, we mark the final states when $j = 0$.

Algorithm 1: Unicode Automaton

Input: An Unicode character class $cc = [c_1, c_2, ..., c_r]$

Output: Initial state *init* of Unicode automaton or $\varnothing$ otherwise

```
1  init ← 0; F ← ∅;
2  if cc = ∅ then
3      return ∅;
4  for i ∈ 1...r do
5      A ← init; j ← len(c_i);
6      while j ≥ 0 ∧ δ(A, c_i^j) ≠ ∅ do
7          A ← δ(A, c_i^j); j ← j − 1;
8      while j ≥ 0 do
9          δ(A, c_i^j) ← S; A ← S;
10         j ← j − 1;
11     if A ∈ F then
12         F ← F ∪ {A}
13     else
14         F ← {A}
15 return init;
```

The Unicode automaton allows us to support a vast class of Unicode-related extended features, which is a major factor of the high usability of our tool. The Unicode character classes effectively define an automaton of a finite language with more succinctness than those translated to standard regular expressions. The cost from Boolean operations among Unicode automata, including intersection [40], subset construction [41], has a major impact on the performance of our tool. To mitigate the cost, we execute those algorithms lazily, e.g. for $[\hat{}\,;"\backslash s]$ in our running example Ψ, the complementation of $[;"\backslash s]$ is computed only if a character is to be generated from this character class, instead of pre-processing and rewritting them in advance [17], thus guarantee the efficiency. The other character classes in Ψ are also compiled into Unicode automata.

4.4 Induction System for Regex

To comprehensively model the semantics of extended operators and generate matching strings, we propose the induction system.

The induction rules are composed of *configurations* and logical constraints, where a triple (E, w, C) is called a configuration, where E is a regex, w is a variable representing strings that E defines, and C is a stack preserving

the generated strings from the referenced subexpressions. To comprehensively model the semantics of regex operators, the extension of the basic functions defined in Sect. 4.1 is necessary for our induction system. Also the syntactic and semantic differences between dialects of regexes can hardly be negligible. To tackle this problem, we designed different induction rules for string generation, according to specified languages from the user. From the categorization in [5], we consider ε-semantics and $\varnothing$-semantics, and differentiate regex dialects in implementation details, e.g. $\backslash w$ is equal to $[a\text{-}zA\text{-}Z0\text{-}9_]$ in Python mode and $[\backslash p\{L\}\backslash p\{Mn\}\backslash p\{Nd\}\backslash p\{Pc\}]$ in C# mode. The induction rules we show here are designed for JavaScript regexes. In our induction system, specialized induction rules for other dialects of regexes can also be found.

$$\text{(CONCAT)}\quad \frac{(E_1E_2, w_1w_2, C)}{(E_1, w_1, C)\ (E_2, w_2, C)}$$

$$\text{(ALTER)}\quad \frac{(E_1|E_2, w_1w_2, C)}{(E_1, w_1, C) \vee (E_2, w_2, C)}$$

$$\text{(GREEDY)}\quad \frac{(E_1^{\{m,n\}}, w_1w_2...w_n, C)}{\bigwedge_{m<j\leq n} (E_1, w_j|\epsilon, C) \quad \bigwedge_{0<i\leq m} (E_1, w_i, C)}$$

$$\text{(LAZY)}\quad \frac{(E_1^{\{m,n\}?}, w_1w_2...w_n, C)}{\bigwedge_{0<i\leq m} (E_1, w_i, C) \quad \bigwedge_{m<j\leq n} (E_1, \epsilon|w_j, C)}$$

$$\text{(CAPTURE)}\quad \frac{((_nE_1)_n, w, C)}{\forall \backslash k, \backslash k \notin E_1 \wedge (E_1, w, C_n \leftarrow w)}$$

$$\text{(BACKREF)}\quad \frac{(\backslash i, w, C)}{(\backslash i, w \leftarrow C_i, C)}$$

$$\text{(POS-LA)}\quad \frac{((\texttt{?=}E_1)E_2, w_1w_2, C)}{(E_1.^*, w_1, C) \wedge (E_2, w_1w_2, C)}$$

$$\text{(NEG-LA)}\quad \frac{((\texttt{?!}E_1)E_2, w_1w_2, C)}{(\neg(E_1.^*), w_1, C) \wedge (E_2, w_1w_2, C)}$$

$$\text{(POS-LB)}\quad \frac{(E_1(\texttt{?<=}E_2), w_1w_2, C)}{(E_1, w_1w_2, C) \wedge (.^*E_2, w_2, C)}$$

$$\text{(NEG-LB)}\quad \frac{(E_1(\texttt{?<!}E_2), w_1w_2, C)}{(E_1, w_1w_2, C) \wedge (\neg(.^*E_2), w_2, C)}$$

$$\text{(WBOUND)}\quad \frac{(E_1\backslash bE_2, w_1xyw_2, C)}{(E_1, w_1x, C) \wedge x \in last(E_1) \cap \backslash w\ \ (E_2, yw_2, C) \wedge y \in first(E_2) \cap \backslash W} \vee \frac{(E_1\backslash bE_2, w_1xyw_2, C)}{(E_1, w_1x, C) \wedge x \in last(E_1) \cap \backslash W\ \ (E_2, yw_2, C) \wedge y \in first(E_2) \cap \backslash w}$$

$$\text{(NON-WBOUND)}\quad \frac{(E_1\backslash BE_2, w_1xyw_2, C)}{(E_1, w_1x, C) \wedge x \in last(E_1) \cap \backslash w\ \ (E_2, yw_2, C) \wedge y \in first(E_2) \cap \backslash w} \vee \frac{(E_1\backslash BE_2, w_1xyw_2, C)}{(E_1, w_1x, C) \wedge x \in last(E_1) \cap \backslash W\ \ (E_2, yw_2, C) \wedge y \in first(E_2) \cap \backslash W} \vee \frac{(E_1\backslash BE_2, w, C)}{(E_1, \epsilon, C) \wedge nullable(E_1)\ \ (E_2, \epsilon, C) \wedge nullable(E_2)}$$

The rules for standard operators are self-explanatory. For rule GREEDY, the original configuration unfolds this operator into a conjunction of series of configurations to generate strings separately. For rule LAZY, it makes the string that the expression matches as short as possible.

We constrain the expressive power of the regex in the rule CAPTURE as follows. When processing a Python/C# regex, we disallow backreferences to appear inside any referenced capturing group. And when the user specifies JavaScript regex for the input, the first assertion in the post-condition is canceled and the unassigned backreference is configured as ε by default. The configuration also pushes the generated string from its configuration into stack C for reference. As we do not rewrite quantifiers like in [29], the generated strings from the same capturing group will overwrite the stack of index i during generation. Notice

that we do not present induction rule for non-capturing groups, since those are considered useless on AST as in regex engines like C#'s, and the unreferenced capture groups are considered non-capturing as in Java regex engine. Thus for those capturing groups not referenced within the regex, we treat them as non-capturing groups. For succinctness, the logic for named capturing groups is also contained in the rule CAPTURE.

In rule BACKREF, the configuration simply reads the context from the latest assigned value of the stack C_i into the string variable. For our running example Ψ, *last* and *nullable* will output *unknown* and proceed to generate a character from $(\texttt{"})^?$, when the generation by the capturing group requires any of those functions from this exact capturing group, our algorithm returns *unknown* to avoid non-termination. Once a character `"` is generated from the sub-regex $(\texttt{"})^?$, the above functions are considered decidable, i.e. $first(\Psi) = \{\texttt{=}\}$, $last(\Psi) = \{\texttt{"}\}$, $nullable(\Psi) = false$. Also, named backreferences are contained in this rule.

From ES2018 [20], lookbehinds are introduced into the standard. In the rules for lookarounds, take positive lookahead as an example, w_1 should belong to $E_1.^*$, $w_1 w_2$ is generated from E_2, the result is the conjunction of two configurations, i.e. $(\texttt{?=}a)\backslash w^+$ can generate *abbbb* even in full match mode. The most intractable case is when lookarounds are decorated by repetitions: the lookarounds also put limitations on each repeating subexpression adjacent to it, our induction system shows a natural advantage in handling these cases.

In rules WBOUND and NON-WBOUND, we categorize the situations by the *first* and *last* functions of subregexes. For instance, in regex $\hat{}\backslash b(\texttt{\&}|ab|c)$, `&` is not satisfiable. Thus we generate a character from $first((\texttt{\&}|ab|c)) \cap \backslash w$ and prune the configuration of `&`. Also, the rules contain the case when the word-boundary appears at the start or end of the regex. And a Non-word boundary operator can generate ε when E_1 and E_2 are both nullable. If none of the situations are satisfied, the induction system will return $\varnothing$.

Furthermore, since the complement of a regular language requires exponential time [21,46], we also apply heuristics for identification of unsatisfiable cases. For $E_1(\texttt{?<!}E_2)$, if E_2 is *nullable*, the complement of E_2 is $\varnothing$, thus the regex is unsatisfiable. Similar strategies are applied to other zero-width assertions.

Back to running example Ψ, the result from applying induction rules is shown as below. Notice we simplified the induction process to improve readability.

$$\dfrac{\dfrac{((\texttt{?:=}(\texttt{"})^{\{0,1\}}[\texttt{\^{};"}\backslash s]^{\{0,\infty\}}\backslash 1), w_1 w_2 w_3 w_4, C)}{(\texttt{=}(\texttt{"})^{\{0,1\}}[\texttt{\^{};"}\backslash s]^{\{0,\infty\}}\backslash 1, w_1 w_2 w_3 w_4, C)}}{(\texttt{=}, w_1, C) \quad \dfrac{\bigwedge_{0<j\le 1} ((\texttt{"}), w_2|\epsilon, C)}{\bigwedge_{0<j\le 1} (\texttt{"}, w_2|\epsilon, C_1 \leftarrow w_2|\epsilon)} \quad \dfrac{([\texttt{\^{};"}\backslash s]^{\{0,\infty\}}, w_3^{0\ldots j}, C)}{\bigwedge_{0<j\le\infty} ([\texttt{\^{};"}\backslash s], w_3^j|\epsilon, C)} \quad (\backslash 1, w_4 \leftarrow C_1, C)}$$

4.5 String Generation Algorithm

In this section, we introduce our algorithm `PowerGen`, which takes a regex, the corresponding language, and the matching function as inputs, and outputs matching strings.

Our algorithm first conducts a syntax check conforming to the regex syntax rules of the provided language, selects the corresponding reduction rules based on the language, and creates an AST. The reduction rules handle the semantic difference of partial/full match calls in the corresponding language. It then forwards the simplified regex AST and the language as input to the Unicode automata compiler. It develops automata to incorporate the Unicode 15.0.0 [51] standard into UTF-8 in Algorithm 1, and compiles the AST leaves into Unicode automata. Depending on the language, we choose the appropriate induction system. This system takes the AST as input and compiles it into induction rules with stack storage. Finally, our random matching string generator receives induction rules and stack information as input and produces a matching string as an output. The generator performs a top-down traversal on the induction rules to generate strings. All of the Boolean operations of induction rules are performed on Unicode automata in Sect. 4.3. If the induction system returns $\varnothing$, `PowerGen` outputs *unsat*. By running the generation function repetitively, multiple matching strings are produced, since our generation strategy is random. Returning to our running example, we can easily obtain random sentences from the string variable of the root configuration. One of the strings generated is `=":`z2L@Q`"`, while the "sound" modeling in [29] mistakenly returns `="`.

5 Evaluation

We implemented our algorithms in C++, and conducted experiments on a machine with 192-core Intel Xeon E7-8850 v2 2.30 GHz processors and 2048 GB of RAM, running Ubuntu 16.04.5 LTS. Our algorithm can generate matching strings in multiple languages, including Python, JavaScript, Java, PCRE2, and C#. Our empirical investigation aims to address the following research questions:

RQ1. When randomly generating strings, does accurate modeling of regexes improve string generation efficiency?

RQ2. Is our support for full matching and partial matching better than other tools, which only support one kind of matching semantics?

RQ3. How does our approach work in practical projects?

To address RQ1, we compare our approach with existing string-generation tools by evaluating our algorithm on publicly available datasets. We select representative examples to demonstrate the correctness of all the tools in generating strings according to their specified matching semantics, thus validating RQ2. Lastly, we assess the performance of our approach in comparison to other tools in practical projects to clarify RQ3.

5.1 Datasets

Our experiment was conducted on a benchmark from [19]. This benchmark contains 537,806 unique regexes extracted from 193,524 programs written in 8

programming languages, including JavaScript, Java, PHP, Python, Ruby, Go, Perl, and Rust. The unique regexes represent the set of expressions after removing duplications.

5.2 Tools for Comparison

We compared nine string generation tools (see Sect. 2.3). We ensured all tools were configured according to their original configurations as stated in their papers or documentation, respectively. Egret, dk.brics, Mutrex and Generex (extended on dk.brics) did not specify the regexes they supported in which programming language, so we classified them as Unspecific support. Meanwhile, Exrex and Xeger focused on Python regexes, and Randexp.js, ExpoSE and Ostrich specialized in JavaScript regexes. ExpoSE supports partial matching semantics, while the others support full matching semantics, as shown in Table 2.

Table 2. Language-Matching Calls-Features Support by String Generation Tools

	Tools[a]	**Egr**	**DK**	**Mut**	**Gen**	**Exr**	**Xeg**	**Rnd**	**Ost**	**Exp**	**PowerGen**
	Language	**Unspecified**				**Python**		**JavaScript**			**Multi-Language**
	Matching Calls	**Full**								**Partial**	**Full&Partial**
Features	Character Class	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Greedy Quantifier	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Lazy Quantifier	✓	✗	✗	✗	✓	✓	✗	✓	✓	✓
	Unicode (\uxxxx)	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Capture/Non-Capturing Group	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
	Input Start/End (^$)	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
	Start/End-of-line Anchors	✓	✗	✗	✗	✓	✓	–	–	–	✓
	Start/Reset match	✗	✗	✗	✗	–	–	–	–	–	✓
	Word/Non-word Boundary	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
	Lookahead	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
	Lookbehind	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
	Backreference	✓	✗	✗	✗	✓	✓	✓	✗	✓	✓
	Ignore Case, Multi-line, Single-line Flags	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓
	Extended Flag	✗	✗	✗	✗	✓	✓	–	–	–	✓
	Unicode Flag	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓
	Comment Group	✗	✗	✗	✗	✓	✓	–	–	–	✓

[a] abbreviations for each tool in this table, along with their respective definitions, are as follows: Egr (Egret), DK (dk.brics), Mut (Mutrex), Gen (Generex), Exr (Exrex), Xeg (Xeger (O'Connor)), Rnd (Randexp.js), Exp (ExpoSE), Ost (Ostrich)

5.3 Evaluation of Random String Generation

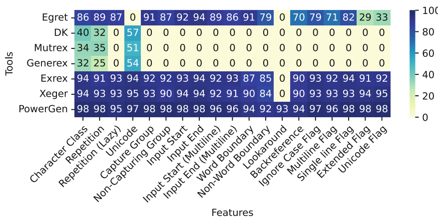
We compared the feature support for each tool. The results are presented in Table 2. The "–" in Table 2 indicates that the programming language specified by the tool lacks support for that feature, and the "✗" signifies that the corresponding tool does not support a feature. Our algorithm supports all the features listed earlier.

To begin with, we evaluate the impact of random generation. Through statistical analysis, the language-specific tools are based on the regexes of the Python or JavaScript languages. Therefore, in full matching comparisons, we evaluate each language-specific tool in its own language, and unspecific tools in both

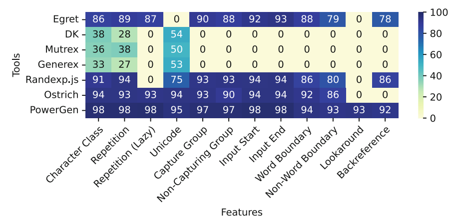
Python and JavaScript. For partial matching comparisons, there is only one tool, ExpoSE, to compare with us. We filtered the dataset to ensure syntax correctness. We assess the accuracy rate and time of the tools, with results depicted in Table 3, where the full match validation are shown on the left and the JavaScript partial match validation on the right. The first line on the left is based on Python validation results, while the second line refers to JavaScript validation results (achieved by adding start and end-of-line anchors before and after the regex). The accuracy rate is defined as: $AccuracyRate = \frac{1}{|D|} \sum_{i=1}^{|D|} Match(r_i, s_i)$. In the given formulas, r_i denotes the i-th regex in the dataset D, where $|D|$ refers to the size of D. s_i denotes the string generated by r_i by the tool. $Match(r_i, s_i)$ represents the validation of string s_i using the language's matcher. It returns 1 for success and 0 for failure. It is noteworthy to mention that Ostrich, ExpoSE, and our approach perform checks for unsatisfiability. We classify regexes as unsatisfiable if all tools determine them to be either unsatisfiable or incorrect. The memory usage refers to the average maximum resident set size.

Table 3. Experimental Results of String Generation Tools under Specific Language

Matching calls	Full match							Partial match	
Tools	Egr	DK	Mut	Gen	Exr	Xeg	PowerGen	Exp	PowerGen
Accuracy (%)	90.90	39.94	40.54	38.87	92.33	92.71	**97.40**	77.84	**97.72**
Time (s)	0.154	0.489	3.346	0.610	0.146	0.159	**0.004**	53.14	**0.004**
Memory (KB)	13535	47522	109772	52020	11021	9652	**5835**	90132	**4975**
Tools	Egr	DK	Mut	Gen	Rnd	Ost	PowerGen		
Accuracy (%)	90.92	35.82	44.62	40.93	91.43	93.69	**97.71**		
Time (s)	0.153	0.471	3.354	0.599	0.172	8.540	**0.004**		
Memory (KB)	13522	47477	110737	52078	32418	138913	**4796**		



(a) Experimental Results of Python String Generation Tools on Feature Dataset



(b) Experimental Results of JavaScript String Generation Tools on Feature Dataset

Fig. 3. Experimental Results of String Generation Tools on Feature Dataset

The experimental results show that `PowerGen` achieves the highest accuracy in random string generators among state-of-the-art tools. In Python full matching validation, `PowerGen` achieved the highest accuracy of 97.40%, followed by Exrex and Xeger (O'Connor) with about 92%, and the worst performer was DK, Mutrex and Generex with about 40%. For JavaScript full matching validation,

to be ranked according to accuracy, PowerGen achieved the highest percentage of 97.71%, followed by Randexp.js and Ostrich with about 92%, while DK, Mutrex and Generex achieved only 35.82%, 44.62% and 40.93% respectively. For JavaScript partial matching validation, PowerGen is about 20% higher than ExpoSE. Additionally, PowerGen generates correct results for all the examples mentioned in Sect. 2.3.

Regarding efficiency, in the full matching comparisons, the fastest among other tools is Exrex with 0.146 s, while our tool only takes 0.004 s, which has achieves a 36.5x speedup over Exrex. Moreover, our tool outperforms the slowest tool, Ostrich, by a factor of 2136. In the partial matching comparisons, we are up to ten thousand times faster than ExpoSE. We also expect our tool to be integrated into other softwares to improve their efficiency. Regarding memory usage, our memory usage is the least among tools in comparison.

To further analyze the experimental results, we then use the AST parsing tool to parse the regexes in the dataset and divide the dataset according to regex features. It should be noted that a regex can have multiple features thus can be split into multiple features categories. Due to the space limitation, we only present the comparison results under full matching calls. We calculate the accuracy rate for each feature and plot the Fig. 3(a) and 3(b) for Python and Javascript. The x-axis represents different features, the y-axis represents different tools, and the values indicate the accuracy rate.

The Fig. 3(a) and 3(b) indicate that our tool achieved the highest accuracy rate for each feature, where lookbehind is only supported by our tool. Among other tools that support Python regex syntax, their support for word boundary is relatively poor. Besides, for the tools that claim to support JavaScript regex syntax, their accuracy for word boundary, non-word boundary, and backreference achieves about 80%, and even 0% in some cases due to lack of support.

Summary to RQ1: By better supporting features, in the full matching comparisons, our algorithm achieves the highest accuracy for both Python and JavaScript. Regarding efficiency, our tool is several dozen times faster than the fastest among existing tools. In the partial matching comparisons, Our accuracy is 20% higher than ExpoSE while being faster by a factor of ten-thousandth.

Table 4. The Results by Each Generator for Examples

<table>
<tr><th rowspan="3">Regex[a]</th><th colspan="2">Possibly Acceptable Results</th><th colspan="11">Tools</th></tr>
<tr><th></th><th></th><th>Egr</th><th>DK</th><th>Mut</th><th>Gen</th><th>Exr</th><th>Xeg</th><th>Rnd</th><th>Ost</th><th>Exp</th><th colspan="2">PowerGen</th></tr>
<tr><th>Full</th><th>Partial</th><th colspan="8">Full</th><th>Partial</th><th>Full</th><th>Partial</th></tr>
<tr><td>\b\ $</td><td>unsat</td><td>a$</td><td>unsat</td><td>–</td><td>–</td><td>–</td><td>$</td><td>$</td><td>$</td><td>unsat</td><td>$</td><td>unsat</td><td>a$</td></tr>
<tr><td>\b\u0023</td><td>unsat</td><td>a#</td><td>unsat</td><td>–</td><td>–</td><td>–</td><td>$</td><td>$</td><td>$</td><td>unsat</td><td>$</td><td>unsat</td><td>a#</td></tr>
<tr><td>node_modules(?=paradigm.*) | (paradigm-gulp-watch)</td><td>paradigm-gupl-watch</td><td>node_ modulesparadigmaaa</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>unsat</td><td>paradigm-gulp-watch</td><td>node_ modulesparadigmaaa</td></tr>
<tr><td>"""(?=")</td><td>unsat</td><td>""""</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>unsat</td><td>unsat</td><td>""""</td></tr>
<tr><td>\n|(?=\?>)</td><td>\n</td><td>?></td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>unsat</td><td>\n</td><td>?></td></tr>
<tr><td>(?=a{2,5})\w</td><td>unsat</td><td>aa</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>unsat</td><td>unsat</td><td>aa</td></tr>
<tr><td>.(?<=\d{2,61})</td><td>unsat</td><td>11</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>unsat</td><td>unsat</td><td>11</td></tr>
<tr><td>https:\/\/(?=\w{2,3}).</td><td>unsat</td><td>https://aa</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>unsat</td><td>unsat</td><td>https://aa</td></tr>
</table>

[a] All the example regexes in this table are from [19]. We simplified some lengthy regexes for presentation.

5.4 Statistics for Full Matching and Partial Matching

The actual project library contains both full and partial matching capabilities. Unfortunately, the tools being compared only support one kind of matching semantics, which is inadequate for dealing with this situation. Furthermore, there are many logical errors in these tools regarding the semantics they claim to support. In this section, we analyse some representative examples from the dataset under full and partial matching calls in Table 4.

1. $\backslash b \backslash \$$
   Under full matching call, the regex above is unsatisfiable, while under partial matching it should return a string from $\backslash w \backslash$ `$`. Among those tools supporting full matching semantic, dk.brics, Mutrex and Generex lack support for word boundaries, Exrex, Xeger(O'Connor) and Randexp.js returns `$`, which is incorrect. ExpoSE's output `$` is erroneous under partial match, while `PowerGen` is capable to find the correct results under both matching semantics.
2. *node_modules*(`?=`*paradigm.**)|(*paradigm-gulp-watch*)
   Under full matching, *paradigm-gulp-watch* is the matching string, while for partial matching, results from *node_modulesparadigm.** or *paradigm-gulp-watch* are acceptable. Although ExpoSE claimed to support lookahead, it returned *unsat*, which is incorrect for both full and partial matching; `PowerGen` generates the correct answers under both cases.
3. `"""(?=")`
   For full matching, this regex is not satisfiable. In partial matching, lookaround requires that after matching three quotes, a quote must follow, resulting in the generation of four quotes. ExpoSE returns *unsat*, which is incorrect in the case of partial matching. In contrast, our result is accurate in both full and partial matching scenarios.
4. (`?=`$a^{\{2,5\}}$)$\backslash w$
   For full matching, this regex is unsatisfiable. And under partial matching, a string *aa* is acceptable. ExpoSE returned *unsat*, which is wrong under partial matching, while our results are correct under both cases.

The other examples in the table are similar to four examples above.

Summary to RQ2: By considering the distinctions between full matching and partial matching, our algorithm can generate correct strings for different semantics.

5.5 Results on Real Projects

Table 5. Examples in PyPI (Python) Project Library

<table>
<tr><th rowspan="2">FileName</th><th rowspan="2">Pattern</th><th rowspan="2">Matching calls</th><th colspan="7">Tools</th></tr>
<tr><th>Egr</th><th>DK</th><th>Mut</th><th>Gen</th><th>Exr</th><th>Xeg</th><th>PowerGen</th></tr>
<tr><td rowspan="2">.../mcdre
forged
_plugin.py</td><td>\w{1,16}</td><td>full matching</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>7</td></tr>
<tr><td>(?<=\u64027)
!!MCDR[\w]*
(?=\u6402)</td><td>partial matching</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>\u64027!!MC
DR\u6402</td></tr>
<tr><td rowspan="3">.../conf
igurati
on.py</td><td>[a-zA-Z0-9._-]+@
[a-zA-Z0-9._ –]{2,}
\.[a-z]{2,4}</td><td>partial matching</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>-</td><td>-@ddg.qq</td></tr>
<tr><td>[a-zA-Z0 – 9._-]+@
[a-zA-Z0-9._-]{2,}
\.[a-z]{2,4}</td><td>full matching</td><td>evil@–.aaaa</td><td>---.aa</td><td>---.ab(✗)</td><td>error</td><td>rhxbe@9
c60bHn7...</td><td>g@HOZGmxj8
.meuy</td><td>_@kzz.dd</td></tr>
<tr><td>(?P<heures>\d+)[h:]
(?P<minutes>\d+).+?
(?P<value>[a-zA-Z0-9\.]+)</td><td>partial matching</td><td>–</td><td>–</td><td>–</td><td>–</td><td>-</td><td>-</td><td>001:77\ud90b
\udf013</td></tr>
<tr><td rowspan="2">.../prep
rocessi
ng.py</td><td>(?:(?<=\s)|(?<=\W)|
(?<=^))(%\w)|(\
{.*? \})(?=\s| \W|$)</td><td>partial matching</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>–</td><td>%1</td></tr>
<tr><td>^{.*?}$</td><td>full matching</td><td>ε</td><td>–</td><td>–</td><td>–</td><td>{bqs-$tx}</td><td>{}</td><td>{}</td></tr>
</table>

We inspected a large number of projects in PyPI [39], Maven [32], npm [36] and other project libraries, and found that many of them contain various matching calls within the same project. In Table 5 and 6, we present a few examples of this phenomenon. Similar to RQ1, we conducted experiments in languages supported by each tool. It can be seen that other tools give wrong results in most examples due to not supporting some features and/or the matching calls (indicated by –), run-time errors (indicated by *error*), or others (indicated by ✗). Our tool consistently produces the correct results thanks to considering different matching semantics and supporting a wider range of extended features.

Summary to RQ3: Our approach is highly effective in real projects due to in-depth understanding of different matching semantics, as well as our comprehensive support for more extended features.

Table 6. Examples in Maven (Java) Project Library

FileName	Pattern	Matching calls	Tools				
			Egr	DK	Mut	Gen	PowerGen
.../JDBCUserStore Manager.java	(*)\1+	full matching	* * *	–	–	–	* * **
	(?<!\\)*	partial matching	–	–	–	–	”*
.../Path.java	/+	partial matching	–	–	–	–	///
	\p{Sc}+:	full matching	–	*error*	*error*	*error*	$:
.../ImdbParser.java	\u00bb	partial matching	–	–	–	–	\u00bb
	(?i)Country.*	full matching	–	–	–	–	Country\udbb4\ude5c
.../OpSumIf.java	.*(?<! ∼)\ * .*	full matching	–	–	–	–	*
	(?<! ∼)*	partial matching	–	–	–	–	@*
	(?<! ∼)\?	partial matching	–	–	–	–	?

6 Related Work

Matching Semantics and Extended Features. Leftmost-longest (POSIX-type) and Leftmost-greedy (PCRE-type) policies are two popular disambiguation strategies for regular expressions. However, POSIX implementations were found error-prone [16]. Okui and Suzuki [38] formalized leftmost-longest semantics and extended position automata [23] with leftmost-longest rule. Sulzmann and Lu extended Brzozowski's derivatives [7] to POSIX parsing problem [48]. Berglund et al. gave a formalization of Boost semantics for its combination of POSIX semantics and capturing groups [4]. Regular expressions with backreferences were first proposed by Aho in 90s [1]. Câmpeanu and colleagues gave rigorous formalisms and various properties [8–10] for regular expressions with backreferences. Recently Berglund and van der Merwe investigated theoretical aspects of regex with backreferences [5]. On the theoretical foundation for lookaheads, Miyazaki and Minamide [33] extended Brzozowski's derivatives [7] to lookaheads. Recently Berglund et al. [6] proposed a model based on Boolean Automata for regular expressions with lookaheads, and gave state complexity results. In 2022, Chida and Terauchi [15] gave the first formal study on regexes with both backreferences and lookaheads.

String Generation Toolkits. The Automaton Library [35] compiles a regex into an ϵ-NFA, and implements interfaces for random string generation. Egret [25] has a partial support for regexes to find inconsistency between regexes and specifications, it was found their tool lacks support for Unicode-related features. Reggae [26] supports string generation for regular expressions with intersection and complement operators; it also mentions that supporting lookarounds and boundaries is challenging. Veanes et al. [52] proposed Rex, which can be used for regular expression testing. Due to the cost of determinization on their proposed symbolic automata based on the construction of ϵ-NFAs, the string generation of Rex is not efficient. Loring et al. [29] claimed their model supports the complete regex language for ES6, but for lookarounds decorated by repetitions, ExpoSE seems to fail to give a correct result. Chen et al. [14] proposed a novel transducer model, namely PSST, to formalize the semantics of regex-dependent string functions, but backreference and lookarounds are still on their future work.

This paper and [47] both propose novel methods for modeling regexes. However, they develop different techniques because the problems they solve are different. The algorithmic differences are listed as follows: firstly our implementation is platform-independent based on induction rules instead of Z3 [34]. Secondly in [47] authors introduced the length constraint in modeling regex operators for checking satisfiability, which is insufficient to deduce fixed-length matched strings for generic purpose. Thirdly since the matching functions for regexes are ubiquitous in practical programs, for the first time we consider different matching semantics in modeling regexes for deducing matching strings. Also we constrained the expressive power of the input regex by induction rules to ensure

the completeness of our algorithm within a fragment of the class of regexes, while in [47], authors introduced a CEGAR (counterexample-guided abstraction refinement) scheme which makes their algorithm incomplete.

7 Conclusion

We propose PowerGen, a tool for deducing matching strings for regexes. It is based on a novel semantic model for regexes, which comprehensively models the extended features, with the awareness of different matching semantics and matching precedence. The evaluation results demonstrate the high effectiveness and efficiency of our algorithms. We aim to develop methods to further deduce the shortest matching strings for regexes and thus get a more refined model for regex in the future.

Acknowledgements. The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. Work supported by the Natural Science Foundation of Beijing, China (Grant No. 4232038) and the National Natural Science Foundation of China (Grant No. 62372439).

References

1. Aho, A.V.: Algorithms for finding patterns in strings. In: Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, pp. 255–300. Elsevier and MIT Press (1990)
2. Arcaini, P., Gargantini, A., Riccobene, E.: MUTREX: a mutation-based generator of fault detecting strings for regular expressions. In: ICST Workshops 2017, pp. 87–96 (2017)
3. Bartoli, A., Lorenzo, A.D., Medvet, E., Tarlao, F.: Inference of regular expressions for text extraction from examples. IEEE Trans. Knowl. Data Eng. **28**(5), 1217–1230 (2016)
4. Berglund, M., Bester, W., van der Merwe, B.: Formalising boost POSIX regular expression matching. In: Fischer, B., Uustalu, T. (eds.) ICTAC 2018. LNCS, vol. 11187, pp. 99–115. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02508-3_6
5. Berglund, M., van der Merwe, B.: Re-examining regular expressions with backreferences. Theor. Comput. Sci. **940**, 66–80 (2023)
6. Berglund, M., van der Merwe, B., van Litsenborgh, S.: Regular expressions with lookahead. J. Univers. Comput. Sci. **27**(4), 324–340 (2021)
7. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964)
8. Câmpeanu, C., Salomaa, K., Yu, S.: A formal study of practical regular expressions. Int. J. Found. Comput. Sci. **14**(6), 1007–1018 (2003)
9. Câmpeanu, C., Santean, N.: On the intersection of regex languages with regular languages. Theor. Comput. Sci. **410**(24–25), 2336–2344 (2009)
10. Câmpeanu, C., Yu, S.: Pattern expressions and pattern automata. Inf. Process. Lett. **92**(6), 267–274 (2004)

11. Caron, P., Champarnaud, J.-M., Mignot, L.: Partial derivatives of an extended regular expression. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 179–191. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21254-3_13
12. Chapman, C., Stolee, K.T.: Exploring regular expression usage and context in python. In: ISSTA 2016, pp. 282–293 (2016)
13. Chapman, C., Wang, P., Stolee, K.T.: Exploring regular expression comprehension. In: ASE 2017, pp. 405–416 (2017)
14. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. POPL **6**, 1–31 (2022)
15. Chida, N., Terauchi, T.: On lookaheads in regular expressions with backreferences. In: FSCD 2022. LIPIcs, vol. 228, pp. 15:1–15:18 (2022)
16. Chris, K.: Regex posix - HaskellWiki. https://wiki.haskell.org/Regex_Posix
17. D'Antoni, L., Veanes, M.: Automata modulo theories. Commun. ACM **64**, 86–95 (2021)
18. Davis, J.C., Coghlan, C.A., Servant, F., Lee, D.: The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In: ESEC/FSE 2018, pp. 246–256 (2018)
19. Davis, J.C., IV, L.G.M., Coghlan, C.A., Servant, F., Lee, D.: Why aren't regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. In: ESEC/FSE 2019, pp. 443–454 (2019)
20. ECMA: ES2018. https://262.ecma-international.org/9.0
21. Ellul, K., Krawetz, B., Shallit, J.O., Wang, M.W.: Regular expressions: new results and open problems. J. Autom. Lang. Comb. **10**(4), 407–437 (2005)
22. Fent: Randexp.js. https://github.com/fent/randexp.js
23. Glushkov, V.M.: The abstract theory of automata. Russ. Math. Surv. **16**, 1–53 (1961)
24. Hooimeijer, P., Veanes, M.: An evaluation of automata algorithms for string analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_18
25. Larson, E., Kirk, A.: Generating evil test strings for regular expressions. In: ICST 2016, pp. 309–319 (2016)
26. Li, N., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Reggae: automated test generation for programs using complex regular expressions. In: ASE 2009, pp. 515–519 (2009)
27. Liu, X., Jiang, Y., Wu, D.: A lightweight framework for regular expression verification. In: HASE 2019, pp. 1–8 (2019)
28. Loring, B., Mitchell, D., Kinder, J.: ExpoSE: practical symbolic execution of standalone JavaScript. In: SPIN 2017, pp. 196–199 (2017)
29. Loring, B., Mitchell, D., Kinder, J.: Sound regular expression semantics for dynamic symbolic execution of javascript. In: PLDI 2019, pp. 425–438 (2019)
30. Luo, B., Feng, Y., Wang, Z., Huang, S., Yan, R., Zhao, D.: Marrying up regular expressions with neural networks: A case study for spoken language understanding. In: ACL 2018, pp. 2083–2093 (2018)
31. Michael, L.G., Donohue, J., Davis, J.C., Lee, D., Servant, F.: Regexes are hard: decision-making, difficulties, and risks in programming regular expressions. In: ASE 2019, pp. 415–426 (2019)
32. Miller, F.P., Vandome, A.F., McBrewster, J.: Apache maven (2010). https://repo1.maven.org/maven2/

33. Miyazaki, T., Minamide, Y.: Derivatives of regular expressions with lookahead. J. Inf. Process. **27**, 422–430 (2019)
34. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
35. Møller, A.: dk.brics.automaton. https://www.brics.dk/automaton/
36. npm Inc: npm. https://www.npmjs.com/
37. O'Connor, C.: Crdoconnor/xeger. https://github.com/crdoconnor/xeger
38. Okui, S., Suzuki, T.: Disambiguation in regular expression matching via position automata with augmented transitions. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 231–240. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18098-9_25
39. Python Software Foundation: Python package index - pypi. https://pypi.org/
40. Rampersad, N., Shallit, J.: Detecting patterns in finite regular and context-free languages. Inf. Process. Lett. **110**(3), 108–112 (2010)
41. Salomaa, K., Yu, S.: NFA to DFA transformation for finite languages over arbitrary alphabets. J. Autom. Lang. Comb. **2**(3), 177–186 (1998)
42. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: S&P 2010, pp. 513–528 (2010)
43. Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X., Lu, J.: ReScue: crafting regular expression DoS attacks. In: ASE 2018, pp. 225–235 (2018)
44. Spishak, E., Dietl, W., Ernst, M.D.: A type system for regular expressions. In: FTfJP 2012, pp. 20–26 (2012)
45. Stanford, C., Veanes, M., Bjørner, N.: Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In: PLDI 2021, pp. 620–635 (2021)
46. Stockmeyer, L.J.: The complexity of decision problems in automata theory and logic. Ph.D. thesis, Massachusetts Institute of Technology, USA (1974)
47. Su, W., Chen, H., Li, R., Chen, Z.: Modeling regex operators for solving regex crossword puzzles. In: Hermanns, H., et al. (eds.) SETTA 2023, LNCS, vol. 14464, pp. 206–225. Springer, Cham (2023)
48. Sulzmann, M., Lu, K.Z.M.: POSIX regular expression parsing with derivatives. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 203–220. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_13
49. Tauber, A.: EXREX. https://github.com/asciimoo/exrex
50. Trinh, M., Chu, D., Jaffar, J.: S3: a symbolic string solver for vulnerability detection in web applications. In: CCS 2014, pp. 1232–1243 (2014)
51. Unicode: Unicode 15.0.0. https://unicode.org/versions/Unicode15.0.0/
52. Veanes, M., de Halleux, P., Tillmann, N.: Rex: symbolic regular expression explorer. In: ICST 2010, pp. 498–507 (2010)
53. Wang, P., Stolee, K.T.: How well are regular expressions tested in the wild? In: ESEC/FSE 2018, pp. 668–678 (2018)
54. Youssef, M.: Generex. https://github.com/mifmif/Generex
55. Yu, S.: Regular languages. In: Handbook of Formal Languages, Vol. 1: Word, Language, Grammar, pp. 41–110 (1997)