A Lightweight Framework for Regular Expression Verification

Xiao Liu, Yufei Jiang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
{xvl5190, yzj107, dwu}@ist.psu.edu

***Abstract*—Regular expressions and finite state automata have been widely used in programs for pattern searching and string matching. Unfortunately, despite the popularity, regular expressions are difficult to understand and verify even for experienced programmers. Conventional testing techniques remain a challenge as large regular expressions are constantly used for security purposes such as input validation and network intrusion detection. In this paper, we present a lightweight verification framework for regular expressions. In this framework, instead of a large number of test cases, it takes in requirements in natural language descriptions to automatically synthesize formal specifications. By checking the equivalence between the synthesized specifications and target regular expressions, errors will be detected and counterexamples will be reported. We have built a web application prototype and demonstrated its usability with two case studies.**

***Index Terms*—regular expression, verification, natural language, formal specification, domain-specific language**

I. Introduction

Regular expressions are widely used in computer programs for pattern searching and string matching due to the high effectiveness and accuracy. A recent technical report shows that regular expressions are used in 42% among nearly 4,000 open-sourced Python projects [1]. Researchers have also explored applying regular expressions to test case generations [2], [3], [4], [5], specifications for string constraint solvers [6], [7], and queries for some data mining framework [8]. As a foundation for lexical analysis, regular expression (regex) has been further applied to advanced functions for security purposes, such as input validation [9] and network intrusion detection [10]. Despite its popularity, it is still considered complicated for users, even experienced programmers, due to the low readability and the hardness to construct correct regular expressions when the size is as large as a single-page document. Because of the gradually increasing adoptions of agile software development, requirements and solutions evolve quickly, which causes the software development more error-prone. In this situation, input validations using regular expressions are constantly being found inefficient due to errors in the constructed expressions, resulting in security vulnerabilities [11], [12].

Due in part to the shared use across software development and how susceptible regexes are to errors, many researchers and practitioners have developed tools to support more robust regular expression creation [12] or allow more comprehensible verification. To verify the correctness of regular expressions, testing is widely adopted. These techniques solve the problem to some extent but are far from sufficient. Assisted with various online and offline tools, e.g., regex101 [13], developers can find errors in regular expressions through test cases. However, this kind of heavyweight frameworks increases their cognitive load to understand both the requirements and the corresponding regular expressions and require a significant amount of human labor to build test suites. In addition, it is difficult to determine the test coverage of self-generated test cases, especially for negative test cases and those for the Kleene star operator. Another critical issue which makes regular expression verification special is that developers are usually the testers, especially when agile development is popular in today's industries. They implement and mostly test a regular expression in the way that it should handle, which unintentionally avoid testing their target in a way that might break.

Alternative assistance is also proposed for regular expression debugging, such as visualization techniques [14], [15]. By parsing target regular expressions and representing them in *Finite State Machines*, it makes the semantics of these complex expressions more understandable. These methods save human labor from composing test cases and increase the readability despite the expressions without clear demarcation, but they overlook the complexity of regular expressions, some of which can be over twenty lines of code. The corresponding visualized automata can be gigantic and complex for users to comprehend. Therefore, the verification task remains difficult.

One may attempt to solve this problem with software verification. However, the regular expression verification problem is *ironic*. The regular languages are in the simplest language family in the formal language classification [16], while the software specifications are typically written in a more expressive or powerful language, e.g., first-order logic. Therefore it is paradoxical we need to write the specification in a more complicated language, which presumably will lead to more error in the specification than in the program (regex).

In this study, we are interested in enabling regular expression verification with natural language descriptions. We present a novel lightweight framework that *compiles* requirements in natural language into domain-specific formal specifications that we defined and *checks* the consistency between formal specifications and corresponding regular expressions using equivalence checking. Incorrect regular expressions will be detected when inequivalence is found and then, counterexamples will be synthesized. We propose this method to answer the following questions:

- **RQ1:** *How to verify the properties stated in the requirements for regular expressions?*
  We propose a domain-specific specification language called *Natural Expression* with which we model the requirements into automata and based on equivalence checking, we can verify the formally specified properties on regular expressions.
- **RQ2:** *How to lower the entrance bar for end-users to compose the formal specifications?*
  We propose a specification synthesis method with a rule-based natural language processing (NLP) technique. By analyzing the natural language requirements with additional user clarifications, formal specifications are generated automatically.

In the following part of the paper, motivated by a practical example which demonstrates the high cost of testing and conventional verification methods, we propose a lightweight verification method for validating regular expressions in Section II. Details about the design and techniques are presented in Section III. We describe the prototype tool (free access online and open-sourced) for the proposed framework in Section IV and we explain how our framework can help users to detect security vulnerabilities in two case studies in Section V. We discuss our contributions and limitations in Section VI. We review existing methods on regular expressions testing and verification and present a comparative summary in Section VII. We draw the conclusions in Section VIII.

II. Motivating Example

Regular expressions are widely used in software industries for input validation and pattern matching, but it is still a tough job to validate a regular expression according to its requirements. Consider the following regular expression:

```
c*(a(ab)*b)d*
```

The requirement is informally stated as:
There is a block of consecutive positions 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; the first letter in X is an a and the last letter is b.
To verify this regular expression, a test suite can be constructed as follows:

```
my @tests = (
    "cccab", "ecabd", "12eac",
    "cababd", "abd", "ccabd"
);
```

This is a test suite which covers three positive test cases ("cababd", "abd", and "ccabd") and three negative ones ("cccab", "ecabd", "12eac"). Testing requires a considerable amount of human labor and expertise to build complete test suites. In another word, few testers will guarantee their test cases cover all the properties as required. Also, it is hard to determine the test coverage, especially for negative test cases and those for testing Kleene stars.

Using first-order logic (FOL) for software verification creates a systematic way to conservatively prove the correctness of a piece of program if we define a language to describe its properties, such as using the *membership predicate*. By introducing the atomic predicate $\mathbf{Q_a(x)}$, where $\mathbf{a}$ is a character and $\mathbf{x}$ ranges over the positions of the word. In order to express the relations between positions, we add the syntax $x < y$ with the semantics "position x is smaller than position y". We can prove that a language over a one-letter alphabet is FO-definable if and only if it is finite or co-finite, where co-finite means the complement of this language is finite. However, simple expressions like $\{A^n | n \textit{ is even}\}$ is not FO-definable. Therefore, people usually extend the verification language with variables $X, Y, Z, ...$ ranging over sets of positions and the newly introduced predicate $x \in X$ means that "position x belongs to the position set X". And generalize the logic to the set of formulas over Σ with the expressions:

$$\varphi := Q_a(x) \mid x < y \mid x \in X \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \exists x\varphi \mid \exists X\varphi.$$

Therefore, it empowers the verifier to verify properties for the target regular expression with the following specifications:

$$\begin{aligned}
\textit{Make}(X) &:= \forall x \in X\ \forall y \in X \\
&\quad (x < y \rightarrow (\forall z\ (x < z \wedge z < y) \rightarrow z \in X)) \\
\textit{Before}(x, X) &:= \forall y \in X\ x < y \\
\textit{After}(x, X) &:= \forall y \in X\ y < x \\
\textit{Before_c}(X) &:= \forall x\ \textit{Before}(x, X) \rightarrow Q_c(x) \\
\textit{After_d}(X) &:= \forall x\ \textit{After}(x, X) \rightarrow Q_d(x) \\
\textit{Alternate}(X) &:= \forall x \in X(Q_a \rightarrow \forall y \in X(y = x + 1 \rightarrow Q_b(y))) \\
&\quad \wedge X(Q_b \rightarrow \forall y \in X(y = x + 1 \rightarrow Q_a(y))) \\
\textit{First_a}(X) &:= \forall x \in X\ \forall y\ (y < x \rightarrow \neg y \in X) \rightarrow Q_a(x) \\
\textit{Last_b}(X) &:= \forall x \in X\ \forall y\ (y > x \rightarrow \neg y \in X) \rightarrow Q_a(x)
\end{aligned}$$

And by putting them together, we have

$$\begin{aligned}
&\exists X.\ (\textit{Make}(X) \wedge \textit{Before_c}(X) \wedge \textit{After_d}(X) \\
&\quad \wedge \textit{Alternate}(X) \wedge \textit{First_a}(X) \wedge \textit{Last_b}(X))
\end{aligned}$$

for abstract the requirements.

Observing the specifications, although the logic behind is clear, it is not designed for light-trained end-users or even engineers. In our study, we consider using another specification language can be an overkill for solving this problem. We propose a method relies on a regex-like specification language and we validate the target regular expression with equivalence checking. The tool works on target examples with the following specifications:

$$\begin{aligned}
&my@spec = (\\
&\quad let\ X \leftarrow [a|b]*, && \textit{// in X, b's and a's alternate} \\
&\quad X \leftarrow (?=a.*b)\ in\ e; && \textit{// in X, begin in a end in b} \\
&\quad let\ e \leftarrow (?=c*X.*), && \textit{// there are only c's before X} \\
&\quad e \leftarrow (?=.*Xd*)\ in\ S; && \textit{// there are only d's after X} \\
&);
\end{aligned}$$

With these regex-like specifications, we will convert it into automata and conduct equivalence checking on this specification automata and target regex converted automata. If they are equivalent, we will consider the target correct; otherwise, we report error as either one of them is problematic. The specifications can be more complicated for a larger regular

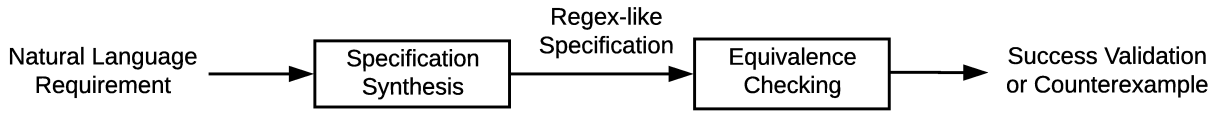

Fig. 1. Framework for regex verification

expression with more properties specified in the requirements, which requires additional professional training for the testers that results in extra cost. To reduce the difficulty for them to compose formal specifications, we incorporate an automatic specification synthesis module in our framework. Because of the versatility of natural language and seamless connection, we propose a rule-based method to synthesize the specifications from the requirements automatically. In another word, by providing the specifications in natural language descriptions similar to the commented part in the specification, one will validate the target regular expression without extra efforts. For this example `c*(ab)*d*`, our tool report `Failed` with a counterexample: `caabd`, which is matched with the specifications but not the target regex.

III. Key Designs

To overcome the barriers for regular expression verification, in this paper, we propose a lightweight verification framework that enables end-users to verify regular expressions with requirements and specifications as shown in Fig. 1. The cores lie in our framework are the *Equivalence Checking* module and the *Specification Synthesis* module. With the target regular expression and synthesized regex-like specifications, we will perform equivalence checking based on an analysis of the inclusion relation between the two regular expressions to validate whether the specified properties hold or not. To lower the bar of entry for specification synthesis, a rule-based natural language processing technique is taken. We elaborate the designs in detail in this section.

A. Regex Equivalence Checking

In our framework, we want to verify all the properties specified in the requirements hold for a given regular expression. We can describe our goal using an algebraic notation. Let E denote the target regular expression to be verified and R denote the requirement specified in natural language. Therefore, we need to validate that the target regular expression conforms to the requirements, We denote this relationship as

$$E \simeq R$$

where, $\simeq$ represents the conformation relation. We also propose to use natural language processing techniques to process the requirements and automatically generate formal specifications S in regular language. Suppose we can get a good formalization from natural language to formal specification, we still need to verify the equivalence or conformation between the target regular expression and the specifications; that is

$$E \equiv S \ \wedge R \ \simeq S$$

We present the formal specifications in regular language, and our goal is to check the equivalence between the regular expression and the formal specifications. We convert both the specifications and the target regular expression into automata and check their equivalence. There are existing tools that we can leverage. Antimirov and Mosses [17] presented a rewrite system for deciding the equivalence of two extended regular expressions (i.e. with intersection) based on a new complete axiomatization of the extended algebra of regular sets. It can be used to construct an algorithm for deciding the equivalence of two regular expressions, but the deduction system is quite inefficient. However, converting regular expressions into finite automata simplifies the target problem.

Algorithm 1 Inclusion of regex algorithm

```
1:  function INCLUSION(r1, r2)                    ▷ Determine r1 ⊆ r2
2:      F1 ← parse(r1)          ▷ Parse the string to finite automata
3:      F2 ← parse(r2)
4:      N2 ←neg(F2)                                ▷ Get negation
5:      intersection ← F1 & N2
6:      if intersection = ∅ then                   ▷ If r1 ∩ ¬r2 = ∅
7:          return True
8:      else
9:          return False
10:     end if
11: end function
```

Essentially, we can abstract the problem as the regular expression inclusion problem. To prove two regular expressions are equivalent, we are proving they are subset of each other; that is

$$r_1 \equiv r_2 \quad \Leftrightarrow \quad r_1 \subseteq r_2, \quad r_2 \subseteq r_1.$$

Here, a pair of regular expressions are in the $\subseteq$ relation if and only if their languages are in the inclusion relation. The inclusion relation in the languages means that all the strings that can be matched with the subset expression can also be matched with the superset expression, e.g., $d* \subseteq .*$. The algorithm for detecting the inclusion of a pair of regular expressions is presented in Algorithm 1.

We rely on the finite state automata for detecting the inclusion relationship between two regular expressions. The inputs are two regular expressions r_1 and r_2 which are represented in strings. We first parses the two regular expressions into finite automata F_1 and F_2. To determine whether the language that a regular expression accepts is the subset of another, we simply determine the intersection of the subset regular expression and the negation of the superset regular expression is an empty set, that is

$$L(r_1) \cap \bar{L}(r_2) = \emptyset.$$

Fig. 2 describes the overall workflow for the equivalence checking. We first convert the regular expressions into the corresponding finite state automata and calculate the equivalence based on testing the two-way inclusion relation. If it passes the equivalence checking, our tool returns “Verified”; otherwise, it returns “Failed” with a string generator. This string generator will output counterexamples which matches one of the two

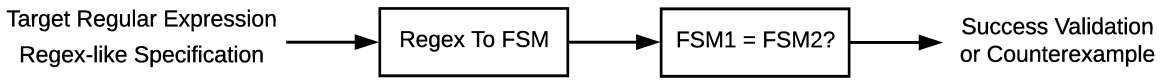

Fig. 2. Flowchart of equivalence checking

regular expressions, but not both at the same time. We output this counterexample for debugging purposes.

We implement the equivalence checker based on an existing automata tool. We extended *greenery 2.0* [18] which is a python library that provides robust methods for automata calculation and we tailored the functions for our purpose.

B. Natural Language Processing

Since we also propose a module that can synthesize formal specifications from natural language descriptions, the key is to understand the requirement description and synthesize the corresponding formal specification.

In recent studies, researchers are prone to adopt statistic-based methods when there are adequate data to train and test. However, in the situation that lacks training data, rule-based systems are alternative to statistic-based ones, to store and manipulate knowledge. The rules are usually based on the linguistic theories which are considered more efficient in specific domains since they do not require many computational resources, and error analysis is easier to perform [19].

In our design, we start with rule-based methods because of the lack of training dataset. The system is built upon an extension of a primitive AI prototype called Eliza [20]. Fig. 3 shows the workflow of our specification synthesis module. The input sentences are processed with a pre-defined script, where there are two basic types of rules: the decomposition rules and the reassemble rules. Decomposition rules are made up of different combinations of keywords. It is used in the tokenizer where a sentence is chopped into chunks and tagged with a token just similar to any modern compilers. Next, according to a set of reassemble rules, which are grammars over the predefined tokens, we will match each providing sentence with a specific semantic meaning. Taking advantage of this semantic meaning, we can synthesize target specifications in an intermediate let-language we defined. We will detail this let-language in Section III-B1.

Although Eliza belongs to the first-generation NLP techniques using a primitive rule-based method to understand users, it works well in specific domains [21], [22]. We have designed the rule script from an extensive study of the requirement descriptions in natural languages from various development documents.

1) Intermediate Representation: In the original framework, we propose to synthesize the regular specifications directly from the natural language requirements. However, it can misinterpret keywords for different applications; meanwhile, there exist different syntaxes for regular language as well. Thus, we add an intermediate level between the natural language requirements and the specifications to represent the semantics we extract from the requirements. The semantics in this level are bound with a property called usage to differentiate applications. We can also parse it for another round to preprocess some special terms. It will deal with the problems such as users describe the numerical range, e.g., for an IPv4 address, not in a digit by digit way. For example, they will describe `[1-2][0-9]` as `[10-29]`.

Fig. 4 is the syntax of the intermediate expression and the each statement in this language consists of two parts: an operator and an expression. The operator can be the verb extracted from the natural language that indicates concatenate, union, kleene, range, negative or position.

Traditionally. programming languages manipulate the value of variables. The values are typically integers, floats, and booleans. In natural expressions, we also manipulate the values, but which are the literal or meta regular expression character combinations. Specifically, our Natural Expression for the generation has the following operator types:

Concat. The manipulate with operator "concat" means to concatenate two expressions of any kinds, literal or meta or merely variable which is another expression. Concatenation is a fundamental part of the regular expression syntax. In the natural language, words like "and", "follow" or "put A and B together" have the meaning "concatenate".

Union. The operator "union" means to create a set for an expression that is described by either "A" or "B". In regular expressions, the symbol to realize this manipulation is "|". In natural language, words like "or" or "either...or..." can be mapped into the union function.

Kleene. The "kleene" is a unary operator. It means to repeat zero or more times. This corresponds to the Kleene star or closure in the regular expression syntax. In natural language, we use words like "repeat" and "for ... times" to describe this function.

Range. The operator "range" means a set which includes a set of elements that in a particular range. It could be a number range or an alphabet range. In regular expression syntax, "{"A"-"B"}" does the same operation. In natural language, we use words like "range in" "from A to B" or "between A and B" to describe this function.

Neg. The operator "neg" means to match a string that does not contain the expressions in a particular set. In regular expression syntax, the symbol to realize this manipulation is " "A and it is quite simple. In natural language, we use words like "not" "none of" or "exclude" to describe this function.

Pos. The operator "Pos" means to match a string that has a specific expression at position d. It is an atomic predicate $\mathbf{Q_a(x)}$, where $\mathbf{a}$ is a character and $\mathbf{x}$ ranges over the positions of the matching string. Although, it is not supported in traditional regular expression but it is very helpful for the verification language as it can be used to support position "before" or "after" for specifications.

2) Basic Library and Extended Library: Since we propose to build an intermediate level, we can build different libraries to process natural language terms for various applications. A basic library will be firstly constructed for cases in general regular expressions and extended libraries. For the prototype,

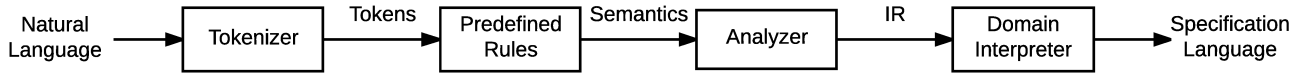

Fig. 3. Flowchart of specification synthesis

stmt S	::=	e		
	|	let $id \leftarrow e$ in S		
expr e	::=	number	|	character
	|	concat(e_1, e_2)	|	union(e_1, e_2)
	|	kleene(e)	|	neg(e)
	|	range(e_1, e_2)	|	pos(e, d)
number d	::=	[0-9]	|	[0-9]d

Fig. 4. The syntax of the intermediate expressions

we build a default library of *high-level abstractions* in Natural Expressions to match some knowledge-based terms in natural language descriptions. Each high-level abstraction can be implemented using basic expressions in the Natural Expression library. As a result, a few high-level abstractions are often sufficient to achieve what many basic expressions can achieve. For instance, "letter" stands for "[a-zA-Z]" and "vowel" means "[aeiou]". On the other hand, no matter for cognitive search or representation, descriptions with low Kolmogorov complexity is of high priority to be chosen by human [23]. Thus, the utilization of high-level abstraction plays an important role as it first reduces the search space of the matching process, second, makes the system more user-friendly.

3) Domain Interpreter: We implement a domain interpreter that translates the *intermediate let-language* into the *regex-like* specification which is used for equivalence checking. The domain interpreter is a lex-yacc style parser that parses the context-free let-language with an LR parser and builds an abstract syntax tree (AST) based on the grammar. We construct the interpreter based on PLY, a python implementation of lex and yacc. We release the source code for the domain interpreter[1]. It can be tailored to synthesize regular expressions directly from the DSL we defined directly as well.

IV. Prototype

To demonstrate the functionality and usability of the verification framework, we build a web application for prototyping which can be seen in Fig. 5. There are four panels in our design, which are the Target Regex Panel, Test Case Panel, Requirement Panel, and Specification Panel. Fig. 5 shows a snapshot on targeting the use of input validation. Before a user starts to interact with the web application, a target regular expression will be provided in the Target Regex Panel and the corresponding requirements in natural language can be referred to in a given document.

To conduct a verification task on a target regular expression, users are supposed to enter the requirements in the Requirement Panel. These descriptions should be in a constraint-based style, and each line only contains one property of the target regular expression. After receiving the requirements, our system will automatically synthesize the corresponding specifications line by line in the Specification Panel. After checking the equivalence between the specifications and the target regular expression, failed verification results will be reported as a reference for debugging. This prototype instantiates the functionalities of the proposed framework. In addition, there is a regular expression tester incorporated in this prototype as the Testing Panel for the evaluation purpose. Each line in this panel is a string for testing the target regular expression and positive cases will be highlighted which is similar to most regular expression debugging tools.

Availability. We have released the prototype tool the source code[2] for public dissemination.

V. Case Study

Regular expressions are widely used for security purposes, such as input validation and intrusion detection. It is of critical importance to verify the correctness of regular expressions to ensure security. In this section, we will conduct three case studies on regular expressions for different purposes, including *web input validation*, and *network intrusion detection*, to demonstrate our proposed framework. These regular expressions and requirements are collected from online development documentation and are currently in practical used by companies like Microsoft and Cisco.

A. Web Input Validation

Regular expressions are widely used for web input validation to constrain input, apply formatting rules, and check lengths. We will examine the regular expression that validates a strong password. The following example, including the regular expression and its requirements, comes from Microsoft technical documentations [24].

```
(?!^[0-9]*$)(?!^[a-zA-Z]*$)^([a-zA-Z0-9]{8,10})$
```

Requirement: *The password must be between 6 and 10 characters, contain at least one digit and one alphabetic character, and must not contain special characters.*

The requirement is from the technical document which is drafted by developers. In his description, there is only one sentence with a few breaks. The sentence is stated in a declarative style. Each of the short sentences specifies a constraint on the target regular expression which is to be verified. By processing the natural desciption, we find four propoerties for this regular expression. With a rule-based natural language processing technique, keywords like `8 to 10`, `characters`, `length`, `at least`, `numeric character`, `alphabetic character`, and `special characters` are detected and parsed with pre-define grammars. Therefore, corresponding specifications will be generated respectively:

`.{6,10}`	From 6 to 10 characters in length
`(?=.*\d)`	must contain at least one numeric character
`(?=.*[a-zA-Z])`	must contain one alphabetic character
`(?=[a-zA-Z0-9]*)`	must not contain special characters

[1] https://github.com/s3team/Regex-Verifier/src/Natural-Expression

[2] https://github.com/s3team/Regex-Verifier

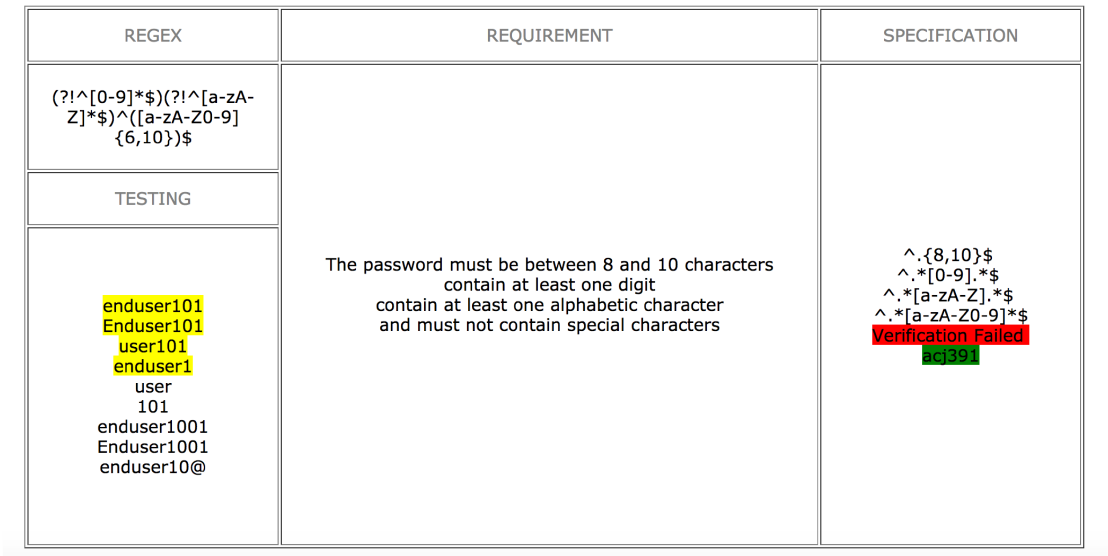

Fig. 5. A web app for prototyping: (1) Target Regex Panel: Takes in target regular expressions to be verified. (2) Test Case Panel: Takes in test cases to test the correctness of a target regular expression. (3) Requirement Panel: Takes in natural language requirements for a target regular expression. (4) Specification Panel: Displays the generated specifications from corresponding requirements.

Then, based on our framework, we will check the equivalence between the target regular expression and the intersection of generated specifications. For this example, results will show `Verification Failed` because of the incorrect length interval in the target regular expression. That is, the target regular expression does not conform the requirements.

B. Network Intrusion Detection

Regular expressions are used in intrusion detection/prevention systems to detect special activities. We will examine two regular expressions in such systems: one is a Cisco IPS signature that detects Yahoo! Messenger Login [25] and the other is from a snort rule [26] that detects packets with special contents.

B1. Cisco IPS Signature

```
[Y][M][S][G][\x00-\xFF]{6}\x00\x54
```

Requirement: *The preceding regular expression detects ASCII characters Y, M, S, and G followed by any six characters followed by hex* `\x00\x54`.

There is only one sentence in this requirement that our natural language processing tool can separate the sentence first indicated by keywords `followed by` and the corresponding specification will be generated:

```
[Y][M][S][G].{6}\x00\x54
```

When we check the equivalence between the two regular expressions, it returns `Verification Failed`. That is because our general library recognizes the keyword `any character` and interprets it as a wild card. However, Cisco uses the hex code to encode characters and compared with general regular expression systems, `any character` has a different meaning in the Cisco system. Therefore, we append a rule in a new library dedicatedly designed for Cisco IPS signature that specified a new interpretation of any character. With this new library loaded, a correct specification can be synthesized and it returns `Verified` for this case.

B2. PCRE from a Snort Rule

```
<OBJECT\s+[^>]*classid\s*=\s*[\x22\x27]?\s*clsid\s*
\x3a\s*A105BD70-BF56-4D10-BC91-41C88321F47C
```

Requirement: *It starts with* “`<OBJECT`” *then a few whitespaces. The next character after the whitespace cannot be* “>”. *Following that is* “`classid`” *and maybe some space after it which is optional and* “=” *then maybe another space here. Then there is an optional* “`'`” *or* “`"`”. *Then we have maybe some whitepace here and* “`clsid`” *and maybe some whitepace here followed by a* “`:`”. *After is some optional spaces and followed by the exact id* “`A105BD70...`”.

This requirement is longer than the previous one and it is also described in a declarative style. In this example, we notice some interesting observations. First, special characters like `'` and `:` are interpreted with their semantic meanings in the requirement description but represented as `\x22` and `\x27` in the target regular expression. It requires our system to process the natural language better with a higher level of abstraction. Therefore, we add new interpretation rules in the extended library for snort rules which will help us successfully translate the requirements to the specifications. In addition, we also observed that complete verification requirements can be harder to process, both for the users and the system compared with previous cases. However, there are still some keywords that we may look for with a rule-based processing system

such as "start with" and "followed by". By partitioning the sentence with the assistance of the keywords, requirements are separated into small pieces, each of which only contains one property of the target regular expression. Since the requirements are described continuously, specifications are synthesized one by one accumulatively:

S1: `^<OBJECT$`	it starts with "`<OBJECT`"
S2: **S1** + `\s+$`	then a few whitespaces
S3: **S2** + `(^>)$`	next character cannot be ">"
S4: **S3** + `^classid$`	following that is "`classid`"
S5: **S4** + `\s*$`	maybe some space after it
S6: **S5** + `=$`	and "="
S7: **S6** + `\s$`	maybe another space here
S8: **S7** + `[\x22\x27]$`	then an optional "'" or "`"`"
S9: **S8** + `\s*$`	then we have some whitepace
S10: **S9** + `clsid$`	and "`clsid`"
S11: **S10** + `\s*$`	and maybe some whitepace
S12: **S11** + `\x3a$`	followed by a "`:`"
S13: **S12** + `\s*$`	After is some optional spaces
S14: **S13** + `/A105B.../`	followed by the exact id "`A105B...`"

Since the specifications are built up accumulatively, the last specification, *S14* contains all the properties that specified in the previous ones. With the synthesized specifications, the system will check the equivalence between the target regular expression and the requirements. In this case, system returns `Verified`.

VI. Discussion

In software development, verification enforces consistency between the current development phase of a software and the initial requirements. However, formal specifications and analysis are often very expensive that requires highly qualified engineers. We propose a conceptual framework to assist with verifying the correctness of general software developments according to their requirements; and we instantiate the conceptual framework with a specific application, regular expression verification, to verify its efficiency and usability.

In typical software verification researches, more powerful languages such as VDM [27] and Z [28] are often used for formal specification. But in our study, regular expressions are simple enough that using these specification languages is an overkill. To verify the correctness of a regular expression, we can specify it in a regular language. Then if we have a specification in a regular language, we have a regular expression already. For an existing regular expression, we would like to avoid a user to write a specification in a regular language or more powerful language. Instead, the user can just tell in natural language what the regular expression should do, which is sort of software requirement, and then we will generate the specification and check the conformance between the implementation (the existing regular expression) and specification (the synthesized regular expression) automatically.

We not only propose a new approach but also solve *the problem of regex verification*. It's new because it's the first preliminary work that proposes to use natural language to validate regex. Typical specification languages are more powerful, or in other words, more complicated than regular languages. So verification of a regex using a specification in such languages is overkill and potentially makes things unnecessarily complicated. But if we simply create another regex according to requirements to act as the specification, and we'll simply need to check the equivalence. However, this solution overlooks the complexity of regular expressions.

There are still some limitations of the current prototype. The biggest issue is the accuracy of specification synthesis. Although we have achieved a relatively high accuracy, it is still the bottleneck for our framework. In our lab study, to exclude the interferes with the incorrect specification synthesis, researchers interrupted when they noticed any errors in the requirement descriptions or synthesis outcome and we conclude with a higher efficiency compared with the conventional testing method. Additionally, rule-based methods are not as scalable as statistical-based methods that the former ones consume more human labor. It is our goal to explore more on the formal specification synthesis and improve the performance in the future study.

VII. Related Work

In this section, we summarize related studies regarding our motivation and corresponding techniques.

A. Regex testing and verification

To verify the correctness of regular expressions, proposed approaches in previous studies focus on testing and debugging methods, which solve the problem to some extent but are far from sufficient. Black-box testing is mostly adopted. Assisted with various online and offline tools, e.g. regex101 [13], users can find bugs in regular expressions through a set of test cases. White-box testing is also discussed by a few researchers that regular expressions can be visualized, for instance, as graph structures [15], [29], [30]. Fabian et al. [14] also introduced the approach that provides advanced visual highlighting of matches in a sample text. However, these techniques rely on test cases where the test set explosion problem still remains. Another issue with white-box testing is that it relies on the constructed regular expression which neglects the properties that specified in the requirements.

Naturally, we think of verification techniques which check the required properties based on mathematical proof but not dynamic test cases. However, only a few studies have been done on regular expressions. Existing papers mostly focus on verification of the syntax level properties. Static program analysis techniques like type systems [12] are adopted for exception detection for regular expressions, e.g. IndexOutOfBound, at the compile time. According to our knowledge, there is *no* previous work on verifying semantic level properties for regular expressions.

B. Formal Language Synthesis

Program synthesis has been widely used in many situations. These situations can all be called a specific domain respectively, where program synthesis could take place. Maoz, Ringert, and Rumpe [31] have solved the NP-hard synthesis problem of the satisfied component and connector model in a bounded scope under some specifications. Gulwani [32] proposes a concrete case of program synthesis as an application for the Microsoft spreadsheet. Singh, Gulwani, and Solar-Lezama [33] show another case that program synthesis is utilized as a method to generate feedback for some simple coding assignments of the introductory programming courses.

Synthesis from natural language was proposed more than a decade ago. In the past few years, researchers devoted more effort to the domain-specific program synthesis, for instance, shell scripts [34], network configuration file [22], games [34], and regular expressions [35], [36] etc. They focus on how to make less ambiguity in the program synthesis from a more abstract language for different purposes.

VIII. Conclusion

We have presented a lightweight verification framework for regular expressions in this paper. It is based on an equivalence checking method between formal specifications and the target regular expressions. To enhance the usability of the proposed framework, we incorporated a specification synthesis module which automatically generates specifications in formal language from natural language descriptions of the requirements. We have built a prototype for the proposed framework and conducted case studies for evaluation. We have also made our tool available and released the source code for public dissemination.

IX. Acknowledgement

We thank Yuyan Bao for the helpful discussions and comments on the manuscript. This research was supported in part by the National Science Foundation (NSF) grants CNS-1652790 and the Office of Naval Research (ONR) grants N00014-13-1-0175, N0001416-1-2265, N00014-16-1-2912, and N00014-17-1-2894.

References

[1] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in Python," in *Proceedings of the 25th Int'l Symposium on Software Testing and Analysis*, ser. ISSTA 2016, 2016.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, 2013.

[3] S. J. Galler and B. K. Aichernig, "Survey on test data generation tools," *Int'l J. on Software Tools for Technology Transfer*, vol. 16, no. 6, pp. 727–751, 2014.

[4] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang, "JST: An automatic test generation tool for industrial Java applications with strings," in *Proceedings of the Int'l Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 992–1001.

[5] N. Tillmann, J. de Halleux, and T. Xie, "Transferring an automated test generation tool to practice: From Pex to fakes and code digger," in *Proceedings of the 29th Int'l Conference on Automated Soft. Eng.*, ser. ASE '14, 2014, pp. 385–396.

[6] A. Kieżun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 4, 2012.

[7] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *Proceedings of the ACM SIGSAC Conference on Computer and Communication Security*, ser. CCS '14, 2014, pp. 1232–1243.

[8] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: Discovering and exploiting relationships in software repositories," in *Proceedings of the 32nd ACM/IEEE Int'l Conference on Software Engineering*, ser. ICSE '10, 2010, pp. 125–134.

[9] A. S. Yeole and B. B. Meshram, "Analysis of different technique for detection of SQL injection," in *Proceedings of the Int'l Conference & Workshop on Emerging Trends in Technology*, ser. ICWET '11, 2011, pp. 963–966.

[10] The Bro Project, "The Bro network security monitor," https://www.bro.org/, 2015.

[11] R. A. Martin, "Common weakness enumeration," *Mitre Corporation*, 2007.

[12] 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*, 2012.

[13] F. Dib, "Regex101," https://regex101.com/, 2014.

[14] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf, "RegViz: Visual debugging of regular expressions," in *Companion Proceedings of the 36th Int'l Conference on Software Engineering*, ser. ICSE Companion 2014, 2014, pp. 504–507.

[15] A. F. Blackwell, "See what you need: Helping end-users to build abstractions," *J. of Visual Language & Computing*, vol. 12, no. 5, pp. 475–499, 2001.

[16] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, September 1956.

[17] V. M. Antimirov and P. D. Mosses, "Rewriting extended regular expressions," *Theoretical Computer Science*, vol. 143, no. 1, pp. 51–72, 1995.

[18] S. Hughes, "greenery 2.0," https://pypi.python.org/pypi/greenery/2.0, 2005.

[19] A. Ranta, "A multilingual natural-language interface to regular expressions," in *Proceedings of the Int'l Workshop on Finite State Methods in Natural Language Processing*, 1998, pp. 79–90.

[20] J. Weizenbaum, "ELIZA—a computer program for the study of natural language communication between man and machine," *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.

[21] X. Liu and D. Wu, "PiE: Programming in Eliza," in *Proceedings of the 29th ACM/IEEE Int'l conference on Automated software engineering*, 2014, pp. 695–700.

[22] X. Liu, B. Holden, and D. Wu, "Automated synthesis of access control lists," in *Proceedings of the 3rd IEEE Int'l Conference on Software Security and Assurance*, 2017.

[23] N. Chater and P. Vitányi, "Simplicity: A unifying principle in cognitive science?" *Trends in Cognitive Sciences*, vol. 7, no. 1, pp. 19–22, 2003.

[24] Microsoft, "How to: Use regular expressions to constrain input in ASP.NET," https://msdn.microsoft.com/en-us/library/ff650303.aspx, 2005.

[25] Cisco, "Writing custom signatures for the cisco intrusion prevention system," http://www.cisco.com/c/en/us/about/security-center/ips-custom-signatures.html, 2016.

[26] J. Esler, "Writing Snort rules correctly," http://www.cisco.com/c/en/us/about/security-center/ips-custom-signatures.html, 2010.

[27] C. B. Jones, *Systematic Software Development using VDM*. Prentice-Hall, 1986, vol. 2.

[28] l. H. Sørensen, "A specification language," in *Program Specification: Proceedings of a Workshop Aarhus, Denmark, August 1981*, ser. Lecture Notes in Computer Science, vol. 134, J. Staunstrup, Ed. Springer, Berlin, Heidelberg, 1982, pp. 381–401.

[29] T. Hung and S. H. Rodger, "Increasing visualization and interaction in the automata theory course," in *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '00, 2000, pp. 6–10.

[30] N. Moreira and R. Reis, "Interactive manipulation of regular objects with FAdo," in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '05, 2005, pp. 335–339.

[31] S. Maoz, J. O. Ringert, and B. Rumpe, "Synthesis of component and connector models from crosscutting structural views," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 444–454.

[32] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11, 2011, pp. 317–330.

[33] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 15–26.

[34] X. Liu, Y. Jiang, L. Wu, and D. Wu, "Natural shell: An assistant for end-user scripting," *Int'l Journal of People-Oriented Programming (IJPOP)*, vol. 5, no. 1, pp. 1–18, 2016.

[35] Z. Zhong, J. Guo, W. Yang, J. Peng, T. Xie, J.-G. Lou, T. Liu, and D. Zhang, "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*, 2018, pp. 1608–1618.

[36] N. Kushman and R. Barzilay, "Using semantic unification to generate regular expressions from natural language," in *North American Chapter of the Association for Computational Linguistics*, 2013.