TRANSREGEX: Multi-modal Regular Expression Synthesis by Generate-and-Repair

Yeting Li[†§], Shuaimin Li[§], Zhiwu Xu[♮], Jialun Cao[‡], Zixuan Chen[†§], Yun Hu[♯§], Haiming Chen[†], Shing-Chi Cheung[‡]
[†] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[§] School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
[♮] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
[‡] The Hong Kong University of Science and Technology, Hong Kong, China
[♯] Science & Technology on Integrated Infomation System Laboratory,
Institute of Software, Chinese Academy of Sciences, Beijing, China
[†]{liyt,chenzx,chm}@ios.ac.cn, [§]lishuaimin17@mails.ucas.ac.cn
[♮]xuzhiwu@szu.edu.cn, [‡]{jcaoap,scc}@cse.ust.hk
[♯]huyun2016@iscas.ac.cn

***Abstract*—Since regular expressions (abbrev. *regexes*) are difficult to understand and compose, automatically generating regexes has been an important research problem. This paper introduces TRANSREGEX, for automatically constructing regexes from both natural language descriptions and examples. To the best of our knowledge, TRANSREGEX is the first to treat the NLP-and-example-based regex synthesis problem as the problem of NLP-based synthesis with regex repair. For this purpose, we present novel algorithms for both NLP-based synthesis and regex repair. We evaluate TRANSREGEX with ten relevant state-of-the-art tools on three publicly available datasets. The evaluation results demonstrate that the accuracy of our TRANSREGEX is 17.4%, 35.8% and 38.9% higher than that of NLP-based approaches on the three datasets, respectively. Furthermore, TRANSREGEX can achieve higher accuracy than the state-of-the-art multi-modal techniques with 10% to 30% higher accuracy on all three datasets. The evaluation results also indicate TRANSREGEX utilizing natural language and examples in a more effective way.**

***Index Terms*—regex synthesis, regex repair, programming by natural languages, programming by example**

I. INTRODUCTION

As a versatile mechanism for pattern matching and searching, regular expressions (abbrev. *regexes*) have been widely used in different fields of computer science such as programming languages, natural language processing (NLP) and databases due to the high effectiveness and accuracy [1]–[6]. Unfortunately, despite their popularity, regexes can be difficult to understand and compose even for experienced programmers [1], [3], [7]–[9].

To alleviate this problem, prior research has proposed techniques to automatically generate regexes. For example, several techniques generate regexes from natural language (NL) descriptions [10]–[13], while others synthesize regexes from examples [14]–[20]. Though these techniques help lessen the difficulties of automatic regex synthesis, they have obvious drawbacks as follows.

Existing NLP-based techniques can only generate regexes similar in shape to the training data and have relatively low accuracy on simple benchmark datasets (e.g., SOFTREGEX [13] achieved only 62.8% accuracy on benchmark NL-RX-Turk [11]). Furthermore, NLP-based techniques are impeded by the ambiguity and imprecision of NL even for stylized English [13], [21], [22]. For example, according to [13], among 921 incorrectly predicted regexes, over 38.4% are caused by ambiguity of NL descriptions and 27.8% are from imprecision. Additionally, Zhong et al. [22] found that NLP-based techniques may not make correct prediction if words in these NL descriptions are not covered by the training data.

On the other hand, the example-based synthesis approaches rely on high quality examples provided by users. The synthesized regexes may be under-fitting or over-fitting when the given examples do not meet the implicit quality requirements (e.g, insufficient or not characteristic enough). However, examples with high quality are often unavailable in practice. It poses difficulties in applying purely example-based approaches. In addition, these approaches have severe restrictions on the kinds of regexes that they can synthesize (e.g., absence of Kleene star [14], [15], limited character occurrences [16]–[18], [20] or constraining to binary alphabet [19]).

Therefore, to better synthesize regexes it would be ideal to take advantage of both NL and examples (called NLP-and-example-based synthesis or multi-modal synthesis): the use of advanced NLP-based techniques can reduce the amount of required (characteristic) examples meanwhile alleviating the amount of effort from users; while the use of examples can effectively disambiguate or correct errors in the descriptions. Further, a survey on posts on regex synthesis shows that many programmers actually use NL descriptions as a major resource, and leverage some example(s) to resolve the ambiguities of NL [23]. On the other hand, insufficient (characteristic) examples limit the generalization ability of example-based approaches, while incorporating NL can improve the generalization ability, and help to drastically narrow down the search space [23]. Actually there have been recent attempts in this direction [21], [24], in which they first translated the

NL description into a sketch[1], then searched the regex space defined by the sketch guided by the given examples. However, the forms of translated sketches are restricted. This prevents regexes from being synthesized correctly when the generated sketches are inappropriate (e.g., logically-incorrect). In such a case, the incorrectness will be inherited from sketches to the subsequent regex. Moreover, while these works [21], [24] have achieved relatively high accuracy on simple datasets, they did not perform well on complex and realistic datasets. In latter datasets, the NL descriptions are longer, more complicated, and describe the regexes which are more complex in terms of length and tree-depth [22], [25].

We observe that most of the incorrect regexes generated by NLP-based techniques are very similar to the target regexes with subtle differences, and can be made equivalent[2] to the target regexes with only minor modifications (e.g., reordering/revising characters or quantifiers). This motivates us to view the NLP-and-example-based regex synthesis problem as the problem of NLP-based synthesis with regex repair, and develop the first framework, TRANSREGEX, to leverage both NL and examples for regex synthesis by using NLP-based and regex repair techniques. TRANSREGEX uses an NLP-based synthesizer to convert the description into a regex. If the synthesized regex is inconsistent with the given examples, it then leverages a regex repairer to modify the synthesized regex guided by the examples, and returns the revised regex.

Considering that the lack of focus on *validity* in previous NLP-based works (e.g., on the dataset StructuredRegex, less than half of the regexes predicted by DEEP-REGEX (Locascio et al.) [11] are valid, see Section IV-C), we propose a two-phase NLP-based synthesis model $\mathrm{S_2RE}$ to solve this problem as follows. By rewarding the $\mathrm{S_2RE}$ model with validity, in addition to the semantic correctness reward used in previous works [10], [13], our model is towards generating more valid regexes than previous models. If the generated regexes are still invalid after that, the invaid2valid model, wherein the structure of $\mathrm{S_2RE}$ is reused, is used to transform them into valid ones to further guarantee the validity of regexes.

There are various drawbacks or limitations in existing repair techniques, such as only supporting positive or negative examples [26], [27], not supporting regexes with the conjunction ($\&$) operator, or having the problems of under-fitting/over-fitting [20], [28]. To overcome these drawbacks or limitations, we present a novel and efficient algorithm, SYNCORR, based on Neighborhood Search (NS) to repair an incorrect regex to achieve that the repaired regex is consistent with the examples. Particularly SYNCORR alleviates the under-fitting/over-fitting problem, via preserving the integrity of the small sub-regexes.

TRANSREGEX can greatly reduce the aforementioned errors caused by ambiguity, imprecision or unknown words in NL descriptions, via using the example-guided regex repairer. In comparison with existing multi-modal works [21], [24], TRANSREGEX avoids their limitations by not restricting the sketches of the generated regex. Furthermore, TRANSREGEX modularizes the synthesis problem as the NLP-based regex synthesis and example-guided regex repair, allowing one to use his own algorithms or any other new algorithms instead.

We evaluate TRANSREGEX by comparing TRANSREGEX against ten state-of-the-art tools on three publicly available datasets. Our evaluation demonstrates the accuracy of our TRANSREGEX is 17.4%, 35.8% and 38.9% higher than that of NLP-based works on the three datasets, respectively. Further, TRANSREGEX can achieve higher accuracy than the state-of-the-art multi-modal works [21], [24], with 10% to 30% higher accuracy on all three datasets. Our evaluation also reveals our NLP-based model $\mathrm{S_2RE}$ can generate 100% valid regexes on complex dataset, whereas other NLP-based tools can synthesize 49.6% to 90.6% valid ones. Finally, the evaluation results on regex repair also show that our SYNCORR has better capability than existing repair tools.

The contributions of this paper are listed as follow.

- We propose TRANSREGEX, an automatic framework which can synthesize regular expressions from both NL descriptions and examples. To the best of our knowledge, TRANSREGEX is the first to treat the NLP-and-example-based regex synthesis problem as the problem of NLP-based synthesis with regex repair.
- We introduce a two-phase algorithm $\mathrm{S_2RE}$ for regex synthesis from NL. By rewarding the $\mathrm{S_2RE}$ model with validity and using the invaid2valid model, $\mathrm{S_2RE}$ generates more valid regexes while having similar or higher accuracy than the state-of-the-art NLP-based models.
- We present a novel algorithm SYNCORR for regex repair that (i) leverages Neighborhood Search (NS) algorithms to guide the search for a better regex which is consistent with the given examples from the neighborhoods of the incorrect regex, and (ii) utilizes some rewriting rules for sub-regexes abstraction to preserve the integrity of some small sub-regexes, thereby alleviating under-fitting/over-fitting and efficiently reducing the search space.
- We conduct a series of comprehensive experiments comparing TRANSREGEX with ten state-of-the-art synthesis tools. The evaluation results demonstrate that the accuracy of our TRANSREGEX is 17.4%, 35.8% and 38.9% higher than that of NLP-based approaches on the three datasets, respectively, while TRANSREGEX can achieve higher accuracy than the state-of-the-art multi-modal techniques with 10% to 30% higher accuracy on all three datasets. The evaluation results also indicate TRANSREGEX utilizing natural language and examples in a more effective way.

II. OVERVIEW

In this section, we present an overview of TRANSREGEX. As illustrated in Fig. 1, TRANSREGEX consists of two steps, namely, the *NLP-based regex synthesis* (Section III-C) and the *example-guided regex repair* (Section III-D). In the first step, *NLP-based regex synthesis* takes the given NL description as input and tries to synthesize a regex from the NL

[1]A sketch is an incomplete regex containing holes to denote missing components.

[2]Two regexes are equal iff their corresponding languages are equivalent.

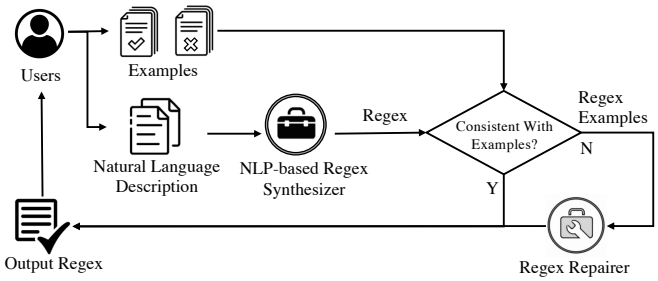

Fig. 1. An overview of framework TRANSREGEX for regex synthesis.

description via an NLP-based synthesizer. After that, if the synthesized regex is consistent with the given examples, then TRANSREGEX outputs the regex. Otherwise, *example-guided regex repair* modifies this synthesized regex based on the provided examples by an example-guided repairer, and returns the repaired regex. Next, we illustrate the main ideas behind TRANSREGEX using a motivating example.

Example II.1. Consider the task of constructing a regex `[AEIOUaeiou].*[0-9]{7,}.*`. The NL description $\mathcal{NL}$, the positive examples $\mathcal{P}$, and the negative examples $\mathcal{N}$ are shown in Fig. 2.

First, TRANSREGEX utilizes our NLP-based synthesizer $\mathrm{S_2RE}$ to translate the NL description $\mathcal{NL}$ into a regex. As shown in Fig. 3, the encoder of $\mathrm{S_2RE}$ generates latent vectors from the given description $\mathcal{NL}$, and the decoder of $\mathrm{S_2RE}$ synthesizes the corresponding regex `([AEIOUaeiou].*[0-9].*){7,,}` according to the latent vectors from the encoder. However, it is clear that this synthesized regex is invalid. $\mathrm{S_2RE}$ then converts this invalid regex into a valid one `([AEIOUaeiou].*[0-9].*){7,}` using our pretrained invalid2valid model. It is easy to verify that the valid regex generated by $\mathrm{S_2RE}$ is inconsistent with the provided examples, such as the positive example `E18043699` $\notin L($`([AEIOUaeiou].*[0-9].*){7,}`$)$. Intrinsically the description $\mathcal{NL}$ shown in Fig. 2 is ambiguous—*i.e.*, it is unclear what part of the string should appear at least 7 times, resulting in the incorrect regex generated by $\mathrm{S_2RE}$.

So after that, TRANSREGEX employs the algorithm SYNCORR that is based on Neighborhood Search (NS) to repair the incorrect regex, given in Fig. 4. As we have mentioned in Section I, the incorrect regexes generated by $\mathrm{S_2RE}$ may be very similar to the target regexes with subtle differences. Therefore, SYNCORR first converts the above incorrect regex to the abstract regex $r =$ `(<VOW><S><NUM><S>)`$<\mathrm{Q}_{7,}>$ with symbolic symbols obtained by executing the function *preprocess*, so that the integrity of small sub-regexes (e.g., `[AEIOUaeiou]` and `[0-9]`) can be retained as much as possible in the subsequent steps. Then SYNCORR calls the function *transformations* to get the neighbours of r, i.e., some abstract regexes (e.g., `<VOW><S>(<NUM>)`$<\mathrm{Q}_{7,}>$`<S>`) that are similar to r through a series of subtle transformations[3] (e.g., *quantifier adjustment* or *element replacement*, etc.). Next, SYNCORR maps these abstract regexes into corresponding concrete regexes via using the function *unpreprocess*, and calculates the f value[4] of each regex. Finally, SYNCORR returns the regex `[AEIOUaeiou].*[0-9]{7,}.*` with f value of 1. Leveraging NLP-based synthesis with regexs repair, our TRANSREGEX is able to synthesize the correct one mentioned above. This success case shows that our TRANSREGEX can well deal with the ambiguity of NL and the invalidity of regexes produced by some NLP-based synthesizers.

In addition, it is worth noting that on the same example and the incorrect regex mentioned above, the repair tool RFIXER [28] produces the incorrect regex `([AEeiO01234U56789].*[0-9].*){2,}`. Specifically, RFIXER cannot generalize for some unseen examples (e.g., a positive example `a1234567`), this will result in the regex produced by RFIXER without some characters like "a", i.e., the generated regexes will be over-fitting. In contrast, SYNCORR avoids over-fitting well by preserving the integrity of the small sub-regexes.

However, considering that SYNCORR is an algorithm based on NS and thus may trap in local optimum, we will continue to use RFIXER to repair if SYNCORR fails. As demonstrated in TABLE III, SYNCORR+RFIXER can achieve more than 10% higher success rate of repair than SYNCORR on the experimental datasets. Further, if there will be more powerful repair tool then RFIXER, by combining SYNCORR we can achieve even higher success rate, which is a future work.

III. REGEX SYNTHESIS ALGORITHM

In this section, we present the details of our synthesis algorithm TRANSREGEX. Before that, we first provide the background.

A. Background

Let Σ be a finite alphabet of symbols. The set of all words over Σ is denoted by Σ^*. The empty word and the empty set are denoted by ε and $\varnothing$, respectively.

Regular Expression (Regex). Expressions of ε, $\varnothing$, and $a \in \Sigma$ are regular expressions; a regular expression is also formed using the operators

$$[C] \quad r_1|r_2 \quad r_1r_2 \quad r_1\&r_2 \quad (r) \quad \sim r_1 \quad r_1\{m,n\}$$

where $C \subseteq \Sigma$, $C \neq \{\varepsilon\}$ or $\varnothing$ is a set of characters, $m \in \mathbb{N}$, $n \in \mathbb{N} \cup \{\infty\}$, and $m \leq n$. Besides, $r?$, r^*, r^+ and $r\{i\}$ where $i \in \mathbb{N}$ are abbreviations of $r\{0,1\}$, $r\{0,\infty\}$, $r\{1,\infty\}$ and $r\{i,i\}$, respectively. $r\{m,\infty\}$ is often simplified as $r\{m,\}$. The *language* $L(r)$ of a regular expression r is defined inductively as follows: $L(\varnothing) = \varnothing$; $L(\varepsilon) = \{\varepsilon\}$; $L(a) = \{a\}$; $L([C]) = C$; $L(r_1|r_2) = L(r_1) \cup L(r_2)$;

[3]In Fig. 4, we only demonstrate transformation *quantifier adjustment* in STEP 3. For all transformations, see Section III.

[4]The f value of a regex represents to which degree the regex meets the given examples. Especially, a regex with f value 1 will accepts all positive examples $\mathcal{P}$ and rejects all negative examples $\mathcal{N}$.

Natural Language Description $\mathcal{NL}$	
items with a vowel preceding a numeral ***at least 7 times***	
Positive Examples $\mathcal{P}$	Negative Examples $\mathcal{N}$
E18043699	u.
U530136382	jz;B
U65972791327	o45
U82433805	FBcW
i3390716928	I4k,S
O789821610	U
U4765749255	I$#].
E6204251	A
e6868266	uV
O50693106874	o20m3u5817

Fig. 2. A pair of a description and examples.

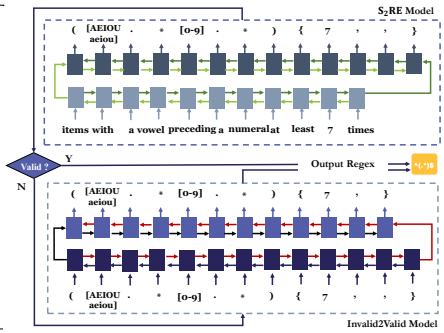

Fig. 3. The process of the algorithm S_2RE.

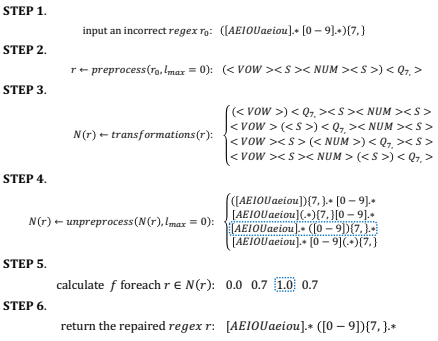

Fig. 4. The process of the algorithm SYNCORR.

$L(r_1r_2) = \{vw \mid v \in L(r_1), w \in L(r_2)\}$; $L(r_1\&r_2) = \{v \mid v \in L(r_1) \wedge v \in L(r_2)\}$; $L(\sim r_1) = \{v \mid v \notin L(r_1)\}$; $L(r\{m,n\}) = \bigcup_{m \leqslant i \leqslant n} L(r)^i$.

If an expression follows the syntax above, then it is a *valid* one, otherwise it is *invalid*. For instance, the expression `ab{1,3}` is valid, but the expression `ab{1,,,,3}` is invalid.

B. The Main Algorithm

Our synthesis algorithm is shown in Algorithm 1, which aims to synthesize regexes from NL descriptions and examples. In detail, our algorithm first employs a NLP-based synthesizer S_2RE to generate a regex r from the given NL description $\mathcal{NL}$ (line 1), which is introduced in Section III-C. Then, if r is consistent with the given positive and negative examples, TRANSREGEX outputs r (line 2). Otherwise, TRANSREGEX leverages two example-guided repairers to fix the incorrect regex r based on the provided positive and negative examples, which are described in Section III-D, and returns the repaired regex (lines 3–6).

Algorithm 1: TRANSREGEX

Input: a natural language description $\mathcal{NL}$, positive examples $\mathcal{P}$, and negative examples $\mathcal{N}$
Output: a regex

1 $r \leftarrow S_2\text{RE}(\mathcal{NL})$;
2 **if** $\mathcal{P} \subseteq L(r)$ *and* $\mathcal{N} \cap L(r) = \varnothing$ **then return** r ;
3 **else**
4 $r \leftarrow$ SYNCORR(r);
5 **if** $\mathcal{P} \subseteq L(r)$ *and* $\mathcal{N} \cap L(r) = \varnothing$ **then return** r ;
6 **else return** RFIXER(r);

C. Regex Synthesis from Natural Language Descriptions

To synthesize a regex from the NL description, we build a seq2seq model with attention mechanism as our regex synthesis model S_2RE. It consists of an encoder and a decoder. The encoder initializes the words in the NL sequences as vectors, then it encodes the vectors as hidden states, which represents the semantic representations of the NL sequences. The decoder generates the corresponding regex according to the latent representations from the encoder.

The training of our neural S_2RE model consists of two stages, using two different strategies.

Maximum Likelihood Estimation (MLE): In the first stage, we use MLE to maximize the likelihood of mapping the NL description to corresponding regex:

$$\theta = argmax \sum_{(\mathcal{NL},r)\in D} log(p(r|\mathcal{NL})) \tag{1}$$

where D is the training set, $p(r|\mathcal{NL})$ is the probability that the seq2seq model generates a regex r from a NL description $\mathcal{NL}$.

Policy Gradient: MLE may fail to consider the semantic equivalence of the regexes that might be different in syntax and the validity of the regexes (especially for complex and realistic datasets). Therefore, in the second stage, we gradually train our S_2RE model via policy gradient [29] by rewarding the model according to the following two indicators.

- Semantic Correctness: following Park et al.'s work [13], we reward the S_2RE model if it generates a regex that is semantically equivalent to the ground truth, that is, the semantic reward $R_C(r)$ is 1 if the regex r is semantically equal with ground truth and 0 otherwise;
- Syntactic Validity: we also reward the S_2RE model if it generates regexes that are valid[5], that is, the syntactic reward $R_V(r)$ is 1 if the regex r is valid and 0 otherwise.

Finally, the objective of the second stage is to maximize the following function:

$$J(\theta) = \sum_{(\mathcal{NL},r)\in D} p(r|\mathcal{NL})R(r) \tag{2}$$

$$R(r) = \alpha R_C(r) + \beta R_V(r) \tag{3}$$

where α and β are hyper-parameters.

The S_2RE model helps to generate more correct and valid regexes than the previous models. However, it still can not guarantee to generate valid regexes for all the input NL descriptions. To solve this problem, we reuse the structure of the S_2RE model to build our invalid2valid model (wherein only R_V is used). Specifically, the invalid2valid model is

[5]The syntax-checking is implemented via the re.compile() function in Python.

trained on $5,000$ invalid and valid regex pairs, which are collected as following:

- Get a valid regex randomly from the dataset *StructuredRegex* [25];
- Make it invalid by performing some minor changes on it : adding or deleting or modifying 1-5 positions randomly;
- If the changed regex is still valid, then discard it.

The whole process of generating regexes from NL descriptions is shown in Fig. 3. It shows that, the algorithm first uses our $\mathrm{S_2RE}$ model to generate a regex. If the generated regex is invalid, it then transform this regex fast to a valid one using the pretrained invalid2valid model. The experiments show that after the $\mathrm{S_2RE}$ model and the invalid2valid model, we obtain 100% valid regexes in syntax.

D. Regex Repair from Examples

The regexes generated by $\mathrm{S_2RE}$ may be incorrect but very similar to the target regexes with subtle differences. In this section, we present our algorithm SYNCORR to repair these incorrect regexes. The key idea is to search for a better regex which accepts more positive examples and rejects more negative examples from the neighborhoods of input regex.

To start with, we define the *neighborhood* and an evaluation criterion of regex r. Given a regex r, we define its *neighborhood*, denoted as $N(r)$, as the set of regexes, which can be obtained by applying a transformation on r (transformations are given later). In order to select a regex r among a set of regexes, we define a measure f on r with respects to positive examples $\mathcal{P}$ and negative examples $\mathcal{N}$ as

$$f(r,\mathcal{P},\mathcal{N}) = \frac{|T_\mathcal{P}| + |T_\mathcal{N}| - |F_\mathcal{P}| - |F_\mathcal{N}|}{|\mathcal{P}| + |\mathcal{N}|} \tag{4}$$

where $T_\mathcal{P} = \{w \in L(r) | w \in \mathcal{P}\}$, $T_\mathcal{N} = \{w \notin L(r) | w \in \mathcal{N}\}$, $F_\mathcal{P} = \{w \notin L(r) | w \in \mathcal{P}\}$, $F_\mathcal{N} = \{w \in L(r) | w \in \mathcal{N}\}$. Intuitively, the higher the f value, the better the regex r. Especially, the regex r with f value of 1 will accepts all positive examples and rejects all negative examples.

The algorithm SYNCORR is shown in Algorithm 2. In order to facilitate neighborhood search, SYNCORR sets the highest level of rewriting rules l_{max} ranges from 2 to 0 (line 1). First, SYNCORR preprocesses the given regex r_0 with the given highest level l_{max} and keeps the result regex in r (line 2). Next, it applies the transforms on r to get the neighborhood $N(r)$ and unpreprocess r and $N(r)$ (lines 4–6). Then SYNCORR selects the regex with the maximum f value, denoted as r_m, from the neighborhoods of r (line 7). After that, it compares the f values between the current regex r and the selected regex r_m (line 8). If the selected regex r_m gets a higher f value, then SYNCORR checks whether the f value equals to 1. If it is, SYNCORR returns r_m (line 9). Otherwise, to start with the next iteration, the regex r_m is preprocessed and assigned to r (line 10). If the selected regex r_m does not get a higher f value, then r_m may be a local maximum, so SYNCORR fails to repair regex r_0 and breaks the loop (line 11). For each l_{max}, the processing above runs until the stop conditions meet (lines 3–11). Finally, SYNCORR returns r_0 if SYNCORR fails for all l_{max} values (line 12).

Algorithm 2: SYNCORR

Input: positive examples $\mathcal{P}$, negative examples $\mathcal{N}$, and an incorrect regex r_0
Output: a regex

```
1  for l_max = 2 to 0 do
2      r ← preprocess(r_0, l_max);
3      while the stop conditions not meet do
4          N(r) ← apply the transformations on r in parallel;
5          r ← unpreprocess(r, l_max);
6          N(r) ← unpreprocess(N(r), l_max);
7          r_m ← arg max_{r'∈N(r)} f(r', P, N);
8          if f(r_m, P, N) > f(r, P, N) then
9              if f(r_m, P, N) == 1 then return r_m ;
10             else r ← preprocess(r_m, l_max);
11         else break;
12 return r_0;
```

Rewriting rules. In order to preserve the integrity of the small sub-regexes (probably the correct part) and reduce the search space, we define some rewriting rules for the regexes to abstract some small sub-regexes. The resulting regexes are called the abstract forms of the original regexes. Specifically, we rewrite some special small regexes to unique symbolic nodes, which are listed as follows:

$[C]$	$\to_0 < C_C >$	$const$	$\to_0 < C_{const} >$
$\{m,n\}$	$\to_0 < Q_{m,n} >$	$\sim(r)$	$\to_1 < N_r >$
$.*r$	$\to_1 < SL_r >$	$r.*$	$\to_1 < SR_r >$
$.*r.*$	$\to_1 < SLR_r >$	$\sim(.*r)$	$\to_2 < NSL_r >$
$\sim(r.*)$	$\to_2 < NSR_r >$	$\sim(.*r.*)$	$\to_2 < NSLR_r >$

where $const$ denotes a string (in regex) consisting of characters in Σ (*e.g.*, "abc"), r is $[C]$ or a $const$, and the suffix indicates the level l of the rule. The rewriting rules are performed greedily and depending on its level. In our implementation, we use some meaningful names for some special regexes (*e.g*, use `<NUM>`, `<LET>`, `<CAP>`, and `<VOW>` for `[0-9]`, `[A-Za-z]`, `[A-Z]`, and `[AEIOUaeiou]`, respectively) and keeps the rewriting mappings in dictionaries (*i.e.*, $Dict_l$ for the level l).

Preprocess and unpreprocess. Given a regex r and the highest level l_{max}, the preprocess is to abstract (*i.e.*, from left to right) r according the rewriting rules whose level l ranging from l_{max} to 0 in order. The rules with high level are performed first. Take r_0 in Fig. 4 as an example. If $l_{max} = 2$, the rules with level $l = 2$ are performed first, but without changing r_0 because of no rules with level $l = 2$ are applicable. Then the rules with level $l = 1$ are applied, yielding $r_1 =$ `(<SRVOW><SRNUM>){7,}`. Finally, the rules with level $l = 0$ is applied, yielding the final preprocessed

regex r =(<SRVOW><SRNUM>)<Q_7,>. If $l_{max} = 1$, the same preprocessed regex r is returned. If $l_{max} = 0$, directly apply the rewriting rules with level $l = 0$, yielding the final preprocessed regex (<VOW><S><NUM><S>)<Q_7,>. The unpreprocess is the reversion of the preprocess.

Stop conditions. We can set the maximum number of iterations, and the maximum running time of the program as independent or mixed stop conditions.

Transformation. We observe that most of the incorrect regexes generated by S_2RE can be made equivalent to the target regexes with only minor modifications. For that, we design a series of transformations on regexes, which are listed bellow, where we use *element*s to denote the small regexes that the rewriting rules can apply on, and *generalized element*s to denote the regexes containing at least a pair of brackets.

- *Binary Element Insertion*: this transformation inserts a binary element (*i.e.*, disjunction, concatenation, or conjunction) from a candidate set into the current regex.
- *(Generalized) Element Deletion*: this transformation deletes a (generalized) element from the current regex.
- *(Generalized) Element Replacement*: this transformation replaces a (generalized) element in the current regex with an element from a candidate set.
- *Quantifier Insertion*: this transformation inserts a quantifier, whose minimum and maximum values are selected according to the positive and negative examples, to a (generalized) element in the current regex.
- *Quantifier Modification*: this transformation modifies a quantifier of a (generalized) element in the current regex, where the minimum and maximum values are set according to the positive and negative examples.
- *Quantifier Adjustment*: this transformation adjusts the (generalized) element restricted by a quantifier in the current regex from its original restrict (generalized) element to another (generalized) element.
- *Operator Insertion*: this transformation inserts an operator (*i.e.*, negation, disjunction, or conjunction) into the current regex.
- *Operator Deletion*: this transformation deletes an operator from the current regex.
- *Element Adjustment*: this transformation adjusts an element in the current regex from its original position to another position.
- *(Generalized) Element Exchanging*: this transformation swaps two (generalized) elements in the current regex.

In our implementation, we take the elements collected in the dictionaries as the candidate set. And to search the neighbour fast, we perform the transformation in parallel, as there are no data races between each transformation.

IV. Evaluation

We implemented TransRegex in Python, and conducted experiments on a machine with 16 cores Intel Xeon CPU E5620 @ 2.40GHz with 12MB Cache, 24GB RAM, running Windows 10 operating system. Under this experiment settings, we then designed our experiments to answer the following research questions:

- RQ1: Can S_2RE model generate correct and valid regexes from natural language descriptions? (§IV-C)
- RQ2: Can SynCorr repair incorrect regexes from examples? (§IV-D)
- RQ3: Can TransRegex synthesize regexes accurately? (§IV-E)
- RQ4: Can TransRegex synthesize regexes efficiently? (§IV-F)

A. Datasets

In the experiment, we evaluate TransRegex on three public datasets: *KB13* [10], *NL-RX-Turk* [11], and *StructuredRegex* [25]. Among them, KB13 consists of 824 pairs of NL descriptions and the corresponding regexes constructed by regex experts. NL-RX-Turk includes 10,000 pairs of NL descriptions and regexes collected through crowdsourcing. StructuredRegex comprises 3,520 long English descriptions which are 2.9 to 4.0 times longer than the first two datasets, paired with complex regexes and associated 6 positive/6 negative examples using crowdsourcing. As our approach requires examples which are absent in the first two datasets, we adopt the corresponding 10 positive and 10 negative examples for the first two datasets provided by Ye et al. [24]. Specifically, the 10 positive examples are enumerated by randomly traversing the deterministic finite automaton (DFA) of the given regex (resp. the 10 negative examples are synthesized by stochastically traversing the DFA of the negation of the given regex).

B. Baselines

We compare three variants of TransRegex (i.e., TransRegex (S_2RE + SynCorr), TransRegex (S_2RE + RFixer), TransRegex (S_2RE + SynCorr + RFixer)) with ten relevant works, including our NLP-based algorithm S_2RE. They are mainly fall into two paradigms. NLP-based works only used NL descriptions to synthesize regexes, i.e., Semantic-unify [10], Deep-Regex (Locascio et al.) [11], Deep-Regex (Ye et al.) [24], SemRegex [12], SoftRegex [13], and our S_2RE. On the other hand, NLP-and-example-based works took both NL and examples for regex synthesis, including Deep-Regex (Ye et al.) + Exs [24], GrammarSketch+ MLE [24], DeepSketch + MLE [24] and DeepSketch + MML [24]. Among them, Semantic-unify learns a probabilistic grammar model to parse NL descriptions into regexes. Deep-Regex (Locascio et al.), Deep-Regex (Ye et al.), SemRegex, SoftRegex, and our S_2RE are a series of algorithms and the main idea of these algorithms is to translate NL into regexes based on the seq2seq model. Deep-Regex (Ye et al.) + Exs is an extension of Deep-Regex (Ye et al.), which takes examples into account by simply filtering the k-best regexes based on whether regexes consistent with the given examples. In addition, GrammarSketch+ MLE, DeepSketch + MLE, and DeepSketch + MML are a family of algorithms and these algorithms first use a grammar-based or neural semantic

parser to parse the NL into sketches, then search the regex space defined by the sketches and find a regex that is consistent with the given examples.

For implementation, DEEP-REGEX (Locascio et al.) [11] and SOFTREGEX [13] are open-source programs, so we are able to reproduce the results of them. While other baselines do not release their source code, so we excerpted the statistics from their paper, and left blanks if they did not report it.

C. RQ1: Effectiveness of S_2RE

To answer the first question, we compare our algorithm S_2RE with five state-of-the-art NLP-based algorithms. TABLE I shows the evaluation results on accuracy of SEMANTIC-UNIFY, DEEP-REGEX (Locascio et al.), DEEP-REGEX (Ye et al.), SEMREGEX, SOFTREGEX, and our model S_2RE. On these three datasets, the accuracy of our S_2RE model is similar to or slightly better than SOFTREGEX, and is always better than the other four NLP-based models (i.e., SEMANTIC-UNIFY, DEEP-REGEX (Locascio et al.), DEEP-REGEX (Ye et al.), and SEMREGEX).

Besides, the evaluation results on validity are summarized in TABLE II. The validity of synthesized regexes is crucial. It guarantees the quality of regex synthesis from NL . Further, in our approach, it ensures that the regex synthesized from NL is provided as a valid input to *example-guided regex repair*. The results show that the validity of existing tools (e.g., DEEP-REGEX (Locascio et al.), and SOFTREGEX) is unsatisfactory on the last dataset StructuredRegex which is much more complex than the first two. As we can see, less than half of the regexes generated by DEEP-REGEX (Locascio et al.) are valid and the most advanced NLP-based model SOFTREGEX achieves 90.6%. By contrast, our model S_2RE achieves 100% validity ratio, because it utilizes both the syntactic validity reward and invalid2valid model.

TABLE I
THE DFA-EQUIVALENT ACCURACY ON THREE DATASETS.

Approach	KB13	NL-RX-Turk	Structured Regex
SEMANTIC-UNIFY	65.5%	38.6%	1.8%
DEEP-REGEX (Locascio et al.)	65.6%	58.2%	23.6%
DEEP-REGEX (Ye et al.)	66.5%	60.2%	24.5%
SEMREGEX	78.2%	62.3%	—
SOFTREGEX	78.2%	62.8%	28.2%
S_2RE	78.2%	62.8%	28.5%
DEEP-REGEX (Ye et al.) + EXS	77.7%	83.8%	37.2%
GRAMMARSKETCH+ MLE	68.9%	69.6%	—
DEEPSKETCH + MLE	84.0%	85.2%	—
DEEPSKETCH + MML	86.4%	84.8%	—
TRANSREGEX (S_2RE + SYNCORR)	92.7%	94.2%	63.3%
TRANSREGEX (S_2RE + RFIXER)	90.3%	94.0%	53.1%
TRANSREGEX (S_2RE + SYNCORR + RFIXER)	**95.6%**	**98.6%**	**67.4%**

TABLE II
THE NUMBER OF VALID REGEXES GENERATED BY THE THREE NLP-BASED MODELS ON THREE DATASETS.

Approach	KB13	NL-RX-Turk	Structured Regex
DEEP-REGEX (Locascio et al.)	205 (99.5%)	**2500 (100%)**	494 (49.6%)
SOFTREGEX	204 (99.1%)	**2500 (100%)**	902 (90.6%)
S_2RE	**206 (100%)**	**2500 (100%)**	**996 (100%)**

Summary to RQ1: S_2RE can achieve similar or better accuracy than the state-of-the-art NLP-based models. Meanwhile, S_2RE can synthesize more valid regexes. The advantage of high validty of S_2RE becomes more obvious on complex datatsets.

D. RQ2: Effectiveness of SynCorr

To answer the second question, we collected 45, 930 and 712 incorrect regexes predicted by S_2RE on KB13, NL-RX-Turk, and StructuredRegex, respectively. We compared SYNCORR with the state-of-the-art tool RFIXER. TABLE III shows the number of successful repairs[6]. RFIXER successfully repaired 55.6%, 83.9%, and 34.4% on dataset KB13, NL-RX-Turk, and StructuredRegex respectively. In contrast, SYNCORR can repair 11.1%, 0.5%, and 14.2% more regexes than RFIXER on the three datasets, respectively. In addition, the repair success rate of SYNCORR + RFIXER is 11.1% (22.2%), 11.8% (12.3%), and 5.8% (20.0%) higher than that of SYNCORR (RFIXER) alone on dataset KB13, NL-RX-Turk, and StructuredRegex respectively.

We further demonstrate the advantages and disadvantages of RFIXER and SYNCORR through a few cases in TABLE IV. As cases #1 and #2 shown in TABLE IV, our algorithm SYNCORR has the high generalization performance compared with RFIXER. Case #3 in TABLE IV illustrates that SYNCORR can handle regexes with & well but RFIXER cannot. Similar to case #4 in TABLE IV, in some cases, SYNCORR will fall into a local optimum, and RFIXER can solve them. Based on the above analysis, we can conclude that SYNCORR and RFIXER are complementary and simultaneously useful for our TRANSREGEX.

Summary to RQ2: SYNCORR can more effectively repair regexes compared with the state-of-the-art tool RFIXER. In addition, SYNCORR and RFIXER are complementary and simultaneously useful for our TRANSREGEX.

TABLE III
THE NUMBER OF SUCCESSFUL REPAIRS BY SYNCORR AND RFIXER ON THREE DATASETS.

Approach	KB13	NL-RX-Turk	Structured Regex
RFIXER	25/45 (55.6%)	780/930 (83.9%)	245/712 (34.4%)
SYNCORR	30/45 (66.7%)	785/930 (84.4%)	346/712 (48.6%)
RFIXER + SYNCORR	35/45 (77.8%)	895/930 (96.2%)	387/712 (54.4%)

E. RQ3: Effectiveness of TransRegex

To answer this question, we compared TRANSREGEX with six NLP-based baselines and four multi-modal baselines. We can see from TABLE I that on average, NLP-based works performed worse than multi-modal works on all three datasets. In general, the accuracy on the first dataset KB13 is much higher than that on the other two datasets for NLP-based methods, and the accuracy achieved by NLP-based methods are up to

[6]The regex that is successfully repaired must not only consistent with the given examples, but also be equal to the target regex.

TABLE IV
EXAMPLES OF REPAIR BY RFIXER AND SYNCORR.

No.	Incorrect Regex	Ground Truth	RFIXER	SYNCORR
#1	([AEIOUaeiou].*[0-9].*){7,}	[AEIOUaeiou].*[0-9]{7,}.*	([AEeiO01234U56789].*[0-9].*){2,} ✗	[AEIOUaeiou].*[0-9]{7,}.* ✓
#2	[A-Za-z]{2,3}[a-z]{2,3}[A-Z]{3,4}	[A-Za-z]{2,3}[a-z]{3}[A-Z]{3,4}	([AabBCDEeFGHIJLnXxy]{2,}[abcdefl npvwxy])+[a-z]{2,3}[A-Z]{3,4} ✗	[A-Za-z]{2,3}[a-z]{3,3}[A-Z]{3,4} ✓
#3	([A-Z]|[a-z]){1,}&.{6,8}&(.*([A-Z]|[a-z]).*)	.{6,8}&(.*[A-Za-z].*)	**Not supporting regexes with &**	.{6,8}&(.*([A-Z]|[a-z]).*) ✓
#4	[A-Za-z]{3,}[0-9]{3,}N[A-Za-z]{2,4}	[A-Z]{3,}[0-9]{3,}(N|g)[A-Za-z]{2,4}	[A-Z]{3,}[0-9]{3,}[gN][a-zA-Z]){2,4} ✓	**Trapping in local optimum**

TABLE V
EXAMPLES ILLUSTRATING THE BENEFITS OF LEVERAGING BOTH NATURAL LANGUAGE AND EXAMPLES IN TRANSREGEX.

No.	Description	Ground Truth	S_2RE	Success Type
#1	items with a capital letter preceding "dog" **at least 3 times**	([A-Z].*dog.*){3,}	[A-Z].*(dog){3,}.*	Ambiguity of NL
#2	the string should start with at least 1 or more **capital i**, then it is followed by 3 letters.	1{1,}[A-Za-z]{3}	[A-Z]{1,}[A-Za-z]{3}	Imprecision of NL
#3	the string must begin with 2 or more letter s. after this grouping, it is a 3 digit number. after this 3 digit number, there a letter Z. after the letter Z, the string must contain _; or ∗##. the string can end with an optional string of 3 to 4 capital letters.	s{2,}[0-9]{3}Z(_;|∗##)([A-Z]{3,4})?	s{2,}[0-9]{3}Z[0-9]{3}([A-Z]{3,4})?	Unknown or rare words
#4	a list of 3 comma separated strings of lowercase letters.	[a-z]+(,[a-z]+)(,[a-z]+)	[a-z]+(,[a-z]+)*	False prediction by S_2RE

30% lower than that achieved by multi-modal methods on average. Particularly, on the first two datasets, the accuracy of NLP-based works range from 38.6% to 78.2%, compared with 68.9% to 86.4% achieved by existing multi-modal works. On comparison, TRANSREGEX achieved approximate 10% higher accuracy than these baselines, reaching 90.3% to 98.6% on the first two datasets. The superiority of our work is more obvious on the last dataset, which is much more complex than the first two. The accuracy achieved by TRANSREGEX (67.4%) almost doubled the accuracy achieved by DEEP-REGEX (Ye et al.) + EXS (37.2%).

Let us present a few example regexes that are synthesized incorrectly from NL descriptions but are synthesized correctly by our TRANSREGEX to illustrate the benefits of leveraging both NL and examples. As case #1 in TABLE V, the description is ambiguous, i.e., it is unclear what part of the string would appear at least 3 times, resulting in that S_2RE predicted a regex embedding other meanings. Our TRANSREGEX utilizes examples (e.g., a positive example `AdogBdogCdog`) to help disambiguate the description, and fix the incorrect regex `[A-Z].*(dog){3,}.*` to the correct regex `([A-Z].*dog.*){3,}`. Similar to cases #2, #3, and #4 in TABLE V, the errors that are caused by imprecision/unknown words in descriptions or false prediction by NLP-based approaches can be corrected by TRANSREGEX through using the example-guided regex repairer.

Further, we analyze the possible reasons why the other multi-modal works are not as effective as ours as follows. As mentioned above, DEEP-REGEX (Ye et al.) + EXS simply uses examples to select the k-best regexes among the candidates produced by DEEP-REGEX (Ye et al.). If there is no correct regex in the candidates, the examples will not help. The three variants algorithms (i.e., GRAMMARSKETCH+ MLE, DEEPSKETCH+ MLE, and DEEPSKETCH+ MML) rely heavily on the quality of the sketches synthesized in their first step. In other words, the incorrection of sketches will be inherited by the generated regex in the next step. While TRANSREGEX does not have the above-mentioned drawbacks. In addition, although TRANSREGEX is also a two-step algorithm, the second step of TRANSREGEX is not only not affected by the errors of the first step, but also specifically correct the errors of the first step guided by the given examples.

Summary to RQ3: TRANSREGEX can achieve higher accuracy than the NLP-based works with 17.4%, 35.8% and 38.9%, and the state-of-the-art multi-modal works with 10% to 30% higher accuracy on all three datasets. The experiment results also indicate TRANSREGEX utilizing natural language and examples in a more effective way than other multi-modal works.

F. RQ4: Efficiency of TransRegex

TABLE VI
AVERAGE RUNNING TIME PER BENCHMARK ON THREE DATASETS.

Approach	KB13	NL-RX-Turk	Structured Regex
DEEP-REGEX (Locascio et al.)	2.621 s	1.104 s	2.108 s
S_2RE	3.578 s	1.656 s	3.313 s
TRANSREGEX (S_2RE + SYNCORR)	4.958 s	3.085 s	8.624 s
TRANSREGEX (S_2RE + RFIXER)	5.821 s	4.737 s	22.460 s
TRANSREGEX (S_2RE + SYNCORR + RFIXER)	6.688 s	4.011 s	13.737 s

To evaluate the efficiency of three variants of TRANSREGEX, we compared the average running time of synthesizing per regex with the open-source baseline DEEP-REGEX (Locascio et al.) and our S_2RE in Table VI. In general, we can see that TRANSREGEX takes longer time to synthesize regexes on the last dataset than on the first two, and NLP-based baselines take less than TRANSREGEX on average. In particular, TRANSREGEX takes an average 5.822 seconds and 3.944 seconds on the first two datasets, compared with 1.104 to 3.578 seconds achieved by baselines. While

on the last dataset, TRANSREGEX takes longer time due to the complexity of the dataset. Considering together with the accuracy, TRANSREGEX achieved the accuracy (67.4%) that doubles the accuracy (28.5%) achieved by S_2RE, taking only around 10 more seconds running time. Table VI also reveals the rationality of our algorithms, the adopting of SYNCORR helps us to accelerate the repair process in some cases, while RFIXER takes care of the rest.

Summary to RQ4: TRANSREGEX can synthesize regex efficiently. Especially when considering together with accuracy, TRANSREGEX can takes an average 3.944 to 5.822 seconds to achieve around 20% more accuracy on simpler datasets, and takes 30% more accuracy at the cost of 10 more seconds on the more complex dataset.

V. THREATS TO TRANSREGEX'S VALIDITY

TRANSREGEX is not guaranteed to generate correct regexes for each benchmark, mainly due to the following aspects:

- **Uncharacteristic examples.** Although TRANSREGEX has greatly reduced the amount of required examples, the quality of TRANSREGEX still depends on characteristic examples. When the examples provided by users are not characteristic, it is difficult for TRANSREGEX to get the correct regexes. For instance, the positive examples from case #1 in TABLE VII belongs to both the incorrect one `[a-z]{3}[A-Za-z]{3,}[A-Za-z]{3,}` and ground truth `[a-z]{3}[A-Za-z]{3,}`, i.e., fail to distinguish between the incorrect one and ground truth. This may cause SYNCORR or RFIXER to fail to repair the incorrect regex. If the user further provides examples (e.g., a positive example `aaaAAA`) that can distinguish the two ones, our methods can easily get the correct one.
- **Wrong examples.** Wrong examples provided by users, resulting in that our algorithms are misled to synthesize regexes in the wrong direction. More concretely, our algorithms repair incorrect regexes in the wrong direction and even fix the correct regexes produced by S_2RE into the incorrect one. As case #2 in TABLE VII, if we only use the NL description, we can get the accurate regex `([A-Za-z]|[-!@#$%^&*()_.]|0){1,}&.{4,}`, which is the same as the ground truth. However, the user provides some wrong examples (e.g., a positive example `==;0#=#+k0-0`), which leads our algorithms to believe that the regex `([A-Za-z]|[-!@#$%^&*()_.]|0){1,}&.{4,}` is not accurate, and then our algorithm uses the wrong examples to fix the accurate regex into an incorrect one.
- **Very complex regexes or descriptions.** Similar to case #3 in TABLE VII, if the NL description is very long and complicated, or the target regex is very complex in terms of length and tree-depth, the regex synthesized by S_2RE may be very different from the target one. This kind of regex, which is very different from the target one and required very large fixes, is difficult for our algorithms to repair into a correct one in a limited time and space.

VI. RELATED WORK

A. Regex Synthesis

Regex synthesis from examples. The problem of automatic regex synthesis from examples has been explored in many domains [4], [16]–[20], [30], [31]. AlphaRegex [19] is a search-based algorithm for synthesizing simple regexes for introductory automata assignments. AlphaRegex exploits over/under-approximations to effectively prune out a large search space. However, all the regexes produced by AlphaRegex are over alphabets of size 2. *RegexGenerator++* [4], [30] is a state-of-the-art approach for the synthesis of regexes from positive and negative examples. The fact that *RegexGenerator++* utilizes genetic programming means that it is not guaranteed to generate a correct solution–*i.e.*, accepting all the positive examples while rejecting all the negative examples. Lots of existing works focus on XML schemas inference [16]–[18], [31], via resorting to infer regexes from examples. These approaches usually aim to tackle restricted forms of regexes from positive examples only. Li et al. [20] presented a novel algorithm FLASHREGEX to generate anti-ReDoS regexes from given positive and negative examples by reducing the ambiguity of these regexes and using SAT techniques.

However, one of the main issues in the above example-based techniques is the quality of the synthesis, i.e., whether it would generalize and correct for unseen examples. Specifically, if users can not provide sufficient and characteristic examples, the synthesized regexes will be under-fitting or over-fitting.

Regex synthesis from NL. Several works from the Natural Language Processing (NLP) community address the problem of generating regexes from (NL) specifications [10]–[13]. Kushman and Barzilay [10] introduced a technique for learning a probabilistic combinatory categorial grammar model to parse a NL description into a regex. To avoid domain-specific feature extraction, Locascio et al. [11] described the DEEP-REGEX model based on standard sequence-to-sequence (seq2seq) model, which regards the problem of generating regexes from NL descriptions as a direct *machine translation* task. To solve the problem that DEEP-REGEX model may not generate semantically correct regexes, the SEMREGEX model [12] based on reinforcement learning method was presented. It leverages DFA equivalence as a reward function to encourage the model to generate semantically correct regexes. To speeds up the training phase of SEMREGEX model, Park et al. [13] devised the SOFTREGEX model, which determines the equivalence of two regexes using deep neural networks.

There are three major bottlenecks in existing NLP-based techniques that affect the quality of synthesis: (i) Ambiguity and imprecision of NL . Ambiguity of NL results in predicting a regex embedding other meanings (resp. imprecision of NL affects the correctness of synthesis); (ii) Unknown words and rare words. There are unknown words or rare words in NL descriptions, which will lead to failure to generate correct regexes; (iii) Seq2Seq model. Seq2Seq-based approaches can only synthesize regexes similar in shape to the training data.

TABLE VII
EXAMPLES OF FAILED CASES GENERATED BY TRANSREGEX.

Failure Pair #1:		**Failure Pair #2:**		**Failure Pair #3:**	
Description	3 lower case letters followed by more than 3 letters.	Description	a string that consists of upper or lower case letters, special characters (-!@#$%&*()_.) or the number 0 and whose length is 4 or more characters long.	Description	a list of 3 semicolon separated strings, the first and second strings begin with any combination of 4 letters or digits, these strings end with 2 to 4 lower case letters, the third part is any number of capital letters.
Positive Examples	fsuhoRdKRUGrFIRj iptHLAdpPnKUXPrWo ...	Positive Examples	==;0#=#+k0-0 ✗ sy=;-0M0=!0T0f!;E ✗ ...	Positive Examples	O0Culdvs;0ctycf;H 80Oymq;2T2myvsz;DZG ...
Negative Examples	qmnVF qmnVa ...	Negative Examples	!;_ #&0-!7!Kx; ...	Negative Examples	mLNWmxydw;x9e5pdvf;A 5xSxe;xTC8jv;Q ...
Ground Truth	[a-z]{3}[A-Za-z]{3,}	Ground Truth:	([A-Za-z]|[-!@#$%&*()_.]|0){1,}&.{4,}	Ground Truth	[A-Za-z]|[0-9]{4}[a-z]{2,4};[A-Za-z]|[0-9]{4}[a-z]{2,4};[A-Z]{1,}
S_2RE	[a-z]{3}[A-Za-z]{3,}[A-Za-z]{3,}	S_2RE	([A-Za-z]|[-!@#$%&*()_.]|0){1,}&.{4,}	S_2RE	([A-Za-z]|[0-9]{1,})&.{4};[a-z]{2,4};[A-Z]{1,}
SYNCORR	[a-z]{3}[A-Za-z]{3,}[A-Za-z]{3,}	SYNCORR	.4,&(([-!@#$%&*()_.].*))	SYNCORR	Time out!
RFIXER	[a-z]{3}[A-Za-z]{3,}[A-Za-z]{3,}	RFIXER	Not supporting regexes with &	RFIXER	Not supporting regexes with &
Failure Type	**Uncharacteristic examples**	Failure Type	**Wrong examples**	Failure Type	**Very complex regexes or descriptions**

Regex synthesis from NL and examples. Ye et al. [25] proposed StructuredRegex, a new dataset for regex synthesis from NL and examples. Ye et al. [24] introduced a baseline model DEEP-REGEX + FILTER, which uses DEEP-REGEX as base model, and considers examples by simply filtering the k-best regexes. However, the positive and negative examples are not considered in the training and inference phase. The latest two works [24], [25] presented new two-step frameworks for regex synthesis from NL and examples. First, a semantic parser converts the NL description into an intermediate sketch. Then a synthesizer searches the regex space defined by the sketch and returns a concrete regex that is consistent with the given examples. Although both adopting a two-step paradigm, these works [24], [25] have an apparent limitation—incorrect sketches generated in the first step will subsequently induce the final regexes. In other words, the incorrection of sketches will be inherited by the synthesized regexes in the next step. On the other hand, our work overcomes this limitation: the second step of TRANSREGEX (i.e., *example-guided regex repair*) fixes the incorrect regex by examples if needed. In this manner, the inherited inconsistencies in our first step (i.e., *NLP-based regex synthesis*) will be fixed in our second step.

B. Regex Repair

Regex repair from examples. There are several works [20], [26]–[28] targeting at repairing regexes from examples. We discuss two main paradigms of them. In the first paradigm, works only consider either positive or negative examples. Li et al. [26] proposed ReLIE, which can modify complex regexes by rejecting the newly-input negative examples. By contrast, Rebele et al. [27] proposed a novel way to generalize a given regex so that it accepts the given positive examples. On the other hand, works in the second paradigm take both positive and negative examples into consideration. Pan et al. [28] designed RFIXER, a tool for repairing incorrect regexes using both examples. It took advantage of skeletons of regexes (i.e., sketches) to effectively prune out the search space, and it employed SMT solvers to efficiently explore the sets of possible character classes and numerical quantifiers. Our work applies RFIXER in our regex synthesis from NL and examples. Li et al. [20] described algorithm REPAIRINGRE based on Neighborhood Search (NS) to repair incorrect or ReDos-vulnerable regexes from positive and negative examples. Similar to REPAIRINGRE, our algorithm SYNCORR also uses NS to repair regexes, but the difference is that REPAIRINGRE uses automaton-directed repair, while we use regex-directed repair.

Like the above example-based synthesis algorithms, these repair algorithms also may cause under-fitting or over-fitting results. To alleviate the problems of under-fitting/over-fitting, our SYNCORR leverages some rewriting rules for sub-regexes abstraction to preserve the integrity of some small sub-regexes.

C. Program Synthesis

Programming by example (PBE). PBE techniques have been the subject of research in the past few decades [32] and successful paradigms for program synthesis, allowing end-users to construct and run new programs by providing examples of the intended program behavior [33]. Recently, PBE techniques have been successfully used for string transformations [34]–[36], data filtering [14], data structure manipulations [37], [38], table transformations [39], [40], SQL queries [41], [42], and MapReduce programs [43], [44].

Programming by NL (PBNL). There has been a lot of progress made in PBNL [45]. Specifically, several techniques have been proposed to translate NL descriptions into Python [46], [47], SQL queries [48]–[50], shell scripts [51], [52], spreadsheet formulas [53], test oracles [54], JavaScript function types [55], and Java expressions [56].

Program synthesis from NL and examples. Sinece program synthesis from NL and examples techniques can well overcome the shortcomings of PBE and PBNL techniques, at the same time, provide a more natural and friendly interface to the users, recent years they have been widely used in several areas, for example, string manipulation programs [23], [57], and program sketches [58]. In this paper, we focus on an important subtask of the program synthesis from NL and examples problem: synthesizing regexes from both NL and examples.

VII. Conclusion

We propose an automatic framework TransRegex, for synthesizing regular expressions from both natural language descriptions and examples. To the best of our knowledge, TransRegex is the first to treat the NLP-and-example-based regex synthesis problem as the problem of NLP-based synthesis with regex repair. For NLP-based synthesis, we devise a two-phase algorithm S_2RE which generates more valid regexes while having similar or higher accuracy than the state-of-the-art NLP-based models. While for regex repair, we present a novel algorithm SynCorr that leverages NS algorithms to guide the search for a target regex and uses rewriting rules to alleviate under-fitting/over-fitting and efficiently reduce the search space. The evaluation results demonstrate that the accuracy of our TransRegex is 17.4%, 35.8% and 38.9% higher than that of NLP-based works on the three publicly available datasets, respectively. Further, TransRegex can achieve higher accuracy than the state-of-the-art multi-modal works with 10% to 30% higher accuracy on all three datasets. The evaluation results also indicate TransRegex utilizing natural language and examples in a more effective way.

References

[1] L. G. M. IV, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, 2019, pp. 415–426.

[2] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu, "ReScue: crafting regular expression DoS attacks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 225–235.

[3] C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 405–416.

[4] A. Bartoli, A. D. Lorenzo, E. Medvet, and F. Tarlao, "Inference of Regular Expressions for Text Extraction from Examples," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 5, pp. 1217–1230, 2016.

[5] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 246–256.

[6] J. C. Davis, L. G. M. IV, C. A. Coghlan, F. Servant, and D. Lee, "Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 2019, pp. 443–454.

[7] 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, FTfJP 2012, Beijing, China, June 12, 2012*, 2012, pp. 20–26.

[8] X. Liu, Y. Jiang, and D. Wu, "A Lightweight Framework for Regular Expression Verification," in *19th IEEE International Symposium on High Assurance Systems Engineering, HASE 2019, Hangzhou, China, January 3-5, 2019*, 2019, pp. 1–8.

[9] B. Luo, Y. Feng, Z. Wang, S. Huang, R. Yan, and D. Zhao, "Marrying Up Regular Expressions with Neural Networks: A Case Study for Spoken Language Understanding," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, 2018, pp. 2083–2093.

[10] N. Kushman and R. Barzilay, "Using Semantic Unification to Generate Regular Expressions from Natural Language," in *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA*, 2013, pp. 826–836.

[11] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman, and R. Barzilay, "Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, 2016, pp. 1918–1923.

[12] Z. Zhong, J. Guo, W. Yang, J. Peng, T. Xie, J. Lou, T. Liu, and D. Zhang, "SemRegex: A Semantics-Based Approach for Generating Regular Expressions from Natural Language Specifications," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, 2018, pp. 1608–1618.

[13] J. Park, S. Ko, M. Cognetta, and Y. Han, "SoftRegex: Generating Regex from Natural Language Descriptions using Softened Regex Equivalence," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, 2019, pp. 6424–6430.

[14] X. Wang, S. Gulwani, and R. Singh, "FIDEX: filtering spreadsheet data using examples," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, 2016, pp. 195–213.

[15] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference*

on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, 2011, pp. 62–73.

[16] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren, "Inference of concise regular expressions and DTDs," *ACM Trans. Database Syst.*, vol. 35, no. 2, pp. 11:1–11:47, 2010.

[17] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren, "Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data," *ACM Trans. Web*, vol. 4, no. 4, pp. 14:1–14:32, 2010.

[18] D. D. Freydenberger and T. Kötzing, "Fast Learning of Restricted Regular Expressions and DTDs," *Theory Comput. Syst.*, vol. 57, no. 4, pp. 1114–1158, 2015.

[19] M. Lee, S. So, and H. Oh, "Synthesizing regular expressions from examples for introductory automata assignments," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, 2016, pp. 70–80.

[20] Y. Li, Z. Xu, J. Cao, H. Chen, T. Ge, S.-C. Cheung, and H. Zhao, "FlashRegex: deducing anti-ReDoS regexes from examples," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, 2020, pp. 659–671.

[21] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, 2020, pp. 487–502.

[22] Z. Zhong, J. Guo, W. Yang, T. Xie, J. Lou, T. Liu, and D. Zhang, "Generating Regular Expressions from Natural Language Specifications: Are We There Yet?" in *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 2018, pp. 791–794.

[23] M. H. Manshadi, D. Gildea, and J. F. Allen, "Integrating Programming by Example and Natural Language Programming," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, 2013.

[24] X. Ye, Q. Chen, X. Wang, I. Dillig, and G. Durrett, "Sketch-Driven Regular Expression Generation from Natural Language and Examples," *Trans. Assoc. Comput. Linguistics*, vol. To appear, 2020.

[25] X. Ye, Q. Chen, I. Dillig, and G. Durrett, "Benchmarking Multimodal Regex Synthesis with Complex Structures," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, 2020, pp. 6081–6094.

[26] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish, "Regular Expression Learning for Information Extraction," in *2008 Conference on Empirical Methods in Natural Language Processing, EMNLP 2008, Proceedings of the Conference, 25-27 October 2008, Honolulu, Hawaii, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*, 2008, pp. 21–30.

[27] T. Rebele, K. Tzompanaki, and F. M. Suchanek, "Adding Missing Words to Regular Expressions," in *Advances in Knowledge Discovery and Data Mining - 22nd Pacific-Asia Conference, PAKDD 2018, Melbourne, VIC, Australia, June 3-6, 2018, Proceedings, Part II*, 2018, pp. 67–79.

[28] R. Pan, Q. Hu, G. Xu, and L. D'Antoni, "Automatic repair of regular expressions," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 139:1–139:29, 2019.

[29] R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Mach. Learn.*, vol. 8, pp. 229–256, 1992.

[30] A. Bartoli, G. Davanzo, A. D. Lorenzo, E. Medvet, and E. Sorio, "Automatic Synthesis of Regular Expressions from Examples," *IEEE Computer*, vol. 47, no. 12, pp. 72–80, 2014.

[31] Y. Li, X. Zhang, J. Cao, H. Chen, and C. Gao, "Learning k-Occurrence Regular Expressions with Interleaving," in *Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Part II*, 2019, pp. 70–85.

[32] D. E. Shaw, W. R. Swartout, and C. C. Green, "Inferring LISP Programs From Examples," in *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*, 1975, pp. 260–267.

[33] K. Ellis and S. Gulwani, "Learning to Learn Programs from Examples: Going Beyond Program Structure," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, pp. 1638–1645.

[34] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, 2011, pp. 317–330.

[35] R. Singh, "BlinkFill: Semi-supervised Programming By Example for Syntactic String Transformations," *Proc. VLDB Endow.*, vol. 9, no. 10, pp. 816–827, 2016.

[36] R. Singh and S. Gulwani, "Learning Semantic String Transformations from Examples," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 740–751, 2012.

[37] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri, "Synthesizing transformations on hierarchically structured data," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 508–521.

[38] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 229–239.

[39] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, 2017, pp. 422–436.

[40] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 317–328.

[41] C. Wang, A. Cheung, and R. Bodík, "Synthesizing highly expressive SQL queries from input-output examples," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, 2017, pp. 452–466.

[42] S. Zhang and Y. Sun, "Automatically synthesizing SQL queries from input-output examples," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 2013, pp. 224–234.

[43] C. Smith and A. Albarghouthi, "MapReduce program synthesis," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 326–340.

[44] M. B. S. Ahmad and A. Cheung, "Leveraging Parallel Data Processing Frameworks with Verified Lifting," in *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016*, 2016, pp. 67–83.

[45] S. Gulwani, O. Polozov, and R. Singh, "Program Synthesis," *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017.

[46] P. Yin and G. Neubig, "A Syntactic Neural Model for General-Purpose Code Generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 440–450.

[47] M. Rabinovich, M. Stern, and D. Klein, "Abstract Syntax Networks for Code Generation and Semantic Parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 1139–1149.

[48] P. Huang, C. Wang, R. Singh, W. Yih, and X. He, "Natural Language to Structured Query Generation via Meta-Learning," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, 2018, pp. 732–738.

[49] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "SQLizer: query synthesis from natural language," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 63:1–63:26, 2017.

[50] Y. Zeng, Y. Gao, J. Guo, B. Chen, Q. Liu, J. Lou, F. Teng, and D. Zhang, "RECPARSER: A Recursive Semantic Parsing Framework for Text-to-SQL Task," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 2020, pp. 3644–3650.

[51] H. Li, Y. Wang, J. Yin, and G. Tan, "SmartShell: Automated Shell Scripts Synthesis from Natural Language," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 2, pp. 197–220, 2019.

[52] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst, "NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System," in *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018*, 2018.

[53] S. Gulwani and M. Marron, "NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 803–814.

[54] M. Motwani and Y. Brun, "Automatically generating precise Oracles from structured natural language specifications," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 188–199.

[55] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: inferring JavaScript function types from natural language information," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 304–315.

[56] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, 2015, pp. 416–432.

[57] M. Raza, S. Gulwani, and N. Milic-Frayling, "Compositional Program Synthesis from Natural Language and Examples," in *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 2015, pp. 792–800.

[58] M. I. Nye, L. B. Hewitt, J. B. Tenenbaum, and A. Solar-Lezama, "Learning to Infer Program Sketches," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019, pp. 4861–4870.