FOREST: An Interactive Multi-tree Synthesizer for Regular Expressions*

(✉) Margarida Ferreira[1,2], Miguel Terra-Neves[2], Miguel Ventura[2], Inês Lynce[1], and Ruben Martins[3]

[1] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
{margaridaacferreira, ines.lynce}@tecnico.ulisboa.pt
[2] OutSystems, Linda-a-Velha, Portugal
{miguel.neves, miguel.ventura}@outsystems.com
[3] Carnegie Mellon University, Pittsburgh, USA
rubenm@cs.cmu.edu

Abstract Form validators based on regular expressions are often used on digital forms to prevent users from inserting data in the wrong format. However, writing these validators can pose a challenge to some users.
We present FOREST, a regular expression synthesizer for digital form validations. FOREST produces a regular expression that matches the desired pattern for the input values and a set of conditions over capturing groups that ensure the validity of integer values in the input. Our synthesis procedure is based on enumerative search and uses a Satisfiability Modulo Theories (SMT) solver to explore and prune the search space. We propose a novel representation for regular expressions synthesis, multi-tree, which induces patterns in the examples and uses them to split the problem through a divide-and-conquer approach. We also present a new SMT encoding to synthesize capture conditions for a given regular expression. To increase confidence in the synthesized regular expression, we implement user interaction based on distinguishing inputs.
We evaluated FOREST on real-world form-validation instances using regular expressions. Experimental results show that FOREST successfully returns the desired regular expression in 70% of the instances and outperforms REGEL, a state-of-the-art regular expression synthesizer.

1 Introduction

Regular expressions (also known as regexes) are powerful mechanisms for describing patterns in text with numerous applications. One notable use of regexes is to perform real-time validations on the input fields of digital forms. Regexes help filter invalid values, such as typographical mistakes ('typos') and format inconsistencies. Aside from validating the format of form input strings, regular expressions can be coupled with capturing groups. A capturing group is a sub-regex within a regex that is indicated with parenthesis and captures the text

* This work was supported by NSF award CCF-1762363 and through FCT under project UIDB/50021/2020, and project ANI 045917 funded by FEDER and FCT.

matched by the sub-regex inside them. Capturing groups are used to extract information from text and, in the domain of form validation, they can be used to enforce conditions over values in the input string. In this paper, we focus on the capture of integer values in input strings, and we use the notation $\$i, i \in \{0, 1, ...\}$ to refer to the integer value of the text captured by the $(i+1)^{\text{th}}$ group.

Form validations often rely on complex regexes which require programming skills that not all users possess. To help users write regexes, prior work has proposed to synthesize regular expressions from natural language [1,9,12,27] or from positive and negative examples [1,7,10,26]. Even though these techniques assist users in writing regexes for search and replace operations, they do not specifically target digital form validation and do not take advantage of the structured format of the data.

In this paper, we propose FOREST, a new program synthesizer for regular expressions that targets digital form validations. FOREST takes as input a set of examples and returns a regex validation. FOREST accepts three types of examples: (i) **valid examples**: correct values for the input field, (ii) **invalid examples**: incorrect values for the input field due to their *format*, and (iii) **conditional invalid examples** (optional): incorrect values for the input field due to their *values*. FOREST outputs a regex validation, consisting of two components: (i) a **regular expression** that matches all valid and none of the invalid examples and (ii) **capture conditions** that express integer conditions that are satisfied by the values on all the valid but none of the conditional invalid examples.

Motivating Example. Suppose a user is writing a form where one of the fields is a date that must respect the format `DD/MM/YYYY`. The user wants to accept:

19/08/1996	22/09/2000	29/09/2003
26/10/1998	01/12/2001	31/08/2015

But not:

19/08/96	22.09.2000	29/9/2003
26-10-1998	1/12/2001	2015/08/31

A regular expression can be used to enforce this format. Instead of writing it, the user may simply use the two sets of values as *valid* and *invalid* input examples to FOREST, who will output the regex `[0-9]{2}/[0-9]{2}/[0-9]{4}`.

Additionally, if the user wants to validate not only the format, but also the values in the date, we can consider as *conditional invalid* the examples:

33/08/1996	22/13/2000	12/31/2003
26/00/1998	00/12/2001	52/03/2015

FOREST will output a regex validation complete with conditions over capturing groups that ensures only valid values are inserted as the day and month: `([0-9]{2})/([0-9]{2})/[0-9]{4}`, $\$0 \leq 31 \wedge \$0 \geq 1 \wedge \$1 \leq 12 \wedge \$1 \geq 1$.

As we can see in the motivating example, data inserted into digital forms is usually structured and shares a common pattern among the valid examples. In this example, the data has the shape `dd/dd/dddd` where `d` represents a digit. This

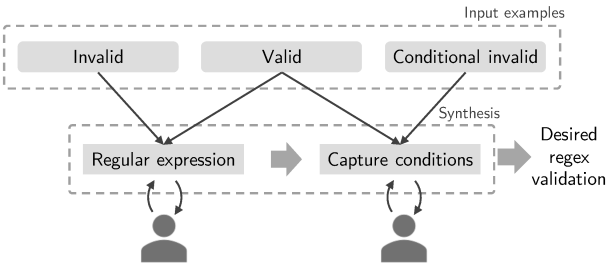


Figure 1: Regex synthesis

contrasts with general regexes for search and replace operations that are often performed over unstructured text. FOREST takes advantage of this structure by automatically detecting these patterns and using a divide-and-conquer approach to split the expression into simpler sub-expressions, solving them independently, and then merging their information to obtain the final regular expression. Additionally, FOREST computes a set of capturing groups over the regular expression, which it then uses to synthesize integer conditions that further constrain the accepted values for that form field.

Input-output examples do not require specialized knowledge and are accessible to users. However, there is one downside to using examples as a specification: they are ambiguous. There can be solutions that, despite matching the examples, do not produce the desired behavior in situations not covered in them. The ambiguity of input-output examples raises the necessity of selecting one among multiple candidate solutions. To this end, we incorporate a user interaction model based on distinguishing inputs for both the synthesis of the regular expressions and the synthesis of the capture conditions.

In summary, this paper makes the following contributions:

- We propose a multi-tree SMT representation for regular expressions that leverages the structure of the input to apply a divide-and-conquer approach.
- We propose a new method to synthesize capturing groups for a given regular expression and integer conditions over the resulting captures.
- We implemented a tool, FOREST, that interacts with the user to disambiguate the provided specification. FOREST is evaluated on real-world instances and its performance is compared with a state-of-the-art synthesizer.

2 Synthesis Algorithm Overview

The task of automatically generating a program that satisfies some desired behavior expressed as a high-level specification is known as Program Synthesis. Programming by Example (PBE) is a branch of Program Synthesis where the desired behavior is specified using input-output examples.

Our synthesis procedure is split into two stages, each relative to an output component. First, FOREST synthesizes the regular expression, which is the basis

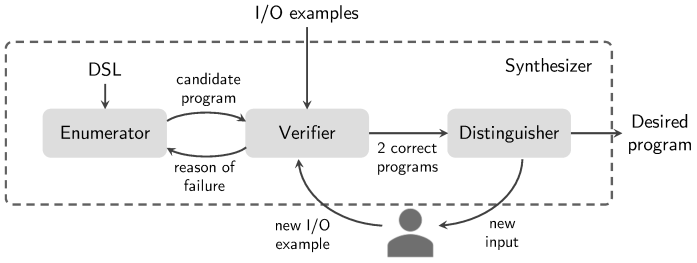


Figure 2: Interactive enumerative search

for the synthesis of capturing groups. Secondly, FOREST synthesizes the capture conditions, by first computing a set of capturing groups and then the conditions to be applied to the resulting captures. The synthesis stages are detailed in sections 3 and 4. Figure 1 shows the regex validation synthesis pipeline. Both stages of our synthesis algorithm employ enumerative search, a common approach to solve the problem of program synthesis [4,5,10,17,21]. The enumerative search cycle is depicted in Figure 2.

There are two key components for program enumeration: the *enumerator* and the *verifier*. The *enumerator* successively enumerates programs from the a predefined Domain Specific Language (DSL). Following the Occam's razor principle, programs are enumerated in increasing order of complexity. The DSL defines the set of operators that can be used to build the desired program. FOREST dynamically constructs its DSL to fit the problem at hand: it is as restricted as possible, without losing the necessary expressiveness. The regular expression DSL construction procedure is detailed in section 3.1.

For each enumerated program, the *verifier* subsequently checks whether it satisfies the provided examples. Program synthesis applications generate very large search spaces; nevertheless, the search space can be significantly reduced by pruning several infeasible expressions along with each incorrect expression found. In the first stage of the regex validation synthesis, the enumerated programs are regular expressions. The enumeration and pruning of regular expressions is described in section 3.2. In the second stage of regex validation synthesis, we deal with the enumeration of capturing groups over a pre-existing regular expression. This process is described in section 4.1.

To circumvent the ambiguity of input-output examples, FOREST implements an interaction model. A new component, the *distinguisher*, ascertains, for any two given programs, whether they are equivalent. When FOREST finds two different validations that satisfy all examples, it creates a *distinguishing input*: a new input that has a different output for each validation. To disambiguate between two programs, FOREST shows the new input to the user, who classifies it as valid or invalid, effectively choosing one program over the other. The new input-output pair is added to the examples, and the enumeration process continues until there is only one solution left. This interactive cycle is described for the synthesis of regular expressions in section 3.3 and capture conditions in section 4.3.

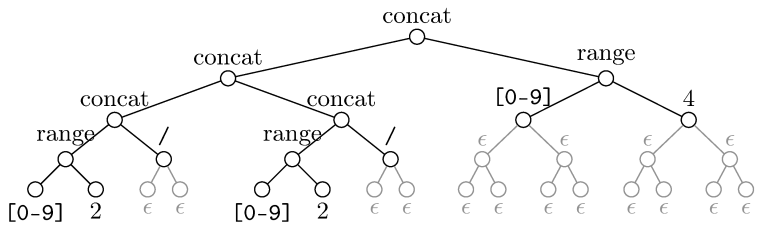


Figure 3: `[0-9]{2}/[0-9]{2}/[0-9]{4}` represented as a k-tree with $k = 2$

3 Regular Expressions Synthesis

In this section we describe the enumerative synthesis procedure that generates a regular expression that matches all valid examples and none of the invalid.

3.1 Regular Expressions DSL

Before the synthesis procedure starts, we define which operators can be used to build the desired regular expression and the values each operator can take as argument. FOREST's regular expression DSL includes the regex union and concatenation operators, as well as several regular expression quantifiers:

- *Kleene* closure: r^* matches r zero or more times,
- positive closure: r^+ matches r one or more times,
- option: $r?$ matches r zero or one times,
- ranges: $r\{m\}$ matches r exactly m times, and $r\{m, n\}$ matches r at least m times and at most n times.

The possible values for the range operators are limited depending on the valid examples provided by the user. For the single-valued range operator, $r\{m\}$, we consider only the integer values such that $2 \leq m \leq l$, where l is the length of the longest valid example string. In the two-valued range operator, $r\{m, n\}$, the values of m and n are limited to integers such that $0 \leq m < n \leq l$. The tuple (0,1) is not considered, since it is equivalent to the option quantifier: $r\{0, 1\} = r?$.

All operators can be applied to regex literals or composed with each other to form more complex expressions. The regex literals considered in the synthesis procedure include the individual letters, digits or symbols present in the examples and all character classes that contain them. The character classes contemplated in the DSL are `[0-9]`, `[A-Z]`, `[a-z]` and all combinations of those, such as `[A-Za-z]` or `[0-9A-Za-z]`. Additionally, `[0-9A-F]` and `[0-9a-f]` are used to represent hexadecimal numbers.

3.2 Regex Enumeration

To enumerate regexes, the synthesizer requires a structure capable of representing every feasible expression. We use a tree-based representation of the search

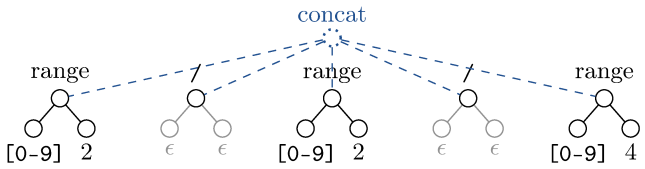


Figure 4: `[0-9]{2}/[0-9]{2}/[0-9]{4}` represented as a multi-tree with $n = 5$ and $k = 2$, resulting from the concatenation of 5 simpler regexes

space. A k-tree of depth d is a tree in which every internal node has exactly k children and every leaf node is at depth d. A program corresponds to an assignment of a DSL construct to each tree node, the node's descendants are the construct's arguments. If k is the greatest arity among all DSL constructs, then a k-tree of depth d can represent all programs of depth up to d in that DSL. The arity of constructs in FOREST's regex DSLs is at most 2, so all regexes in the search space can be represented using 2-trees. To allow constructs with arity smaller than k, some children nodes are assigned the *empty* symbol, ϵ. In Figure 3, the regex from the motivating example, `[0-9]{2}/[0-9]{2}/[0-9]{4}`, is represented as a 2-tree of depth 5.

To explore the search space in order of increasing complexity, we enumerate k-trees of lower depths first and progressively increase the depth of the trees as previous depths are exhausted. The enumerator encodes the k-tree as an SMT formula that ensures the program is well-typed. A model that satisfies the formula represents a valid regex. Due to space constraints we omit the k-tree encoding but further details can be found in the literature [2,17].

Multi-tree representation. We considered several validators for digital forms and observed that many regexes in this domain are the concatenation of relatively simple regexes. However, the successive concatenation of simple regexes quickly becomes complex in its k-tree representation. Recall the regex for date validation presented in the motivating example: `[0-9]{2}/[0-9]{2}/[0-9]{4}`. Even though this is the concatenation of 5 simple sub-expressions, each of depth at most 2, its representation as a k-tree has depth 5, as shown in Figure 3.

The main idea behind the multi-tree constructs is to allow the number of concatenated sub-expressions to grow without it reflecting exponentially on the encoding. The multi-tree structure consists of n k-trees, whose roots are connected by an artificial root node, interpreted as an n-ary concatenation operator. This way, we are able to represent regexes using fewer nodes. Figure 4 is the multi-tree representation of the same regex as Figure 3, and shows that the multi-tree construct can represent this expression using half the nodes.

The k-tree enumerator successively explores k-trees of increasing depth. However, multi-tree has two measures of complexity: the depth of the trees, d, and the number of trees, n. FOREST employs two different methods for increasing these values: static multi-tree and dynamic multi-tree.

Static multi-tree. In the static multi-tree method, the synthesizer fixes n and progressively increases d. To find the value of n, there is a preprocessing step, in which FOREST identifies patterns in the valid examples. This is done by first identifying substrings common to all examples. A substring is considered a dividing substring if it occurs exactly the same number of times and in the same order in all examples. Then, we split each example before and after the dividing substrings. Each example becomes an array of n strings.

Example 1. Consider the valid examples from the motivating example. In these examples, '/' is a dividing substring because it occurs in every example, and exactly twice in each one. '0' is a common substring but not a dividing substring because it does not occur the same number or times in all examples. After splitting on '/', each example becomes a tuple of 5 strings:

('19', '/', '08', '/', '1996')	('01', '/', '12', '/', '2001')
('26', '/', '10', '/', '1998')	('29', '/', '09', '/', '2003')
('22', '/', '09', '/', '2000')	('31', '/', '08', '/', '2015')

Then, we apply the multi-tree method with n trees. For every $i \in \{1, ..., n\}$, the i^{th} sub-tree represents a regex that matches all strings in the i^{th} position of the split example tuples and the concatenation of the n regexes will match the original example strings. Since each tree is only synthesizing a part of the original input strings, a reduced DSL is recomputed for each tree.

Dynamic multi-tree. The dynamic multi-tree method is employed when the examples cannot be split because there are no dividing substrings. In this scenario, the enumerator will still use a multi-tree construct to represent the regex. However, the number of trees is not fixed and all trees use the original, complete DSL. A multi-tree structure with n k-trees of depth d has $n \times (k^d - 1)$ nodes. FOREST enumerates trees with different values of (n, d) in increasing order of number of nodes, starting with $n = 1$ and $d = 2$, a simple k-tree of depth 2.

Pruning. We prune regexes which are provably equivalent to others in the search space by using algebraic rules of regular expressions like the following:

$$(r*)* \equiv r* \qquad (r?)? \equiv r? \qquad (r+)+ \equiv r+$$
$$(r+)* \equiv (r*)+ \equiv r* \qquad (r?)* \equiv (r*)? \equiv r* \qquad (r?)+ \equiv (r+)? \equiv r*$$
$$(r*)\{m\} \equiv (r\{m\})* \qquad (r+)\{m\} \equiv (r\{m\})+ \qquad (r?)\{m\} \equiv (r\{m\})?$$
$$r\{n\}\{m\} \equiv r\{m\}\{n\} \equiv r\{m \times n\}$$

To prevent the enumeration of equivalent regular expressions, we add SMT constraints that block all but one possible representation of each regex. Take, for example, the equivalence $(r?)+ \equiv r*$. We want to consider only one way to represent this regex, so we add a constraint to block the construction $(r?)+$ for any regex r. Another such equivalence results from the idempotence of union:

$r|r = r$. To prevent the enumeration of expressions of the type $r|r$, every time the union operator is assigned to a node i, we force the sub-tree underneath i's left child to be different from the sub-tree underneath i's right child by at least one node. When we enumerate a regex that is not consistent with the examples, it is eliminated from the search space. Along with the incorrect regex, we want to eliminate regexes that are equivalent to it. The union operator in the regular expressions DSL is commutative: $r|s = s|r$, for any regexes r and s. Thus, whenever an expression containing $r|s$ is discarded, we eliminate the expression that contains $s|r$ in its place as well.

3.3 Regex Disambiguation

To increase confidence in the synthesizer's solution, FOREST disambiguates the specification by interacting with the user. We employ an interaction model based on distinguishing inputs, which has been successfully used in several synthesizers [11,24,25,14]. To produce a distinguishing input, we require an SMT solver with a regex theory, such as Z3 [15,23]. Upon finding two regexes that satisfy the user-provided examples, r_1 and r_2, we use the SMT solver to solve the formula:

$$\exists s : r_1(s) \neq r_2(s), \tag{1}$$

where $r_1(s)$ (resp. $r_2(s)$) is True if and only if r_1 (resp. r_2) matches the string s. A string s that satisfies (1) is a distinguishing input. FOREST asks the user to classify this input as valid or invalid, and s is added to the respective set of examples, thus eliminating either r_1 or r_2 from the search space. After the first interaction, the synthesis procedure continues only until the end of the current depth and number of trees.

4 Capturing Groups Synthesis

In this section we describe the synthesis procedure of the second component of a regex validation: a set of integer conditions over captured values that are satisfied by all valid examples but none of the conditional invalid examples.

4.1 Capturing Groups Enumeration

To enumerate capturing groups, FOREST starts by identifying the regular expression's atomic sub-regexes: the smallest sub-regexes whose concatenation results in the original complete regex. For example, `[0-9]{2}` is an atomic sub-regex: there are no smaller sub-regexes whose concatenation results in it. It does not make sense to place a capturing group inside atomic sub-regexes: `([0-9]){2}` does not have a clear meaning. Once identified, the atomic sub-regexes are placed in an ordered list. Enumerating capturing groups over the regular expression is done by enumerating non-empty disjoint sub-lists of this list. The elements inside each sub-list form a capturing group.

Example 2. Recall the date regex: `[0-9]{2}/[0-9]{2}/[0-9]{4}`. The respective list of atomic sub-regexes is [`[0-9]{2}`, `/`, `[0-9]{2}`, `/`, `[0-9]{4}`]. The following are examples of sub-lists of the atomic sub-regexes list and their resulting capturing groups:

[[`[0-9]{2}`], `/`, `[0-9]{2}`, `/`, `[0-9]{4}`] → `([0-9]{2})/[0-9]{2}/[0-9]{4}`

[[`[0-9]{2}`], `/`, [`[0-9]{2}`], `/`, [`[0-9]{4}`]] → `([0-9]{2})/([0-9]{2})/([0-9]{4})`

4.2 Capture Conditions Synthesis

To compute capture conditions, we need all conditional invalid examples to be matched by the regular expression. After, capturing groups are enumerated as described in section 4.1. The number of necessary capturing groups is not known beforehand, so we enumerate capturing groups in increasing number.

A capture condition is a 3-tuple: it contains the captured text, an integer comparison operator and an integer argument. FOREST considers only two integer comparison operators, $\leq$ and $\geq$. However, the algorithm can be easily expanded to include other operators. Let $\mathcal{C}$ be a set of capturing groups and $\mathcal{C}(x)$ the integer captures that result from applying $\mathcal{C}$ to example string x. Let $\mathcal{D}_\mathcal{C}$ be the set of all possible capture conditions over capturing groups $\mathcal{C}$. $\mathcal{D}_\mathcal{C}$ results from combining each capturing group with each integer operator. Finally, let $\mathcal{V}$ be the set of all valid examples, $\mathcal{I}$ the set of all conditional invalid examples, and $\mathcal{X} = \mathcal{V} \cup \mathcal{I}$ the union of these two sets.

Given capturing groups $\mathcal{C}$, FOREST uses Maximum Satisfiability Modulo Theories (MaxSMT) to select from $\mathcal{D}_\mathcal{C}$ the minimum set of conditions that are satisfied by all valid examples and none of the conditional invalid. To encode the problem, we define two sets of Boolean variables. First, we define $s_{cap,x}$ for every $cap \in \mathcal{C}(x)$ and $x \in \mathcal{X}$. $s_{cap,x}$ = True if capture cap in example x satisfies all used conditions that refer to it. We also define u_{cond} for all $cond \in \mathcal{D}_\mathcal{C}$. u_{cond} = True means condition $cond$ is used in the solution. Additionally, we define a set of integer variables b_{cond}, for all conditions $cond \in \mathcal{D}_\mathcal{C}$ that represent the integer argument present in each condition.

Let SMT($cond, x$) be the SMT representation of condition $cond$ for example x: the capture is an integer value, and the integer argument is the corresponding b_{cond} variable. Let $\mathcal{D}_{cap} \subseteq \mathcal{D}_\mathcal{C}$ be the set of capture conditions that refer to capture cap. Constraint (2) states that a capture cap in example x satisfies all conditions if and only if for every condition that refers to cap either it is not used in the solution or it is satisfied for the value of that capture in that example:

$$s_{cap,x} \leftrightarrow \bigwedge_{cond \in \mathcal{D}_{cap}} u_{cond} \rightarrow \text{SMT}(cond, x). \tag{2}$$

Example 3. Recall the first valid string from the motivating example: x_0 = "19/08/1996". Suppose FOREST has already synthesized the desired regular expression and enumerated a capturing group that corresponds to the day: `([0-9]{2})/[0-9]{2}/[0-9]{4}`. Let $cond_0$ and $cond_1$ be the conditions that

refer to the first (and only) capturing group, \$0, and operators $\leq$ and $\geq$ respectively. The SMT representation for $cond_0$ and x_0 is $\text{SMT}(cond_0, x_0) = 19 \leq b_{cond_0}$. Constraint (2) is:

$$s_{0,x_0} \leftrightarrow (u_{cond_0} \rightarrow 19 \leq b_{cond_0}) \wedge (u_{cond_1} \rightarrow 19 \geq b_{cond_1}).$$

Then, we ensure the used conditions are satisfied by all valid examples and none of the conditional invalid examples:

$$\bigwedge_{x \in \mathcal{V}} \bigwedge_{cap \in \mathcal{C}(x)} s_{cap,x} \wedge \bigwedge_{x \in \mathcal{I}} \bigvee_{cap \in \mathcal{C}(x)} \neg s_{cap,x}. \tag{3}$$

Since we are looking for the minimum set of capture conditions, we add soft clauses to penalize the usage of capture conditions in the solution:

$$\bigwedge_{cond \in \mathcal{D}_\mathcal{C}} \neg\, u_{cond}. \tag{4}$$

We consider part of the solution only the capture conditions whose u_{cond} is True in the resulting SMT model. We also extract the values of the integer arguments in each condition from the model values of the b_{cond} variables.

4.3 Capture Conditions Disambiguation

To ensure the solution meets the user's intent, FOREST disambiguates the specification using, once again, a procedure based on distinguishing inputs. Once FOREST finds two different sets of capture conditions $\mathcal{S}_1$ and $\mathcal{S}_2$ that satisfy the specification, we look for a distinguishing input: a string c which satisfies all capture conditions in $\mathcal{S}_1$, but not those in $\mathcal{S}_2$, or vice-versa. First, to simplify the problem, FOREST eliminates from $\mathcal{S}_1$ and $\mathcal{S}_2$ conditions which are present in both: these are not relevant to compute a distinguishing input. Let $\mathcal{S}_1^*$ (resp. $\mathcal{S}_2^*$) be the subset of $\mathcal{S}_1$ (resp. $\mathcal{S}_2$) containing only the distinguishing conditions, i.e., the conditions that differ from those in $\mathcal{S}_2$ (resp. $\mathcal{S}_1$).

We do not compute the distinguishing string c directly. Instead, we compute the integer value of the distinguishing captures in c, i.e., the captures that result from applying the regular expression and its capturing groups to the distinguishing input string. We define $|\mathcal{C}|$ integer variables, c_i, which correspond to the values of the distinguishing captures: $c_0, c_1, ..., c_{|\mathcal{C}|} = \mathcal{C}(c)$.

As before, let $\text{SMT}(cond, c)$ be the SMT representation of each condition $cond$. Each capture in $\mathcal{C}(c)$ is represented by its respective c_i, the operator maintains it usual semantics and the integer argument is its value in the solution to which the condition belongs. Constraint (5) states that c satisfies the conditions in one solution but not the other.

$$\bigwedge_{cond \in \mathcal{S}_1^*} \text{SMT}(cond, c) \quad \neq \quad \bigwedge_{cond \in \mathcal{S}_2^*} \text{SMT}(cond, c). \tag{5}$$

In the end, to produce the distinguishing string c, FOREST picks an example from the valid set, applies the regular expression with the capturing groups to it, and replaces its captures with the model values for c_i.

FOREST asks the user to classify c as valid or invalid. Depending on the user's answer, c is added as a valid or conditional invalid example, effectively eliminating either S_1 or S_2 from the search space.

Example 4. Recall the examples from the motivating example. No example invalidates a date with the day 32, so FOREST will find two correct sets of capture conditions over the regular expression `([0-9]{2})/([0-9]{2})/[0-9]{4}`: $\mathcal{S}_1 = \{\$0 \leq 31, \$0 \geq 1, \$1 \leq 12, \$1 \geq 1\}$, and $\mathcal{S}_2 = \{\$0 \leq 32, \$0 \geq 1, \$1 \leq 12, \$1 \geq 1\}$. First, we define two sets containing only the distinguishing captures: $\mathcal{S}_1^* = \{\$0 \leq 31\}$ and $\mathcal{S}_2^* = \{\$0 \leq 32\}$. Then, to find c_0, the value of the distinguishing capture for these solutions, we solve the constraint:

$$\exists c_0 : c_0 \leq 31 \neq c_0 \leq 32$$

and get the value $c_0 = 32$ which satisfies $\mathcal{S}_2^*$ (and $\mathcal{S}_2$), but not $\mathcal{S}_1^*$ (or $\mathcal{S}_1$).

If we pick the first valid example, "19/08/1996" as basis for c, the respective distinguishing input is $c =$ "32/08/1996". Once the user classifies c as invalid, c is added as a conditional invalid example and $\mathcal{S}_2$ is removed from consideration.

5 Related Work

Program synthesis has been successfully used in many domains such as string processing [8,19,7,26], query synthesis [11,25,17], data wrangling [2,5], and functional synthesis [3,6]. In this section, we discuss prior work on the synthesis of regular expressions [10,1] that is most closely related to our approach.

Previous approaches that perform general string processing [7,26] restrict the form of the regular expressions that can be synthesized. In contrast, we support a wide range of regular expressions operators, including the Kleene closure, positive closure, option, and range. More recent work that targets the synthesis of regexes is done by ALPHAREGEX [10] and REGEL [1]. ALPHAREGEX performs an enumerative search and uses under- and over-approximations of regexes to prune the search space. However, ALPHAREGEX is limited to the binary alphabet and does not support the kind of regexes that we need to synthesize for form validations. REGEL [1] is a state-of-the-art synthesizer of regular expressions based on a multi-modal approach that combines input-output examples with a natural language description of user intent. They use natural language to build hierarchical sketches that capture the high-level structure of the regex to be synthesized. In addition, they prune the search space by using under- and over-approximations and symbolic regexes combined with SMT-based reasoning. REGEL's evaluation [1] has shown that their PBE engine is an order of magnitude faster than ALPHAREGEX. While REGEL targets more general regexes that are suitable for search and replace operations, we target regexes for form validation which usually have more structure. In our approach, we take advantage

of this structure to split the problem into independent subproblems. This can be seen as a special case of sketching [22] where each hole is independent. Our pruning techniques are orthogonal to the ones used by Regel and are based on removing equivalent regexes prior to the search and to remove equivalent failed regexes during search. To the best of our knowledge, no previous work focused on the synthesis of conditions over capturing groups.

Instead of using input-output examples, there are other approaches that synthesize regexes solely from natural language [9,12,27]. We see these approaches as orthogonal to ours and expect that Forest can be improved by hints provided by a natural language component such as was done in Regel.

6 Experimental Results

Implementation. Forest is open-source and publicly available at https://github.com/Marghrid/FOREST. Forest is implemented in Python 3.8 on top of Trinity, a general-purpose synthesis framework [13]. All SMT formulas are solved using the Z3 SMT solver, version 4.8.9 [15]. To find distinguishing inputs in regular expression synthesis, Forest uses Z3's theory of regular expressions [23]. To check the enumerated regexes against the examples, we use Python's regex library [18]. The results presented herein were obtained using an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz, with 64GB of RAM, running Debian GNU/Linux 10. All processes were run with a time limit of one hour.

Benchmarks. To evaluate Forest, we used 64 benchmarks based on real-world form-validation regular expressions. These were collected from regular expression validators in validation frameworks and from `regexlib` [20], where users can upload their own regexes. Among these 64 benchmarks there are different formats: national IDs, identifiers of products, date and time, vehicle registration numbers, postal codes, email and phone numbers. For each benchmark, we generated a set of string examples. All 64 benchmarks require a regular expression to validate the examples, but only 7 require capture conditions. On average, each instance is composed of 13.2 valid examples (ranging from 4 to 33) and 9.3 invalid (ranging from 2 to 38). The 7 instances that target capture conditions have on average 6.3 conditional invalid examples (ranging from 4 to 8).

The goal of this experimental evaluation is to answer the following questions:
Q1: How does Forest compare against Regel? (section 6.1)
Q2: How does pruning affect multi-tree's time performance? (section 6.2)
Q3: How does static multi-tree improve on dynamic multi-tree? (section 6.2)
Q4: How does multi-tree compare against other encodings? (section 6.3)
Q5: How many examples are required to return a correct solution? (section 6.4)

Forest, by default, uses static multi-tree (when possible) with pruning. It correctly solves 31 benchmarks (48%) in under 10 seconds. In one hour, Forest solves 47 benchmarks (73%), with 96% accuracy: only two solutions did not correspond to the desired regex validation. Forest disambiguates only among programs at the same depth, and so if the first solution is not at the same depth

Table 1: Comparison of time performance using different synthesis methods

Timeout (s)	**10**	**60**	**3600**
FOREST (with interaction)	31	39	47
FOREST's 1st regex (no interaction)	40	46	50
Multi-tree w/o pruning	20	32	38
Dynamic-only multi-tree	5	10	18
k-tree	4	9	15
Line-based (w/o pruning)	4	4	12
REGEL	29	38	47
REGEL PBE	5	7	23

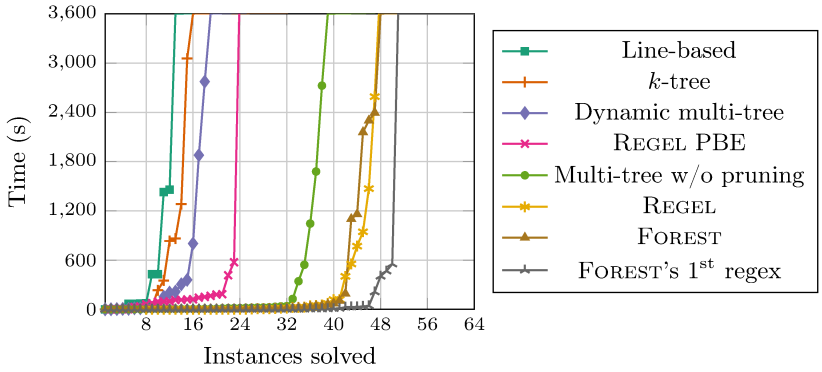


Figure 5: Instances solved using different methods

as the correct one, the correct solution is never found. After 1 hour of running time, FOREST is interrupted, but it prints its current best validation before terminating. After the timeout, FOREST returned 3 more regexes, 2 of which the correct solution for the benchmark. In all benchmarks to which FOREST returns a solution, the first matching regular expression is found in under 10 minutes. In 40 benchmarks, the first regex is found in under 10 seconds. The rest of the time is spent disambiguating the input examples. FOREST interacts with the user to disambiguate the examples in 27 benchmarks. Overall, it asks 1.8 questions and spends 38.6 seconds computing distinguishing inputs, on average.

Regarding the synthesis of capture conditions, in 5 of the benchmarks, we need only 2 capturing groups and at most 4 conditions. In these instances, the conditions' synthesis takes under 2 seconds. The remaining 2 benchmarks need 4 capturing groups and take longer: 99 seconds to synthesize 4 conditions and 1068 seconds for 6 conditions. During capture conditions synthesis, FOREST interacts 7.14 times and takes 0.1 seconds to compute distinguishing inputs, on average.

Table 1 shows the number of instances solved in under 10, 60 and 3600 seconds using FOREST, as well as using the different variations of the synthesizer which will be described in the following sections. The cactus plot in Figure 5

shows the cumulative synthesis time on the y-axis plotted against the number of benchmarks solved by each variation of FOREST (on the x-axis). The synthesis methods that correspond to lines more to the right of the plot are able to solve more benchmarks in less time. We also compare solving times with REGEL [1]. REGEL takes as input examples and a natural description of user intent. We consider not only the complete REGEL synthesizer, but also the PBE engine of REGEL by itself, which we denote by REGEL PBE.

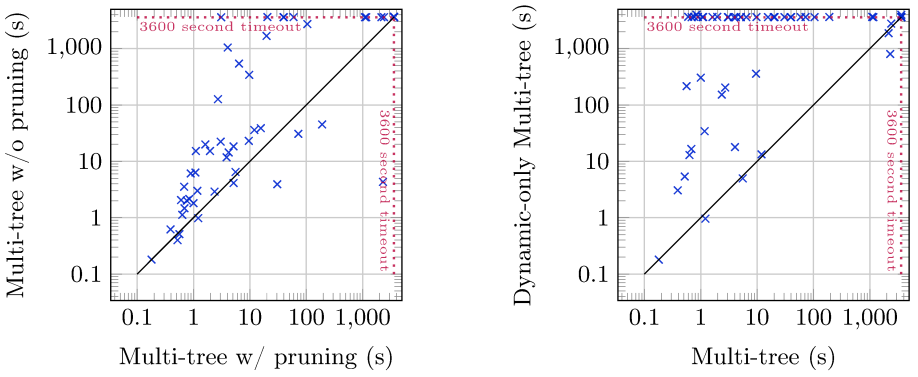
6.1 Comparison with REGEL

As mentioned in section 5, REGEL's synthesis procedure is split into two steps: sketch generation (using a natural language description of desired behavior) and sketch completion (using input-output examples). To compare REGEL and FOREST, we extended our 64 form validation benchmarks with a natural language description. To assess the importance of the natural language description, we also ran REGEL using only its PBE engine. Sketch generation took on average 60 seconds per instance, and successfully generated a sketch for 63 instances. The remaining instance was run without a sketch. We considered only the highest ranked sketch for each instance. In Table 1 we show how many instances can be solved with different time limits for sketch completion; note that these values do not include the sketch generation time. REGEL returned a regular expression for 47 instances within the time limit. Since REGEL does not implement a disambiguation procedure, the returned regular expression does not always exhibit the desired behavior, even though it correctly classifies all examples. Of the 47 synthesized expressions, 31 exhibit the desired intent. This is a 66% accuracy, which is the same as FOREST without disambiguation (FOREST's 1^{st} regex) but it is much lower than FOREST with disambiguation at 96%. We also observe that REGEL's performance is severely impaired when using only its PBE engine.

51 out of the 63 generated sketches are of the form $\square\{S_1, ..., S_n\}$, where each S_i is a concrete sub-regex, i.e., has no holes. This construct indicates the desired regex must contain *at least* one of $S_1, ..., S_n$, and contains no information about the top-level operators that are used to connect them. 22 of the 47 synthesized regexes are based on sketches of that form, and they result from the direct concatenation of *all* components in the sketch. No new components are generated during sketch completion. Thus, most of REGEL's sketches could be integrated into FOREST, whose multi-tree structure holds precisely those top-level operators that were missing from REGEL's sketches.

6.2 Impact of pruning the search space and splitting examples

To evaluate the impact of pruning the search space as described in section 3.2, we ran FOREST with all pruning techniques disabled. In the scatter plot in Figure 6a, we can compare the solving time on each benchmark with and without pruning. Each mark in the plot represents an instance. The value on the y-axis shows the synthesis time of multi-tree with pruning disabled and the value on the x-axis the synthesis time with pruning enabled. The marks above the $y = x$ line



(a) Impact of pruning

(b) Impact of example-splitting

Figure 6: Comparison of synthesis time using different variations of FOREST.

(also represented in the plot) represent problems that took longer to synthesize without pruning than with pruning. On average, with pruning, FOREST can synthesize regexes in 42% of the time and enumerates about 15% of the regexes before returning. There is no significant change in the number of interactions before returning the desired solution.

FOREST is able to split the examples and use static multi-tree as described in section 3.2 in 52 benchmarks (81%). The remaining 12 are solved using dynamic multi-tree. To assess the impact of using static multi-tree we ran FOREST with a version of the multi-tree enumerator that does not split the examples, and jumps directly to dynamic multi-tree solving. In the scatter plot in Figure 6b, we compare the solving times of each benchmark. Using static multi-tree when possible, FOREST requires, on average, less than two thirds of the time (59.1%) to return the desired regex for benchmarks solved by both methods. Furthermore, with static multi-tree FOREST can synthesize more complex regexes: the maximum number of nodes in a solution returned by dynamic multi-tree is 12 (avg. 6.7), while complete multi-tree synthesizes regexes of up to 24 nodes (avg. 10.3).

6.3 Multi-tree versus *k*-tree and line-based encodings

To evaluate the performance of multi-tree enumeration, we ran FOREST with two other enumeration encodings: k-tree and line-based. The latter is a state of the art encoding for the synthesis of SQL queries [17]. k-tree is the default enumerator in TRINITY [13], and the line-based enumerator is available in SQUARES [16]. The k-tree encoding has a very similar structure to that of multi-tree, so our pruning techniques were easily applied to this encoding. On the other hand, line-based encoding is intrinsically different, so the pruning techniques were not implemented. We compare the line-based encoding to multi-tree without pruning. In every other aspect, the three encodings were run in the same conditions, using FOREST's regex DSL. k-tree is able to synthesize programs with up to

10 nodes, while the line-based encoding synthesizes programs of up to 9 nodes. Neither encoding outperforms multi-tree.

As seen in Table 1, line-based encoding does not outperform the tree-based encodings for the domain of regexes while it was much better for the domain of SQL queries [17]. We conjecture this disparity arises from the different nature of DSLs. Most SQL queries, when represented as a tree, leave many branches of the tree unused, which results in a much larger tree and SMT encoding.

6.4 Impact of fewer examples

To assess the impact of providing fewer examples on the accuracy of the solution, we ran Forest with modified versions of each benchmark. First, each benchmark was run with at most 10 valid and 10 invalid examples, chosen randomly among all examples. Conditional invalid examples are already very few per instance, so these were not altered. The accuracy of the returned regexes is slightly lower.

With only 10 valid and 10 invalid examples, Forest returns the correct regex in 93.5% of the benchmarks, which represents a decrease of only 2.5% relative to the results with all examples. We also saw an increase in the number of interactions before returning, since fewer examples are likely to be more ambiguous. With only 10 examples, Forest interacts on average 2.2 times per benchmark, which represents an increase of about a fifth. The increase in the number of interactions reflects on a small increase in the synthesis time (less than 1%).

After, we reduced the number of examples even further: only 5 valid and 5 invalid. The accuracy of Forest in this setting was reduced to 71%. On average, it interacted 4.3 times per benchmark, which is over two times more than before.

7 Conclusions and Future Work

Regexes are commonly used to enforce patterns and validate the input fields of digital forms. However, writing regex validations requires specialized knowledge that not all users possess. We have presented a new algorithm for synthesis of regex validations from examples that leverages the common structure shared between valid examples. Our experimental evaluation shows that the multi-tree representation synthesizes three times more regexes than previous representations in the same amount of time and, together with the user interaction model, Forest solves 70% of the benchmarks with the correct user intent. We verified that Forest maintains a very high accuracy with as few as 10 examples of each kind. We also observed that our approach outperforms Regel, a state-of-the-art synthesizer, in the domain of form validations.

As future work, we would like to explore the synthesis of more complex capture conditions, such as conditions depending on more than one capture. This would allow more restrictive validations; for example, in a date, the possible values for the day could depend on the month. Another possible extension to Forest is to automatically separate invalid from conditional invalid examples, making this distinction imperceptible to the user.

References

1. Chen, Q., Wang, X., Ye, X., Durrett, G., Dillig, I.: Multi-modal synthesis of regular expressions. In: PLDI. ACM (2020)
2. Chen, Y., Martins, R., Feng, Y.: Maximal multi-layer specification synthesis. In: ESEC/SIGSOFT FSE. pp. 602–612. ACM (2019)
3. Fedyukovich, G., Gupta, A.: Functional synthesis with examples. In: CP. Lecture Notes in Computer Science, vol. 11802, pp. 547–564. Springer (2019)
4. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. In: PLDI. pp. 420–435. ACM (2018)
5. Feng, Y., Martins, R., Geffen, J.V., Dillig, I., Chaudhuri, S.: Component-based synthesis of table consolidation and transformation tasks from examples. In: PLDI. pp. 422–436. ACM (2017)
6. Golia, P., Roy, S., Meel, K.S.: Manthan: A data driven approach for boolean function synthesis. In: CAV. Springer (2020)
7. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL. pp. 317–330. ACM (2011)
8. Kini, D., Gulwani, S.: Flashnormalize: Programming by examples for text normalization. In: IJCAI. pp. 776–783. AAAI Press (2015)
9. Kushman, N., Barzilay, R.: Using semantic unification to generate regular expressions from natural language. In: HLT-NAACL. pp. 826–836. The Association for Computational Linguistics (2013)
10. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. In: GPCE. pp. 70–80. ACM (2016)
11. Li, H., Chan, C., Maier, D.: Query from examples: An iterative, data-driven approach to query construction. Proc. VLDB Endow. **8**(13), 2158–2169 (2015)
12. Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., Barzilay, R.: Neural generation of regular expressions from natural language with minimal domain knowledge. In: EMNLP. pp. 1918–1923. The Association for Computational Linguistics (2016)
13. Martins, R., Chen, J., Chen, Y., Feng, Y., Dillig, I.: Trinity: An Extensible Synthesis Framework for Data Science. PVLDB **12**(12), 1914–1917 (2019)
14. Mayer, M., Soares, G., Grechkin, M., Le, V., Marron, M., Polozov, O., Singh, R., Zorn, B.G., Gulwani, S.: User interaction models for disambiguation in programming by example. In: UIST. pp. 291–301. ACM (2015)
15. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
16. Orvalho, P., Terra-Neves, M., Ventura, M., Martins, R., Manquinho, V.M.: Squares. https://squares-sql.github.io, accessed on May 27, 2020
17. Orvalho, P., Terra-Neves, M., Ventura, M., Martins, R., Manquinho, V.M.: Encodings for enumeration-based program synthesis. In: CP. Lecture Notes in Computer Science, vol. 11802, pp. 583–599. Springer (2019)
18. Python Software Foundation: Python3's regular expression module `re`. https://docs.python.org/3/library/re.html, accessed on October 11, 2020
19. Raza, M., Gulwani, S.: Automated data extraction using predictive program synthesis. In: AAAI. pp. 882–890. AAAI Press (2017)
20. Regular Expression Library: www.regexlib.com, accessed on May 27, 2020
21. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: CAV. Lecture Notes in Computer Science, vol. 11562, pp. 74–83. Springer (2019)

22. Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. **15**(5-6), 475–495 (2013)
23. Stanford, C., Veanes, M., Bjørner, N.: Symbolic boolean derivatives for efficiently solving extended regular expression constraints. Tech. Rep. MSR-TR-2020-25, Microsoft (August 2020), updated November 2020.
24. Wang, C., Cheung, A., Bodík, R.: Interactive query synthesis from input-output examples. In: SIGMOD Conference. pp. 1631–1634. ACM (2017)
25. Wang, C., Cheung, A., Bodík, R.: Synthesizing highly expressive SQL queries from input-output examples. In: PLDI. pp. 452–466. ACM (2017)
26. Wang, X., Gulwani, S., Singh, R.: FIDEX: filtering spreadsheet data using examples. In: OOPSLA. pp. 195–213. ACM (2016)
27. Zhong, Z., Guo, J., Yang, W., Peng, J., Xie, T., Lou, J., Liu, T., Zhang, D.: Semregex: A semantics-based approach for generating regular expressions from natural language specifications. In: EMNLP. pp. 1608–1618. Association for Computational Linguistics (2018)