Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript

Blake Loring
Information Security Group
Royal Holloway, University of London
United Kingdom

Duncan Mitchell
Department of Computer Science
Royal Holloway, University of London
United Kingdom

Johannes Kinder
Department of Computer Science
Royal Holloway, University of London
United Kingdom

Abstract

Existing support for regular expressions in symbolic execution-based tools for test generation and bug finding is insufficient. Common aspects of mainstream regular expression engines, such as backreferences or greedy matching, are commonly ignored or imprecisely approximated, leading to poor test coverage or missed bugs. In this paper, we present a model for the complete regular expression language of ECMAScript 2015 (ES6) that is sound for dynamic symbolic execution of `test` and `exec`. We model regular expression operations using string constraints and classical regular expressions and use a refinement scheme to address the problem of matching precedence and greediness. We implemented our model in ExpoSE, a dynamic symbolic execution engine for JavaScript, and evaluated it on over 1,000 Node.js packages containing regular expressions, demonstrating that the strategy is effective and can significantly increase the number of successful regular expression queries and therefore boost coverage.

1 Introduction

Regular expressions are popular with developers for matching and substituting strings and are supported by many programming languages. For instance, in JavaScript, one can write `/goo+d/.test(s)` to test whether the string value of `s` contains `"go"`, followed by one or more occurrences of `"o"` and a final `"d"`. Similarly, `s.replace(/goo+d/,"better")` evaluates to a new string where the first such occurrence in `s` is replaced with the string `"better"`.

Several testing and verification tools include some degree of support for regular expressions because they are so common [21, 24, 26, 31, 34]. In particular, SMT (satisfiability modulo theory) solvers now often support theories for strings and classical regular expressions [1, 2, 4, 14, 22, 23, 31, 35–37], which allow expressing constraints such as $\mathtt{s} \in \mathcal{L}(\mathtt{/goo+d/})$ for the `test` example above. Although any general theory of strings is undecidable [5], many string constraints are efficiently solved by modern SMT solvers.

SMT solvers support regular expressions in the language-theoretical sense, but "regular expressions" in programming languages like Perl or JavaScript are not limited to representing regular languages [3]. For instance, the expression `/<(\w+)>.*?<\/\1>/` parses any pair of matching XML tags, which is a context-sensitive language (because the tag is an arbitrary string that must appear twice). Problematic features that prevent a translation to the word problem in regular languages include capture groups (the parentheses around `\w+` in the example above), backreferences (the `\1` referring to the capture group), and greedy/non-greedy matching precedence of subexpressions (the `.*?` is non-greedy). In addition, any such expression could also be included in a lookahead `(?=)`, which effectively encodes intersection of context sensitive languages. In tools reasoning about string-manipulating programs, these features are usually ignored or imprecisely approximated. This is a problem, because complex regular expressions are widespread, as we demonstrate in §7.1.

In the context of dynamic symbolic execution (DSE) for test generation, this lack of support can lead to loss of coverage or missed bugs where constraints would have to include membership in non-regular languages. The difficulty arises from the typical mixing of constraints in path conditions—simply *generating* a matching word for a standalone regular expression is easy (without lookaheads). To date, there has been only limited progress on this problem, mostly addressing immediate needs of implementations with approximate solutions, e.g., for capture groups [26] and backreferences [24, 27]. However, neither matching precedence nor lookaheads have been addressed before.

In this paper, we propose a novel scheme for supporting ECMAScript regular expressions in dynamic symbolic execution and show that it is effective in practice. We rely on the specification of regular expressions and their associated methods in ECMAScript 2015 (ES6). However, our methods and findings should be easily transferable to most other existing implementations of regular expressions.In particular, we make the following contributions:

- We fully model ES6 regular expressions in terms of classical regular languages and string constraints (§4) and cover several aspects missing from previous work [24, 26, 27]. We introduce the notion of a *capturing language* to make the problem of matching and capture group assignment self-contained.
- We introduce a counterexample-guided abstraction refinement (CEGAR) scheme to address the effect of greediness on capture groups (§5), which allows to deploy our model in DSE without sacrificing soundness for under-approximation.
- We present the first systematic study of JavaScript regular expressions, examining usage of regular expression features across 415,487 packages from the NPM software repository. We show that non-regular features are widely used (§7.1).

In the remainder of the paper we review regular expressions (§2) and present an overview of our approach by example (§3). We then detail our model of regular expressions using a novel formulation (§4), and we propose a CEGAR scheme to address matching precedence (§5). We discuss an implementation of the encoding as part of the ExpoSE symbolic execution engine for JavaScript (§6) and evaluate its practical impact on DSE (§7). Finally, we review related work (§8) and conclude (§9).

2 Regular Expressions

We review ES6 regular expressions, focusing on differences to classical regular expressions. We begin with the regular expression API and its matching behavior (§2.1) and then explain capture groups (§2.2), backreferences (§2.3), and operator precedence (§2.4). ES6 regular expressions are comparable to those of other languages but lack Perl's recursion and lookbehind and do not require POSIX-like longest matches.

2.1 Methods, Anchors, Flags

ES6 regular expressions are `RegExp` objects, created from literals or the `RegExp` constructor. `RegExp` objects have two methods, `test` and `exec`, which expect a string argument; String objects offer the `match`, `split`, `search` and `replace` methods that expect a `RegExp` argument.

A regular expression accepts a string if any portion of the string matches the expression, i.e., it is implicitly surrounded by wildcards. The matched string's position in the text can be controlled with *anchors*, with `^` and `$` matching the start and end, respectively.

Flags in regular expressions can modify the behavior of matching operations. The *ignore* case flag `i` ignores character cases when matching. The *multiline* flag `m` redefines anchor characters to match either the start and end of input or newline characters. The *unicode* flag `u` changes how unicode literals are escaped within an expression.

The meaning of the *global* flag `g` varies. It extends the effects of `match` and `replace` to include all matches on the string and it is equivalent to the sticky flag for the `test` and `exec` methods of `RegEx`. The *sticky* flag `y` forces matching to start at `RegEx.lastIndex`, which is updated with the index of the previous match. Therefore, `RegEx` objects become stateful as shown in the following example:

```
r = /goo+d/y;
r.test("goood"); // true; r.lastIndex = 6
r.test("goood"); // false; r.lastIndex = 0
```

2.2 Capture Groups

Parentheses in regular expressions not only change operator precedence (e.g., `(ab)*` matches any number of repetitions of the string `"ab"` while `ab*` matches the character `"a"` followed by any number of characters `"b"`), but they also create *capture groups*. Capture groups are implicitly numbered from left to right by order of the opening parenthesis, for example, `/a|((b)*c)*d/` is numbered as $\texttt{/a|}(^1(^2\texttt{b})\texttt{*c})\texttt{*d/}$. Where only bracketing is required, a non-capturing group can be created by using the syntax `(?: ...)`.

Table 1. Regular expression operators, separated by classes of precedence.

Operator	Name	Rewriting
(r)	Capturing parentheses	
$\backslash n$	Backreference	
$(?:r)$	Non-capturing parentheses	
$(?=r)$	Positive lookahead	
$(?!r)$	Negative lookahead	
\b	Word boundary	
\B	Non-word boundary	
$r*$	Kleene star	
$r*?$	Lazy Kleene star	
$r+$	Kleene plus	$r*r$
$r+?$	Lazy Kleene plus	$r*?r$
$r\{m,n\}$	Repetition	$r^n \mid \ldots \mid r^m$
$r\{m,n\}?$	Lazy repetition	$r^m \mid \ldots \mid r^n$
$r?$	Optional	$r \mid \epsilon$
$r??$	Lazy optional	$\epsilon \mid r$
$r_1 r_2$	Concatenation	
$r_1 \mid r_2$	Alternation	

For real-world regular expressions, capture groups are important because the regular expression engine will record the *most recent* substring matched against each capture group. Capture groups can be referred to from within the expression using backreferences (see §2.3); the last matched substrings for each capture group are also returned by some of the API methods. In JavaScript, the return value of `match` is an array, with the whole match at index 0, and the last matched instance of the i^{th} capture group at index i. In the example above, `"bbbbcbcd".match(/a|((b)*c)*d/)` will evaluate to the array `["bbbbcbcd", "bc", "b"]`.

2.3 Backreferences

A *backreference* in a regular expression refers to a numbered capture group and will match whatever the engine last matched the capture group against. In general, the addition of backreferences to regular expressions makes the accepted languages non-regular [3].

Inside quantifiers (Kleene star/plus and other repetition operators), the string matched by the backreference can change across multiple matches. For example, the regular expression `/((a|b)\2)+/` can match the string `"aabb"`, with the backreference `\2` being matched twice: the first time, the capture group contains `"a"`, the second time it contains `"b"`. This logic applies recursively, and it is possible for backreferences to in turn be part of outer capture groups.

2.4 Operator Evaluation

We explain the operators of interest for this paper in Table 1; the implementation described in §6 supports the full ES6 syntax [13]. Some operators can be rewritten into semantically equivalent expressions to reduce the number of cases to handle (shown in the **Rewriting** column).

Real-world regular expressions distinguish between *greedy* and *lazy* evaluation. Greedy operators consume as many characters as possible such that the entire regular expression still matches; lazy operators consume as few characters as possible. This distinction—called *matching precedence*—is unnecessary for classical regular languages, but does affect the assignment of capture groups and therefore backreferences.

Zero-length assertions or *lookarounds* are regular expressions that do not consume any characters but still restrict the accepted word, enforcing a language intersection. Positive or negative *lookaheads* can contain any regular expression, including capture groups and backreferences. In ES6, *lookbehind* is only available through `\b` (word boundary), and `\B` (non-word boundary), which are commonly used to only (or never) match whole words in a string.

3 Overview

We now give an overview of our approach. We first define the word problem for ES6 regular expressions (§3.1) and how it arises in DSE (§3.2). We introduce our model for complex regular expressions by example (§3.3) and explain how to eliminate spurious solutions by refinement (§3.4).

3.1 The Word Problem and Capturing Languages

For any given classical regular expression r, we write $w \in \mathcal{L}(r)$ whenever w is a word within the (regular) language generated by r. For an ES6 regular expression R, we also need to record values of capture groups within the regular expression. To this end, we introduce the following notion:

Definition 1 (Capturing Language)**.** The *capturing language* of an ES6 regular expression R, denoted $\mathcal{L}_c(R)$, is the set of tuples $(w, C_0, \ldots, C_n)$ such that w is a word of the language of R and each $C_0, \ldots, C_n$ is the substring of w matched by the corresponding numbered capture group in R.

A word w is therefore matched by a regular expression R if and only if $\exists C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \in \mathcal{L}_c(R)$. It is not matched if and only if $\forall C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \notin \mathcal{L}_c(R)$. For readability, we will usually omit quantifiers for capture variables where they are clear from the context.

3.2 Regular Expressions In DSE

The code in Listing 1 parses numeric arguments between XML tags from its input variable `args`, an array of strings. The regular expression in line 4 breaks each argument into two capture groups, the tag and the numeric value (`parts[0]` is the entire match). When the tag is "`timeout`", it sets the `timeout` value accordingly (lines 6-7). Line 12 uses a runtime assertion to check that the timeout value is truly numeric after the arguments have been processed. The assertion can fail because the program contains a bug: the regular expression in line 4 uses a Kleene star and therefore also admits the empty string as the number to set, and JavaScript's dynamic type system will allow setting `timeout` to the empty string.

```
1  let timeout = '500';
2  for (let i = 0; i < args.length; i++) {
3    let arg = args[i];
4    let parts = /<(\w+)>([0-9]*)<\/\1>/.exec(arg);
5    if (parts) {
6      if (parts[1] === "timeout") {
7        timeout = parts[2];
8      } ...
9  }}
10 assert(/^[0-9]+$/.test(timeout) == true);
```

Listing 1. Code example using regular expressions.

DSE finds such bugs by systematically enumerating paths, including the failure branches of assertions [7, 15]. Starting from a concrete run with input, say, `args[0] = "foo"`, the DSE engine will attempt to build a *path condition* that encodes the branching decisions in terms of the input values. It then attempts to systematically flip clauses in the path condition and query an SMT solver to obtain input assignments covering different paths. This process repeats forever or until all paths are covered (this program has an unbounded number of paths as it is looping over an input string).

Without support for regular expressions, the DSE engine will *concretize* `arg` on the call to `exec`, assigning the concrete result to `parts`. With all subsequent decisions therefore concrete, the path condition becomes $pc = \texttt{true}$ and the engine would be unable to cover more paths and find the bug.

Implementing regular expression support ensures that `parts` is *symbolic*, i.e., its elements are represented as formulas during symbolic execution. The path condition for the the initial path thus becomes $pc = (\texttt{args[0]}, C_0, C_1, C_2) \notin \mathcal{L}_c(R)$ where $R = \texttt{/<(\w+)>([0-9]*)<\/\1>/}$. Negating the only clause and solving yields, e.g., `args[0] = "<a>0</a>"`. DSE then uses this input assignment to cover a second path with $pc = (\texttt{args[0]}, C_0, C_1, C_2) \in \mathcal{L}_c(R) \wedge C_1 \neq \texttt{"timeout"}$. Negating the last clause yields, e.g., "`<timeout>0</timeout>`", entering line 7 and making `timeout` and therefore the assertion symbolic. This leads to $pc = (\texttt{args[0]}, C_0, C_1, C_2) \in \mathcal{L}_c(R) \wedge C_1 = \texttt{"timeout"} \wedge (C_2, C_0') \in \mathcal{L}_c(\texttt{^[0-9]+})$, which, after negating the last clause, triggers the bug with the input "`<timeout></timeout>`".

3.3 Modeling Capturing Language Membership

Capturing language membership constraints in the path condition cannot be directly expressed in SMT. We model these

in terms of classical regular language membership and string constraints. For a given ES6 regular expression R, we first rewrite R (see Table 1) in atomic terms only, i.e., `|`, `*`, capture groups, backreferences, lookaheads, and anchors. For consistency with the JavaScript API, we also introduce an additional outer capture group. Consider the regular expression R = `(?:a|(b))\1`. After preprocessing, the capturing language membership problem becomes $(w, C_0, C_1) \in \mathcal{L}_c$(`(:?.|n)*?((?:a|(b))\1)(?:.|n)*?`), a generic rewriting that allows for characters to precede and follow the match in the absence of anchors.

We recursively reduce capturing language membership to regular membership. To begin we translate the purely regular Kleene stars and the outer capture group to obtain $(w, C_0, C_1) \in \mathcal{L}_c(R') \implies w = w_1 \mathbin{++} w_2 \mathbin{++} w_3 \wedge w_1 \in \mathcal{L}$(`(:?.|\n)*?`)$\wedge (w_2, C_1) \in \mathcal{L}_c$(`(?:a|(b))\1`)$\wedge C_0 = w_2 \wedge w_3 \in \mathcal{L}$(`(:?.|\n)*?`). where ++ is string concatenation. We continue by decomposing the regular expression until there are only purely regular terms or standard string constraints. We translate the capturing language for `(?:a|(b))\1` next to obtain $w = w_1 \mathbin{++} w_2 \wedge (w_1, C_1) \in \mathcal{L}_c$(`a|(b)`)$\wedge (w_2) \in \mathcal{L}_c$(`\1`), When treating the alternation, either the left is satisfied and the capture group becomes undefined (which we denote $\varnothing$), or the right is satisfied and the capture is locked to the match, which we model as $(w_1 \in \mathcal{L}(a) \wedge C_1 = \varnothing) \vee (w_1 \mathcal{L}(b) \wedge C_1 = w_1)$. Finally we model the backreference, which is case dependent on whether the capture group it refers to is defined or not: $(C_1 = \varnothing \implies w_2 = \epsilon) \wedge (C_1 \neq \varnothing \implies w_2 = C_1)$. Putting this together we obtain a model for R:

$$
\begin{aligned}
&(w, C_0, C_1) \in \mathcal{L}_c(R) \implies w = w_1 \mathbin{++} w_2 \mathbin{++} w_3 \mathbin{++} w_4 \\
&\wedge C_0 = w_2 \mathbin{++} w_3 \\
&\wedge \big((w_2 \in \mathcal{L}(a) \wedge C_1 = \varnothing) \vee (w_2 \in \mathcal{L}(b) \wedge C_1 = w_2)\big) \\
&\wedge (C_1 = \varnothing \implies w_3 = \epsilon) \wedge (C_1 \neq \varnothing \implies w_3 = C_1) \\
&\wedge w_1 \in \mathcal{L}(\texttt{(:?.|n)*?}) \wedge w_4 \in \mathcal{L}(\texttt{(:?.|n)*?}).
\end{aligned}
$$

3.4 Refinement

Because of matching precedence (greediness), this type of model permits assignments to capture groups that are impossible in real executions. For example, for `/a*(a)?/`, we model $(w, C_0, C_1) \in \mathcal{L}_c$(`a*(a)?`) $\implies w = w_1 \mathbin{++} w_2 \wedge w_1 \in \mathcal{L}$(`a*`)$\wedge w_2 \in \mathcal{L}$(`a|`$\epsilon$)$\wedge C_0 = w \wedge C_1 = w_2$. This allows C_1 to be either `a` or the empty string ϵ, i.e., the tuple (`"aa"`, `"aa"`, `"a"`) would be a spurious member of the capturing language under our model. Because `a*` is *greedy*, it will always consume both characters in the string `"aa"`; therefore, `(a)?` can only match ϵ. This problem posed by *greedy* and *lazy* operator semantics remains unaddressed by previous work [24, 26, 27, 31]. To address this, we use a counterexample-guided abstraction refinement scheme that validates candidate assignments with an ES6-compliant matcher. Continuing the example, the candidate element (`"aa"`, `"aa"`, `"a"`) is validated by running a concrete matcher on the string `"aa"`, which contradicts the candidate captures with C_0 = `"aa"` and $C_1 = \epsilon$. The model is refined with the counter-example to the following:

$$
\begin{aligned}
w = w_1 \mathbin{++} w_2 \wedge w_1 \in \mathcal{L}(\texttt{a*}) \wedge w_2 \in \mathcal{L}(\texttt{a|}\epsilon) \wedge C_0 = w \wedge C_1 = w_2 \\
\wedge \big(w = \texttt{"aa"} \implies (C_0 = \texttt{"aa"} \wedge C_1 = \epsilon)\big).
\end{aligned}
$$

We then generate and validate a new candidate (w, C_0, C_1) and repeat the refinement until a satisfying assignment passes the concrete matcher.

4 Modeling ES6 Regular Expressions

We now detail the process of modeling capturing languages. After preprocessing a given ES6 regular expression R to R' (§4.1), we model constraints $(w, C_0, \ldots, C_n) \in \mathcal{L}_c(R')$ by recursively translating terms in the abstract syntax tree (AST) of R' to classical regular language membership and string constraints (§4.2). Finally, we show how to model negated constraints $(w, C_0, \ldots, C_n) \notin \mathcal{L}_c(R')$ (§4.4).

4.1 Preprocessing

For illustrative purposes, we make the concatenation $R_1 R_2$ of terms R_1, R_2 explicit as the binary operator $R_1 \cdot R_2$. Any regular expression can then be split into combinations of atomic elements, capture groups and backreferences (referred to collectively as *terms*, in line with the ES6 specification [13]), joined by explicit operators. Using the rules in Table 1, we rewrite any regular expression R to an equivalent regular expression R' containing only alternation, concatenation, Kleene star, capture groups, non-capturing parentheses, lookarounds, and backreferences. Note that some rules duplicate capture groups; we therefore have to renumber capture groups during the rewriting (see Appendix A for details).

We rewrite any remaining lazy quantifiers to their greedy equivalents, as the models are agnostic to matching precedence (this is dealt with in refinement).

4.2 Operators and Capture Groups

Let t be the next term to process in the AST of R'. If t is capture-free and purely regular, there is nothing to do in this step. If t is non-regular, it contains $k + 1$ capture groups (with $k \geq -1$) numbered i through $i + k$. At each recursive step, we express membership of the capturing language $(w, C_i, \ldots, C_{i+k}) \in \mathcal{L}_c(t)$ through a model consisting of string and regular language membership constraints, and a set of remaining capturing language membership constraints for subterms of t. Note that we record the locations of capture groups within the regular expression in the preprocessing step. When splitting t into subterms t_1 and t_2, capture groups $C_i, \ldots, C_{i+j}$ are contained in t_1 and $C_{i+j+1}, \ldots, C_{i+k}$ are contained in t_2 for some j. The models for individual operations are given in Table 2; we discuss specifics of the rules below.

When matching an alternation `|`, capture groups on the non-matching side will be undefined, denoted by $\varnothing$, which is distinct from the empty string ϵ).

Table 2. Models for regular expression operators.

Operation	t	**Overapproximate Model for** $(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t)$
Alternation	$t_1 \mid t_2$	$\big((w, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1) \wedge C_{i+j+1} = ... = C_{i+k} = \varnothing\big)$ $\vee \big((w, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2) \wedge C_i = ... = C_{i+j} = \varnothing\big)$
Concatenation	$t_1 \cdot t_2$	$w = w_1 \mathbin{+\!\!+} w_2 \wedge (w_1, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1) \wedge (w_2, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$
Backreference-free Quantification	t_1*	$w = w_1 \mathbin{+\!\!+} w_2 \wedge w_1 \in \mathcal{L}(\hat{t_1}*) \wedge (w_2, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1 \mid \epsilon)$ $\wedge \big(w_2 = \epsilon \implies (w_1 = \epsilon \wedge C_i = \ldots = C_{i+k} = \varnothing)\big)$
Positive Lookahead	$(?{=}t_1)\, t_2$	$(w, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1 .*) \wedge (w, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$
Negative Lookahead	$(?!{=}t_1)\, t_2$	$(w, C_i, ..., C_{i+j}) \notin \mathcal{L}_c(t_1 .*) \wedge (w, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$
Input Start	$t_1\hat{}$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, C_i, ..., C_{i+k}) \in \mathcal{L}(. * \langle)$
Input Start (Multiline)	$t_1\hat{}$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, C_i, ..., C_{i+k}) \in \mathcal{L}(. * \langle \mid \backslash n)$
Input End	$\$t_1$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, C_i, ..., C_{i+k}) \in \mathcal{L}(\rangle .*)$
Input End (Multiline)	$\$t_1$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge (w, C_i, ..., C_{i+k}) \in \mathcal{L}(\rangle \mid \backslash n .*)$
Word Boundary	$t_1 \backslash \mathtt{b}\ t_2$	$w = w_1 \mathbin{+\!\!+} w_2 \wedge (w_1, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1) \wedge (w_2, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$ $\wedge \Big(\big((w_1 \in \mathcal{L}(.*\backslash \mathtt{W}) \vee w_1 = \epsilon\big) \wedge w_2 \in \mathcal{L}(\backslash \mathtt{w}.*)\big) \vee \big(w_1 \in \mathcal{L}(.*\backslash \mathtt{w}) \wedge (w_2 \in \mathcal{L}(\backslash \mathtt{W}.*) \vee w_2 = \epsilon)\big)\Big)$
No Word Boundary	$t_1 \backslash \mathtt{B}\ t_2$	$w = w_1 \mathbin{+\!\!+} w_2 \wedge (w_1, C_i, ..., C_{i+j}) \in \mathcal{L}_c(t_1) \wedge (w_2, C_{i+j+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_2)$ $\wedge \big((w_1 \notin \mathcal{L}(.*\backslash \mathtt{W}) \wedge w_1 \neq \epsilon) \vee w_2 \notin \mathcal{L}(\backslash \mathtt{w}.*)\big) \wedge \big(w_1 \notin \mathcal{L}(.*\backslash \mathtt{w}) \vee (w_2 \notin \mathcal{L}(\backslash \mathtt{W}.*) \wedge w_2 \neq \epsilon)\big)$
Capture Group	(t_1)	$(w, C_{i+1}, ..., C_{i+k}) \in \mathcal{L}_c(t_1) \wedge C_i = w$
Non-Capturing Group	$(?:t_1)$	$(w, C_i, ..., C_{i+k}) \in \mathcal{L}_c(t_1)$
Base Case	t regular	$w \in \mathcal{L}(t)$

When modeling quantification $t = t_1*$, we assume t_1 does not contain backreferences (we address this case in §4.3). In this instance, we model t via the expression $\hat{t_1}* t_1 \mid \epsilon$, where $\hat{t_1}$ is a regular expression corresponding to t_1 except each set of capturing parentheses is rewritten as a set of non-capturing parentheses. In this way, $\hat{t_1}$ is regular (it is backreference-free by assumption). However, $\hat{t_1}* t_1 \mid \epsilon$ is not semantically equivalent to t: if possible, capturing groups must be satisfied, so $\hat{t_1}*$ cannot consume all matches of the expression. We encode this constraint with the implication that $\hat{t_1}*$ must match the empty string whenever $t_1 \mid \epsilon$ does.

Lookahead constrains the word to be a member of the languages of both the assertion expression and t_2. The word boundary \b is effectively a single-character lookaround for word and non-word characters. Because the boundary can occur both ways, the model uses disjunction for the end of w_1 and the start of w_2 being word and non-word, or non-word and word characters, respectively.

For capture groups, we bind the next capture variable C_i to the string matched by t_1. The i^{th} capture group must be the outer capture and the remaining captures $C_{i+1}, \ldots, C_{i+k}$ must therefore be contained within t_1. There is nothing to be done for non-capturing groups and recursion continues on the contained subexpression.

Anchors assert the start (^) and end ($) of input; we represent the beginning and end of a word via the meta-characters $\langle$ and $\rangle$, respectively. In most instances when handling these operations, t_1 will be ϵ; this is because it is rare to have regular expression operators prior to those marking the end of input (or after marking the end of input, respectively). In both these cases, we assert that the language defines the start or end of input—and that as a result of this, the language of t_1 must be an empty word, though the capture groups may be defined (say through t_1 containing assertions with nested captures). We detail separate rules for matching a regular expression with the multiline flag set. In this anchor behavior is modified to accept either our meta-characters or a line break.

4.3 Backreferences

Table 3 lists our models for different cases of backreferences in the AST of regular expression R; $\backslash k$ is a backreference to the k^{th} capture group of R. Intuitively, each instance of a backreference is a variable that refers to a capture group and has a type that depends on the structure of R.

We call a backreference *immutable* if it can only evaluate to a single value when matching; it is *mutable* if it can take on multiple values, which is a rare but particularly tricky case.

Table 3. Modeling backreferences.

Type of $\backslash k$	Capturing Language	Approximation	Model
Empty	$(w) \in \mathcal{L}_c(\backslash k)$	Exact	$w = \epsilon$
Immutable	$(w) \in \mathcal{L}_c(\backslash k)$	Overapproximate	$(C_k = \varnothing \implies w = \epsilon) \wedge (C_k \neq \varnothing \implies w = C_k)$
Immutable	$(w) \in \mathcal{L}_c(\backslash k*)$	Overapproximate	$(C_k = \varnothing \implies w = \epsilon) \wedge (C_k \neq \varnothing \implies \exists m \geq 0 : w = {+\!+}_{i=0}^{m} C_k)$
Mutable	$(w,C_k) \in \mathcal{L}_c((?:(t_1)\backslash k)*)$ t_1 is capture group-free	Overapproximate	$\big(w = \epsilon \wedge C_k = \varnothing\big) \vee \big(\exists m \geq 1 : w = {+\!+}_{i=1}^{m}(\sigma_{i,1} {+\!+} \sigma_{i,2})$ $\wedge \forall i > 1, \big((\sigma_{i,1}, C_{k,i}) \in \mathcal{L}_c(t_1) \wedge \sigma_{i,2} = C_{k,i}\big) \wedge C_k = C_{k,m}\big)$
Mutable	$(w,C_k) \in \mathcal{L}_c((?:(t_1)\backslash k)*)$ t_1 is capture group-free	Unsound	$\big(w = \epsilon \wedge C_k = \varnothing\big) \vee \big(\exists m \geq 1 : w = {+\!+}_{i=1}^{m}(\sigma_{i,1} {+\!+} \sigma_{i,2})$ $\wedge (\sigma_{i,1}, C_k) \in \mathcal{L}_c(t_1) \wedge \forall i \geq 1, (\sigma_{i,1} = \sigma_{1,1} \wedge \sigma_{i,2} = \sigma_{1,1})\big)$

For example, consider `/((a|b)\2)+\1\2/`. Here, the backreference `\1` and the second instance of `\2` are immutable. However, the first instance of `\2` is mutable: each repetition of the outer capture group under the Kleene plus can change the value of the second (inner) capture group, in turn changing the value of the backreference inside this quantification. For example, the string `"aabbaabbb"` satisfies this regular expression, but `"aabaaabaa"` does not. To fully characterize these distinctions, we introduce the following definition:

Definition 2 (Backreference Type)**.** Let t be the k^{th} capture group of a regular expression R. Then

1. $\backslash k$ is *empty* if either k is greater than the number of capture groups in R, or $\backslash k$ is encountered before t in a post-order traversal of the AST of R;
2. $\backslash k$ is *mutable* if $\backslash k$ is not empty, and both t and $\backslash k$ are subterms of some term Q in R which is quantified;
3. otherwise, $\backslash k$ is *immutable*.

When a backreference is empty, it is defined as ϵ, because it refers to a capture group that either is a superterm, e.g., `/(a\1)*/`, or appears later in the term, e.g., `/\1(a)/`.

There are two cases for immutable backreferences. In the first case, the backreference is not quantified. In our model for R, C_k has already been modeled with an equality constraint so we can bind the backreference to it. In the second case, the backreference occurs within a quantification; here, the matched word is a finite concatenation of identical copies of the referenced capture group. Both models also incorporate the corner case where the capture group is $\varnothing$ due to alternation or an empty Kleene star. Following the ES6 standard, the backreference evaluates to ϵ in this case.

Mutable backreferences appear in the form $(\ldots t_1 \ldots \backslash k \ldots)*$ where t_1 is the k^{th} capture group; ES6 does not support forward referencing of backreferences, so in $(\ldots \backslash k \ldots t_1 \ldots)*$, $\backslash k$ is empty. For illustration purposes, the fourth entry of Table 3 describes the simplest case for mutable backreferences, other patterns are straightforward generalizations. In this case, we assume t_1 is the k^{th} capture group but is otherwise capture group free. We can treat the entirety of this term at once: as such, any word in the language is either ϵ, or for some number of iterations, we have the concatenation of a word in the language of t_1 followed by a copy of it. We introduce new variables $C_{k,i}$ referring to the values of the capture group in each iteration, which encodes the repeated matching on the string until settling on the final value for C_k. In this instance, we need not deal with the possibility that any $C_{k,i}$ is $\varnothing$, since the quantification ends as soon as t_1 does not match.

Unfortunately, constraints generated from this model are hard to solve and not feasible for current SMT solvers, because they require to "guess" a partition of the matched string variable into individual and varying components. To make solving such queries practical, we introduce an alternative to the previous rule where we treat quantified backreferences as immutable. The resulting model is shown in the last row of Table 3. Returning to the example of `/((a|b)\2)+\1\2/`, this model accepts (`"aaaaaaaaa"`, `"aaaaaaaaa"`, `"aaaa"`, `"a"`), but not (`"aabbaabbb"`, `"aabbaabbb"`, `"aabb"`, `"b"`). We discuss the soundness implications in §5.4. Quantified backreferences are rare (see §7.1), so the effect is limited in practice.

4.4 Modeling Non-Membership

The model described so far addresses membership of the capturing language, i.e., constraints of the form $\exists C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \in \mathcal{L}_c(R)$. We analogously define a model for non-membership, i.e., $\forall C_0, \ldots, C_n : (w, C_0, \ldots, C_n) \notin \mathcal{L}_c(R)$. The models for alternation, looakaheads, word boundaries, backreferences, non-capturing groups, and the base case are negated. In concatenation and quantification, only the language and emptiness constraints are negated, so the models take the form $w = w_1 {+\!+} w_2 \wedge (\ldots \notin \mathcal{L}_c(\ldots) \vee \ldots \notin \mathcal{L}_c(\ldots) \vee (w_2 = \epsilon \wedge \neg(w_1 = \epsilon \ldots)))$. In the same manner, the model for capture groups is $(w, C_{i+1}, \ldots, C_{i+k}) \notin \mathcal{L}_c(t_1) \wedge C_i = w$.

Returning to the example of §3.3, the negated model for $\mathcal{L}_c((a|(b)\backslash 1))$ becomes

$$\forall C_0, C_1 : w = w_1 {+\!+} w_2 \wedge C_0 = w$$
$$\wedge \big(\neg\big((w_1 \in \mathcal{L}(a) \wedge C_1 = \varnothing) \vee (w_1 \in \mathcal{L}(b) \wedge C_1 = w_1)\big)$$
$$\vee \neg(C_1 = \varnothing \implies w_2 = \epsilon) \vee \neg(C_1 \neq \varnothing \implies w_2 = C_1)\big).$$

5 Matching Precedence Refinement

We now explain the issue of matching precedence (§5.1) and introduce a counterexample-guided abstraction refinement scheme (§5.2) to address it. We discuss termination (§5.3) and the overall soundness of our approach (§5.4).

5.1 Matching Precedence

The model in Tables 2 and 3 does not account for matching precedence (see §3.4). A standards-compliant ES6 regular expression matcher will derive a unique set of capture group assignments when matching a string w, because matching precedence dictates that greedy (non-greedy) expressions match as many (as few) characters as possible before moving on to the next [13]. These requirements are not part of our model, as encoding them directly into SMT would require nesting of quantifiers for each operator, making them impractical for automated solving.

5.2 CEGAR for ES6 Regular Expression Models

We eliminate infeasible elements of the capturing language admitted by our model through counter example-guided abstraction refinement (CEGAR).

Algorithm 1 is a CEGAR-based satisfiability checker for constraints modeled from ES6 regular expressions, which relies on an external SMT solver with classical regular expression and string support, and on a ES6-compliant regular expression matcher. The algorithm takes an SMT problem P (in DSE, this will be derived from the path condition), which may contain encoded regular expressions. The list E of tuples $(e, \mathit{Membership})$ explicitly enumerates (raw) capturing language membership constraints e and whether they are positive or negated. The algorithm returns `undefined` if P is unsatisfiable or a satisfying model with correct matching precedence.

L is a worklist of SMT (sub-)problems, initialized to P. While L is non-empty, we extract the next problem P' from L and pass it to an external SMT solver. The solver returns a satisfying assignment M for the problem or `undefined` if the problem is unsatisfiable (lines 3-4). If M is not undefined, the algorithm uses a concrete matcher (e.g., Node.js's built-in matcher) to populate concrete capture variables $C_i^\natural$ for the matched strings of all subproblems.

Lines 7-22 describe how the assignments of capture groups are checked for each regular expression e in the original problem P. We first check if we obtained a match as a result of executing the concrete matcher. If we have, then w is a member of the language generated by e. If e was a positive membership constraint, then we must check if the capture group assignments are consistent with those from M (line 13). If they do match, we move on to the next regular expression, otherwise we refine the constraint problem and add it to L (line 15). To develop the refinement we have to identify why the language membership operation failed. If the regular expression matched but capture groups are inconsistent, then we refine the problem by fixing capture group assignments for the matched word and add it to the worklist (line 15). Dually, if a modeled non-membership constraint was satisfiable but the assigned word $M[w]$ did match concretely, we refine the problem in the same way (line 18). Otherwise, if a positive membership constraint was not satisfied, we block the currently assigned word (line 22).

After this process completes, we either have confirmed the overall assignment satisfies P, in which case we are done (line 24), or we must continue with further entries in L. If L has been exhausted without finding a satisfying model, we know the problem P is unsatisfiable (line 25).

Algorithm 1: Counterexample-guided abstraction refinement scheme for matching precedence.

Input : Constraint Problem, P
Language Membership Instances, E
Output : `undefined` if P is unsatisfiable, or a satisfying assignment for P otherwise

```
1  L := [P];
2  while L ≠ ∅ do
3      select and remove P' from L ;
4      M := Solve(P');
5      if M ≠ undefined then
6          Failed := false;
7          foreach (e, Membership) ∈ E do
8              select (w, C_0, ..., C_n) from P for e;
9              (C_0^♮, ..., C_n^♮) := ConcreteMatch(M[w], e) ;
10             if (C_0^♮, ..., C_n^♮) then
11                 R := (w = M[w] ⟹ ∀0 ≤ i ≤ n, C_i = C_i^♮);
12                 if Membership then
13                     if ∃0 ≤ i ≤ n : C_i^♮ ≠ M[C_i] then
14                         Failed := true;
15                         L := (P' ∧ R) ∪ L ;
16                 else          // Non-membership query
17                     Failed := true;
18                     L := (P' ∧ R) ∪ L ;
19             else
20                 if Membership then
21                     Failed := true;
22                     L := (P' ∧ (w ≠ M[w])) ∪ L ;
23         if Failed then
24             return M;
25 return undefined;
```

5.3 Termination

Unsurprisingly, CEGAR may require arbitrarily many refinements on pathological formulas and never terminate. This is

unavoidable due to undecidability [5]). In practice, we therefore impose a limit on the number of refinements, leading to *unknown* as a possible third result. SMT solvers already may timeout or report *unknown* for complex string formulas, so this does not lead to additional problems in practice.

5.4 Soundness

When constructing the rules in Tables 2 and 3, we followed the semantics of regular expressions as laid out in the ES6 standards document [13]. The ES6 standard is written in a semi-formal fashion, so we are confident that our translation into logic is accurate, but cannot have formal proof. Existing attempts to encode ECMAScript semantics into logic such as JSL [6] or JaVerT [25] do not include regular expressions.

With the exception of the optimized rule for mutable backreferences, our models are overapproximate, because they ignore matching precedence. When the CEGAR loop terminates, any spurious solutions from overapproximation are eliminated. As a result, we have an *exact* procedure to decide (non)-membership for capturing languages of ES6 regular expressions without quantified backreferences.

In the presence of quantified backreferences, the model after CEGAR termination becomes *underapproximate*. Since DSE itself is an underapproximate program analysis (due to concretization, solver timeouts, and partial exploration), our model and refinement strategy are *sound for DSE*.

6 Implementation

We now describe an implementation of our approach in the DSE engine ExpoSE [24][1]. We explain how to model the regular expression API with capturing language membership (§6.1) and give a brief overview of ExpoSE (§6.2).

6.1 Modeling the Regular Expression API

The ES6 standard specifies several methods that evaluate regular expressions [13]. We follow its specified pseudocode for `RegExp.exec(s)` to implement matching and capture group assignment in terms of capturing language membership in Algorithm 2. Notably, our algorithm implements support for all flags and operators specified for ES6.

The function `RegExp.test(s)` is precisely equivalent to `{return this.exec(s)!== undefined}`. In the same manner, one can construct models for other regular expression functions defined for ES6. Our implementation includes partial models for the remaining functions that allow effective test generation in practice but are not semantically complete.

Algorithm 2 first processes flags to begin from the end of the previous match for sticky or global, and rewrites the regular expression to accept lower and upper case variants of characters for ignore case.

We introduce the ⟨ and ⟩ characters to *input* which act as meta-character markers for the start of and end of string during matching. Next, if the sticky or global flags are set we slice *input* at lastIndex so that the new match begins from the end of the previous. Due to the introduction of our meta-characters the lastIndex needs to be offset by 1 if it is greater than zero. We then rewrite the regular expression source to allow for characters to precede and succeed the match. Note that we use `(?:.|\n)*?` rather than `.*?` because the wildcard . consumes all characters except line breaks in ECMAScript regular expressions. To avoid adding these characters to the final match we place the original regular expression source inside a capture group. This forms C_0, which is defined to be the whole matched string [13]. Once preprocessing is complete we test whether the input string and fresh string for each capture group are within the capturing language for the expression. If they are then a results object is created which returns the correctly mapped capture groups, the input string, and the start of the match in the string with the meta-characters removed. Otherwise `lastIndex` is reset and `undefined` is returned.

Algorithm 2: RegExp.exec(input)

```
1  input' := ⟨ + input + ⟩;
2  if sticky or global then
3  |   offset := lastIndex > 0 ? lastIndex + 1 : 0;
4  |   input' := input'.substring(offset);
5  source' := '(:?.|\n)*?(' + source + ')(:?.|\n)*?';
6  if caseIgnore then
7  |   source' := treatIgnoreCase(source');
8  if let (input', C0, ..., Cn) ∈ Lc(source') then
9  |   Remove ⟨ and ⟩ from (input', C0, ..., Cn);
10 |   lastIndex := lastIndex + C0.startIndex + C0.length;
11 |   result := [C0, ..., Cn];
12 |   result.input := input;
13 |   result.index := C0.startIndex;
14 |   return result;
15 else
16 |   lastIndex := 0;
17 |   return undefined;
```

6.2 ExpoSE

ExpoSE is a DSE tool which uses the Jalangi2 [17] framework to instrument a piece of JavaScript software in order to create a program trace. As the program terminates, ExpoSE calls the SMT solver Z3 [12] to identify all feasible alternate test-cases from the trace. These new test cases are then queued and the next test case is selected for execution, in the manner of generational search [16]. The ExpoSE framework allows for the parallel execution of individual test cases, aggregating coverage and alternative path information as each test case terminates. This parallelization is achieved by executing each

[1] Source code is available at https://github.com/ExpoSEJS

test case as a unique process allocated to a dedicated single core; as such the analysis is highly scalable.

Our strategy for test case selection is similar to the CUPA strategy proposed by Bucur et al. [8]. We use program fork points to prioritize unexplored code: each expression is given a unique identifier and scheduled test cases are sorted into buckets based upon which expression was being executed when they were created. We select the next test case by choosing a random test case from the bucket that has been accessed least during the analysis; this prioritizes test cases triggered by less common expressions.

7 Evaluation

We now empirically answer the following research questions:

(**RQ1**) Are non-classical regular expressions an important problem in JavaScript?
(**RQ2**) Does accurate modeling of ES6 regular expressions make DSE-based test generation more effective?
(**RQ3**) Does the performance of the model and the refinement strategy enable practical analysis?

We answer the first question with a survey of regular expression usage in the wild (§7.1). We address RQ2 by comparing our approach against an existing partial implementation of regular expression support in ExpoSE [24] on a set of widely used libraries (§7.2). We then measure the contribution of each aspect of our approach on over 1,000 JavaScript packages (§7.3). We answer RQ3 by analyzing solver and refinement statistics per query (§7.4).

7.1 Surveying Regular Expression Usage

We focus on code written for Node.js, a popular framework for standalone JavaScript. Node.js is used for both server and desktop applications, including popular tools *Slack* and *Skype*. We analyzed 415,487 packages from the NPM repository, the primary software repository for open source Node.js code. Nearly 35% of NPM packages contain a regular expression, 20% contain a capture group and 4% contain a backreference.

Methodology We developed a lightweight static analysis that parses all source files in a package and identifies regular expression literals and function calls. We do not detect expressions of the form `new RegExp(...)`, as they would generally require a more costly analysis. Our numbers therefore provide a lower bound for regular expression usage.

Results We found regular expression usage in JavaScript to be widespread, with 145,100 packages containing at least one regular expression out of a total 415,487 scanned packages. Table 4 lists the number of NPM packages containing regular expressions, capture groups, backreferences, and backreferences appearing within quantification. Note that a significant number of packages make use of capture groups and backreferences, confirming the importance of supporting them.

Table 4. Regular expression usage by NPM package.

Feature	Count	%
Packages on NPM	415,487	100.0%
. . . with source files	381,730	91.9%
. . . with regular expressions	145,100	34.9%
. . . with capture groups	84,972	20.5%
. . . with backreferences	15,968	3.8%
. . . with quantified backreferences	503	0.1%

Table 5. Feature usage by unique regular expression.

Feature	Total	%	Unique	%
Total Regex	9,552,546	100%	305,691	100%
Capture Groups	2,360,178	24.71%	119,051	38.94%
Global Flag	2,620,755	27.44%	90,356	29.56%
Character Class	2,671,565	27.97%	71,040	23.24%
Kleene+	1,541,336	16.14%	67,508	22.08%
Kleene*	1,713,713	17.94%	66,526	21.76%
Ignore Case Flag	1,364,526	14.28%	58,831	19.25%
Ranges	1,273,726	13.33%	52,155	17.06%
Non-capturing	1,236,533	12.94%	25,946	8.49%
Repetition	360,578	3.7%	17,068	5.58%
Kleene* (Lazy)	230,060	2.41%	13,250	4.33%
Multiline Flag	137,366	1.44%	10,604	3.47%
Word Boundary	336,821	3.53%	9,677	3.17%
Kleene+ (Lazy)	148,604	1.56%	6,072	1.99%
Lookaheads	176,786	1.85%	3,123	1.02%
Backreferences	64,408	0.67%	2,437	0.80%
Repetition (Lazy)	2,412	0.03%	221	0.07%
Quantified BRefs	1,346	0.01%	109	0.04%
Sticky Flag	98	<0.01%	60	0.02%
Unicode Flag	73	<0.01%	48	0.02%

Table 5 reports statistics for all 9M regular expressions collected, giving for each feature the fraction of expressions including it. Many regular expressions in NPM packages are not unique; this appears to be due to repeated inclusion of the same literal (instead of introduction of a constant), the use of online solutions to common problems, and the inclusion of dependencies (foregoing proper dependency management). To adjust for this, we provide data for both all expressions encountered and for just unique expressions. In both cases, there are significant numbers of capture groups, backreferences, and other non-classical features. As the occurrence rate of quantified backreferences is low, we do not differentiate between mutable and immutable backreferences.

Conclusions Our findings confirm that regular expressions are widely used and often contain complex features. Of particular importance is a faithful treatment of capture groups, which appear in 20.45% of the packages examined.

Table 6. Statement coverage with our approach (**New**) vs. [24] (**Old**) and the relative increase (+) on popular NPM packages (**Weekly** downloads). **LOC** are lines loaded and **RegEx** are regular expression functions symbolically executed.

Library	Weekly	LOC	RegEx	Old(%)	New(%)	+(%)
babel-eslint	2,500k	23,047	902	21.0	26.8	27.6
fast-xml-parser	20k	706	562	3.1	44.6	1,338.7
js-yaml	8,000k	6,768	78	4.4	23.7	438.6
minimist	20,000k	229	72,530	65.9	66.4	0.8
moment	4,500k	2,572	21	0.0	52.6	∞
query-string	3,000k	303	50	0.0	42.6	∞
semver	1,800k	757	616	51.7	46.2	−10.6
url-parse	1,400k	322	448	60.9	71.8	17.9
validator	1,400k	2,155	94	67.5	72.2	7
xml	500k	276	1,022	60.2	77.5	28.7
yn	700k	157	260	0.0	54.0	∞

On the flip side, since the occurrence of quantified backreferences is low, at just 0.01% of regular expressions found, the unsound optimized rule introduced in §4.3 will rarely lead to additional underapproximation during DSE.

7.2 Improvement Over State of the Art

We now compare our approach against the only available and functional implementation of regular expression support in JavaScript, which is part of the original ExpoSE [24].

Methodology We evaluated the statement coverage achieved by both versions of ExpoSE on a set of libraries, which we chose for their popularity (with up to 20M weekly downloads) and use of regular expressions. This includes the three libraries minimist, semver, and validator evaluated by Loring et al. [24]. To fairly compare original ExpoSE against our extension, we use the original automated library harness of Loring et al. [24] for both. Therefore we do not take advantage of other improvements for test generation, such as symbolic array support, which we have added in the course of our work. We re-executed each package six times for one hour each on both versions, using 32-core machines with 256GB of RAM, and averaged the results. We limited the refinement scheme to 20 iterations, which we identified as effective in preliminary testing (see §7.4).

Results Table 6 contains the results of our comparison. To provide an indication of program size, we use the number of lines of code loaded at runtime (JavaScript's dynamic method of loading dependencies makes it hard to determine meaningful LOC statically).

The results show that ExpoSE extended with our model and refinement strategy can improve coverage more than 10-fold on our sample of widely-used libraries. In some cases, the lack of ES6 support in the existing ExpoSE prohibited meaningful analysis, leading to 0% coverage. In the case of semver, we see a decrease in coverage if stopped after 1 hour. This is due to the modeling of complex regular expressions increasing solving time (see also §7.4). The coverage deficit disappears when executing both versions of ExpoSE with a timeout of 2 hours.

Conclusions We find that our modifications to ExpoSE make test generation more effective in widely used libraries using regular expressions. This suggests that the new method of solving regular expression queries presented in this paper has a substantial impact on practical problems in DSE. We also see that other improvements to ExpoSE, such as ES6 support, have affected coverage. Therefore, we continue with an evaluation of the individual aspects of our model.

7.3 Breakdown of Contributions

We now drill down into the contributions our individual improvements in regular expression support are making to increases in coverage.

Methodology From the packages with regular expressions from our survey §7.1, we developed a test suite of 1,131 NPM libraries for which ExpoSE is able to automatically generate a meaningful test harness. In each of the libraries selected, ExpoSE executed at least one regular expression operation on a symbolic string, which ensures that the library contains some behavior relevant to the scope of this paper. The test suite constructed in this manner contains numerous libraries that are dependencies of packages widely used in industry, including `Express` and `Lodash`.[2]

Automatic test generation typically requires a bespoke test harness or set of parameterized unit tests [30] to achieve high coverage in code that does not have a simple command line interface, including libraries. ExpoSE's harness explores libraries fully automatically by executing all exported methods with symbolic arguments for the supported types `string`, `boolean`, `number`, `null` and `undefined`. Returned objects or functions are also subsequently explored in the same manner.

We executed each package for one hour, which typically allowed to reach a (potentially initial) coverage plateau, at which additional test cases do not increase coverage further. We break down our regular expression support into four levels and measure the contribution and cost of each one to line coverage and test execution rate (Table 7). As baseline, we first execute all regular expression methods concretely, concretizing the arguments and results. In the second configuration, we add the model of ES6 regular expressions and their methods, including support for word boundaries and lookaheads, but remove capture groups and concretize any accesses to them, including backreferences. Third, we also enable full support for capture groups and backreferences.

[2]Raw data for the experiments, including all package names, is available at https://github.com/ExpoSEJS/Targets/tree/master/stdouts

Table 7. Breakdown of how different components contribute to testing 1, 131 NPM packages, showing number (#) and fraction (%) of packages with coverage improvements, average coverage increase (**Cov**), and test execution rate.

RegEx Support Level	Improved #	Improved %	Cov	Tests min
Concrete Regular Expressions	-	-	-	11.46
+ Modeling RegEx	528	46.68%	+3.09%	10.14
+ Captures & Backreferences	194	17.15%	+2.30%	9.42
+ Refinement	63	5.57%	+2.18%	8.70
All Features vs. Concrete	617	54.55%	+3.39%	

Fourth, we finally also add the refinement scheme to address over-approximation.

Results Table 7 shows, for each level of support, the number and percentage of target packages where coverage improved; the geometric mean of the absolute increase in coverage; and the mean test execution rate. The final row shows the effect of enabling full support compared to the baseline. Note that the number of packages improved is less than the sum of the rows above, since the coverage of a package can be improved by multiple features.

In a dataset of this size that includes many libraries making only little use of regular expressions, average coverage increases are expected to be small. Nevertheless, we see that dedicated support improves the coverage of more than half of packages that symbolically executed at least one regular expression function. As expected, the biggest improvement comes from supporting basic symbolic execution of regular expressions, even without capture groups or regard for matching precedence. However, we see further improvements when adding capture groups, which shows that they indeed affect program semantics. The advantages from the refinement scheme are less pronounced. This is because generated models have a chance of generating correct inputs on the first attempt, even in ambiguous settings.

On some libraries in the dataset, the approach is highly effective: For example, in the manifest parser *n4mf-parser*, full support improves coverage by 29% over concrete; in the format conversion library *sbxml2json*, by 14%; and in the browser detection library *mario*, by 16%. In each of these packages the refinement scheme contributed to the improvement in coverage. In general, the largest increases are seen in packages including regular expression-based parsers.

Each additional feature causes a small decrease in average test execution rate. Although a small fraction (~1%) of queries can take longer than 300s to solve, concurrent test execution prevents DSE from stalling on a single query.

Conclusions Full support for ES6 regular expressions improves performance of DSE of JavaScript in practice at a cost of a 16% increase in execution time (RQ2). An increase in coverage at lower execution rate in a fixed time window suggests that full regular expression support increases the quality of individual test cases.

Table 8. Solver times per package and query.

Packages/Queries	Constraint Solver Time Minimum	Maximum	Mean
All packages	0.04s	12h 15m	2h 34m
With capture groups	0.20s	12h 15m	2h 40m
With refinement	0.46s	12h 15m	2h 48m
Where refinement limit is hit	3.49s	11h 07m	3h 17m
All queries	0.001s	22m 26s	0.15s
With capture groups	0.001s	22m 26s	5.53s
With refinement	0.005s	18m 51s	22.69s
Where refinement limit is hit	0.120s	18m 51s	58.85s

7.4 Effectiveness on Real-World Queries

We now investigate the performance of the model and refinement scheme to answer RQ3. Finally, we also discuss the refinement limit and how it affects analysis.

Methodology We collected data on queries during the NPM experiments (§7.3) to provide details on SMT query success rates and execution times, as well as on the usage of the refinement scheme.

Results We found that 753 (66%) of the 1,131 packages tested executed at least one query containing a capture group or backreference. Of these packages, 653 (58% overall) contained at least one query to the SMT solver requiring refinement, and 134 (12%) contained a query that reached the refinement limit.

In total, our experiments executed 58,390,184 SMT queries to generate test cases. As expected, the majority do not involve regular expressions, but they form a significant part: 4,489,581 (7.6%) queries modeled a regular expression, 645,295 (1.1%) modeled a capture group or backreference, 74,076 (0.1%) required use of the refinement scheme and 2,079 (0.003%) hit the refinement limit. The refinement scheme was overwhelmingly effective: only 2.8% of queries with at least one refinement also reached the refinement limit (0.003% of all queries where a capture group was modeled). Of the refined SMT queries, the mean number of refinements required to produce a valid satisfying assignment was 2.9; the majority of queries required only a single refinement.

Table 8 details time spent processing SMT problems per-package and per-query. We provide the data over the four key aspects of the problem: we report the time spent in the constraint solver both per package and per query in total, as well as the time in the constraint solver for the particularly challenging parts of our strategy. We found that the use of refinements increased the average per-query solving time by a factor of four; however, this is dominated by SMT queries

that hit the refinement limit, which took ten times longer to run on average. The low minimum time spent in the solver in some packages can be attributed to packages where a regular expression was encountered early in execution but limitations in the test harness or function models (unrelated to regular expressions) prevented further exploration.

Conclusions We find the refinement scheme is highly effective, as it is able to solve 97.2% of encountered constraint problems containing regular expressions. It is also necessary, as 10% of queries containing a capture group had led to a spurioussatisfying assignment and required refinement.

Usually, only a small number of refinements required to produce a correct satisfying assignment. Therefore, even refinement limits of five or fewer are feasible and may improve performance with low impact on coverage.

8 Related Work

Closest to our work is that of Loring et al. [24], who introduce ExpoSE and initial support for encoding a subset of JavaScript regular expressions in terms of classical regular language membership and string constraints. The paper omits key features such as lookaheads, word boundaries, and anchors, which we incorporated in our complete model of ES6 regular expressions. They provide no solution to matching precedence, which we address with a refinement scheme. Furthermore, their lack of support for the full ES6 regular expression standard severely limits the ability to systematically explore arbitrary JavaScript programs (see §7.2).

In principle, regular expression engines can be symbolically executed themselves through the interpreter [8]. While this removes the need for modeling, in practice the symbolic execution of the entire interpreter and regular expression engine quickly becomes infeasible due to path explosion.

Several other approaches for symbolic execution of JavaScript were described in the literature; most include support for classical regular expressions, although to a more limited extent than the encoding of Loring et al. [24]. Li et al. [21] presented an automated test generation scheme for programs with regular expressions by on-line generation of a matching function for each regular expression encountered, exacerbating path explosion. Saxena et al. [26] proposed the first scheme to encode capture groups through string constraints. Sen et al. [28] presented Jalangi, a tool based on program instrumentation and concolic values. Li and Ghosh [20] and Li et al. [19] describe a custom browser and symbolic execution engine for JavaScript and the browser DOM, and a string constraint solver *PASS* with support for most JavaScript string operations. Although all of these approaches feature some support for ECMAScript regular expressions (such as limited support for capture groups), they ignore matching precedence and do not support backreferences or lookaheads.

Thomé et al. [29] propose a heuristic approach for solving constraints involving unsupported string operations. We choose to model unsupported operations and use of a CEGAR scheme to ensure correctness. Abdulla et al. [2] propose the use of a refinement scheme to solve complex constraint problems, including support for context-free languages. The language of regular expressions with backreferences is not context-free [9] and, as such, their scheme does not suffice for encoding all regular expressions; however, their approach could serve as richer base theory than classic regular expressions. Scott et al. [27] suggest backreferences can be eliminated via concatenation constraints, however they do not present a method for doing so.

Further innovations from the string solving community, such as work on the decidability of string constraints involving complex functions [11, 18] or support for recursive string operations [32, 33], are likely to improve the performance of our approach in future. We incorporate our techniques at the level of the DSE engine rather than the constraint solver, which allows our tool to leverage advances in string solving techniques; at the same time, we can take advantage of the native regular expression matcher and can avoid having to integrate implementation language-specific details for regular expressions into the solver.

A previous survey of regular expression usage across 4,000 Python applications [10] also provides a strong motivation for modeling regular expressions. Our survey extends this work to JavaScript on a significantly larger sample size.

9 Conclusion

We presented an approach to translating constraints involving real-world regular expressions. To the best of our knowledge, ours is the first comprehensive solution for ES6. We use a novel CEGAR scheme to address matching precedence, which so far had been largely ignored in related work. Evaluating our approach, we demonstrate that regular expressions are extensively used in JavaScript and that our novel solution outperforms existing partial approaches to the problem.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT Solver for String Constraints. In *Computer Aided Verification (CAV)*.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. 2017. Flatten and Conquer: A Framework for Efficient Analysis of String Constraints. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[3] Alfred V. Aho. 1990. Algorithms for Finding Patterns in Strings. In *Handbook of Theoretical Computer Science (Vol. A)*, Jan van Leeuwen (Ed.). MIT Press, Cambridge, MA, USA, 255–300.

[4] Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. 2012. SMT-LIB Sequences and Regular Expressions. In *Int. Workshop on Satisfiability Modulo Theories (SMT)*.

[5] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path Feasibility Analysis for String-Manipulating Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[6] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 87–100.

[7] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT – A formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software (ICRS 1975)*. ACM, 234–245.

[8] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[9] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. 2003. A Formal Study of Practical Regular Expressions. *Int. J. Foundations of Computer Science* 14, 06 (2003).

[10] Carl Chapman and Kathryn T. Stolee. 2016. Exploring Regular Expression Usage and Context in Python. In *Int. Symp. on Software Testing and Analysis (ISSTA)*.

[11] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. 2018. What is decidable about string constraints with the ReplaceAll function. *PACMPL* 2, POPL (2018), 3:1–3:29.

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[13] ECMA International. 2015. *ECMAScript 2015 Language Specification*. ECMA International.

[14] Xiang Fu, Michael C. Powell, Michael Bantegui, and Chung-Chih Li. 2013. Simple linear string constraints. *Formal Asp. Comput.* 25, 6 (2013).

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[16] Patrice Godefroid, Michael Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symp. (NDSS)*.

[17] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Int. Symp. on Software Testing and Analysis (ISSTA)*.

[18] Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. 2018. String constraints with concatenation and transducers solved efficiently. *PACMPL* 2, POPL (2018), 4:1–4:32.

[19] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: automatic symbolic testing of JavaScript web applications. In *Foundations of Software Engineering (FSE)*.

[20] Guodong Li and Indradeep Ghosh. 2013. PASS: String solving with parameterized array and interval automaton. In *Haifa Verification Conference (HVC)*.

[21] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Reggae: Automated test generation for programs using complex regular expressions. In *Automated Software Engineering (ASE)*.

[22] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *Computer Aided Verification (CAV)*.

[23] Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2015. A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings. In *Int. Symp. on Frontiers of Combining Systems (FroCoS)*.

[24] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *Proc. Int. SPIN Symposium on Model Checking Software (SPIN)*.

[25] José Fragoso Santos, Petar Maksimovic, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript verification toolchain. *PACMPL* 2, POPL (2018), 50:1–50:33. https://doi.org/10.1145/3158138

[26] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *IEEE Symp. Sec. and Privacy (S&P)*.

[27] Joseph D. Scott, Pierre Flener, and Justin Pearson. 2015. Constraint Solving on Bounded String Variables. In *Integration of AI and OR Tech. in Constraint Prog. (CPAIOR)*.

[28] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Foundations of Software Engineering (FSE)*.

[29] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel C. Briand. 2017. Search-driven string constraint solving for vulnerability detection. In *Int. Conf. on Software Engineering, (ICSE)*.

[30] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *Foundations of Software Engineering (FSE)*.

[31] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Conf. Computer and Commun. Sec. (CCS)*.

[32] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. Progressive Reasoning over Recursively-Defined Strings. In *Computer Aided Verification (CAV)*.

[33] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2017. Model Counting for Recursively-Defined Strings. In *Computer Aided Verification (CAV)*.

[34] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST)*.

[35] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. 2017. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design* 50, 2-3 (2017).

[36] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. 2015. Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints. In *Computer Aided Verification (CAV)*.

[37] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Foundations of Software Engineering (FSE)*.

A Renumbering Capture Groups

When syntactically rewriting regular expressions (see Table 1), we may introduce additional capture groups. For example, `/(a){1,3}/` becomes `/(a)(a)(a)|(a)(a)|(a)/`, introducing five additional capture groups. To correct this, we construct a correspondence between the two regular expressions relating their capture groups.

We distinguish two cases, Kleene plus and repetition. When rewriting a Kleene plus expression $S+$ containing K capturing parentheses, $S{*}S$ has $2K$ capture groups (the lazy case is similar). The original membership constraint would be of the form $C_1, \ldots, C_K \in \mathcal{L}_c(S+)$. After rewriting, the constraint becomes $C_0, C_{1,1}, \ldots, C_{K,1}, C_{1,2}, \ldots, C_{K,2} \in \mathcal{L}_c(S{*}S)$, where C_0 is defined as in §4.1. Since $S{*}S$ contains two copies of S, $C_{i,j}$ corresponds to the i^{th} capture in the j^{th} copy of S in $S{*}S$. We assert the direct correspondence between captures as

$$\begin{aligned}(w, C_0, C_1, \ldots, C_K) \in \mathcal{L}_c(S+) \iff \\ (w, C_0, C_{1,1}, \ldots, C_{K,1}, C_{1,2}, \ldots, C_{K,2}) \in \mathcal{L}_c(S{*}S) \\ \wedge \forall i \in \{1, \ldots, K\},\ C_i = C_{i,2}.\end{aligned}$$

For repetition (the lazy case is similar), if $S\{n, m\}$ has K capture groups then $S' = S^m \mid \ldots \mid S^n$ has $\frac{K}{2}(n+m)(n-m+1)$ captures. In S', suppose we index our captures as $C_{i,j,k}$ where index $i \in \{1, \ldots, K\}$ is the number of capture group in S, index $j \in \{0, \ldots, n-m\}$ denotes which alternate the capture group is in, and $k \in \{0, \ldots, m+j\}$ indexes the copies of S within each alternate. Intuitively, we pick a single $x \in \{0, \ldots, n-m\}$ that corresponds to the first satisfied alternate. Comparing the assignment of captures in $r\{n, m\}$ to S', we know that the value of the capture is the last possible match, so $C_i = C_{i,x,m+x}$ for all $i \in \{1, \ldots, K\}$. Formally, this direct correspondence can be expressed as:

$$
\begin{aligned}
&(w, C_0, C_1, \ldots, C_K) \in \mathcal{L}_c(S\{m, n\}) \iff \\
&\quad (w, C_0, C_{1,0,0}, \ldots, C_{K,n-m,n}) \in \mathcal{L}_c(S^m \mid \ldots \mid S^n) \\
&\wedge \exists x \in \{0, \ldots, n-m\} : \\
&\quad \big((w, C_0, C_{1,x,0}, \ldots, C_{K,x,m+x}) \in \mathcal{L}_c(S^{m+x}) \\
&\qquad \wedge \forall x' > x,\ (w, C_0, C_{1,x',0}, \ldots, C_{K,x',m+x'}) \notin \mathcal{L}_c(S^{m+x'}) \\
&\qquad \wedge \forall i \in \{1, \ldots, K\},\ C_i = C_{i,x,m+x}\big).
\end{aligned}
$$