Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints

Caleb Stanford*
University of Pennsylvania
Philadelphia, PA, USA
castan@cis.upenn.edu

Margus Veanes
Microsoft
Redmond, WA, USA
margus@microsoft.com

Nikolaj Bjørner
Microsoft
Redmond, WA, USA
nbjorner@microsoft.com

Abstract

The manipulation of raw string data is ubiquitous in security-critical software, and verification of such software relies on efficiently solving string and regular expression constraints via SMT. However, the typical case of Boolean combinations of regular expression constraints exposes blowup in existing techniques. To address solvability of such constraints, we propose a new theory of derivatives of symbolic extended regular expressions (extended meaning that complement and intersection are incorporated), and show how to apply this theory to obtain more efficient decision procedures. Our implementation of these ideas, built on top of Z3, matches or outperforms state-of-the-art solvers on standard and handwritten benchmarks, showing particular benefits on examples with Boolean combinations.

Our work is the first formalization of derivatives of regular expressions which both handles intersection and complement and works symbolically over an arbitrary character theory. It unifies existing approaches involving derivatives of extended regular expressions, alternating automata and Boolean automata by lifting them to a common symbolic platform. It relies on a parsimonious augmentation of regular expressions: a construct for symbolic conditionals is shown to be sufficient to obtain relevant closure properties for derivatives over extended regular expressions.

CCS Concepts: • **Security and privacy** → **Logic and verification**; • **Computing methodologies** → *Symbolic and algebraic algorithms*; • **Theory of computation** → *Regular languages.*

Keywords: regex, SMT, regular expression, derivative, automaton, string

*Work done during an internship at MSR.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454066

ACM Reference Format:
Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454066

1 Introduction

Regular expressions and finite automata play a fundamental role in many areas, ranging from applications in natural sciences [29] and NLP [48] to core problems in applied computer science, such as matching [25, 51, 57], model-checking [30], and solving of string constraints in SMT [31]. Recent years have seen a resurgence of interest in solvers for quantifier-free string and regular expression constraints, driven by software verification and security applications [3, 8, 16, 44]. However, there remains a gap between the theory of regular expressions (or regexes) and the constraints that arise in practice in such applications. We focus here on two aspects of this gap: (1) in typical applications, regexes exist over a symbolic potentially complex character theory rather than over a finite alphabet; and (2) in typical applications, multiple regex membership constraints may be combined using Boolean connectives. Modern SMT solvers thus need to efficiently solve Boolean combinations of regex constraints over a symbolic alphabet, rather than solving individual constraints in isolation over a finite one.

Although regexes are widely supported in most modern SMT string solvers [1, 2, 4, 9, 11, 19–21, 26, 32, 42, 50, 68–71], no current state-of-the-art tool provides a satisfactory solution to both of these challenges simultaneously. With respect to (1), modern strings that arise in applications are generally written in Unicode, but as of today, no SMT solver supports even the Basic Multilingual Plane (*BMP* or also known as *Plane 0*), while most widely used regex standards, e.g., the .NET regex standard [46] are based on BMP. Additionally, regexes that arise in practice employ *character classes* such as `\w` which denotes a word character, i.e. the subset of the character space (e.g. Unicode) which includes the Latin alphabet `a-z` and other alphabetic symbols. With respect to (2), we follow existing work by defining *extended regexes* to be those that allow intersection and complement. As we will see shortly, an efficient treatment of extended regexes has eluded existing techniques. One reason behind this is that adding intersection and complement fundamentally affects the difficulty of decision procedures of regexes. Recall that emptiness of extended regexes (*ERE*) is non-elementary [62], and (among other restricted fragments of *ERE*) *ERE* without complement is already PSPACE-hard [33] and PSPACE-complete when restricted further to intersections of classical regexes [39]. See also the more recent studies [27, 28, 41].

We believe that Boolean combinations of constraints represent the norm, rather than the exception, in practice. To give one illustrative application domain: cloud policy languages, such as Amazon AWS policies [8] and Microsoft Azure resource manager policies [45] utilize regexes for lightweight pattern matching. For example, Figure 1 shows a combination of constraints used to match a *date format*: a string which appears like a date, such as `2020-Nov-25`. The syntax elements `\d{4}`, `[a-zA-Z]{3}`, and `\d{2}` denote a sequence of four digits, a sequence of three letters, and a sequence of two digits, respectively; the remaining constraints then enforce that the first four digits should be either `2019` or `2020`. A sanity check here for SMT would be to make sure that the constraint is indeed satisfiable — for example, if we made a mistake and wrote `.*2019` and `.*2020` instead of `2019.*` and `2020.*`, then it would be unsatisfiable because this accidentally conflicts with the earlier constraint `\d{4}-`... which enforces that the year is at the beginning of the string. This would render this hypothetical audit policy useless (never activated) and would not match the user's intention. To combine the date constraints into a single *classical* regex (i.e., without any use of complement or intersection), is theoretically possible because regular languages are closed under Boolean operations. However, this might lead to an at-least-exponential blowup factor (due to the complexity results referenced in the previous paragraph). In addition, we cannot simply expect users not to write Boolean combinations. In fact in practice, industrial policy languages encourage and sometimes mandate the use of Boolean combinations by restricting regex syntax in various ways. For example, both the Amazon AWS and Microsoft Azure languages, as of 2020, among other restrictions, allow Kleene star in `.*` only (here `.*` is the regex matching any string). In particular, the disjunction (`anyOf`) in Figure 1 cannot for example be rewritten as `(2019|2020).*` using a single `like` or `match` expression. This makes the use of top level conjunction and complement, as in this date example, the native language for complex regular constraints, and raises the need to deal with Boolean combinations of regex constraints for analysis.

Existing Solutions. The way that current state-of-the-art solvers deal with Boolean combinations (intersection and complement) can be summarized by two main approaches:

1. Convert a regex r into an automaton M_r and then propagate the logical connectives into corresponding Boolean operations over automata: $(s \in r_1) \wedge (s \in r_2)$ is converted into $s \in L(M_{r_1} \times M_{r_2})$ and $\neg(s \in r)$ is converted into $s \in L(M_r^{\complement})$ [66].
2. Propagate the operations over regexes, by considering extended regexes, such as `(.*\d.*)&(.*[a-z].*)`, where & is intersection. Then, algebraically manipulate such extended regexes using *derivatives* [43].

```
{"if":{"allOf":[{"field":"date", "match":"####-???-##"},
               {"anyOf":[{"field":"date", "like":"2019*" },
                         {"field":"date", "like":"2020*"}]}]}
 "then":{"effect":"audit"}}
```

meaning : $\quad$ *date* $\in$ `\d{4}-[a-zA-Z]{3}-\d{2}` $\wedge$ (*date* $\in$ `2019.*` $\vee$ *date* $\in$ `2020.*`).

Figure 1. Example Boolean combination of regex constraints arising in practice: users of the Azure resource policy language [45] write a restricted form of regexes to control when a cloud resource should be audited. The semantics of the policy (top) is a Boolean combination of regex membership constraints (bottom), where # denotes a number (`\d`), ? denotes a letter (`[a-zA-Z]`), * denotes any sequence (`.*`), and we write `{n}` for n-fold iteration of a regex. Large Boolean combinations are either challenging or beyond reach for existing SMT string solvers (see Section 6).

While it is possible to extend classical automata algorithms to work modulo a character theory [24], the first approach has the following fundamental bottleneck. The construction of M_r is typically eager (the entire state space is constructed), and intersection and complement cause state space blowup for most automata models that are used. This means that constructing the state space for M_r is infeasible, such as for r = `~(.*a.{100})` (where `.*` matches any string, $\{n\}$ is n-fold repetition, and ~ is complement). This is a limitation because constructing M_r eagerly might not be needed in the first place: for example if checking satisfiability of r, it may be that an accepting state of M_r can be reached through exploration without constructing all states. On the other hand, if checking unsatisfiability of r, in product and complement constructions on automata, many more states are constructed than may actually be reachable (these can be eliminated through minimization of automata, but only after the fact). This suggests that we may be able to avoid constructing them in the first place.

On the other hand, the second approach addresses this state space blowup by leveraging *derivatives*, a syntactic way of exploring the state space of a regex without converting it to automata, pioneered by Brzozowski [14] and Antimirov [6]. The summary of the approach is that the derivatives of a regex correspond to the states of M_r, but they are constructed *lazily*. However, the second approach has another fundamental drawback: the lack of an appropriate formalism which both works symbolically and incorporates intersection and complement. As shown in [36], the classical theory of derivatives does not directly extend to the symbolic setting, because taking a symbolic derivative

(derivative with respect to a character predicate denoting a set B of characters) of an extended *symbolic* regex r does not in general lead to the desired semantics: it either results in an *over-approximation* or an *under-approximation* of the actual language, depending on whether the positive derivative $\Delta_B(r)$ or the negative derivative $\nabla_B(r)$ is taken [36, Lemma 3]. On the other hand, a classical generalization of Antimirov derivatives to extended regexes is possible (over a finite alphabet Σ) although challenging [17]; however, leveraging this work for the symbolic SMT setting would require explicitly enumerating (finitizing) the entire alphabet upfront (also known as *mintermization* in the literature [23, 24]; see Section 8.3). Local mintermization is also discussed in [36] in form of the *next literal* computation that creates a finite partition of the relevant predicates for computing a derivative precisely in the classical sense. For a general Boolean combination with n relevant predicates, this computation can in the worst case yield 2^n next literals. These techniques thus may be prohibitively expensive (e.g. for Unicode), and they additionally require considering all regex constraints in an SMT formula globally. Considering only intersection, and not complement, avoids some of this complexity and represents a state-of-the-art approach [43], but this loses the full generality of the Boolean operations.

Kleene algebras with tests (KAT) [40] have been studied in the context of derivative based automata constructions [53] when applied to standard regexes, i.e., regular expressions without intersection or complement. In KAT, character predicates can be represented succinctly by *tests*, e.g., by encoding predicates as BDDs [53]. However, Boolean operations over regular languages, in particular complement, appear to be incompatible with KAT because Boolean operations in KAT are defined on the Boolean-algebra subset, not the entire algebra. Recall that $\neg 1 = 0$ and $\neg 0 = 1$ in KAT where 1 is ε and 0 is $\bot$. However, $[\![{\sim}\bot]\!] = D^*$ and $[\![{\sim}\varepsilon]\!] = D^* \setminus \{\varepsilon\}$. Thus, identifying regular expression complement (~) with negation ($\neg$), in an extended regex such as ${\sim}((\varphi \cdot R_1) \mid (\neg\psi \cdot R_2))$, would break the Kleene algebra laws and the intended semantics.

This Work. We fill these gaps by proposing the first theory of derivatives of symbolic regexes which incorporates intersection and complement. Unlike previous work, our approach can be used to avoid the state-space blowup of automata-based solvers without assuming a finite alphabet and without under- and over-approximation. The *key new insight* that enables us to define derivatives of regexes directly, while allowing Boolean operations, is that we augment regexes with *conditionals* (if-then-else), and define the derivative of a regex to be a regex with conditionals, called a *transition regex*. We show that transition regexes allow for efficient algebraic manipulation rules for complementation and intersection: for example, given a regex which is a Boolean combination of classical regexes, we show that the number of derivatives is strictly linear (Theorem 7.3). We give a decision procedure based on our derivatives which integrates into a broader SMT context: a set of inference rules that incrementally unfolds regex constraints into symbolic constraints over the background character theory. Derivatives enable this lazy unfolding; the symbolic conditionals directly map to the underlying character theory; and the succinct handling of Boolean combinations via extended regexes avoids the blowup in existing techniques. We also introduce an accompanying theory of symbolic Boolean finite automata (SBFAs): the derivatives of an extended regex correspond to the states in the SBFA. This is used to prove the succinctness theorem and to study the connection with classical approaches and other techniques.

We have implemented symbolic Boolean derivatives in a new regular expression solver, dZ3, which is built on top of Z3 and fully replaces the existing solver. We show that the lack of blowup shows the expected benefits in practice. Compared to an array of state-of-the-art solvers, we show that our decision procedure matches or outperforms other solvers in terms of number of benchmarks solved and average time per benchmark. It shows particular benefits on examples with Boolean combinations: although CVC4 and Ostrich are competitive on subsets of the benchmarks, no solver consistently shows good performance across benchmark sets involving Boolean combinations. For example, dZ3 is 1.54x faster than the next best solver (CVC4) on average for existing benchmarks with Boolean combinations, and solves 88% of handwritten examples such as the date example in Figure 1, compared to 57% for CVC4.

Contributions.

- We introduce a new theory of symbolic derivatives of extended regexes, which avoids the blowup in existing techniques. It works via translation to *transition regexes* which augment extended regexes with a conditional construct. (Section 4)
- We propose a sound and conditionally complete decision procedure for solving extended regular expression constraints in an SMT context. (Section 5)
- We provide a proof-of-concept open-source implementation on top of Z3, called dZ3.[1] Using standard existing benchmark sets, existing benchmarks focused on Boolean combinations, and additional handwritten examples, we show that our solver matches or outperforms state-of-the-art solvers for string constraints and shows particular performance and solvability improvements on Boolean combinations. (Section 6)
- To formally study the benefits of our approach, we introduce a theory of Symbolic Boolean Finite Automata

[1] The solver is available with the latest release of Z3 at https://github.com/Z3Prover/z3, with experimental scripts at https://github.com/cdstanford/dz3-artifact. Benchmarks used for the paper can be found at https://github.com/cdstanford/regex-smt-benchmarks and at https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks.

(SBFAs) that generalizes the classical approaches of alternating and Boolean automata to the symbolic setting. In particular, we use SBFAs to show that for a common subclass of extended regexes, the set of symbolic derivatives has linear size (**Theorem 7.3**). (Section 7)

- We provide an in-depth comparison of our theory of derivatives with the classical theory. (Section 8).

2 Motivating Running Example

We discuss here a motivating example that helps us highlight some of the main ideas behind *transition regexes*, the key to defining derivatives for symbolic extended regular expressions. The example also serves as a running example and is referenced in the later sections. It is similar in spirit to the date example in Figure 1 and is typical to many of the benchmarks used in Section 6.

Suppose we are given a membership constraint $in(s, R)$, where s is a string term over an alphabet type Σ, i.e., s has type Σ^*, and R is a concrete regex over Σ^*. (The syntax $in(s, R)$, corresponding to SMT-LIB `str.in_re`, denotes that s matches the regex R.) Our goal is to solve the *satisfiability* problem for that membership constraint: does there exist a concrete instance of s in Σ^* such that R accepts that instance? Using the approach of derivatives, we plan to attack the problem by calculating the derivatives of R, by deducing the following case split: [2]

$$(|s| = 0 \wedge nullable(R)) \vee (|s| > 0 \wedge in(s_{1..}, \delta(R)(s_0))),$$

where $nullable(R)$ is true if R accepts the empty string, and $\delta(R)$ is a function of R called its *derivative*: it takes a regex R and a character s_0, and returns a regex for the language of *suffixes* w such that $s_0 w \in L(R)$ holds.

However, the classical theory of derivatives does not directly apply here: *the problem is that the string s may be uninterpreted (we don't know the first character s_0)*, and classical derivatives are only defined for a given input character. In other words, s is a variable of type string, which does not have a value yet, as opposed to being a fixed string like "cat". We could naively enumerate all possible characters $\bigvee_{a\in\Sigma}(s_{1..}{\in}D^{Brz}_a(R) \wedge s_0{=}a)$, where $D^{Brz}_a(R)$ is the classical Brzozowski derivative [14] (defined independently for each character a), but this does not scale.

Our contribution is to address this by providing a closed definition of $\delta(R)$ above: in particular, we want to be able to evaluate $\delta(R)$ symbolically, before knowing s_0. We call this the *symbolic derivative*, and we call the resulting term a *transition regex*: it denotes a function from Σ to regexes.

More concretely, take R to be a typical *password constraint*:

$$in(s, \texttt{.*\textbackslash d.*}) \wedge \neg in(s, \texttt{.*01.*})$$

This constraint states that s contains at least one digit but not the subsequence `01`. Regular expressions such as this one are used in the generation and validation of password strings. In typical real-world cases, they may involve many more similar simultaneous constraints (cf. [52]), which can be encoded as large intersections (cf. [61]). The motivation for derivative-based approaches is that such constraints — in particular because they are also combined with bounded loops such as `.{8,128}` — cause an explosion of the state space when converted to automata [23]. By unfolding the derivatives of R, we will explore possible strings for s without constructing the state space up front.

We now show how to solve the constraint $in(s, R)$ for this example, using our approach, and following our implementation in dZ3. The negation is first converted into a regex complement and then the conjunction into an intersection: $in(s, \texttt{(.*\textbackslash d.*) \& \textasciitilde(.*01.*)})$. Let $R_1 = \texttt{.*\textbackslash d.*}$, $R_2 = \texttt{\textasciitilde(.*01.*)}$ and $R = R_1 \,\texttt{\&}\, R_2$. Since R is not nullable (does not accept the empty string), the case split we started from reduces to the assertion $|s| > 0 \wedge in(s_{1..}, \delta(R)(s_0))$. To calculate $\delta(R)$ as a transition regex, we need to deal with the problem that we do not know s_0. The solution is to *augment regexes with conditionals (if-then-else)*, and then allow conditionals in transition regexes. When taking the derivative of a regex such as `01`, we let $\varphi_{\texttt{0}}$ be the predicate $\lambda x.x = \texttt{0}$ and we construct the term $\textsc{if}(\varphi_{\texttt{0}}, \texttt{1}, \bot)$, read as: given input x, *if* $x = \texttt{0}$ *then* `1` *else* $\bot$. This idea allows for the derivative of R to be computed using algebraic rules as follows. The $\equiv$ relation below indicates simplification steps using distributivity, De Morgan's laws, and other properties that are not part of the derivation itself. Below, φ_d stands for the predicate for the character class `\d`, i.e., digits. The conditional expression $\textsc{if}(\varphi, \tau, \rho)$ formally abbreviates $\lambda x.\textsc{if}(\varphi(x), \tau(x), \rho(x))$.

$$\begin{aligned}
\delta(R) &= \delta(R_1) \,\texttt{\&}\, \delta(R_2) \\
\delta(R_1) &= R_1 \mid \textsc{if}(\varphi_d, \texttt{.*}, \bot) \equiv \textsc{if}(\varphi_d, \texttt{.*}, R_1) \\
\delta(R_2) &= \texttt{\textasciitilde}(\delta(\texttt{.*01.*})) = \texttt{\textasciitilde}(\texttt{.*01.*} \mid \delta(\texttt{01.*})) \\
&= \texttt{\textasciitilde}(\texttt{.*01.*} \mid \textsc{if}(\varphi_{\texttt{0}}, \texttt{1.*}, \bot)) \\
&\equiv \texttt{\textasciitilde}(\texttt{.*01.*}) \,\texttt{\&}\, \texttt{\textasciitilde}(\textsc{if}(\varphi_{\texttt{0}}, \texttt{1.*}, \bot)) \\
&\equiv R_2 \,\texttt{\&}\, \textsc{if}(\varphi_{\texttt{0}}, \texttt{\textasciitilde(1.*)}, .*) \\
&\equiv \textsc{if}(\varphi_{\texttt{0}}, R_2 \,\texttt{\&}\, \texttt{\textasciitilde(1.*)}, R_2) \\
\delta(R) &\equiv \textsc{if}(\varphi_d, \texttt{.*}, R_1) \,\texttt{\&}\, \textsc{if}(\varphi_{\texttt{0}}, R_2 \,\texttt{\&}\, \texttt{\textasciitilde(1.*)}, R_2) \\
&\overset{\text{(i)}}{\equiv} \textsc{if}(\varphi_{\texttt{0}}, R_2 \,\texttt{\&}\, \texttt{\textasciitilde(1.*)}, \textsc{if}(\varphi_d, \texttt{.*}, R_1) \,\texttt{\&}\, R_2) \\
&\equiv \textsc{if}(\varphi_{\texttt{0}}, R_2 \,\texttt{\&}\, \texttt{\textasciitilde(1.*)}, \textsc{if}(\varphi_d, R_2, R))
\end{aligned}$$

Observe that all conditional predicates are extracted from the regex itself: e.g. $\varphi_{\texttt{0}}$ in a conditional arises from `0` in the original regex. Step (i) uses (among other properties) that $\neg\varphi_d \wedge \varphi_{\texttt{0}}$ is unsat. Note also that $\texttt{\textasciitilde}\bot \equiv \texttt{.*}$ and $\texttt{.*|}\ldots \equiv \texttt{.*}$.

There is no direct classical counterpart to the above derivation sequence, because classical regexes do not have *if-then-else*. In particular, there is no direct classical counterpart which handles complement. For example, consider the regex `01.*` above. Classically, we would take the derivative as $D^{Brz}_{\texttt{0}}(\texttt{01.*}) = \texttt{1.*}$. But what if we want to now take the derivative of the complement of `01.*`? Then we need to know

[2] We write s_i for the i'th element of s and $s_{i..}$ for the suffix from i. These can be purely symbolic expressions; s itself may be a variable (uninterpreted).

not just this derivative where the first character is `0` but also the derivative if the first character is *not* `0`, because while the latter case was impossible before it becomes relevant when considering the complement. Using conditionals solves this problem: we write the derivative as $\textsc{if}(\varphi_0, \texttt{1.*}, \bot)$, which has the case where the first character is not `0` present. Then, when complementing this, we get $\textsc{if}(\varphi_0, \texttt{\~{}(1.*)}, \texttt{.*})$. Thus, viewing the derivative as a conditional (transition regex) is what enables us to treat complement algebraically.

Having calculated $\delta(R)$ as above, we continue as follows. Let $R_3 = R_2 \,\&\, \texttt{\~{}(1.*)}$. So $in(s_{1..}, \delta(R)(s_0))$ reduces to

$$in(s_{1..}, \textsc{if}(s_0 = \texttt{0}, R_3, \textsc{if}(\varphi_d(s_0), R_2, R)))$$

Expanding the if-then-else creates the further case split:

$$(s_0 = \texttt{0} \wedge in(s_{1..}, R_3)) \vee (s_0 \neq \texttt{0} \wedge in(s_{1..}, \textsc{if}(\varphi_d(s_0), R_2, R)))$$

where $in(s_{1..}, R_3)$ splits further into two subcases:

$$(|s_{1..}| = 0 \wedge nullable(R_3)) \vee (|s_{1..}| > 0 \wedge in(s_{2..}, \delta(R_3)(s_1)))$$

where $(s_{1..})_{1..} = s_{2..}$ and $(s_{1..})_0 = s_1$, and the procedure repeats. Here R_3 is nullable so dZ3 can generate a model for $|s| > 0 \wedge |s_{1..}| = 0 \wedge s_0 = \texttt{0}$ — provided that these constraints are consistent with other constraints on s in the context. For example if there was a constraint $s_0 > \texttt{0}$, this case would be blocked and the search would backtrack to the other case.[3]

3 Preliminaries

Sequences. When working with sequences over a domain Σ we make the standard simplifying assumption that $\Sigma^{(1)} = \Sigma$, and let $\Sigma^{(0)} = \{\epsilon\}$, $\Sigma^{(k+1)} = \Sigma \cdot \Sigma^{(k)}$, for $k \geq 0$, and $\Sigma^* = \bigcup_{k\geq 0} \Sigma^{(k)}$, $\Sigma^+ = \bigcup_{k\geq 1} \Sigma^{(k)}$. Moreover, for $v \in \Sigma^{(k)}$, the length of v is k, $|v| = k$. In contrast, when Σ^* is implemented in an SMT solver the type Σ^* is *sequence over* Σ that is disjoint from Σ. For $X, Y \subseteq \Sigma^*$, define $X \cdot Y \subseteq \Sigma^*$ such that $X \cdot Y = \{x \cdot y \mid x \in X, y \in Y\}$ where concatenation $\cdot$ is associative and ϵ is the empty sequence. We write xy for $x \cdot y$ when it is clear from the context that juxtaposition stands for concatenation. Also, X^* stands for the closure of X under concatenation when it is clear from the context that $X \subseteq \Sigma^*$.

Boolean Algebras. Given a nonempty *universe* D, *Boolean algebra over* D is a tuple $\mathcal{A} = (D, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where Ψ is a set of *predicates* closed under the Boolean connectives; $[\![\_]\!] : \Psi \to 2^D$ is a *denotation function*; $\bot, \top \in \Psi$; $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = D$, and for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = D \setminus [\![\varphi]\!]$. For $\varphi, \psi \in \Psi$ we write $\varphi \equiv \psi$ (φ is *equivalent* to ψ) to mean $[\![\varphi]\!] = [\![\psi]\!]$. In particular, if $\varphi \equiv \bot$ then φ is *unsatisfiable* and if $\varphi \equiv \top$ then φ is *valid*. $\mathcal{A}$ is *effective* if all components of $\mathcal{A}$ are recursively enumerable, and satisfiability of $\varphi \in \Psi$ ($\varphi \not\equiv \bot$) is decidable. $\mathcal{A}$ is *extensional* if $\varphi \equiv \psi$ implies that $\varphi = \psi$.

[3]The condition $s_0 > \texttt{0}$ is possible because the underlying character theory (for example bitvectors) is equipped with a total order.

Given a finite set $S \subseteq \Psi$ of predicates, a *minterm* of S is a *satisfiable* predicate $\bigwedge_{\psi\in S} \psi'$ where $\psi' \in \{\psi, \neg\psi\}$. Let $Minterms(S)$ stand for a fixed set of all pairwise inequivalent minterms of S. Observe that $|Minterms(S)| \leq 2^{|S|}$ and that $\{[\![\alpha]\!] \mid \alpha \in Minterms(S)\}$ is a *partition* of D.

Boolean Combinations. If Q is a set of basic syntax elements then $\mathbb{B}(Q)$ denotes the *Boolean closure* over Q using `|` for disjunction, `&` for conjunction, and `~` for complement: i.e. the grammar generated by $R ::= Q \mid R \,\texttt{|}\, R \mid R \,\&\, R \mid \texttt{\~{}}R$. Similarly, $\mathbb{B}^+(Q)$ denotes the *positive* Boolean closure of Q (without use of `~`). In the context of a particular Q where syntactic rewrites are allowed, we will sometimes view `&` and `|` as *idempotent*, *associative* and *commutative* operators, and also rewrite `~~`q to q. We also lift `&` and `|` to *finite nonempty* subsets $S \subseteq Q$ through $AND(S)$ and $OR(S)$, respectively.

Symbolic Regexes. Let $\mathcal{A} = (\Sigma, \Psi, [\![\_]\!], \bot, ., \vee, \wedge, \neg)$ be a fixed effective Boolean algebra called an *alphabet theory*. Note that Σ may be infinite. We first recall the definitions of the two standard subclasses of regexes and extended regexes, where $\varphi \in \Psi$. We always work *modulo* $\mathcal{A}$ and we do not mention this explicitly every time.

$$RE ::= \varphi \mid \varepsilon \mid \bot \mid RE_1 \cdot RE_2 \mid RE{*} \mid RE_1 \,\texttt{|}\, RE_2$$
$$ERE ::= \varphi \mid \varepsilon \mid \bot \mid ERE_1 \cdot ERE_2 \mid ERE{*} \mid \mathbb{B}(ERE)$$

The class RE corresponds to all standard regexes. The fragment $\mathbb{B}(RE) \subset ERE$ comprises all Boolean combinations over RE and covers *all* of our practical scenarios. The *language accepted by* R, denoted $\mathbf{L}(R) \subseteq \Sigma^*$, is defined by:

$\mathbf{L}(\varphi) = [\![\varphi]\!]$, $\mathbf{L}(\varepsilon) = \{\epsilon\}$, $\mathbf{L}(\bot) = \emptyset$,
$\mathbf{L}(R_1 \cdot R_2) = \mathbf{L}(R_1) \cdot \mathbf{L}(R_2)$, $\mathbf{L}(R{*}) = \mathbf{L}(R)^*$,
$\mathbf{L}(R_1 \,\texttt{|}\, R_2) = \mathbf{L}(R_1) \cup \mathbf{L}(R_2)$, $\mathbf{L}(R_1 \,\&\, R_2) = \mathbf{L}(R_1) \cap \mathbf{L}(R_2)$,
$\mathbf{L}(\texttt{\~{}}R) = \Sigma^* \setminus \mathbf{L}(R)$

A regex R is *nullable* ($\nu(R)$) iff $\epsilon \in \mathbf{L}(R)$. Nullability can be computed inductively: $\nu(\varphi) = \nu(\bot) = \mathit{false}$; $\nu(\varepsilon) = \nu(R{*}) = \mathit{true}$; $\nu(R_1 \cdot R_2) \Leftrightarrow \nu(R_1)$ *and* $\nu(R_2)$; $\nu(R_1 \& R_2) \Leftrightarrow \nu(R_1)$ *and* $\nu(R_2)$; $\nu(R_1 \,\texttt{|}\, R_2) \Leftrightarrow \nu(R_1)$ *or* $\nu(R_2)$; $\nu(\texttt{\~{}}R) \Leftrightarrow$ *not* $\nu(R)$. Given $R \in ERE$ we let Ψ_R denote the set of all predicates φ that occur in R.

4 Symbolic Derivatives

Here we formally introduce the key concept of *transition regexes TR*, in which regexes are augmented with conditionals. We define *symbolic derivatives* for $R \in ERE$ in terms TR, and prove their correctness in Theorem 4.3. We also discuss some algebraic laws that hold in TR — used as simplification rules in dZ3 — as illustrated in Section 2.

Transition Regexes. The definition of TR depends on a type parameter Q; for the present section $Q = ERE$, but the general case will be useful later in Section 7. Let $\diamond \in \{\texttt{|}, \&\}$, $\bar{\&} = \texttt{|}$ and $\bar{\texttt{|}} = \&$. Then TR, or TR_Q, is defined by the grammar

$$TR ::= Q \mid \textsc{if}(\varphi, TR_1, TR_2) \mid \mathbb{B}(TR)$$

We call $\textsc{if}(\varphi, \tau_1, \tau_2)$ a *conditional regex*. A transition regex τ denotes the function $\tau : \Sigma \to \mathbb{B}(Q)$ defined as follows.[4]

$$
\begin{aligned}
R(x) &= R \quad (\text{for } R \in Q)\\
\textsc{if}(\varphi,\tau,\rho)(x) &= \begin{cases} \tau(x), & \text{if } x \in [\![\varphi]\!];\\ \rho(x), & \text{otherwise.}\end{cases}\\
\tau \diamond \rho(x) &= \tau(x) \diamond \rho(x)\\
{\sim}\tau(x) &= {\sim}(\tau(x))
\end{aligned}
$$

Transition regexes τ and ρ are *equivalent*, denoted $\tau \equiv \rho$, when $\forall x \in \Sigma$, $\mathbf{L}(\tau(x)) = \mathbf{L}(\rho(x))$. Concatenation of regexes is lifted to transition regexes τ in $\tau \cdot R$ for $R \in \mathit{ERE}$:

$$
\begin{aligned}
\textsc{if}(\varphi,\tau,\rho)\cdot R &= \textsc{if}(\varphi, \tau\cdot R, \rho \cdot R)\\
(\tau \mid \rho)\cdot R &= (\tau \cdot R) \mid (\rho \cdot R)\\
{\sim}\tau \cdot R &= \overline{\tau} \cdot R\\
(\tau \mathbin{\&} \rho)\cdot R &= \mathit{lift}(\tau \mathbin{\&} \rho)\cdot R
\end{aligned}
$$

The definition of $\mathit{lift}(\tau)$ is such that if $\tau \in Q$ then $\mathit{lift}(\tau) = \tau$ else τ is transformed into an equivalent conditional regex by lifting the character predicates to the top while pushing conjunction into the leaves. (The precise lift rules are discussed in Section 4.1.) Finally, *negation* $\overline{\tau}$ of τ is defined as follows.

$$\overline{R} = {\sim}R,\ \ \overline{{\sim}\tau} = \tau,\ \ \overline{\tau \diamond \rho} = \overline{\tau} \mathbin{\bar{\diamond}} \overline{\rho},\ \ \overline{\textsc{if}(\varphi,\tau,\rho)} = \textsc{if}(\varphi, \overline{\tau}, \overline{\rho})$$

The following lemmas represent key semantic properties that are used in several contexts. Lemma 4.1 is used in the proof of Theorem 4.3 and Lemma 4.2 is correctness of negation that is for example exploited in normal forms. Both lemmas are proved by induction over τ using various algebraic laws of *TR*.

Lemma 4.1. $\mathbf{L}(\tau \cdot R(x)) = \mathbf{L}(\tau(x)) \cdot \mathbf{L}(R)$

Lemma 4.2. ${\sim}\tau \equiv \overline{\tau}$

The *symbolic derivative* $\delta(R)$ of a regex $R \in \mathit{ERE}$ is defined as the following transition regex, where $\varphi \in \Psi$.

$$
\begin{aligned}
\delta(\varepsilon) = \delta(\bot) &= \bot\\
\delta(\varphi) &= \textsc{if}(\varphi, \varepsilon, \bot)\\
\delta(R \cdot R') &= \begin{cases} \delta(R)\cdot R' \mid \delta(R'); & \text{if } R \text{ is nullable,}\\ \delta(R)\cdot R'; & \text{otherwise.}\end{cases}\\
\delta(R*) &= \delta(R) \cdot R*\\
\delta(R \diamond R') &= \delta(R) \diamond \delta(R') \quad (\text{for } \diamond \in \{\&, \mid\})\\
\delta({\sim}R) &= {\sim}\delta(R)
\end{aligned}
$$

Theorem 4.3 is the correctness theorem of symbolic derivatives. For $L \subseteq \Sigma^*$ and $a \in \Sigma$, recall the classical definition of the *derivative of* L *w.r.t.* a, $\mathbf{D}_a(L) = \{v \mid av \in L\}$, and for $R \in \mathit{ERE}$ we use *Brzozowski derivatives* $D_a^{Brz}(R) \in \mathit{ERE}$ (modulo $\mathcal{A}$ [36]), and the classical result $\mathbf{L}(D_a^{Brz}(R)) = \mathbf{D}_a(\mathbf{L}(R))$ [14, Theorem 3.1]. Let $\mathbf{D}_a(R) = \mathbf{L}(D_a^{Brz}(R))$. Let $\hat{\mid} = \cup$ and $\hat{\&} = \cap$.

Theorem 4.3. $\mathbf{L}(\delta(R)(a)) = \mathbf{L}(D_a^{Brz}(R))$.

[4]Function application of (x) binds weakest, so $\tau \diamond \rho(x)$ stands for $(\tau \diamond \rho)(x)$.

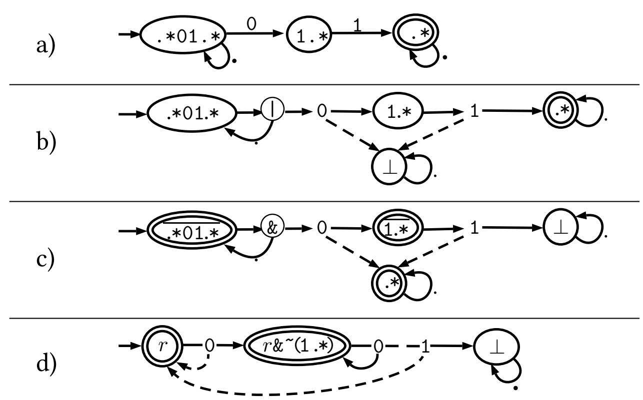


Figure 2. Example symbolic derivations on extended regexes, viewed as transitions between regexes.

Proof. By induction over R. The base cases $\bot$ and ε are trivial. **Base case** φ**:** $\delta(\varphi) = \textsc{if}(\varphi, \varepsilon, \bot)$. If $a \in [\![\varphi]\!]$ then $\textsc{if}(\varphi, \varepsilon, \bot)(a)$ becomes $\varepsilon(a) = \varepsilon = D_a^{Brz}(\varphi)$, else it becomes $\bot = D_a^{Brz}(\varphi)$. **Induction case** $R \cdot R'$**:** If R is nullable, then $\mathbf{L}(\delta(R{\cdot}R')(a)) = \mathbf{L}(\delta(R){\cdot}R' \mid \delta(R')(a)) = \mathbf{L}(\delta(R){\cdot}R'(a) \mid \delta(R')(a))$. Now this is $\mathbf{L}(\delta(R)(a)){\cdot}\mathbf{L}(R') \cup \mathbf{L}(\delta(R')(a))$, which by IH equals $\mathbf{D}_a(R){\cdot}\mathbf{L}(R') \cup \mathbf{D}_a(R') = \mathbf{D}_a(R{\cdot}R')$. If R is not nullable, then $\mathbf{L}(\delta(R{\cdot}R')(a)) = \mathbf{L}(\delta(R){\cdot}R'(a)) = \mathbf{L}(\delta(R)(a)){\cdot}\mathbf{L}(R')$; applying IH, $\mathbf{D}_a(R){\cdot}\mathbf{L}(R') = \mathbf{D}_a(R{\cdot}R')$. **Induction case** $R*$: We first simplify $\mathbf{L}(\delta(R*)(a)) = \mathbf{L}(\delta(R){\cdot}R*(a)) = \mathbf{L}(\delta(R)(a)){\cdot}\mathbf{L}(R*)$. Applying IH again, $= \mathbf{D}_a(R){\cdot}\mathbf{L}(R*) = \mathbf{D}_a(R*)$. **Induction case** $R{\diamond}R'$ **(where** $\diamond \in \{\mid, \&\}$**):** here, $\mathbf{L}(\delta(R{\diamond}R')(a)) = \mathbf{L}(\delta(R)(a)) \mathbin{\hat{\diamond}} \mathbf{L}(\delta(R')(a))$, and by IH we get $\mathbf{D}_a(R) \mathbin{\hat{\diamond}} \mathbf{D}_a(R') = \mathbf{D}_a(R{\diamond}R')$. **Induction case** ${\sim}R$: Finally, we have $\mathbf{L}(\delta({\sim}R)(a)) = \Sigma^* \setminus \mathbf{L}(\delta(R)(a))$, which by IH becomes $\Sigma^* \setminus \mathbf{D}_a(R) = \mathbf{D}_a({\sim}R)$. ◻

Corollary 4.4. *If* $R \in \mathbb{B}(\mathit{RE})$ *then* $\delta(R)(a) \in \mathbb{B}(\mathit{RE})$.

Proof. If $R \in \mathbb{B}(\mathit{RE})$ then lifting (see Section 4.1) is never invoked, because concatenations never arise with a &-term on the left. Inductively, this implies that ~ and & remain as top-level operators, never nested inside $\cdot$ or $*$. ◻

Example 4.5. Consider the regex `.*01.*` from above. We write individual characters also for the corresponding singleton predicates when this is unambiguous, except that $[\![.]\!] = \Sigma$. We implicitly use the simplification rule that $\textsc{if}(., \tau, \_) \equiv \tau$. Thus, e.g., $\delta(.)$ simplifies to ε (and so $\delta(.)r$ simplifies to r).

$$
\begin{aligned}
\delta(.{*}01.{*}) &= \delta(.{*}){\cdot}01.{*} \mid \delta(01.{*})\\
&= \delta(.){\cdot}.{*}01.{*} \mid \delta(0){\cdot}1.{*} = .{*}01.{*} \mid \textsc{if}(0, 1.{*}, \bot)\\
\delta(1.{*}) &= \textsc{if}(1, .{*}, \bot)
\end{aligned}
$$

The two transition regexes are shown as classical transitions in Figure 2a where $\bot$ is hidden. The equivalent *complete* view of the transition regexes is shown in Figure 2b where the dashed arrows represent the false branches of conditional regexes. The negation of the complete form is seen in Figure 2c as the dual of Figure 2b, where $\overline{\bot} = .*$, and

$\overline{.*} = \bot$. Finally, Figure 2d is the DNF form of Figure 2c, where $r =$ `~(.*01.*)`. Regexes which are *nullable* (or *final*) are denoted with double boundary. ◻

Algebraic Properties. Transition regexes form a particular kind of effective Boolean algebra. [5] The regex $.*$ is treated as the *absorbing* element of `|` and the *unit* element of `&`. Conversely, $\bot$ is treated as the unit element of `|` and the absorbing element of both `&` and $\cdot$. For example r & $.* = r$ and $\bot \cdot r = \bot$. We also treat `|`, `&`, $\cdot$ as associative operators and `|`, `&` as commutative idempotent operators. This is important in reducing the number of different but equivalent regexes from arising during search. However, the algebra is only modulo these syntactic rules, and not *all* possible simplifications: this means that the algebra is *not extensional*, i.e., $\tau \equiv \rho$ does not in general imply $\tau = \rho$.

We exploit this algebra for different algebraic simplifications and normal forms. The most important one is *disjunctive normal form* or *DNF*. Here we consider $\tau = \delta(R)$ for $R \in \mathbb{B}(RE)$ but DNF generalizes to all $R \in ERE$ by using $lift(\tau)$ (Section 4.1). We consider DNF *with respect to the Boolean structure of TR* where any element of *ERE* is considered to be atomic. Moreover, any nested conditional regex whose leaves are all in *ERE* is already in DNF. Thus, a transition regex is in DNF if it is a disjunction of conditional regexes whose leaves are all in *ERE*.

For example $\text{IF}(\varphi, \tau_1, \tau_2)$ & ρ is not in DNF but expands to $\text{IF}(\varphi, \tau_1 \,\&\, \rho, \tau_2 \,\&\, \rho)$ and is also subject to simplifications discussed next that integrate satisfiability checks of $\mathcal{A}$ into the rules.

1. If $\varphi \wedge \psi \equiv \bot$ then $\text{IF}(\varphi, \tau, \bot) \,\&\, \text{IF}(\psi, \rho, \bot) \equiv \bot$ else $\text{IF}(\varphi, \tau, \bot) \,\&\, \text{IF}(\psi, \rho, \bot) \equiv \text{IF}(\varphi \wedge \psi, \tau \,\&\, \rho, \bot)$.
2. *Cleaning* of unsatisfiable branches of a nested conditional regex. For example if $\tau = \text{IF}(\varphi, \text{IF}(\psi, \tau_1, \tau_2), \rho)$ and $\varphi \wedge \psi \equiv \bot$ then τ simplifies to $\text{IF}(\varphi, \tau_2, \rho)$ or if $\varphi \wedge \neg\psi \equiv \bot$ then τ simplifies to $\text{IF}(\varphi, \tau_1, \rho)$.
3. It is useful to push complement into $\mathcal{A}$ when possible, e.g., by using the rule $\sim\text{IF}(\varphi, .*, \bot) \equiv \text{IF}(\neg\varphi, .*, \bot)$.

When working with the two algebras $\mathcal{A}$ and *TR*, it is important to keep in mind that their Boolean operations have different semantics.[6] For example, the predicate $\neg\varphi$ as a singleton regex denotes the language $\mathbf{L}(\neg\varphi) = \Sigma \setminus [\![\varphi]\!]$, while the regex $\sim\varphi$ denotes the language $\mathbf{L}(\sim\varphi) = \Sigma^* \setminus [\![\varphi]\!]$.

We show in Theorem 7.3 that for $R \in \mathbb{B}(RE)$ the number of individual regexes that are formed after computing the fixpoint of all regexes through derivation is *linear* in R.

4.1 Lift Rules

The lifting rule $lift(\tau)$ propagates intersection into the leaves and thus lifts conditionals to the top level. Here we also pass the branch condition ψ that is initially ., that can be maintained to be satisfiable, so that dead branches are eliminated on-the-fly and the resulting transition regex is *clean* — in all conditional regexes all branches are satisfiable. Assume here that τ is in NNF. The NNF rules are specified below.

$$\begin{array}{rcl} lift(\tau) & = & lift_{.}(\tau) \\ lift_{\psi}(\tau) & = & \bot \quad \text{if } \psi \equiv \bot \end{array}$$

In the remainder ψ is assumed satisfiable ($\psi \not\equiv \bot$).

$$\begin{array}{rcl} lift_{\psi}(R) & = & R \quad \text{if } R \in ERE \text{ and } \psi \equiv . \\ lift_{\psi}(R) & = & \text{IF}(\psi, R, \bot) \quad \text{if } R \in ERE \text{ and } \psi \not\equiv . \\ lift_{\psi}(\text{IF}(\varphi, t, f)) & = & \text{IF}(\varphi, lift_{\psi\wedge\varphi}(t), lift_{\psi\wedge\neg\varphi}(f)) \\ lift_{\psi}(\text{IF}(\varphi, t, f) \,\&\, \rho) & = & lift_{\psi}(\text{IF}(\varphi, t \,\&\, \rho, f \,\&\, \rho)) \\ lift_{\psi}((\tau_1 \mid \tau_2) \,\&\, \rho) & = & lift_{\psi}(\tau_1 \,\&\, \rho) \mid lift_{\psi}(\tau_2 \,\&\, \rho) \end{array}$$

NNF. The *negation normal form* of a transition regex τ, $NNF(\tau)$, is defined as follows. The correctness of these rules rests on Lemma 4.2.

$$\begin{array}{rcl} NNF(\text{IF}(\varphi, \tau, \rho)) & = & \text{IF}(\varphi, NNF(\tau), NNF(\rho)) \\ NNF(\sim\text{IF}(\varphi, \tau, \rho)) & = & \text{IF}(\varphi, NNF(\sim\tau), NNF(\sim\rho)) \\ NNF(\sim\sim\tau) & = & NNF(\tau) \\ NNF(\sim R) & = & \sim R \quad \text{if } R \in ERE \end{array}$$

The remaining cases are given by De Morgan's laws.

5 Solving Extended Regular Expression Constraints in SMT

Here we show that symbolic derivatives of extended regexes, defined in Section 4, form the basis for a decision procedure that can be integrated in the context of an SMT solver to solve Boolean combinations of *ERE* constraints. A brief overview was given in Section 2.

The regex solver for *ERE* constraints is part of the sequence theory solver in Z3. Regex solving works through *membership propagation rules* that are triggered from the main sequence solver when a membership constraint $in(s, r)$ is encountered. Here s is a term whose type (called its *sort* in Z3) is *sequence over* Σ, or Σ^*, and r is an *ERE* over Σ^*. The regex solver maintains a *graph* G whose nodes are regexes seen so far, and edges from a node are all of its possible (partial) derivatives – G is introduced formally below. In other words, G starts out as an empty graph, and whenever a propagation on $in(s, r)$ occurs, r is added to the graph if it is not present. Propagation then triggers the rewrite rules in Figure 3a. In brief, if $in(s, r)$ has already been determined to be unsatisfiable, as recorded by the graph G, no additional work is done and we rewrite it to false (the BOT rule). Otherwise, $in(s, r)$ is expanded into two cases through the DER rule: s is empty or nonempty. In the latter case the constraint becomes $in\_tr(s, t)$ (read *in transition regex* and analogous to $in(s, r)$), where t is now the *derivative of* r and all the terminals of t are added to G as being reachable from r.

[5] In particular, *TR* is a Boolean algebra over Σ^+ where $\tau : \Sigma \to ERE$ has denotation $[\![\tau]\!] = \bigcup_{a\in\Sigma} aL(\tau(a)) \subseteq \Sigma^+$, where $aL = \{av \mid v \in L\}$.

[6] This is also true in the context of SMT where they are distinct primitive operators. Here we avoid ambiguities by not overloading the operators.

As these rewrites are processed in the regex solver, constraints are accumulated to be handled by the sequence solver. In particular, the DER rule generates formulas about the length of s: $|s| = 0$ and $|s| > 0$, and the ITE rule generates *character constraints* through the predicate φ that is extracted from a conditional regex. Membership constraints exist in a broader context of formulas, including possibly other string constraints on s, so we cannot in general prove that our solver is complete with respect to any set of formulas. However, what we show (Theorem 5.2) is that the decision procedure is sound and complete for a *single* regex constraint if the character theory (satisfiability of predicates φ) is decidable. That is, focusing only on a single constraint $in(s, r)$, then the procedure proves false if and only if r is empty. This result also extends to a Boolean combination of constraints on the same string, e.g. $in(s, r_1) \wedge in(s, r_2)$: this is because, as described in Section 2, we can rewrite it to $in(s, r_1 \,\&\, r_2)$. However, this last rule is done *prior* to applying the decision procedure, and we do not describe it in this section. Instead, we focus on how to propagate a given constraint $in(s, r)$.

We also assume that regexes are *concrete* (i.e. there are no variables of type regex or equations between regexes, only membership constraints for concrete regexes). While this restriction is standard, it can be partially relaxed without additional work: for example, *inequivalence* constraints of the form $r \not\equiv r'$ for regexes r, r' (this includes nonemptiness constraints) can also be reduced to membership using the Boolean operators. In particular $r \not\equiv \bot$ iff $\exists x(x \in r)$, and $r_1 \not\equiv r_2$ iff $(r_1 \,\&\, {\sim}r_2) \mid (r_2 \,\&\, {\sim}r_1) \not\equiv \bot$.

The graph maintained by the regex solver has the form $G = (V, E, F, C)$, with additional derived components *Dead* and *Alive*. The vertices $V \subseteq ERE$ represent the set of all encountered regexes so far, and $E \subseteq V \times V$ is a set of directed edges such that $(v, w) \in E$ implies that $w \in \mathbf{Q}(\delta^{\text{DNF}}(v))$, i.e., w is *derived from* v. In this context $\delta^{\text{DNF}}(v)$ is equivalent to the abstract definition $\delta(v)$ (defined in Section 4) but in a normal form; the required normal form is discussed further below, with an example. Here, $\mathbf{Q}(\delta^{\text{DNF}}(v))$ denotes the set of leaves of the DNF. We write E^* for the *reflexive and transitive closure* of E and we write $E^*(v)$ for $\{w \mid (v, w) \in E^*\}$, i.e., $E^*(v)$ is the set of all vertices in G that are reachable from v.

- $F \subseteq V$ is a set of *final* vertices (*nullable* regexes).
- $C \subseteq V$ is the set of all *closed* v: $\forall w {\in} \mathbf{Q}(\delta^{\text{DNF}}(v)) : (v, w) {\in} E$. In other words, a closed vertex is a vertex all of whose outgoing edges have been added to E.
- $Alive \subseteq V$ is the set of all v s.t. $E^*(v) \cap F \neq \emptyset$.
- $Dead \subseteq V$ is the set of all v s.t. $E^*(v) \subseteq (C \setminus Alive)$. In other words, all vertices in *Dead* are dead-end regexes whose status can never change because all of them are closed (have been fully explored).

For modularity, G does not have knowledge of its vertices being regexes, but they are treated as abstract elements.

$$\frac{in\_tr(s, \textbf{IF}(\varphi, t, f))}{(\varphi(s_0) \wedge in\_tr(s, t)) \vee (\neg\varphi(s_0) \wedge in\_tr(s, f))}\ (\text{ITE})$$

$$\frac{in\_tr(s, r)}{in(s_{1..}, r)}\ (\text{ERE}) \qquad \frac{in\_tr(s, t_1 \mid t_2)}{in\_tr(s, t_1) \vee in\_tr(s, t_2)}\ (\text{OR})$$

$$\frac{in(s, r) \qquad r \notin G.Dead}{\begin{array}{c}(|s| = 0 \wedge \nu(r)) \vee \\ (|s| > 0 \wedge in\_tr(s, \delta^{\text{DNF}}(r)) \wedge \boldsymbol{Upd}[r {\to} \mathbf{Q}(\delta^{\text{DNF}}(r))])\end{array}}\ (\text{DER})$$

$$\frac{in(s, r) \qquad r \in G.Dead}{\bot}\ (\text{BOT})$$

(a) Membership propagation rules for *ERE*s and transition predicates. Here $r \in ERE$, $in(s, r)$ denotes a membership constraint (s matches regex r), and $in\_tr(s, t)$ denotes analogous membership in a transition regex t. Recall that $\nu(r)$ iff r is *nullable*. All rules are equivalence preserving in their respective contexts. In particular $in\_tr(s, t)$ rules are applied only when $|s| > 0$. An implicit assumption is that $r \in G.V$.

$$\frac{\boldsymbol{Upd}[r {\to} Q] \qquad G = (V, E, F, C)}{G {:=} (V {\cup} Q, E {\cup} \{(r, q) \mid q {\in} Q\}, F {\cup} \{q {\in} Q \mid \nu(q)\}, C {\cup} \{r\})}\ (\text{UPD})$$

(b) Graph update rule. An implicit assumption is that $r \in G.V$. Observe that the rule has no effect if $r \in G.C$.

Figure 3. Decision procedure propagation rules.

Therefore, for the abstract description here, we consider the sets F and C to be represented explicitly. The event that all immediate (partial) derivatives from v have been added then causes v to be added to the set C. On the other hand, we consider *Alive* and *Dead* to be inferred from (V, E, F, C) rather than being explicitly represented here.

The primary purpose of G is to enable *dead-end detection*, that is to block search and to infer unsatisfiability of dead-end regexes, as indicated by the BOT rule in Figure 3a. Conveniently, G can be maintained globally and persistently (a single graph for the entire solver and across different logical scopes). In particular, G is independent of the current logical scope because the property of a vertex in G being dead is independent of other side constraints that may exist on the input sequence s.

Initially $G = (V_0, \emptyset, \{r \in V_0 \mid r \text{ is nullable}\}, \emptyset)$ where V_0 is some initial set of regexes that occur in initial membership constraints. When the regex solver is called on a constraint $in(s, r)$, we perform the following steps.

1. As shown in Figure 3a the DER rule either allows the solution $s = \varepsilon$ if r is nullable, or it propagates the goal $in\_tr(s, \delta^{\text{DNF}}(r))$ provided that r is not dead and s is nonempty.
2. The propagation rules for $in\_tr(s, \delta^{\text{DNF}}(r))$ (ITE and OR) rewrite the derivative into a disjunction of cases, where

the leaves are new membership subgoals for $s_{1..}$ as shown by the ERE rule.

3. In this process G is incrementally updated, triggered by $\boldsymbol{Upd}[r{\to}Q]$ where Q is the set $\mathbf{Q}(\delta^{\text{DNF}}(r))$ of all the derivative regexes for r and r is consequently marked closed, as shown by the UPD rule in Figure 3b.

Transition Regex Normal Form. Ensuring that these rules eventually prove unsatisfiability for regexes r denoting the empty language requires care. Notice that Figure 3a does not contain propagation rules for conjunction (intersection) and negation (complement) of transition regexes. This is because such rules would result in incompleteness. For example, consider the hypothetical rule that we reduce $in\_tr(s, r_1\&r_2)$ to $in\_tr(s, r_1) \wedge in\_tr(s, r_2)$. Then, if we apply this to the constraint $in\_tr(s, \texttt{(.*a)\&(.*b)})$, we obtain two separate constraints which propagate separately, and we never arrive at the required contradiction and conclude the original transition regex is unsatisfiable. More specifically, this would occur after propagating rules DER and then ITE starting from $in(s, \texttt{(a.*a)\&(a.*b)})$, since $\delta^{\text{DNF}}(r) = \textbf{IF}(\texttt{a}, \texttt{(.*a)\&(.*b)}, \bot)$.

To avoid such issues with intersection and complement propagation is why we require that $\delta^{\text{DNF}}(r)$ is a normal form of $\delta(r)$: specifically, we require a DNF form where union and if-then-else are always pushed outwards over complement and intersection, and we enforce this when computing derivatives. In particular this requires using the *lift* rules for $r \in ERE$.[7] The implication is that when simplifying $in\_tr(s, r)$, after applying ITE and OR as necessary, we can directly apply rule ERE to the conjunctions, which are plain regexes not involving if-then-else.

Recall the definition of a disjunctive normal form (DNF) of transition regexes from Section 4. Observe that a conditional regex whose terminals are all in *ERE* is already in DNF. The following example illustrates computation of DNF.

Example 5.1. Recall the computation of $\delta(\texttt{.*01.*})$ from Example 4.5. Let $r = \sim(\texttt{.*01.*})$. Let φ_0 be $\lambda x.x = \texttt{0}$ and let φ_1 be $\lambda x.x = \texttt{1}$. In Section 2 we showed that $\delta(r)$ can be computed initially as $\sim\delta(\texttt{.*01.*}) = \sim(\texttt{.*01.*} \mid \textbf{IF}(\texttt{0}, \texttt{1.*}, \bot))$. Hence

$$\begin{aligned}\delta^{\text{DNF}}(\sim(\texttt{.*01.*})) &= DNF(\sim(\texttt{.*01.*} \mid \textbf{IF}(\varphi_0, \texttt{1.*}, \bot)))\\ &= DNF(r \;\&\; \sim\textbf{IF}(\varphi_0, \texttt{1.*}, \bot))\\ &= DNF(r \;\&\; \textbf{IF}(\varphi_0, \sim(\texttt{1.*}), \sim(\bot)))\\ &= DNF(r \;\&\; \textbf{IF}(\varphi_0, \sim(\texttt{1.*}), \texttt{.*}))\\ &= \textbf{IF}(\varphi_0, r \;\&\; \sim(\texttt{1.*}), r)\end{aligned}$$

It is also easy to compute that $\delta^{\text{DNF}}(\sim(\texttt{1.*})) = \textbf{IF}(\varphi_1, \bot, \texttt{.*})$. We continue with the regex $r \;\&\; \sim(\texttt{1.*})$ and get that

$$\begin{aligned}\delta^{\text{DNF}}(r \;\&\; \sim(\texttt{1.*})) &= DNF(\delta(r) \;\&\; \delta(\sim(\texttt{1.*})))\\ &= DNF(\textbf{IF}(\varphi_0, r \;\&\; \sim(\texttt{1.*}), r) \;\&\; \textbf{IF}(\varphi_1, \bot, \texttt{.*}))\\ &= DNF(\textbf{IF}(\varphi_0, r \;\&\; \sim(\texttt{1.*}) \;\&\; \textbf{IF}(\varphi_1, \bot, \texttt{.*}),\\ &\qquad r \;\&\; \textbf{IF}(\varphi_1, \bot, \texttt{.*})))\\ &= \textbf{IF}(\varphi_0, r \;\&\; \sim(\texttt{1.*}), \textbf{IF}(\varphi_1, \bot, r))\end{aligned}$$

where the last equality uses, among other simplifications, the fact that $\varphi_0 \wedge \varphi_1 \equiv \bot$ to keep the resulting conditional regex clean. The resulting transitions are shown in Figure 2(d). □

Finally, we can prove the following summary theorem about the properties of the membership propagation rules. Here $\vdash$ refers to inference with respect to the rules in Figure 3a and Figure 3b. Recall that $r \equiv \bot$ means that $\mathbf{L}(r) = \emptyset$. The rewrite procedure ($\vdash$) is necessarily terminating because the total number of derivatives is finite; see *complexity* below, and the later Theorem 7.1.

Theorem 5.2. *If the character theory is decidable, $r \in ERE$, and s is an uninterpreted constant then $in(s, r) \vdash \bot$ iff $r \equiv \bot$.*

Proof sketch. The proof relies on Symbolic Boolean Finite Automata (SBFA), which we define in Section 7. In particular, we show that G represents an accurate reachability graph of the underlying symbolic automaton, constructed incrementally, where states that end up in $G.Dead$ are equivalent to $\bot$, and where states may be intersection regexes. □

Complexity. Theorem 5.2 states that the decision procedure is sound and complete for regex emptiness, but does not discuss its complexity. In the worst case, complexity relates to the number of regexes in the space of all derivatives (recursively) of a regex. Studying this is a primary motivation for why we develop a theory of automata corresponding to symbolic extended regexes in Section 7. In particular, we give a complexity bound for the common case in practice of regexes in $\mathbb{B}(RE)$ in Theorem 7.3: for this class, that the number of states in an SBFA is linear. As leaves in the DNF $\delta^{\text{DNF}}(r)$ correspond to conjunctions of states in $\mathbb{B}(RE)$, this implies exponential worst-case complexity for the decision procedure here, for $\mathbb{B}(RE)$. For extended regexes, nonemptiness is known to be non-elementary [62], so we can only hope for concrete complexity bounds in practical subclasses.

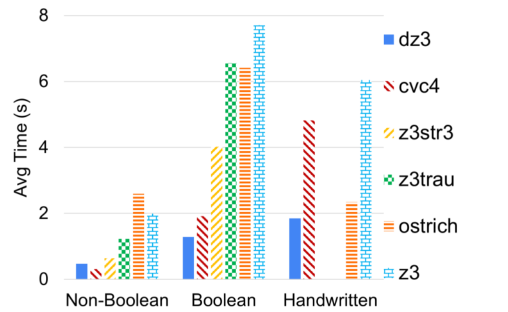
Alive and Dead State Detection. In the implementation the graph G incrementally maintains a DAG of strongly connected components (SCCs) using the Union-Find data structure [63] for implementing SCCs, and it implements explicit marking of SCCs corresponding to the *Dead* and *Alive* subsets of V. The event of adding a new batch of edges to E causes an incremental cycle detection algorithm to be executed, in order to identify new SCCs, followed by recursively marking new *Dead* and *Alive* vertices. For the incremental cycle detection and SCC maintenance, we implemented a simplified variant of known efficient graph algorithms, similar in spirit to what is described in [10].

6 Experiments

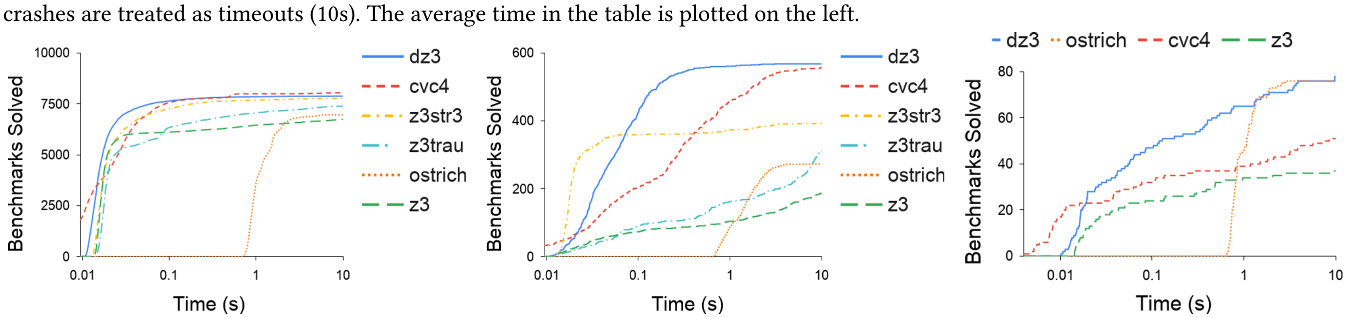
We have implemented symbolic Boolean derivatives as an extension to Z3, together with the strategies for normalizing derivatives and the sound decision procedure described in Section 5. Our solver, dZ3, fully replaces the existing solver

[7] Lift rules are given in Section 4.1. The rules are *not* needed for $r \in \mathbb{B}(RE)$.

Solver	Solved			Avg (s)			Med (s)		
	NB	B	H	NB	B	H	NB	B	H
dz3	95.6%	**88.1%**	**87.6%**	0.47	**1.28**	**1.85**	**0.016**	0.06	**0.08**
cvc4	**97.6%**	86.4%	57.3%	**0.31**	1.92	4.82	0.019	0.30	3.18
z3str3	94.3%	60.9%	–	0.64	4.02	–	0.018	**0.03**	–
z3trau	89.6%	48.7%	–	1.22	6.56	–	0.020	TO	–
ostrich	84.5%	42.3%	85.4%	2.59	6.41	2.34	1.091	TO	0.92
z3	81.8%	29.0%	41.6%	1.99	7.70	6.05	0.018	TO	TO



(a) Summary of the experimental results on non-Boolean (NB), Boolean (B), and additional handcrafted benchmarks (H): percent of benchmarks solved, average time to solve, and median time to solve. Best solver is in bold. For comparison, errors, wrong answers, and crashes are treated as timeouts (10s). The average time in the table is plotted on the left.



(b) Cumulative plots on non-Boolean (left), Boolean (middle), and handcrafted (right) benchmarks. The x-axis is time on a log-scale, and the y-axis shows number of benchmarks solved in that amount of time or less.

Benchmark	Quantity	Benchmark	Quantity	Benchmark	Quantity
Kaluza	5452	Norn	147	Date	20
Slog	1976	SyGuS-qgen	343	Password	34
Norn	813	RegExLib Intersection	55	Boolean + Loops	21
		RegExLib Subset	100	Determinization Blowup	14
Total Non-Boolean	8241	Total Boolean	645	Total Handwritten	89

(c) Benchmarks used for the evaluation. Existing benchmark suites (Kaluza, Slog, Norn, SyGuS, RegExLib) are classified as Boolean if they contain multiple constraints on the same regex.

Figure 4. Results of the experimental evaluation (a-b), and benchmarks used (c).

in Z3 for regular expression constraints which is based on symbolic automata. We carried out a series of experiments to compare our solver with Z3 and other state-of-the-art string solvers. Our interest is in evaluating the following questions:

- Q1 Overall, does dZ3 match the performance of existing regular expression solvers on standard string constraint benchmarks?
- Q2 How does dZ3 specifically fare on standard benchmarks which contain *Boolean combinations* of regular expression constraints on the same regex (which are equivalent to Boolean operations on *ERE*), compared to the state of the art?
- Q3 Finally, how does dZ3 fare on handcrafted difficult examples, designed to showcase the interaction of Boolean operations with other regex operators, compared to the state of the art?

To evaluate Q1, we assembled a collection of standard benchmark suites from the literature: Kaluza, Norn, Slog, and SyGuS-qgen, as collected by SMT-LIB [59, 60]. We add to this an existing set of benchmarks provided in [12, 58], which we call RegExLib: these ask for the answer to an intersection or containment problem between regular expressions taken from regexlib.com, an online library of regular expressions. From all of these sets, we removed benchmarks that do not contain any regular expression constraints, and some Norn benchmarks which contained existential quantification, as this was not allowed by the stated logic. We note that the Kaluza benchmarks represent the easiest cases of these, dominated by constraints that can be simplified to word equations, and serve as a baseline reference in this regard.

To evaluate Q2, the challenge arises of how to fairly compare with solvers which do not support explicit intersection and complement. To address this issue, we observe that although most standard benchmarks do not explicitly contain intersection and complement, a large number of benchmarks contain multiple regex membership constraints on the same string, which is logically equivalent to (and can be treated as) a Boolean combination. Therefore, we parsed the benchmarks from Q1 to divide them into *simple* benchmarks, which do not contain multiple regular expression constraints on the same string variable, and *Boolean* benchmarks, which contain at least one instance of multiple regular expression constraints on the same string. Our hypothesis is that our solver is particularly suited to the Boolean case, as it translates such constraints succinctly to *ERE*s.

To evaluate Q3, we wrote four sets of examples. Unlike in Q2, we incorporate explicit (rather than implicit) intersection and complement. The first set contains problems involving *date* constraints, where a string is constrained to look like a date, as in Figure 1: the questions ask, e.g. whether one such constraint implies another or whether an intersection of such constraints is satisfiable. Such constraints naturally incorporate Boolean combinations: for example, if the month is February, then the day must not be 30 or 31. The second set contains problems involving *password* constraints, e.g. a password must contain at least one number and a letter, and no more than 20 characters, like the example in Section 2. Third, we have a set of regexes where Boolean operations interact with concatenation and iteration, in particular to create nontrivial unsatisfiable regexes. These also serve to test the dead state elimination described in Section 5. Finally, we include classical examples which have small nondeterministic state spaces but blowup when determinized, to test efficiency of derivatives in avoiding determinization: these include variants of `(.*a.{k})&(.*b.{k})` where k is constant. Together with the benchmarks for Q1 and Q2, the number of benchmarks from various sources is summarized in Figure 4(c). The RegExLib benchmarks as well as the additional handwritten examples have been made available at https://github.com/cdstanford/regex-smt-benchmarks.

For all experiments, we compared dZ3 with a representative list of state-of-the-art and actively maintained solvers: Z3 [26, 68], Z3str3[11, 70], Z3-Trau [1, 69], CVC4 [9, 21, 42, 43], and Ostrich [20, 50]. Ostrich represents the most modern tool in the line of solvers including Sloth [32] and Norn [4]. We exclude Z3str3 and Z3-Trau from the Q3 handwritten examples, since explicit intersection and complement were not supported at the time of evaluation. We ran each solver with a 10s timeout, and compared the answer with the correct label (if provided with the benchmark); otherwise, we compared with the answer provided by a baseline solver that appears to be trained (and sound) for the benchmark set in question: for this purpose we used Ostrich for the Norn benchmarks and CVC4 for Kaluza, Slog, and SyGuS-qgen (all others were labeled). If the baseline solver did not return a result, we marked the answer as "unchecked" and conservatively considered it correct. An answer of "unknown" is counted as an error (i.e. unsupported case). In summary, a correct result can be either sat, unsat, or unchecked, while an incorrect result can be either wrong, a timeout, or an error. We further manually inspected solver errors and incorrect answers to ensure fair classification: we checked to ensure that these are unsupported cases, bugs, or crashes, and not a result of a malformed input (we corrected instances of the latter by replacing the input in question). We followed existing SMT community practices [13] in our methodology to summarize and plot the resulting comparisons. The experiments were run on a Dell XPS13 with an Intel Core i7 CPU and 16GB of RAM.

Results. The results are summarized in Figure 4. dZ3 shows state-of-the-art performance and is consistently the best or near the best solver — in terms of average time, median time, or number of benchmarks solved, across our three benchmark sets (Figure 4(a)). dZ3 shows particularly good performance on Boolean and handwritten benchmarks, where only CVC4 (on Boolean) and Ostrich (on handwritten) compare. However, compared to CVC4, dZ3 solves 87% of the handwritten benchmarks rather than 57.3%; and compared to Ostrich, dZ3 solves 88% of the Boolean benchmarks rather than 42.3%. No other solver does consistently well in all three categories. Overall, the plots in Figure 4(b) demonstrate that our implementation of symbolic Boolean derivatives achieves state-of-the-art performance in practice.

7 Symbolic Boolean Finite Automata

In order to formally study the efficiency of our implementation, and in particular, the state space of the set of derivatives, we explore a connection to automata. In particular, we formally define *symbolic Boolean finite automata* or SBFAs, a variant of alternating automata adapted to the symbolic setting. We show that derivatives of symbolic extended regexes correspond to states in a corresponding SBFA, and in the case of $R \in \mathbb{B}(RE)$, we prove a theorem that the state space size is linear in the size of R. This allows us to analyze the worst-case complexity of our decision procedure. SBFAs will also prove useful in comparing with alternative approaches and existing extensions of automata in Section 8.

SBFA. A *Symbolic Boolean Finite Automaton* or SBFA is a tuple $M = (\mathcal{A}, Q, \iota, F, q_\perp, \Delta)$ where $\mathcal{A}$ is the *alphabet theory*; Q is a finite set of *states*; $\iota \in \mathbb{B}(Q)$ is the *initial state combination*; $F \subseteq Q$ is the set of *final states*; $q_\perp \in Q \setminus F$ is the *bottom state*; $\Delta : Q \to TR_Q$ is the *transition function* such that $\Delta(q_\perp) = q_\perp$, where TR_Q is defined in Section 4.

We lift the *final* condition to $\mathbf{q} \in \mathbb{B}(Q)$ denoted $\nu_F(\mathbf{q})$ as follows: $\nu_F(q)$ iff $q \in F$, $\nu_F(\mathbf{p}|\mathbf{q})$ iff $\nu_F(\mathbf{p})$ or $\nu_F(\mathbf{q})$, $\nu_F(\mathbf{p}\&\mathbf{q})$

iff $\nu_F(\mathbf{p})$ and $\nu_F(\mathbf{q})$, and $\nu_F({\sim}\mathbf{q})$ iff not $\nu_F(\mathbf{q})$. The definition of Δ is lifted similarly to $\mathbb{B}(Q) \to TR_Q$.

Semantics. The *language* accepted by M is $\mathbf{L}(M) = \mathbf{M}(\iota)$, where $\mathbf{M} : \mathbb{B}(Q) \to \Sigma^*$ is given by the following equations:

$$\forall \mathbf{q} \in \mathbb{B}(Q) : \ \mathbf{M}(\mathbf{q}) = \{\epsilon \mid \nu_F(\mathbf{q})\} \cup \bigcup_{a\in\Sigma} a \cdot \mathbf{M}(\Delta(\mathbf{q})(a))$$

Construction from Regexes. Construction of an SBFA from a regex $R \in ERE$ starts with the initial state combination $\iota = R$ and computes the rest of the states in Q as the fixpoint of all the states reachable as *terminals* of $\delta(q)$ for $q \in Q$, where what constitutes as a terminal depends on the state granularity and/or normal form of the intended SBFA. In the granularity that is used below, a terminal of $\textsc{if}(\varphi, \tau, \rho)$ is a terminal of τ or ρ, a terminal of ${\sim}\tau$ is a terminal of τ, and a terminal of $\tau \diamond \rho$ is a terminal of τ or ρ. If $\tau \in RE$ then τ is a terminal. In this case, states (other than potentially ι and ${\sim}\bot = .*$) are themselves not conjunctions or negations.

The regex $\bot$, that is the bottom state $q_\bot$, and the dual *top* state regex $.*$ ($= {\sim}\bot$) are called *trivial*. Let $\mathbf{Q}(\tau)$ denote the set of all *nontrivial* terminals of a transition regex τ.

Given a regex R, let $\delta^+(R)$ denote $\mathbf{Q}(\delta(R))$ unioned with all states of derivatives that can be reached from $\mathbf{Q}(\delta(R))$. Formally, $\delta^+(R)$, is the least fixed point of the following equations, where S is a set of regexes,

$$\delta^+(R) = \mathbf{Q}(\delta(R)) \cup \delta^+(\mathbf{Q}(\delta(R))), \quad \delta^+(S) = \bigcup_{R\in S} \delta^+(R).$$

Observe that $\delta^+(R)$ is the set of regexes reached after *one or more* derivations, which may but need not include R itself, e.g., $\delta^+(\mathtt{b(ab)*}) = \{\mathtt{(ab)*}, \mathtt{b(ab)*}\}$ includes the start regex while $\delta^+(\mathtt{ab}) = \{\mathtt{b}, \varepsilon\}$ does not.

Theorem 7.1. *$\delta^+(R)$ is finite.*

Proof. Let $\Gamma = Minterms(\Psi_R)$. For $a \in \Sigma$ let $\hat{a}$ be the minterm in Γ that contains a. It follows that if $\hat{a} = \hat{b}$ then $\delta(R)(a) = \delta(R)(b)$ because then $\textsc{if}(\varphi, \varepsilon, \bot)(a) = \textsc{if}(\varphi, \varepsilon, \bot)(b)$, since for all $\varphi \in \Psi_R$ and $\gamma \in \Gamma$: $[\![\gamma]\!] \subseteq [\![\varphi]\!]$ iff $[\![\gamma]\!] \cap [\![\varphi]\!] \neq \emptyset$. Thus, by Theorem 4.3, we can treat Γ as a finite alphabet with $\delta(R)(a) = D^{Brz}_{\hat{a}}(R)$ where each predicate $\varphi \in \Psi_R$ in $D^{Brz}_{\hat{a}}(R)$ is treated equivalently as a choice $OR\{\hat{a} \mid a \in [\![\varphi]\!]\}$. The statement now follows from [14, Theorem 5.2] that the number of dissimilar derivatives of R is finite, where R_1 and R_2 are called *similar* when they are equal modulo & and | being idempotent, associative, and commutative. □

***SBFA**(R).* The SBFA of $R \in ERE$ is defined as follows, where $Q = \delta^+(R) \cup \{R, \bot, .*\}$ and $F = \{q \in Q \mid q \text{ is nullable}\}$.[8]

$$SBFA(R) = (\mathcal{A}, Q, R, F, \bot, \delta{\restriction}Q)$$

The following is the correctness theorem of $SBFA(R)$.

[8] We write $\delta{\restriction}Q$ to denote δ restricted to the finite set Q — to follow the SBFA definition strictly.

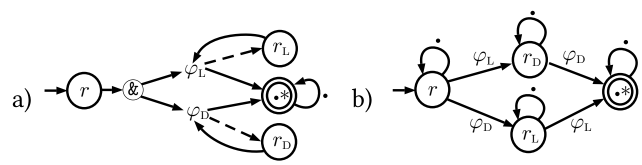


Figure 5. Two Example Symbolic Boolean Finite Automata (SBFA) derived from the same regex r.

Theorem 7.2. *Let $R \in ERE$ and $M = SBFA(R)$. Then for all $\mathbf{q} \in \mathbb{B}(Q_M)$, $\mathbf{M}(\mathbf{q}) = \mathbf{L}(\mathbf{q})$. In particular $\mathbf{L}(M) = \mathbf{L}(R)$.*

Proof. The statement follows by proving that $\forall \mathbf{q} \in \mathbb{B}(Q)$: $v \in \mathbf{M}(\mathbf{q}) \Leftrightarrow v \in \mathbf{L}(\mathbf{q})$ by induction over $|v|$. The base case $v = \epsilon$ follows because $\nu_F(\mathbf{q}) \Leftrightarrow \nu(\mathbf{q})$. The induction case is: $av \in \mathbf{M}(\mathbf{q})$ iff $v \in \mathbf{D}_a(\mathbf{M}(\mathbf{q}))$ iff $v \in \mathbf{M}(\delta(\mathbf{q})(a))$ iff (by the IH) $v \in \mathbf{L}(\delta(\mathbf{q})(a))$ iff (by Theorem 4.3) $v \in \mathbf{L}(D^{Brz}_a(\mathbf{q}))$ iff (by [14, Theorem 3.1]) $av \in \mathbf{L}(\mathbf{q})$. □

Theorem 7.3 is another key result. Here a regex is *normalized* when all concatenations are in right-associative form. A regex is *clean* if it contains no $\bot$ and no unsat predicates. Let $\sharp(R)$ denote the number of predicate nodes in R.

Theorem 7.3. *Let $R \in \mathbb{B}(RE)$. If R is clean and normalized then $|Q_{SBFA(R)}| \le \sharp(R) + 3$.*

Proof sketch. The proof relies on the following lemma: if $X, Z \in RE$ are clean and normalized, then $\delta^+(XZ) = \delta^+(X)Z \cup \delta^+(Z)$; and if $X = S*$ then $\delta^+(X) = \delta^+(S)X$. The proof of the lemma proceeds by induction over X. For the case $X = S{*}Y$, we need to show the equation $\delta^+(S{*}W) = \delta^+(S)S{*}W \cup \delta^+(W)$. The main result then proceeds by induction on R, where by normalization we write $R = R_1{\cdot}Z$ where R_1 is not a concatenation and possibly $Z = \varepsilon$, and do casework on R_1. □

In contrast, for a general $R \in ERE$ we do not have a linear bound on $|Q_{SBFA(R)}|$ because the lifting in $(\tau\&\rho){\cdot}R = lift(\tau\&\rho){\cdot}R$ that first transforms $\tau\&\rho$ into DNF may lead to an exponential blowup.

Example 7.4. Recall $r_D = .*\backslash d.*$ from Section 2 and let $r_L = .*[a\text{-}z].*$. So r_L matches any string containing at least one lower-case letter. Let $\varphi_L = $ `[a-z]` and $\varphi_D = $ `\d`. Let $r = r_L \,\&\, r_D$. Then

$$\begin{aligned} \delta(r_L) &= r_L \mid \textsc{if}(\varphi_L, .*, \bot) \equiv \textsc{if}(\varphi_L, .*, r_L) \\ \delta(r_D) &= r_D \mid \textsc{if}(\varphi_D, .*, \bot) \equiv \textsc{if}(\varphi_D, .*, r_D) \\ \delta(r) &= \delta(r_L) \,\&\, \delta(r_D) = \textsc{if}(\varphi_L, .*, r_L) \,\&\, \textsc{if}(\varphi_D, .*, r_D) \end{aligned}$$

$SBFA(r)$ is shown in Figure 5a. The DNF equivalent is shown in Figure 5b where the default operation is disjunction. □

8 Related Work

Here we provide a formal study of the relationship between symbolic derivatives and related formalisms that can be used in the context of decision procedures for *ERE*. In particular, we first compare with classical derivatives of regular expressions and existing extensions. Next, we compare with existing extensions of classical finite automata and symbolic automata. Then we discuss work related to string solvers and implementation of the proposed techniques in the context of SMT solvers. Finally, we compare to the use of derivatives in matching, and to existing work on extended regular expressions over a finite alphabet.

8.1 Relation to Classical Derivatives

The theory of derivatives of regular expressions has evolved in parallel and largely independently of the mainstream automata research. One of the key features of derivatives is that they provide a lazy and a more algebraic perspective on how finite automata and their regular expression counterparts are related; basic theoretical properties between various classical automata and their derivatives are discussed in [5].

The connection between *ERE* (modulo $\mathcal{A}$) and symbolic derivatives was initially studied in-depth in [36], with the main application of language containment in *ERE*. An important side result [36, Section 5] is that classical derivatives do not directly generalize to predicates, and a workaround is to combine *positive* and *negative* derivatives. We have shown here that a remedy is to use *conditionals*.

In the following we discuss the exact relationship to well-established related classical notions, first Brzozowski derivatives [14] and then Antimirov derivatives [6] and its generalization to *ERE* [17]. We show how they relate to $\delta(R)$ for $R \in RE$. Assume Σ is *finite*, let $a \in \Sigma$, and let $R_a = \delta(R)(a)$.

Brzozowski Derivatives. R_a is precisely the *Brzozowski* derivative [14, Theorem 3.1] $D_a(R)$ of R w.r.t. a.[9] If regexes are viewed as DFA states, D_a is the transition function for a.

Antimirov Derivatives. If $R_a = \bot$ then $\partial_a(R) = \emptyset$ else $R_a = |_{i=1}^{n} R_i$ and $\partial_a(R) = \{R_i\}_{i=1}^{n}$ is the *Antimirov* derivative [6, Definition 2.8] of R w.r.t. a as a set of *partial* derivatives R_i. When viewed as states, each R_i corresponds to a separate target state of a transition (R, a, R_i) of an NFA.

Partial Derivatives of ERE. The Antimirov construction is extended to *ERE* in [17]. The formal construction $\frac{\partial}{\partial_a}(R)$ in [17, Definition 2] inlines negation, inlines concatenation propagation, and inlines conjunction distribution, in the definition of $\frac{\partial}{\partial_a}$ so that the result is essentially an |-set of &-sets. Intuitively $\frac{\partial}{\partial_a}(R) = DNF(R_a)$.

8.2 Relation to Classical Automata

Parallel finite automata by Kozen [38], subsequently renamed to *alternating finite automata* or *AFAs* in [18], and *Boolean finite automata* or *BFAs* by Brzozowski and Leiss [15], were introduced independently (cf [15, p.25]) and use fairly different formalizations and application contexts in doing so. While both work over a finite state space Q and are equivalent classically, their differing notation becomes important symbolically: BFAs use transitions to $\mathbb{B}(Q)$ while AFAs use transitions to 2^{2^Q} encoding $DNF(\mathbb{B}^+(Q))$. We provide a description of SBFAs over finite alphabets as BFAs next.

BFA. Let $M = (\mathcal{A}, Q, \iota, F, q_\bot, \Delta)$ be a SBFA. The equivalent BFA of M is $BFA(M) = (\Sigma, Q, \lambda(q,a).\Delta(q)(a), \iota, F)$.

Proposition 8.1. $\mathbf{L}(M) = L(BFA(M))$ *with* L *as in [15, p.25].*

8.3 Relation to Symbolic Extensions of Automata

Symbolic alternating finite automata (SAFAs) [22] and alternating data automata (ADAs) [35] are two orthogonal symbolic extensions of finite automata, in the former case via SFAs and in the latter case via data automata [34].

Symbolic Alternating Finite Automata. An SAFA [22] (modulo $\mathcal{A}$) is a generalization of an SFA by allowing transition targets to be elements in $\mathbb{B}^+(Q)$ where Q is a finite set of states. There is an initial state combination $\iota \in \mathbb{B}^+(Q)$, a set of final states $F \subseteq Q$, and a finite set of transitions $\Delta \subseteq Q \times \Psi \times \mathbb{B}^+(Q)$. Let $M_{\text{SAFA}} = (\mathcal{A}, Q, \iota, F, \Delta)$

The equivalent SBFA of M_{SAFA} is defined as follows with a bottom state $q_\bot \notin Q$, and where $OR(\emptyset) = q_\bot$.

$$\begin{array}{l} SBFA(M_{\text{SAFA}}) = (\mathcal{A}, Q \cup \{q_\bot\}, \iota, F, q_\bot, \\ \quad \{q_\bot \mapsto q_\bot\} \cup \bigcup_{q \in Q} \{q \mapsto OR\{\textbf{IF}(\psi, \boldsymbol{p}, q_\bot) \mid (q, \psi, \boldsymbol{p}) \in \Delta\}\}) \end{array}$$

Proposition 8.2. $\mathbf{L}(SBFA(M_{SAFA})) = \mathscr{L}(M_{SAFA})$

Going from SBFA $M = (\mathcal{A}, Q, \iota, F, q_\bot, \Delta)$ to SAFA is possible but not easy in general. This is also related to why ~ is not supported in SAFA [22]. W.l.o.g., assume that Δ does not contain complement. This is achieved by adding negated states $\bar{q}$ to Q and for each negated state $\bar{q}$ letting $\Delta(\bar{q}) = NNF({\sim}\Delta(q))$ where $NNF(\tau)$ computes the *negation normal form* of τ meaning that all negations are pushed down to states. In particular, $NNF({\sim}\textbf{IF}(\varphi, \tau, \rho)) = \textbf{IF}(\varphi, NNF({\sim}\tau), NNF({\sim}\rho))$, and $NNF({\sim}q) = \bar{q}$. The other cases are standard.

The equivalent SAFA of M is defined as follows where $\tau(\alpha) = \tau(a)$ for some $a \in [\![\alpha]\!]$ — which is well-defined (independent of choice) due to the local mintermization.

$$\begin{array}{l} SAFA(M) = (\mathcal{A}, Q, NNF(\iota), F, \\ \quad \{(q, \alpha, \Delta(q)(\alpha)) \mid q \in Q, \alpha \in Minterms(Guards(\Delta(q)))\}) \end{array}$$

Proposition 8.3. $\mathbf{L}(M) = \mathscr{L}(SAFA(M))$

The problem with this construction is that $|Minterms(\Gamma)|$ can be exponential in $|\Gamma|$ so the construction of $SAFA(M)$ is exponential in the worst case. The same problem arises in SAFA *normalization* [22] used for complementation.

Alternating Data Automata. The expressiveness of this class of automata goes far beyond regular languages, because registers are allowed to carry information across state

[9] D_a applies to the whole *ERE* class.

boundaries, so that consecutive data elements in traces are functionally related. Data automata, as defined in [34], use registers and have the expressive power of general Turing machines. In an *alternating* data automaton [35], arbitrary Boolean combinations of predicates can be used to relate before and after values of registers. It is stated in [34] that complement of alternating data automata is linear unlike in [22]. We are not aware of work relating *ERE* with ADAs.

Conditional Branching. Conditional transitions (without Boolean combinations of states) have been used before in a special class of deterministic symbolic transducers called *Branching Symbolic Transducers* or *BSTs* [55]. The main motivation behind BSTs is in the context of data processing pipelines where they preserve condition evaluation order and in this way support more direct and efficient serial code generation. A BST has a finite state space Q, and when the BST acts as a finite state automaton, its rules correspond to a subset of TR_Q without Boolean operations. Conditional transitions are also used in the implementation of MONA [37] where transitions are multi-terminal BDDs whose terminals are states. We apply similar principles in dZ3 to represent transition regexes in a canonical way.

8.4 Related Work in SMT

String and regex constraints have been the focus of both SMT and CP solving communities, with several tools being developed over the past decade. A theory of strings with regexes is a standard part of the SMT-LIBv2 format [64]. String solvers are integrated in the CDCL(T) architecture [42, 49]. From the CP community, the MiniZinc format integrates membership constraints over regular languages presented as either DFAs or NFAs [47]. The solver presented in [43] is closely related to ours in that it relies on Antimirov derivatives to reduce positive regular expression membership constraints. It diverges from our approach as it handles intersection similar to [17], instead of using symbolic derivatives. Consistent with what the empirical evaluation suggests, complementation is not treated in depth and is essentially out of scope of this work. Ostrich [20] is advertised as a symbolic solver for string formulas that come from path constraints, and its solver is based on solving for pre-images (see also the earlier solvers Sloth [32] and Norn [4]). Our evaluation suggests that Ostrich performs extremely well, restricted to certain benchmark sets. While full handling of regexes seems out of scope of Z3-Trau, *flat* automata were recently applied [1] for solving symbolic constraints that include string-to-int and int-to-string conversions. Z3str3 [11] and its predecessor Z3str2 [71] integrate several innovations around string equality solving. Many of the advances previously developed in S3 [65] are now integrated within Z3's default string solver, and hence dZ3 benefits from these results. ZELKOVA is a tool used internally by Amazon to check AWS policy configurations, it uses a custom NFA engine based extension of Z3 to handle regex constraints [8].

8.5 Related Use of Derivatives in Matching

Regular expression matchers over symbolic alphabets, such as SRM [67], use Brzozowski derivatives. There is a stark contrast between the problems of matching and analysis (SMT solving). In matching, the next concrete character is always known, whereas in solving, the next character in the string may be unknown. This is why transition regexes (conditionals) are necessary in our case, whereas they are not necessary in matching. It is crucial to not assume that the next character is known in SMT, because the regex engine is part of a broader solver loop where different constraints exist on various characters. Moreover, SRM builds *minterms* upfront for a given regex R — which is one of the main problems we want to avoid using conditionals.

8.6 Related Work on Extended Regular Expressions

Regular expressions over a finite alphabet extended with intersection and complement (*ERE*) have previously been studied from a complexity standpoint and for practical use cases. From a complexity viewpoint, emptiness is shown to be non-elementary [62] for general *ERE* and PSPACE-hard [33, 39] for the subclass with intersection only. Succinctness of *ERE* over classical regular expressions is also non-elementary [62] (see Theorem 25 in [27]), and is studied in more detail in [28]. More efficient solutions can be given focused on the *ERE* membership problem [41], or in a separate vein, focused on algebraic rewrites rather than complexity [7]. Finally, *ERE* membership specifications have been fruitfully applied to the problems of testing and monitoring [54, 56].

9 Conclusion

In this paper, we generalized the finite-alphabet based work of derivatives to work over a symbolic alphabet and to incorporate Boolean combinations, and showed how to use such symbolic Boolean derivatives to solve regular expression membership constraints in SMT. Our solver, dZ3, achieves state-of-the-art performance on standard benchmark sets, and significant speedup on constraints involving intersection and complement, where no existing solver does consistently well across benchmark sets. While we have experimentally validated the main ideas, many further promising directions remain to be explored: for example, generalizing concrete regex constraints to constraints over regex and string variables, and designing heuristics that capture common usage patterns and that can be exploited by CDCL-based solvers.

Acknowledgments

We would like to thank our shepherd, Viktor Kunčak, and all of the anonymous reviewers for their valuable help in improving the paper.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janku, Hsin-Hung Lin, Lukás Holík, and Wei-Cheng Wu. 2020. Efficient handling of string-number conversion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 943–957. https://doi.org/10.1145/3385412.3386034

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. 2018. Trau: SMT solver for string constraints. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 1–5.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String constraints for verification. In *International Conference on Computer Aided Verification*. Springer, 150–166.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT solver for string constraints. In *International Conference on Computer Aided Verification*. Springer, 462–469.

[5] Cyril Allauzen and Mehryar Mohri. 2006. A unified construction of the Glushkov, Follow, and Antimirov automata. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 110–121.

[6] Valentin Antimirov. 1995. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science* 155 (1995), 291–319.

[7] Valentin M Antimirov and Peter D Mosses. 1995. Rewriting extended regular expressions. *Theoretical Computer Science* 143, 1 (1995), 51–72.

[8] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–9. https://doi.org/10.23919/FMCAD.2018.8602994

[9] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[10] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert Endre Tarjan. 2011. A New Approach to Incremental Cycle Detection and Related Problems. *CoRR* abs/1112.0784 (2011). http://arxiv.org/abs/1112.0784

[11] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 55–59.

[12] Nikolaj Bjørner, Vijay Ganesh, Raphael Michel, and Margus Veanes. 2012. An SMT-LIB Format for Sequences and Regular Expressions. In *SMT'12*, P. Fontaine and A. Goel (Eds.). 76–86.

[13] Martin Brain, James H Davenport, and Alberto Griggio. 2017. Benchmarking Solvers, SAT-style.. In *SC2 ISSAC*.

[14] Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *JACM* 11 (1964), 481–494.

[15] J. A. Brzozowski and E. Leiss. 1980. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science* 10 (1980), 19–35.

[16] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. 2017. *String Analysis for Software Verification and Security*. Springer.

[17] Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. 2011. Partial Derivatives of an Extended Regular Expression. In *Language and Automata Theory and Applications, LATA 2011 (LNCS, Vol. 6638)*. Springer, 179–191.

[18] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. 1981. Alternation. *JACM* 28, 1 (1981), 114–133.

[19] Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 325–342.

[20] Taolue Chen, Matthew Hague, Anthony W Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[21] CVC4. 2020. https://github.com/CVC4/CVC4.

[22] Loris D'Antoni, Zachary Kincaid, and Fang Wang. 2018. A Symbolic Decision Procedure for Symbolic Alternating Finite Automata. *Electronic Notes in Theoretical Computer Science* 336 (2018), 79–99.

[23] Loris D'Antoni and Margus Veanes. 2014. Minimization of Symbolic Automata. *ACM SIGPLAN Notices – POPL'14* 49, 1 (2014), 541–553. https://doi.org/10.1145/2535838.2535849

[24] Loris D'Antoni and Margus Veanes. 2020. Automata Modulo Theories. *Commun. ACM* (2020).

[25] James C Davis. 2019. Rethinking Regex engines to address ReDoS. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1256–1258.

[26] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08 (LNCS)*. Springer, 337–340.

[27] Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming-Wei Wang. 2005. Regular expressions: New results and open problems. *J. Autom. Lang. Comb.* 10, 4 (2005), 407–437.

[28] Wouter Gelade and Frank Neven. 2008. Succinctness of the complement and intersection of regular expressions. *arXiv preprint arXiv:0802.2869* (2008).

[29] Dan Gusfield. 1997. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News* 28, 4 (1997), 41–60.

[30] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. 1995. Mona: Monadic Second-order logic in practice. In *TACAS '95 (LNCS, Vol. 1019)*. Springer.

[31] Hossein Hojjat, Philipp Rümmer, and Ali Shamakhi. 2019. On Strings in Software Model Checking. In *APLAS (LNCS, Vol. 11893)*, A. Lin (Ed.). Springer.

[32] Lukáš Holík, Petr Janků, Anthony W Lin, Philipp Rümmer, and Tomáš Vojnar. 2017. String constraints with concatenation and transducers solved efficiently. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–32.

[33] H. B. Hunt III. 1973. *The equivalence problem for regular expressions with intersections is not polynomial in tape.* TR 73-161. Department of Computer Science, Cornell University, Ithaca, New York.

[34] R. Iosif, A. Rogalewicz, and T. Vojnar. 2016. Abstraction refinement and antichains for trace inclusion of infinite state systems. In *TACAS'16 (LNCS, Vol. 9636)*. Springer, 71–89.

[35] Radu Iosif and Xiao Xu. 2018. Abstraction Refinement for Emptiness Checking of Alternating Data Automata. In *TACAS'18*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 93–111.

[36] Matthias Keil and Peter Thiemann. 2014. Symbolic Solving of Extended Regular Expression Inequalities. In *FSTTCS'14 (LIPIcs)*. 175–186.

[37] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. 2002. MONA Implementation Secrets. *International Journal of Foundations of Computer Science* 13, 4 (2002), 571–586.

[38] Dexter Kozen. 1976. On parallelism in Turing machines. In *17th Annual Symposium on Foundations of Computer Science, FOCS'76*. IEEE Xplore, 89–97.

[39] Dexter Kozen. 1977. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*. 254–266. https://doi.org/10.1109/SFCS.1977.16

[40] Dexter Kozen. 1997. Kleene algebra with tests. *Transactions on Programming Languages and Systems* 19 (1997), 427–443.

[41] Orna Kupferman and Sharon Zuhovitzky. 2002. An improved algorithm for the membership problem for extended regular expressions. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 446–458.

[42] 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 *International Conference on Computer Aided Verification*. Springer, 646–662.

[43] 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 *FroCoS 2015: Frontiers of Combining Systems (LNCS, Vol. 9322)*. Springer, 135–150.

[44] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound regular expression semantics for dynamic symbolic execution of JavaScript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 425–438.

[45] Microsoft. 2020. *Azure Resource Manager documentation*. https://docs.microsoft.com/en-us/azure/azure-resource-manager/.

[46] Microsoft. 2020. *.NET regular expressions*. https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions.

[47] MiniZinc. 2020. https://www.minizinc.org.

[48] Mehryar Mohri. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering* 2, 1 (1996), 61–80.

[49] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM* 53, 6 (2006), 937–977. https://doi.org/10.1145/1217856.1217859

[50] Ostrich. 2020. https://github.com/uuverifiers/ostrich/.

[51] Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *Journal of Functional Programming* 19, 2 (2009), 173–190.

[52] passwords generator.org. 2020. https://passwords-generator.org/.

[53] Damien Pous. 2015. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. *ACM SIGPLAN Notices – POPL'15* 50, 1 (2015), 357âĂŞ368. https://doi.org/10.1145/2775051.2677007

[54] Grigore Roşu and Mahesh Viswanathan. 2003. Testing extended regular language membership incrementally by rewriting. In *International Conference on Rewriting Techniques and Applications*. Springer, 499–514.

[55] Olli Saarikivi, Margus Veanes, Todd Mytkowicz, and Madan Musuvathi. 2017. Fusing Effectful Comprehensions. In *ACM SIGPLAN Notices – PLDI'17*. ACM.

[56] Koushik Sen and Grigore Roşu. 2003. Generating optimal monitors for extended regular expressions. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003), 226–245.

[57] Reetinder Sidhu and Viktor K Prasanna. 2001. Fast regular expression matching using FPGAs. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, 227–238.

[58] SMT. 2012. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/nbjorner-microsoft.automata.smtbenchmarks.zip.

[59] SMTLib. 2020. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_S.

[60] SMTLib. 2020. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA.

[61] stackoverflow.com. 2020. Regex for password must contain at least eight characters, at least one number and both lower and uppercase letters and special characters. https://stackoverflow.com/questions/19605150/regex-for-password-must-contain-at-least-eight-characters-at-least-one-number-a.

[62] L. J. Stockmeyer and A. R. Meyer. 1973. Word Problems Requiring Exponential Time(Preliminary Report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, STOC'73*. ACM, 1âĂŞ9. https://doi.org/10.1145/800125.804029

[63] Robert E. Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *JACM* 22 (1975), 215–225.

[64] Cesare Tinelli, Clark Barrett, and Pascal Fontaine. 2020. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml.

[65] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1232–1243. https://doi.org/10.1145/2660267.2660372

[66] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. 2010. Symbolic Automata Constraint Solving. In *Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2010 (LNCS, Vol. 6397)*, C.G. Fermüller and A. Voronkov (Eds.). Springer, 640–654.

[67] Margus Veanes, Olli Saarikivi, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *TACAS'19 (LNCS)*. Springer.

[68] Z3. 2020. https://github.com/z3prover/z3.

[69] Z3-Trau. 2020. https://github.com/diepbp/z3-trau.

[70] Z3str3. 2020. https://sites.google.com/site/z3strsolver/.

[71] 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), 249–288.