Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints


CALEB STANFORD†, Microsoft, USA and University of Pennsylvania, USA
MARGUS VEANES, Microsoft, USA
NIKOLAJ BJØRNER, Microsoft, USA

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.

1 INTRODUCTION

Regular expressions and finite automata play a fundamental role in many areas, ranging from applications in natural sciences [21] and NLP [33] to core problems in applied computer science, such as matching [19, 36, 39], model-checking [22], and solving of string constraints in SMT [23]. 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 [4, 11]. 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, 5, 7, 14, 15, 20, 35, 49–51], 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 [31] 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.

We believe that Boolean combinations of constraints represent the norm, rather than the exception, in practice. To give one example: cloud policy languages, such as Amazon AWS policies [4] and Microsoft Azure resource manager policies [30] 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. 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 the year was accidentally specified to be both at the beginning and at the end 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 not only might be less succinct, but interestingly, industrial policy languages actually restrict regex syntax in various ways, forcing users to write Boolean combinations. For example, both the Amazon AWS and Microsoft Azure languages, as of 2020, allow Kleene star in .* only (here .* is the regex matching any string). One rationale behind such language restrictions is to simplify the regex matcher engine implementation in order to avoid performance bottlenecks that could otherwise be exploited for regex denial-of-service attacks (ReDoS). This makes the use of top level conjunction and complement, as used in this date example, a way to safely express more complex regular constraints, while at the same time raises the need to deal with Boolean combinations of regex constraints for analysis.

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

(1) Convert a regular expression $r$ into an automaton $M_r$ and then propagate the logical connectives into corresponding Boolean operations over automata. Thus $(s \in r_1) \land (s \in r_2)$ can be converted into $s \in L(M_{r_1} \times M_{r_2})$ and $\neg(s \in r)$ can be converted into $s \in L(M_r^C)$ [48].
Symbolic Boolean Derivatives

Fig. 1. Example Boolean combination of regex constraints arising in practice: users of the Azure resource policy language \[30\] 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).

(2) Propagate the operations over regexes, by considering extended regexes, such as \((.*\d.*)&(.*[a-z].*)\), where & is intersection. Then, directly algebraically manipulate such extended regexes using derivatives \[29\].

While it is possible to extend classical automata algorithms to work modulo a character theory \[18\], 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 regular expression without converting it to automata, pioneered by Brzozowski \[9\] and Antimorov \[3\]. The summary of the approach is that the derivatives of a regular expression 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 \[26\], 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 regular expression \(r\) does in general not preserve its language 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 \[26, Lemma 3\]. On the other hand, a classical generalization of Antimorov derivatives to extended regular expressions is possible (over a finite alphabet \(\Sigma\)) although challenging \[12\]; however, leveraging this work for the symbolic SMT setting would require explicitly
enumerating (finitizing) the entire alphabet upfront (also known as \textit{mintermization} in the literature [17, 18]), as we explain further in Section 2. Doing so may be infeasible or prohibitively expensive (e.g. for Unicode), requires considering all regex constraints in an SMT formula globally, and for general predicates may cause another exponential blowup [18]. Considering only intersection, and not complement, avoids some of this complexity and represents a state-of-the-art approach [29], but this loses the full generality of the Boolean operations.

In 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 \textit{key new insight} that enables us to define derivatives of regexes directly, while allowing Boolean operations, is that we augment regexes with \textit{conditionals} (if-then-else), and define the derivative of a regex to be a regex with conditionals, called a \textit{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 7.4). We give a decision procedure based on our derivatives which integrates into a broader SMT context: a set of inference rules that incrementally unfolds regular expression 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 regular expression 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. Using a large benchmark suite and 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.

\textbf{Contributions}

- We introduce a new theory of symbolic derivatives of extended regexes, which avoids blowup in existing techniques. It works via translation to \textit{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. Using existing benchmark sets, 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 (SBFAs) that generalizes the various 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 7.4). (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 $s \in R$, where $s$ is a string term over an alphabet type $\Sigma$, i.e., $s$ has type $\Sigma^*$, and $R$ is a concrete regex over $\Sigma^*$. Our goal is to solve the satisfiability problem for that membership constraint: does there exist a concrete instance of $s$ in $\Sigma^*$ 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:

$$\left(\mid s \mid = 0 \land IsNullable(R)\right) \lor \left(\mid s \mid > 0 \land s_1 \ldots \in \delta(R)(s_0)\right),$$

where $IsNullable(R)$ 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 first character ($s_0$), and returns a regex for the language of suffixes $w$ such that $s_0w \in R$ 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. We could naively enumerate all possible characters

$$\bigvee_{a \in \Sigma}(s_1 \ldots \in D_a(R) \land s_0 = a),$$

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 $\Sigma$ to regexes.

---

1We write $s_i$ for its $i$'th element and $s_L$ for its suffix from $i$. Note that these can be purely symbolic expressions, $s$ itself may be uninterpreted.
More concretely, take \( R \) to be a typical password constraint:

\[
(s \in .*\d.*) \land \neg(s \in .*01.*)
\]

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. [37]), which can be encoded as large intersections (cf. [43]). 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 [17]. 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 \( s \in R \) 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:

\[
s \in (.*\d.*) \land \neg(.*01.*)
\]

Let \( R_1 = .\ast\d\ast \), \( R_2 = \neg(.*01.*) \) and \( R = R_1 \land R_2 \). Since \( R \) is not nullable (does not accept the empty string), the case split we started from reduces to the assertion

\[
|s| > 0 \land s_L \in \delta(R)(s_0)
\]

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 generate the term \( \text{IF}(x = 0, 1, \bot) \), read as if \( x = 0 \) then \( 1 \) else \( \bot \). This idea allows for the derivative of \( R \) to be computed using algebraic rules as follows. The \( \equiv \) below also shows simplification steps using distributivity, DeMorgan’s laws, and other properties.

Below, \( \phi_d \) is the predicate for \( \backslash d \) (characters that are digits).

\[
\begin{align*}
\delta(R) &= \delta(R_1) \land \delta(R_2) \\
\delta(R_1) &= R_1 | \text{IF}(\phi_d(x), .\ast, \bot) \equiv \text{IF}(\phi_d(x), .\ast, R_1) \\
\delta(R_2) &= \neg(\delta(.*01.*)) = \neg(.*01.* | \delta(01.*)) \\
&= \neg(.*01.* | \text{IF}(x = 0, 1, \ast, \bot)) \\
&= R_2 \land \text{IF}(x = 0, \neg(1\ast), .\ast) \\
&= \text{IF}(x = 0, R_2 \land \neg(1\ast), R_2) \\
\delta(R) &\equiv \text{IF}(\phi_d(x), .\ast, R_1) \land \text{IF}(x = 0, R_2 \land \neg(1\ast), R_2) \\
&\equiv \text{IF}(x = 0, R_2 \land \neg(1\ast), \text{IF}(\phi_d(x), .\ast, R_1) \land R_2) \\
&\equiv \text{IF}(x = 0, R_2 \land \neg(1\ast), \text{IF}(\phi_d(x), R_2, R))
\end{align*}
\]
Observe that all conditional predicates are extracted from the regex itself: e.g. \( \emptyset \) in a conditional arises from \( \emptyset \) in the original regex. Step (i) uses (among other properties) that \( \neg \varphi_d(x) \land x = \emptyset \) is unsat. Note that \( \neg \bot \equiv \ast \) and \( .|\ldots\ast \).

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 regular expression \( \emptyset1.\ast \). But what if we want to now take the derivative of the complement of \( \emptyset1.\ast \)? Then we need to know not just this derivative where the first character is \( \emptyset \) but also the derivative if the first character is not \( \emptyset \), 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 \( \text{IF}(x = \emptyset, 1.\ast, \bot) \), which has the case where the first character is not \( \emptyset \) present. Then when complementing this, we get \( \text{IF}(x = \emptyset, \neg(1.\ast), \ast) \).

Thus, viewing the derivative as a conditional regex (transition regex) is what enables us to treat complement algebraically.

Having calculated the derivative \( \delta(R) \), we then continue as follows. Let \( R_3 = R_2 \land \neg(1.\ast) \). So \( s_L \in \delta(R)(s_0) \) reduces to

\[
s_L \in \text{IF}(s_0 = \emptyset, R_3, \text{IF}(\varphi_d(s_0), R_2, R))
\]

Going forward, this creates the further case split:

\[
(s_0 = \emptyset \land s_L \in R_3) \lor (s_0 \neq \emptyset \land s_L \in \text{IF}(\varphi_d(s_0), R_2, R))
\]

where \( s_L \in R_3 \) splits further into two subcases:

\[
(|s_L| = 0 \land \text{IsNullable}(R_3)) \lor (|s_L| > 0 \land s_L \in \delta(R)(s_1))
\]

where \( (s_L)_L = s_2 \) and \( (s_L)_0 = s_1 \), and the procedure repeats. Here \( R_3 \) is nullable so \( dZ_3 \) can generate a model for \( |s| > 0 \land |s_L| = 0 \land s_0 = \emptyset \) — provided that these constraints are consistent with other constraints on \( s \) in the context. For example if there was a constraint \( s_0 > \emptyset \), this case would be blocked and the search would backtrack to the other case.\(^2\)

3 PRELIMINARIES

3.1 Sequences

When working with sequences over a domain \( \Sigma \) 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 \( \Sigma^* \) is implemented in an SMT solver the type \( \Sigma^* \) is sequence over \( \Sigma \) that is disjoint from \( \Sigma \). 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 \( \epsilon \) is the empty sequence. We write \( xy \) for \( x \cdot y \) when it is clear.

\(^2\)The condition \( s_0 > \emptyset \) is possible because the underlying character theory (by default bitvectors in \( dZ_3 \)) is equipped with a total order.
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 $X \subseteq \Sigma^*$.

### 3.2 Boolean Algebras

Let $D$ be a nonempty universe. A Boolean algebra over $D$ is a tuple $\mathcal{A} = (D, \Psi, \bot, \top, \lor, \land, \neg)$ where $\Psi$ is a set of predicates closed under the Boolean connectives; $\bot, \top \in \Psi; \bot \land \top = \emptyset, \bot \lor \top = D$, and for all $\varphi, \psi \in \Psi, [\varphi \lor \psi] = [\varphi] \cup [\psi], [\varphi \land \psi] = [\varphi] \cap [\psi]$, and $[\neg \varphi] = D \setminus [\varphi]$. For $\varphi, \psi \in \Psi$ we write $\varphi \equiv \psi$ ($\varphi$ is equivalent to $\psi$) to mean that $[\varphi] = [\psi]$. In particular, if $\varphi \equiv \bot$ then $\varphi$ is unsatisfiable and if $\varphi \equiv \top$ then $\varphi$ is valid. $\mathcal{A}$ being effective means that all components of $\mathcal{A}$ are recursively enumerable, and satisfiability of $\varphi \in \Psi (\varphi \not\equiv \bot)$ is decidable.

### 3.3 Boolean Combinations

If $Q$ is a set of basic elements or atoms then $\mathbb{B}(Q)$ denotes the Boolean closure over $Q$ using $|$ for disjunction, $\&$ for conjunction, and $\neg$ for complement. $\mathbb{B}^+(Q)$ denotes the positive Boolean closure of $Q$ (without use of $\neg$). Both $\&$ and $|$ are treated as idempotent, associative and commutative operators and lifted to finite nonempty subsets $S \subseteq Q$ through $\text{AND}(S)$ and $\text{OR}(S)$, respectively.

### 3.4 Symbolic Regexes

Let $\mathcal{A} = (\Sigma, \Psi, \bot, \top, \lor, \land, \neg)$ be a fixed effective Boolean algebra called an alphabet theory. Note that $\Sigma$ may be infinite. We first recall the definitions of the two standard subclasses of regexes and extended regexes, where $\phi \in \Psi$. We always work modulo $\mathcal{A}$ and we do not mention this explicitly every time.

$$RE ::= \phi \mid \epsilon \mid \bot \mid RE_1 \cdot RE_2 \mid RE^* \mid RE_1 | RE_2$$

$$ERE ::= \phi \mid \epsilon \mid \bot \mid ERE_1 \cdot ERE_2 \mid ERE^* \mid \mathbb{B}(ERE)$$

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 $R$ is $L(R) \subseteq \Sigma^*$:

$L(\phi) = [\varphi], L(\epsilon) = \{\epsilon\}, L(\bot) = \emptyset,$

$L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2), \ L(R^*) = L(R)^*,$

$L(R_1 | R_2) = L(R_1) \cup L(R_2), \ L(R_1 \& R_2) = L(R_1) \cap L(R_2),$

$L(\neg R) = \Sigma^* \setminus L(R)$

A regex $R$ is nullable ($v(R)$) iff $\epsilon \in L(R)$: $v(\phi) = v(\bot) = false; v(\epsilon) = v(\epsilon^*) = true; v(R_1 \cdot R_2) \iff v(R_1)$ and $v(R_2)$; $v(R_1 \& R_2) \iff v(R_1)$ and $v(R_2)$; $v(R_1 | R_2) \iff v(R_1)$ or $v(R_2)$; $v(\neg R) \iff not \ v(R)$. 

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner
4 SYMBOLIC DERIVATIVES

Here we formally introduce the concept of transition regexes TR, define symbolic derivatives for \( R \in \text{ERE} \) 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.

4.1 Transition Regexes

In order to define symbolic derivatives we first introduce the key concept of transition regexes TR in which regexes are augmented with conditionals. The definition of TR depends on a parameter \( Q \) — referred to below as TR\(_Q\) — here \( Q = \text{ERE} \). Let \( \varnothing \in \{ |, \& \} \) and \( \bar{\varnothing} = | \) and \( \bar{|} = \& \).

\[
TR ::= Q \mid \textbf{IF}(\varnothing, \tau_1, \tau_2) \mid \exists(\text{TR})
\]

We call \( \textbf{IF}(\varnothing, \tau_1, \tau_2) \) a conditional regex. A transition regex \( \tau \) denotes the function \( \tau : \Sigma \to \mathbb{B}(Q) \) defined as follows.

\[
\begin{align*}
R(x) &= R \quad \text{(for \( R \in Q \))} \\
\textbf{IF}(\varnothing, \tau, \rho)(x) &= \begin{cases} \\
\tau(x), & \text{if } x \in \llbracket \varnothing \rrbracket; \\
\rho(x), & \text{otherwise.}
\end{cases} \\
\tau \circ \rho(x) &= \tau(x) \circ \rho(x) \\
\bar{\tau}(x) &= \bar{\tau}(\tau(x))
\end{align*}
\]

Transition regexes \( \tau \) and \( \rho \) are equivalent, denoted \( \tau \equiv \rho \), when \( \forall x \in \Sigma, \tau(x) \equiv \rho(x) \). The concatenation operation of regexes is lifted to transition regexes \( \tau \) in \( \tau \cdot R \) for \( R \in \text{ERE} \).

\[
\begin{align*}
\textbf{IF}(\varnothing, \tau, \rho) \cdot R &= \textbf{IF}(\varnothing, \tau \cdot R, \rho \cdot R) \\
(\tau \mid \rho) \cdot R &= (\tau \cdot R) \mid (\rho \cdot R) \\
\bar{\tau} \cdot R &= \bar{\tau} \cdot R \\
(\tau \& \rho) \cdot R &= \text{lift}(\tau \& \rho) \cdot R
\end{align*}
\]

Negation \( \bar{\tau} \) of \( \tau \) is defined as follows.

\[
\bar{R} = \neg R, \quad \bar{\tau} = \tau, \quad \bar{\tau} \circ \rho = \bar{\tau} \circ \bar{\rho}, \quad \textbf{IF}(\varnothing, \tau, \rho) = \textbf{IF}(\varnothing, \bar{\tau}, \bar{\rho})
\]

The definition of lift(\( \tau \)) is such that if \( \tau \in Q \) then lift(\( \tau \)) = \( \tau \) else \( \tau \) is transformed into an equivalent conditional regex by lifting the character predicates to the top while pushing conjunction into the leaves. Lift rules are discussed in Section 4.3 below.

\footnote{Function application of \( (x) \) binds weakest, so \( \tau \circ \rho(x) \) stands for \( (\tau \circ \rho)(x) \).}
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 \( \tau \) using various algebraic laws of TR.

**Lemma 4.1.** \( L(\tau \cdot R(x)) = L(\tau(x)) \cdot L(R) \)

**Lemma 4.2.** \( \sim \tau \equiv \overline{\tau} \)

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

\[
\begin{align*}
\delta(\epsilon) &= \delta(\perp) = \perp \\
\delta(\varphi) &= \text{IF}(\varphi, \epsilon, \perp) \\
\delta(R \cdot R') &= \begin{cases} \\
\delta(R) \cdot R' | \delta(R'); & \text{if } R \text{ is nullable,} \\
\delta(R) \cdot R'; & \text{otherwise.} \\
\end{cases} \\
\delta(R^*) &= \delta(R) \cdot R^* \\
\delta(R \circ R') &= \delta(R) \circ \delta(R') \quad \text{(for } \circ \in \{\cup, |\}} \\
\delta(\sim R) &= \sim \delta(R)
\end{align*}
\]

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* \( L \) *wrt* \( a \), \( D_a(L) = \{ v : av \in L \} \), and for \( R \in ERE \) we use Brzozowski derivatives \( D_a(R) \in ERE \) (modulo \( A \) [26]), and the classical result \( L(D_a(L)) = D_a(L(R)) \) [9, Theorem 3.1]. Let \( D_a(R) = L(D_a(R)) \).

**Theorem 4.3.** \( \delta(R)(a) \equiv D_a(R) \).

**Proof.** By induction over \( R \). The base cases \( \perp \) and \( \epsilon \) are trivial.

**Base case \( \varphi \):** \( \delta(\varphi) = \text{IF}(\varphi, \epsilon, \perp) \). If \( a \in \varphi \), then \( \text{IF}(\varphi, \epsilon, \perp)(a) = \epsilon(a) = \epsilon = D_a(\varphi) \). If \( a \notin \varphi \), then \( \text{IF}(\varphi, \epsilon, \perp)(a) = \perp(a) = \perp = D_a(\varphi) \).

**Induction case \( R \cdot R' \) and \( R \) nullable:**

\[
\begin{align*}
L(\delta(R \cdot R')(a)) &= L(\delta(R) \cdot R' | \delta(R')(a)) \\
&= L(\delta(R) \cdot R'(a) | \delta(R')(a)) \\
&= L(\delta(R)(a)) \cdot L(R') \cup L(\delta(R')(a)) \\
&\equiv D_a(R) \cdot L(R') \cup D_a(R') = D_a(R \cdot R')
\end{align*}
\]

**Induction case \( R \cdot R' \) and \( R \) not nullable:**

\[
\begin{align*}
L(\delta(R \cdot R')(a)) &= L(\delta(R) \cdot R'(a)) = L(\delta(R)(a)) \cdot L(R') \\
&\equiv D_a(R) \cdot L(R') = D_a(R \cdot R')
\end{align*}
\]
Symbolic Boolean Derivatives

Induction case $R^*$:

\[ L(\delta(R^*)(a)) = L(\delta(R)\cdot R^*(a)) = L(\delta(R)(a)) \cdot L(R^*) \]

\[ \text{IH} \quad D_a(R) \cdot L(R^*) = D_a(R^*) \]

Induction case $R \circ R'$: Let $\circ \in \{\cup, \&\}$. $\hat{\cup} = \cup$ and $\hat{\&} = \cap$.

\[ L(\delta(R \circ R')(a)) = L(\delta(R)(a)) \circ \delta(\delta(R')(a)) \]

\[ \text{IH} \quad D_a(R) \circ D_a(R') = D_a(R \circ R') \]

Induction case $\neg R$:

\[ L(\delta(\neg R)(a)) = \Sigma^* \setminus L(\delta(R)(a)) \quad \text{IH} \quad \Sigma^* \setminus D_a(R) = D_a(\neg R) \]

The statement follows by the induction principle. \(\square\)

A useful property to observe about the proof of Theorem 4.3 is the following corollary.

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

Proof. If $R \in \mathbb{B}(RE)$ then lifting is never invoked. Complement and conjunction remain as top level operators only and are never nested within a concatenation or loop. \(\square\)

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 $\text{IF}(., \tau, .) \equiv \tau$. Thus, e.g., $\delta(.)$ simplifies to $\varepsilon$ (and so $\delta(.)$ simplifies to $r$).

\[ \delta(.*01.*) = \delta(.*01.*) | \delta(01.*) \]

\[ = \delta(.*01.*) | \delta(01.*) | \text{IF}(01, ., \bot) \]

\[ \delta(1.*) = \text{IF}(1, ., \bot) \]

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 transitions.
branches of conditional regexes. The negation of the complete form is seen in Figure 2c as the dual of Figure 2b, where \( \overline{1} = .*, \) and \( \overline{v} = \bot. \) A regex \( q \) is final (has double boundary) when \( q \) is nullable.

4.2 Algebraic Properties

Transition regexes form a particular kind of an effective Boolean algebra.\(^4\) 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 \&. For example \( r \& .* = r \) and \( \bot \cdot r = \bot. \) We also treat \( |, \& \) 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 not extensional, i.e., \( \tau \equiv \rho \) does in general not 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 \( R \in \mathbb{R}(RE) \) but DNF generalizes to all \( R \in ERE \) by using \( \text{lift}(r). \) For DNF we apply standard laws of distributivity. Perhaps the most relevant case here is \( \text{IF}(\varphi, \tau_1, \tau_2) \& \rho \) that in general expands to \( \text{IF}(\varphi, \tau_1 \& \rho, \tau_2 \& \rho) \) but is also subject to simplifications discussed next that integrate satisfiability checks of \( \mathcal{A} \) into the rules.

1. If \( \varphi \land \psi \equiv \bot \) then \( \text{IF}(\varphi, \tau, \bot) \& \text{IF}(\psi, \rho, \bot) \equiv \bot \) else \( \text{IF}(\varphi, \tau, \bot) \& \text{IF}(\psi, \rho, \bot) \equiv \text{IF}(\varphi \land \psi, \tau \& \rho, \bot). \)

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 \( \varphi \land \psi \equiv \bot \) then \( \tau \) simplifies to \( \text{IF}(\varphi, \tau_2, \rho) \) or if \( \varphi \land \neg \psi \equiv \bot \) then \( \tau \) 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 \( \neg \text{IF}(\varphi, .*, \bot) \equiv \text{IF}(\neg \varphi, .*, \bot). \)

Example 4.6. Recall the computation of \( \delta(.01.*) \) from Example 4.5. Let \( r = \neg(.01.*) \). In Section 2 we showed that \( \delta(r) \) can be computed initially as \( \neg \delta(.01.*) \) and then we take its DNF so that in the end \( \delta(r) \equiv \text{IF}(\emptyset, r \& \neg(1.*), r). \) It is also easy to see that \( \delta(\neg(1.*)) \equiv \text{IF}(1, .*, \bot). \) We continue with the regex \( r \& \neg(1.*) \) and get that

\[
\delta(r \& \neg(1.*)) = \delta(r) \& \delta(\neg(1.*))
\]

\[
= \text{IF}(\emptyset, r \& \neg(1.*), r) \& \text{IF}(1, .*, \bot)
\]

\[
= \text{IF}(\emptyset, r \& \neg(1.*), \text{IF}(1, .*, \bot, r))
\]

where the last equality uses, among other simplifications, the fact that \( \emptyset \land 1 \equiv \bot \) to keep the resulting conditional regex clean. The resulting transitions are shown in Figure 2(d).

\(^4\)One can view \( TR \) as a Boolean algebra over \( \Sigma^* \) where \( f : \Sigma \to 2^{\Sigma^*} \) is represented by \( \bigcup_{a \in \Sigma} a f(a) \subseteq \Sigma^* \) where \( aL = \{av \mid v \in L\}. \)
When working with the two algebras $\mathcal{A}$ and $\mathcal{TR}$, it is important to keep in mind that their Boolean operations have different semantics. For example, the predicate $\neg \varphi$ as a singleton regex denotes the language $L(\neg \varphi) = \Sigma \setminus \{\varphi\}$, while the regex $\neg \varphi$ denotes the language $L(\neg \varphi) = \Sigma^* \setminus \{\varphi\}$.

We show in Theorem 7.4 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 $R$.

### 4.3 Lift rules

The lifting rule $\text{lift}(\tau)$ propagates the intersection into the leaves and thus lifts the conditionals to the top level. Here we also pass the branch condition $\psi$ that is initially $\bot$, 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 $\tau$ is in NNF. The NNF rules are specified below.

$$\text{lift}(\tau) = \text{lift} (\tau)$$

$$\text{lift}_\psi(\tau) = \bot \text{ if } \psi \equiv \bot$$

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

$$\text{lift}_\psi(R) = R \text{ if } R \in \text{ERE} \text{ and } \psi \equiv .$$

$$\text{lift}_\psi(R) = \text{IF}(\psi, R, \bot) \text{ if } R \in \text{ERE} \text{ and } \psi \not\equiv .$$

$$\text{lift}_\psi(\text{IF}(\varphi, t, f)) = \text{IF}(\varphi, \text{lift}_\psi(\varphi \land \varphi)(t), \text{lift}_\psi(\varphi \land \varphi)(f))$$

$$\text{lift}_\psi(\text{IF}(\varphi, t, f) \land \rho) = \text{lift}_\psi(\text{IF}(\varphi, t \land \rho, f \land \rho))$$

$$\text{lift}_\psi((\tau_1 | \tau_2) \land \rho) = \text{lift}_\psi(\tau_1 \land \rho) \lor \text{lift}_\psi(\tau_2 \land \rho)$$

### 4.3.1 NNF

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

$$\text{NNF}(\text{IF}(\varphi, \tau, \rho)) = \text{IF}(\varphi, \text{NNF}(\tau), \text{NNF}(\rho))$$

$$\text{NNF}(\neg \text{IF}(\varphi, \tau, \rho)) = \text{IF}(\varphi, \text{NNF}(\neg \tau), \text{NNF}(\neg \rho))$$

$$\text{NNF}(\neg \tau) = \text{NNF}(\neg \tau) \text{ if } R \in \text{ERE}$$

The remaining cases are standard applications of DeMorgan’s laws.

### 5 SOLVING EXTENDED REGULAR EXPRESSION CONSTRAINTS IN SMT

Here we show that 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. The regex solver for ERE constraints is part of the sequence theory solver in dZ3. One challenge here is that the problem is not an isolated decision procedure but needs to be integrated into the main satisfiability engine of the solver, and in particular, interact with the solver for the given background theory of characters. We describe our algorithm following our implementation that builds on Z3. A brief overview was given in Section 2.

We focus here on assertions of the form of \( s \in r \), called membership constraints, where \( s \) is a term whose sort\(^6\) is sequence over \( \Sigma \) or \( \Sigma^* \) and \( r \) is an ERE over \( \Sigma^* \). Such constraints exist in a broader context of formulas, including possibly other string constraints on \( s \). We 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 \( r \neq r' \) for regexes \( r, r' \) (this includes nonemptiness constraints) can also be reduced to membership using the Boolean operators. In particular \( r \neq \bot \) iff \( \exists x (x \in r) \), and \( r_1 \neq r_2 \) iff \( (r_1 \land \neg r_2) \lor (r_2 \land \neg r_1) \neq \bot \).

The regex solver dynamically maintains a graph \( G = (V, E, F, C) \) with additional derived components Dead and Alive. The vertices \( V \subseteq \text{ERE} \) 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 \( w \in Q(\delta_{\text{dnf}}(v)) \), 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.

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 \( v \) s.t. \( \forall w \in Q(\delta_{\text{dnf}}(v)) : (v, w) \in E \). In other words, a closed vertex is a vertex all of whose outgoing edges have been added to \( E \).
- \( \text{Alive} \subseteq V \) is the set of all \( v \) s.t. \( E^*(v) \cap F \neq \emptyset \).
- \( \text{Dead} \subseteq V \) is the set of all \( v \) s.t. \( E^*(v) \subseteq (C \setminus \text{Alive}) \). 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. 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 \( \text{Alive} \) and \( \text{Dead} \) 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 and to block search and to infer unsatisfiability of dead-end regexes, as indicated by the bor-rule in Figure 3a. It is important to note that \( G \) 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 \( s \), i.e., this means that any satisfiability checks of branches

\(^6\)We say sort for type as is custom in the context of SMT.
we apply this to the constraint in-tr

An implicit assumption is that (a) Membership propagation rules for ERE

5.1 Transition regex normal form

For example, consider the hypothetical rule that we reduce 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 \( in-tr(s, r_1) \land in-tr(s, r_2) \). Then, if we apply this to the constraint \( in-tr(s, (\cdot a) & (\cdot b)) \), we obtain two separate constraints which propagate separately, and we never arrive at the required contradiction and conclude the original transition regex is

\[
\begin{align*}
\text{in-tr}(s, \text{TR}(\phi, t, f)) & \quad \text{(ITE)} \\
(\phi(s_0) \land \text{in-tr}(s, t)) \lor (\neg \phi(s_0) \land \text{in-tr}(s, f)) & \quad \text{in-tr}(s, t_1) \lor \text{in-tr}(s, t_2) & \quad \text{(OR)} \\
\text{in}(s_0, r) & \quad \text{(ERE)} \\
\text{in}(s_0, r) \implies \text{in-tr}(s, t) & \quad \text{in-tr}(s_0, r) \lor \text{in-tr}(s, f) & \quad \text{in}(s, r) \implies \text{in-tr}(s, t_2) \lor \text{in-tr}(s, t_1) & \quad \text{(DER)} \\
\text{in}(s, r) & \quad r \notin G.\text{Dead} \\
(|s| = 0 \land \nu(r)) \lor (|s| > 0 \land \text{in-tr}(s, \delta^{\text{DNF}}(r)) \land \text{Upd}[r \rightarrow Q(\delta^{\text{DNF}}(r))] \lor (\text{Upd}[r \rightarrow Q(\delta^{\text{DNF}}(r))] \lor (\text{Upd}[r \rightarrow Q(\delta^{\text{DNF}}(r))] & \quad \text{(BOT)} \\
G := (V \cup Q, E \cup \{(r, q) \mid q \in Q\}, F \cup \{q \in Q \mid \nu(q)\}, C \cup \{r\}) & \quad \text{(UPD)} \\
\text{Udp}[r \rightarrow Q] & \quad G = (V, E, F, C) \\
\text{in}(s, r) & \quad r \in G.\text{Dead} \\
\bot & \quad (|s| > 0 \land \text{in-tr}(s, \delta^{\text{DNF}}(r)) \land \text{Upd}[r \rightarrow Q(\delta^{\text{DNF}}(r))] & \quad \text{(DER)} \\
\end{align*}
\]

(a) Membership propagation rules for EREs and transition predicates. Here \( r \in ERE \). Recall that \( \nu(r) \iff r \text{ is nullable} \). All rules are equivalence preserving in their respective contexts. In particular \( \text{in-tr}(s, t) \) rules are applied only when \(|s| > 0\). An implicit assumption is that \( r \in G.V \).

\[
\text{Udp}[r \rightarrow Q] = (G \cup (r, q) \mid q \in Q), E \cup \{q \in Q \mid \nu(q)\}, C \cup \{r\}) & \quad \text{(UPD)} \\
\text{in}(s, r) & \quad r \in G.\text{Dead} \\
\bot & \quad (|s| > 0 \land \text{in-tr}(s, \delta^{\text{DNF}}(r)) \land \text{Upd}[r \rightarrow Q(\delta^{\text{DNF}}(r))] & \quad \text{(DER)} \\
\end{align*}
\]

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

Fig. 3. Decision procedure propagation rules.

are performed in a global scope, independent of local assertions. Therefore \( G \) can persist across different logical scopes.

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. An unsolved membership constraint \( \text{in}(s, r) \) trigger a call to the regex solver that performs the steps below.

1. As shown in Figure 3a the der-rule either allows the solution \( s = \epsilon \) if \( r \) is nullable, or it propagates the goal \( \text{in-tr}(s, \delta^{\text{DNF}}(r)) \) provided that \( r \) is not dead and \( s \) is nonempty.

2. The propagation rules for \( \text{in-tr}(s, \delta^{\text{DNF}}(r)) \) create a search space where the leaves of \( \delta^{\text{DNF}}(r) \) eventually trigger new membership subgoals for \( s_1 \) as shown by the ere-rule.

3. In this process \( G \) is incrementally updated, triggered by \( \text{Udp}[r \rightarrow Q] \) where \( Q \) is the set \( Q(\delta^{\text{DNF}}(r)) \) of all the derivative regexes for \( r \) and \( r \) is consequently closed, as shown by the upd-rule in Figure 3b.

5.1 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.
unsatisfiable. More specifically, this would occur after propagating rules \texttt{der} and then \texttt{ite} starting from \texttt{in}(s, (a.*a)&(a.*b)), since \(\delta^{\text{DNF}}(r) = \texttt{if}(a, (a.*a)&(a.*b), \bot)\).

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 \texttt{lift} rules for \(r \in \text{ERE}\) (not for \(r \in \mathbb{B}(\text{RE})\)). The implication is that when simplifying \texttt{in-tr}(s, r), after applying \texttt{ite} and \texttt{or} as necessary, we can directly apply rule \texttt{ere} to the conjunctions, which are plain regexes not involving if-then-else.

Using this strategy, we can then 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 \(L(r) = \emptyset\). The theorem states that the rules provide a decision procedure for emptiness of EREs modulo any decidable character theory. The proof then uses the property that \(G\) represents an accurate reachability graph of the underlying symbolic automaton, where states that end up in \(G.\text{Dead}\) are equivalent to \(\bot\), and where states may be intersection regexes.

**Theorem 5.1.** Let \(r \in \text{ERE}\) and \(s\) be an uninterpreted constant. Then \(\text{in}(s, r) \vdash \bot\) iff \(r \equiv \bot\).

### 5.2 Complexity

Theorem 5.1 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 \(\mathbb{B}(\text{RE})\) in Theorem 7.4: for this case, we show 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 \(\mathbb{B}(\text{RE})\), this implies exponential worst-case complexity for the decision procedure here, for \(\mathbb{B}(\text{RE})\). For general extended regexes, nonemptiness is known to be non-elementary [44], so we can only hope for concrete complexity bounds in practical subclasses.

### 5.3 Alive and dead state detection

In the implementation the graph \(G\) incrementally maintains a DAG of strongly connected components (SCCs) using the Union-Find datastructure [45] for implementing SCCs, and it implements explicit marking of SCCs corresponding to the \texttt{Dead} and \texttt{Alive} subsets of \(V\). We let \(\text{Find}(v)\) denote the SCC that contains \(v\). The event of adding a new batch of edges to \(E\) causes an incremental cycle detection algorithm to be executed, that is immediately followed by an algorithm that incrementally updates the DAG of SCCs and propagates the markings of \texttt{Dead} and \texttt{Alive} vertices.

We implemented a custom variant of incremental cycle detection and SCC maintenance algorithms, that is similar in spirit to the algorithm described in [6]. A unique aspect of our algorithm is that it makes use
of an additional dissimilarity heuristic asserting that certain states \( p \) and \( q \) can never belong to the same SCC, denoted by \( p \sim q \), because they can never be both in the same cycle. For example if \( p = \text{abc} \) then \( \delta(p) = \text{ir}(\text{a,b,c}, \bot) \) and, let \( q = \text{bc} \), trivially \( p \sim q \). This information is used by the DFS search algorithms in our incremental SCC algorithm to prune the search space during cycle detection.

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 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 SEREs), 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 SMTlib [41, 42]. We add to this an existing set of benchmarks provided in [8, 40], 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.

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 SEREs.

To evaluate Q3, we wrote four sets of examples. Unlike in Q2, we incorporate explicit intersection and complement. The first set contains problems involving date constraints, where a string is constrained to look
### Table 1: Experimental Results

<table>
<thead>
<tr>
<th>Solver</th>
<th>Solved</th>
<th>Avg (s)</th>
<th>Med (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NB</td>
<td>B</td>
<td>H</td>
</tr>
<tr>
<td>dz3</td>
<td>95.6%</td>
<td>88.1%</td>
<td>87.6%</td>
</tr>
<tr>
<td>cvc4</td>
<td>97.6%</td>
<td>86.4%</td>
<td>57.3%</td>
</tr>
<tr>
<td>z3str3</td>
<td>94.3%</td>
<td>60.9%</td>
<td>–</td>
</tr>
<tr>
<td>z3trau</td>
<td>89.6%</td>
<td>48.7%</td>
<td>–</td>
</tr>
<tr>
<td>ostrich</td>
<td>84.5%</td>
<td>42.3%</td>
<td>85.4%</td>
</tr>
<tr>
<td>z3</td>
<td>81.8%</td>
<td>29.0%</td>
<td>41.6%</td>
</tr>
</tbody>
</table>

(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 number of benchmarks solved 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.

(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.

Fig. 4. Results of the experimental evaluation. (A full table of results can be found in the appendix.)
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).

For all experiments, we compared dZ3 with a representative list of state-of-the-art and actively maintained solvers: Z3 [20, 49], Z3str3 [7, 51], Z3-Trau [1, 50], CVC4 [5, 15], and OSTRICH [14, 35]. We exclude Z3str3 and Z3-Trau from the Q3 handwritten examples, since explicit intersection and complement are not supported. 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. 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 manually inspected solver errors and incorrect answers to ensure that they all appear to be unsupported cases, bugs, or crashes, and never a result of a malformed input (which we correct by replacing the input in question). The experiments were run on a Dell XPS13 with an Intel Core i7 CPU and 16GB of RAM.

6.1 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 all 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 symbolic Boolean derivatives reach state-of-the-art performance in practice, while on benchmarks with Boolean combinations the solver solves more benchmarks faster than any existing tool.
7 SYMBOLIC BOOLEAN FINITE AUTOMATA

In order to formally study the efficiency of our ERE 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.

7.1 SBFA

A Symbolic Boolean Finite Automaton or SBFA is a tuple \( M = (A, Q, \iota, F, q_\bot, \Delta) \) where \( A \) 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; \( q_\bot \in Q \setminus F \) is the bottom state; \( \Delta : Q \rightarrow \mathcal{T}\mathcal{R}_Q \) is the transition function such that \( \Delta(q_\bot) = q_\bot \), where \( \mathcal{T}\mathcal{R}_Q \) is defined in Section 4.

We lift the final condition to \( q \in \mathbb{B}(Q) \) denoted \( \nu_F(q) \) as follows: \( \nu_F(q) \) iff \( q \in F \), \( \nu_F(p | q) \) iff \( \nu_F(p) \) or \( \nu_F(q) \), \( \nu_F(p \& q) \) iff \( \nu_F(p) \) and \( \nu_F(q) \), and \( \nu_F(\neg q) \) iff not \( \nu_F(q) \).

The definition of \( \Delta \) is lifted similarly to \( \mathbb{B}(Q) \rightarrow \mathcal{T}\mathcal{R}_Q \).

7.1.1 Semantics. \( M \) denotes \( M : \mathbb{B}(Q) \rightarrow \Sigma^* \) by the equations

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

The language accepted by \( M \) is \( L(M) = M(\iota) \).

7.2 Construction from Regexes

The 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. With respect to the granularity that is as assumed below, a terminal of \( \text{if}(\varphi, \tau, \rho) \) is a terminal of \( \tau \) or \( \rho \), a terminal of \( \neg \tau \) is a terminal of \( \tau \), and a terminal of \( \tau \circ \rho \) is a terminal of \( \tau \) or \( \rho \). If \( \tau \in RE \) then \( \tau \) is a terminal. In this case, states (other than potentially \( \iota \) and \( \neg \bot = \ast \)) are themselves not conjunctions or negations.

The regex \( \bot \), that is the bottom state \( q_\bot \), and the dual top state regex \( \ast (= \neg \bot) \) are called trivial. Let \( Q(\tau) \) denote the set of all nontrivial terminals of a transition regex \( \tau \).
Symbolic Boolean Derivatives

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

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

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., $\delta^+((ab)+) = \{(ab)^*, b(ab)^*\}$ includes the start regex while $\delta^+(ab) = \{b, e\}$ does not.

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

**Proof.** The are finitely many different states reached in $\delta^+(R)$ because $L(R)$ is regular and because the various algebraic operations are represented concisely, e.g., $\&$ is idempotent, associative, commutative with unit element $\cdot$ and absorbing element $\bot$. Similarly for $|$ and $\cdot$. $\square$

$SBFA(R)$. The SBFA of $R \in ERE$ is defined as follows, where $Q = \delta^+(R) \cup \{R, \bot, .+\}$ and $F = \{q \in Q \mid q$ is nullable\}.$^7$ $SBFA(R) = (A, Q, R, F, \bot, \delta|Q)$

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

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

**Proof.** The statement follows by proving that $\forall q \in \mathbb{B}(Q) : v \in M(q) \iff v \in L(q)$ by induction over $|v|$. The base case $v = \epsilon$ follows because $v_F(q) \iff v(q)$. The induction case is: $av \in M(q)$ iff $v \in D_a(M(q))$ iff $v \in M(\delta(q)(a))$ iff (by the IH) $v \in L(\delta(q)(a))$ iff (by Theorem 4.3) $v \in L(D_a(q))$ iff (by [9, Theorem 3.1]) $av \in L(q)$. $\square$

Theorem 7.4 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 $\#(R)$ denote the number of predicate nodes in $R$. We need the following lemma.

**Lemma 7.3.** If $X, Z \in RE$ are clean and normalized then $\delta^+(XZ) = \delta^+(X)Z \cup \delta^+(Z)$; if $X = S^*$ then $\delta^+(X) = \delta^+(S)X$.

**Proof.** We prove by induction over $X$ that

$$\delta^+(XZ) = \delta^+(X)Z \cup \delta^+(Z).$$

It follows from working with normalized regexes that in a concatenation node the first element is not a concatenation and we apply case analysis over the first element, that is not an intersection or complement because here we only consider standard regexes.

---

$^7$We write $\delta|Q$ to denote $\delta$ restricted to the finite set $Q$ — to follow the SBFA definition strictly.
Base case $X = \varepsilon$. Follows immediately because $\delta(\varepsilon) = \emptyset$.

Induction case $X = \psi Y$.
Then $\delta(XZ) = \text{ir}((\psi, \varepsilon, \bot) \cdot YZ) = \text{ir}(\psi, YZ, \bot)$, so $Q(\delta(XZ)) = \{YZ\}$ and thus

$$
\delta^+(XZ) = \{YZ\} \cup \delta^+(YZ)
\overset{IH}{=} \{YZ\} \cup \delta^+(Y)Z \cup \delta^+(Z)
= \{(Y) \cup \delta^+(Y))Z \cup \delta^+(Z)
= \delta^+(X)Z \cup \delta^+(Z)
$$

Induction case $X = (X_1|X_2)Y$.
Then $\delta(XZ) = (\delta(X_1YZ) \cup \delta(X_2YZ))$, so

$$
\delta^+(XZ) = \delta^+(X_1YZ) \cup \delta^+(X_2YZ)
\overset{2xIH}{=} \delta^+(X_1YZ) \cup \delta^+(X_2Y)Z \cup \delta^+(Z)
\overset{2xIH}{=} (\delta^+(X_1)Y \cup \delta^+(Y))Z \cup
\delta^+(X_2)Y \cup \delta^+(Y))Z \cup \delta^+(Z)
= (\delta^+(X_1)Y \cup \delta^+(X_2Y) \cup \delta^+(Y))Z \cup \delta^+(Z)
\overset{(*)}{=} \delta^+(X_1X_2Y) \cup \delta^+(Y)Z \cup \delta^+(Z)
$$

In ($\star$), if $Y = \varepsilon$, the equality holds by definition of derivatives of a choice node. If $Y \neq \varepsilon$, then $X_1|X_2$ is smaller than $X$, and one can apply the IH on $(X_1|X_2)Y$ with $X_1|X_2$ as $X$ and $Y$ as an instance of the universal variable $Z$ in the lemma.

Induction case $X = S*Y$.
Then $\delta(X) = \delta(SX) \cup \delta(Y)$ because $S*$ is nullable. The proof step uses (1), for any normalized $W$:

$$
\delta^+(S*W) = \delta^+(S)S*W \cup \delta^+(W)
$$

Equation (1) is proved first as follows:

$$
\delta^+(S*W) = \delta^+(SS*W) \cup \delta^+(W)
\overset{(IH)}{=} \delta^+(S)S*W \cup \delta^+(S*W) \cup \delta^+(W)
\overset{(lfp)}{=} \delta^+(S)S*W \cup \delta^+(W)
$$
where (lfp) holds because \( \delta^+(S\cdot W) \subseteq \delta^+(S)S\cdot W \cup \delta^+(W) \) that can be shown separately. It follows that

\[
\delta^+(XZ) = \delta^+(S\cdot(YZ)) \\
\stackrel{(1)}{=} \delta^+(S)S\cdot YZ \cup \delta^+(YZ) \\
\stackrel{IH}{=} \delta^+(S)S\cdot YZ \cup \delta^+(Y)Z \cup \delta^+(Z) \\
= (\delta^+(S)S\cdot Y \cup \delta^+(Y))Z \cup \delta^+(Z) \\
\stackrel{(1)}{=} \delta^+(S\cdot Y)Z \cup \delta^+(Z) \\
= \delta^+(X)Z \cup \delta^+(Z)
\]

The statement follows by the induction principle. Observe that (1) implies the second part of the lemma by setting \( W = \varepsilon \).

**Theorem 7.4.** Let \( R \in \mathbb{B}(RE) \). If \( R \) is clean and normalized then \( |Q_{SBFA}(R)| \leq \#(R) + 3 \).

**Proof.** If \( R \) is normalized we can use a slightly condensed definition of \( \delta(R) \) that is inlined in the proof. We prove (2)

\[
|\delta^+(R)| \leq \#(R)
\]

by induction over \( R = R_1 \cdot Z \) where \( R_1 \) is not a concatenation and possibly \( Z = \varepsilon \), corresponding to the case that \( R \) is not a concatenation or that \( R \) is a conjunction or complement.

**Base case** \( R = \varepsilon \). Then \( |\delta^+(R)| = 0 = \#(R) \).

**Induction case** \( R = \psi Z \). Then \( \delta(\psi Z) = \mathbf{1}(\psi, Z, \perp) \) and so \( \delta^+(R) = \{Z\} \cup \delta^+(Z) \). Here \( Z \in RE \) counts for one terminal and \( \psi \) counts for one predicate node. Thus

\[
|\delta^+(R)| = |\delta^+(Z)| + 1 \leq \#(Z) + 1 = \#(R).
\]

**Induction case** \( R = (X|Y)Z \). Then \( \delta(R) = \delta(XZ) \mid \delta(YZ) \) and so \( \delta^+(R) = \delta^+(XZ) \cup \delta^+(YZ) \). Then, by Lemma 7.3,

\[
\delta^+(R) = \delta^+(XZ) \cup \delta^+(YZ) = \delta^+(X)Z \cup \delta^+(Y)Z \cup \delta^+(Z)
\]

which implies that (observe that, for a set \( S \), \( |S-Z| = |S| \))

\[
|\delta^+(R)| \leq |\delta^+(X)| + |\delta^+(Y)| + |\delta^+(Z)| \leq \\
\#(X) + \#(Y) + \#(Z) = \#(R).
\]

**Induction case** \( R = S\cdot Z \). Then \( \delta(R) = \delta(S)S\cdot Z \mid \delta(Z) \) and so, by using Lemma 7.3, \( \delta^+(R) = \delta^+(S)S\cdot Z \cup \delta^+(Z) \). Then

\[
|\delta^+(R)| \leq |\delta^+(S)| + |\delta^+(Z)| \leq \#(S) + \#(Z) = \#(R).
\]
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 [2].

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. Finally, we discuss work related to string solvers and implementation of the proposed techniques in the context of SMT solvers.

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 [2].
The connection between \( ERE \) (modulo \( \mathcal{A} \)) and symbolic derivatives was initially studied in-depth in [26], with the main application of language containment in \( ERE \). An important side result [26, 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 [9] and then Antimirov derivatives [3] and its generalization to \( ERE \) [12]. We show how they relate to \( \delta(R) \) for \( R \in RE \). Assume \( \Sigma \) is finite, let \( a \in \Sigma \), and let \( R_a = \delta(R)(a) \).

### 8.1.1 Brzozowski Derivatives

\( R_a \) is precisely the Brzozowski derivative [9, Theorem 3.1] \( D_a(R) \) of \( R \) wrt \( a \).\(^5\) If regexes are viewed as DFA states, \( D_a \) is the transition function for \( a \).

#### 8.1.2 Antimirov Derivatives

If \( R_a = \top \) then \( \partial_a(R) = \emptyset \) else \( R_a = \cup_{i=1}^n R_i \) and \( \partial_a(R) = \{ R_i \}_{i=1}^n \) is the Antimirov derivative [3, Definition 2.8] of \( R \) wrt \( a \) 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.

#### 8.1.3 Partial Derivatives of \( ERE \)

The Antimirov construction is extended to \( ERE \) in [12]. The formal construction \( \frac{\partial}{\partial_a}(R) \) in [12, 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 \( \cup \)-set of \&-sets. Intuitively \( \frac{\partial}{\partial_a}(R) = DNF(R_a) \).

### 8.2 Relation to Classical Automata

Parallel finite automata by Kozen [28], subsequently renamed to alternating finite automata or AFAs in [13], and Boolean finite automata or BFAs by Brzozowski and Leiss [10], were introduced independently ([10, 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 \( \Xi(Q) \) while AFAs use transitions to \( 2^{2^Q} \) encoding \( DNF(\Xi(\epsilon)(Q)) \).

Algebra \( \mathcal{A} \) is assumed to be such that \( \Sigma \) is finite and for each \( a \in \Sigma \) there is a predicate \( \hat{a} \) such that \( \|\hat{a}\| = \{a\} \). In a pure classical setting of finite automata, transition functions are usually parameterized by single characters, so the notion of character predicates seems vacuous. In our translation below we will make use of \( \mathcal{A} \), where input predicates such as \( \neg \hat{a} \) arise implicitly, because for example, a transition predicate \( \neg \text{if}(\hat{a}, q, \bot) \) simplifies to \( \text{if}(\hat{a}, q, \top) \) that logically corresponds to \( \text{if}(\hat{a}, \neg q, \bot) | \text{if}(\neg \hat{a}, q, \bot) \). Perhaps the most common use of predicates is that \( \text{if}(a, q, \bot) | \text{if}(\beta, q, \bot) \) reduces to \( \text{if}(a \lor \beta, q, \bot) \) and, analogously, \( \text{if}(a, q, \bot) \land \text{if}(\beta, q, \bot) \) reduces to \( \text{if}(a \land \beta, q, \bot) \).

We provide a description of SBFAs over finite alphabets as BFAs next.

#### 8.2.1 SSBFA to BFA

Let \( M = (\mathcal{A}, Q, \iota, F, q_\bot, \Lambda) \) be a SSBFA. The equivalent BFA of \( M \) is \( BFA(M) = (\Sigma, Q, \lambda(q, a), \Delta(q)(a), \iota, F) \).\(^6\)

\(^5\) \( D_a \) applies to the whole \( ERE \) class.
Proposition 8.1. \( L(M) = L(BFA(M)) \) with \( L \) as in [10, p.25].

8.2.2 BFA to SBFA. BFAs over \( \Sigma \) have a finite set of states \( Q \) an initial function \( i \in \mathbb{B}(Q) \), a set of final states \( F \subseteq Q \), and a transition function \( \delta : Q \times \Sigma \rightarrow \mathbb{B}(Q) \).

We translate \( M_{bfa} = (\Sigma, Q, \delta, i, F) \) into an equivalent SBFA as follows. The translation uses the fresh state \( q_\perp \notin Q \).

\[
SBFA(M_{bfa}) = (A, Q \cup \{q_\perp\}, i, F, q_\perp, \{q_\perp \rightarrow q_\perp\} \cup \{p \mapsto OR_{v \in \Sigma} 1f(\hat{a}, \delta(p, a), q_\perp)\})
\]

where \( \hat{a} \) is the predicate in \( A \) such that \( \llbracket \hat{a} \rrbracket = \{a\} \).

Proposition 8.2. \( L(SBFA(M_{bfa})) = L(M_{bfa}) \) with \( L \) as in [10].

8.2.3 AFA to SBFA. Alternating finite automata [13, 28] (AFAs) have a finite input alphabet \( \Sigma \), a finite set of states \( Q = \{q_i\}_{i=0}^{k-1} \), a start state \( i \in Q \), a set of final states \( F \subseteq Q \), and a transition function \( g : Q \rightarrow (\Sigma \times \{0,1\}^k) \rightarrow \{0,1\} \). Let \( g_p = g(p) \) for \( p \in Q \). A sequence \( v \in \{0,1\}^k \) represents the conjunction

\[
q_v = AND(q_i \in Q \mid v_i = 1)
\]

and for \( a \in \Sigma, p \in Q, \lambda v.g_p(a, v) \) represents the disjunction

\[
g_{p,a} = OR(q_v \mid g_p(a, v) = 1, v \in \{0,1\}^{(k)})
\]

where \( OR(\emptyset) = q_\perp \) is a new state and \( AND(\emptyset) = \neg q_\perp \). The translation of \( M_{afa} = (Q, i, F, \Sigma, g) \) into an equivalent SBFA is as follows

\[
SBFA(M_{afa}) = (A, Q \cup \{q_\perp\}, i, F, q_\perp, \{q_\perp \rightarrow q_\perp\} \cup \{p \mapsto OR_{v \in \Sigma} 1f(\hat{a}, g_{p,a}, q_\perp)\})
\]

Proposition 8.3. \( L(SBFA(M_{afa})) = L(M_{afa}) \) with \( L \) as in [13].

8.2.4 Remarks. Observe that the main difference between \( M_{afa} \) and \( M_{bfa} \) besides the initial state formula is that \( g \) relies essentially on DNF(\( \mathbb{B}^+(Q) \)) while \( \delta \) uses the full \( \mathbb{B}(Q) \) for state predicates. In that respect, the BFA formulation is closer in spirit to SBFAs. Thus, because of DNF, the size of \( \delta \) can be exponentially more succinct than \( g \) (if \( g \) is represented explicitly as its type suggests). Negation does not play a big role here because it can be eliminated at a linear cost. Therefore, AFAs and BFAs are to a large extent considered to be equivalent notions. As we know, this is not true in the symbolic case, when comparing SAFAs and SBFAs.

8.3 Relation to Symbolic Extensions of Automata

Symbolic alternating finite automata (s-AFAs) [16] and alternating data automata (ADAs) [25] are two orthogonal symbolic extensions of finite automata, in the former case via SFAs and in the latter case via data automata [24].

8.3.1 Symbolic Alternating Finite Automata. An s-AFA [16] (modulo \( A \)) 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
We apply similar principles in dZ3 to represent transition regexes in a canonical way. The implementation of MONA [27] where transitions are multi-terminal BDDs whose terminals are states. We apply similar principles in dZ3 to represent transition regexes in a canonical way. Where they preserve condition evaluation order and in this way support more direct and efficient serial well-defined (independent of choice) due to the local mintermization.

BSTs have been used before in a special class of deterministic symbolic transducers called alternating symbolic transducers. Proposition 8.4. \( L(SBFA(M_{\text{ Safa}})) = \mathcal{L}(M_{\text{ Safa}}) \)

Going from SBFA \( M = (Q, \tau, F) \) to s-AFA is possible but not easy in general. This is also related to why \( \sim \) is not supported in s-AFA [16]. W.l.o.g., assume that \( \Delta \) does not contain complement. This is achieved by adding negated states \( \check{q} \) to \( Q \) and for each negated state \( \check{q} \) letting \( \Delta(\check{q}) = \text{NNR}(\Delta(q)) \) where \( \text{NNR}(\tau) \) computes the negation normal form of \( \tau \) meaning that all negations are pushed down to states. In particular, \( \text{NNR}(\text{IF}(\varphi, \rho)) = \text{IF}(\varphi, \text{NNR}(\neg \varphi), \text{NNR}(\neg \rho)) \), and \( \text{NNR}(\neg q) = \check{q} \). The other cases are standard.

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

\[ \text{SAFA}(M) = (Q, \tau, \text{NNF}(i), F) \quad (q, \Delta(q) \in \text{Minterms}([\llbracket \Delta(q) \rrbracket])) \]

Proposition 8.5. \( \mathcal{L}(M) = \mathcal{L}(\text{SAFA}(M)) \)

The problem with this construction is that \( |\text{Minterms}(\Gamma)| \) can be exponential in \( |\Gamma| \) so the construction of \( \text{SAFA}(M) \) is exponential in the worst case. The same problem arises in s-AFA normalization [16] used for complementation.

8.3.2 Alternating Data Automata. This class of automata goes far beyond regular languages because registers are allowed to carry information across state boundaries so that consecutive data elements in traces are functionally related. Data automata, as defined in [24], use registers and have the expressive power of general Turing machines. In an alternating data automaton [25], arbitrary Boolean combinations of predicates can be used to relate before and after values of registers. It is stated in [24] that complement of alternating data automata is linear unlike in [16]. We are not aware of work relating ERE with ADAs.

8.3.3 Conditional Branching. Conditional transitions (although without Boolean combinations of states) have been used before in a special class of deterministic symbolic transducers called Branching Symbolic Transducers or BSTs [38]. 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 [27] where transitions are multi-terminal BDDs whose terminals are states.
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 SMTLIB2 format [46]. String solvers are integrated in the CDCL(T) architecture [34]. From the CP community, the MiniZinc format integrates membership constraints over regular languages presented as either DFAs or NFAs [32]. The solver presented in [29] 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 [12], 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 is advertised as a symbolic solver for string formulas that come from path constraints [14], and its solver is based on solving for pre-images. Our evaluation suggests that it performs either extremely well, or not at all, depending on categories. 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 [7] integrates several innovations around string equality solving. Many of the advances previously developed in S3 [47] and now integrated within Z3’s default string solver, 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 [4].

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 optimizations remain to be explored; in particular taking advantage of algebraic laws of derivatives of EREs and designing heuristics that capture common usage patterns and that can be exploited by CDCL-based solvers.

REFERENCES


Symbolic Boolean Derivatives


## A Full Experimental Results

Figure 6 contains the full experimental results, which were described in Section 6 and summarized in Figure 4.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Time (s)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt; .04</td>
<td>&lt; .12</td>
</tr>
<tr>
<td>dz3</td>
<td>5018</td>
<td>71</td>
</tr>
<tr>
<td>z3</td>
<td>4325</td>
<td>582</td>
</tr>
<tr>
<td>zstr3</td>
<td>4439</td>
<td>569</td>
</tr>
<tr>
<td>z3trau</td>
<td>3998</td>
<td>728</td>
</tr>
<tr>
<td>cvc4</td>
<td>3744</td>
<td>1323</td>
</tr>
<tr>
<td>ostrich</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kalua</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slag</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; .04</td>
<td>&lt; .12</td>
</tr>
<tr>
<td>dz3</td>
<td>1884</td>
<td>55</td>
</tr>
<tr>
<td>z3</td>
<td>934</td>
<td>101</td>
</tr>
<tr>
<td>zstr3</td>
<td>1143</td>
<td>542</td>
</tr>
<tr>
<td>z3trau</td>
<td>1224</td>
<td>178</td>
</tr>
<tr>
<td>cvc4</td>
<td>1887</td>
<td>61</td>
</tr>
<tr>
<td>ostrich</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Norn</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; .04</td>
<td>&lt; .12</td>
</tr>
<tr>
<td>dz3</td>
<td>366</td>
<td>282</td>
</tr>
<tr>
<td>z3</td>
<td>76</td>
<td>103</td>
</tr>
<tr>
<td>zstr3</td>
<td>626</td>
<td>2</td>
</tr>
<tr>
<td>z3trau</td>
<td>170</td>
<td>78</td>
</tr>
<tr>
<td>cvc4</td>
<td>544</td>
<td>132</td>
</tr>
<tr>
<td>ostrich</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Norm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; .04</td>
<td>&lt; .12</td>
</tr>
<tr>
<td>dz3</td>
<td>82</td>
<td>13</td>
</tr>
<tr>
<td>z3</td>
<td>44</td>
<td>30</td>
</tr>
<tr>
<td>zstr3</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>z3trau</td>
<td>47</td>
<td>50</td>
</tr>
<tr>
<td>cvc4</td>
<td>96</td>
<td>25</td>
</tr>
<tr>
<td>ostrich</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SyGuS-qgen</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; .04</td>
<td>&lt; .12</td>
</tr>
<tr>
<td>dz3</td>
<td>126</td>
<td>176</td>
</tr>
<tr>
<td>z3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>zstr3</td>
<td>277</td>
<td>4</td>
</tr>
<tr>
<td>z3trau</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cvc4</td>
<td>21</td>
<td>17</td>
</tr>
<tr>
<td>ostrich</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RegExLib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; .04</td>
<td>&lt; .12</td>
</tr>
<tr>
<td>dz3</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>z3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>zstr3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>z3trau</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>cvc4</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>ostrich</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RegExLib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; .04</td>
<td>&lt; .12</td>
</tr>
<tr>
<td>dz3</td>
<td>26</td>
<td>28</td>
</tr>
<tr>
<td>z3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>zstr3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>z3trau</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cvc4</td>
<td>17</td>
<td>46</td>
</tr>
<tr>
<td>ostrich</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Handwr.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; .04</td>
<td>&lt; .12</td>
</tr>
<tr>
<td>dz3</td>
<td>35</td>
<td>14</td>
</tr>
<tr>
<td>z3</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>zstr3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>z3trau</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cvc4</td>
<td>28</td>
<td>6</td>
</tr>
<tr>
<td>ostrich</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 6. Full results of the experiments, divided by double lines into non-Boolean benchmarks (regular expression constraints are on separate variables, top), Boolean benchmarks (multiple regular expression constraints on the same variable, middle), and additional handcrafted Boolean examples (bottom).