Kalpa Publications in Computing

Volume 18, 2024, Pages 152–162

LPAR 2024 Complementary Volume

On Symbolic Derivatives and Transition Regexes

Margus Veanes

Microsoft Research, Redmond, USA
margus@microsoft.com

Abstract

Symbolic derivatives of extended regular expressions or regexes provide a mechanism for incremental unfolding of regexes into symbolic automata. The underlying representation is in form of so called *transition regexes*, that are particularly useful in the context of SMT for supporting lazy propagation of Boolean operations such as *complement*. Here we introduce symbolic derivatives of further operations on regexes, including *fusion* and *lookaheads*, that are currently not supported in the context of SMT but have been used in many application domains ranging from matching to temporal logic specifications. We argue that those operators could also be supported in the context of SMT and motivate why such direct support would be beneficial.

1 Introduction

We consider two new applications of the use of symbolic derivatives of extended regular expressions or regexes (**ERE**) in two distinct but related domains. Symbolic derivatives of **ERE** were introduced in [13] and allow lazy unfolding of regexes into symbolic automata [3] by using so called *transition regexes* that are nested if-then-else expressions whose conditions are predicates in a given alphabet theory $\mathcal{A}$ and whose leaves are regexes. A good example of how such a symbolic derivative and transition regex can look like is taking the (symbolic) derivative of the regex $\tilde{}R$ where $R = \texttt{.*[a-d][a-z]}\{k\}$, denoted by $\delta(\tilde{}R)$, that results in the transition regex $\tilde{}\delta(R) = \text{ITE}(\texttt{[a-d]}, \tilde{}(R|\texttt{[a-z]}\{k\}), \tilde{}R)$ that can further be transformed into $\text{ITE}(\texttt{[a-d]}, \tilde{}R\texttt{\&}\tilde{}\texttt{([a-z]}\{k\}), \tilde{}R)$ via de Morgan's law applied to the left leaf. Observe that regex complement ($\tilde{}$) has just been propagated into the leaves of the transition regex, i.e., no determinization is required in order to handle complement as it is propagated lazily when derivatives are constructed on a need-to-know basis.

We start by considering *counters*, that are currently not supported in SMT, but whose derivative rule is otherwise well-known. We then extend the notion of derivatives for two new constructs *fusion* and *lookahead* with motivations why these constructs are interesting and practically relevant, but are currently not supported in the context of SMT either.

Counters or *bounded loops* are used heavily in practice but are also exposing an Achilles heel in many regex matching engines [14]. The example of the regex R above illustrates use of a counter. Sometimes the involved loop bounds are prohibitively large, and *unwinding* the loops upfront becomes infeasible, in particular when the counters are nested. Therefore, in order to properly support such regexes in SMT, counters should be supported natively.

Fusion of regexes $R{:}S$ denotes the language of all words uav such that a is an alphabet element, ua is accepted by R, and av is accepted by S. Fusion has primarily been used in SERE, that is a regular expression based sublanguage of PSL [6]. For us, this extension is related to work in generalizing temporal logic beyond the classical (propositional) setting to modulo theories [17]. Here the motivation is related to ongoing work in cloud reliability.

Lookaheads are used in practice in regex matching but are neither supported in any *non-backtracking* matching engines, nor in SMT. We show that derivatives can provide elegant and efficient ways to implement lookahead, and can potentially enhance the current derivative based implementation of IsMatch in the new nonbacktracking regular expression engine in .NET [8].

A general observation about symbolic derivatives is that, adding new constructs such as counters or fusion to **ERE** together with their derivation rules, does not affect any of the other constructs or rules. Such locality of changes is a very powerful principle when dealing with derivatives in general. A shining example of that is the derivative of complement that simply propagates complement to the leaves of the transition regex. Symbolic automata based treatment of complement would otherwise require *determinization* of the whole automaton as the fixpoint of all the derivatives. This is, in the worst case, not only exponential in the number of NFA states but can also be exponential in the number of transitions due to mintermization of predicates. For the concrete regex R above, determinization would produce 2^{k+1} states. Moreover, since the loop bound k is not written in unary, this amounts to $O(2^{2^n})$ many states where n is the size of R. Loop bounds beyond a hundred or even thousand are not uncommon [14].

Overview Section 2 defines basic background material, including a formal definition of *transition terms* that is based on transition regexes from [13]. Section 3 recalls $\mathbf{ERE}_{\mathcal{A}}$ and symbolic derivatives from [13] where $\mathcal{A}$ is an alphabet theory. Section 4 discusses *counters*. Section 5 discusses and motivates a new derivative rule for *fusion* of regexes. Section 6 discusses and motivates a new derivative rule for handling of *lookahead* in regexes. Section 7 discusses ongoing and future work.

2 Preliminaries

As a meta-notation throughout the paper we write $lhs \stackrel{\text{DEF}}{=} rhs$ to let *lhs* be *equal by definition* to *rhs*. In the following let $\mathbb{D}$ be a nonempty (possibly infinite) domain of core elements.

2.1 Effective Boolean algebra

An *effective Boolean algebra (EBA) over* $\mathbb{D}$ [3] is a tuple

$$\mathcal{A} = (\mathbb{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$$

where Ψ is a set of *predicates* closed under the Boolean connectives; $[\![\_]\!] : \Psi \to 2^{\mathbb{D}}$ is a *denotation function*; $\bot, \top \in \Psi$; $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathbb{D}$, and for all $\alpha, \beta \in \Psi$, $[\![\alpha \vee \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$, $[\![\alpha \wedge \beta]\!] = [\![\alpha]\!] \cap [\![\beta]\!]$, and $[\![\neg\alpha]\!] = \mathbb{D} \setminus [\![\alpha]\!]$. We also write $a \vDash \alpha$ for $a \in [\![\alpha]\!]$. For $\alpha, \beta \in \Psi$ we write $\alpha \equiv \beta$ to mean $[\![\alpha]\!] = [\![\beta]\!]$. In particular, if $\alpha \equiv \bot$ then α is *unsatisfiable*, also denoted by $\mathbf{UNSAT}(\alpha)$, else $\mathbf{SAT}(\alpha)$ if $\alpha \not\equiv \bot$. We require that the connectives of $\mathcal{A}$ are computable and that $a \vDash \alpha$ is decidable. $\mathcal{A}$ is *decidable* if $\mathbf{SAT}(\alpha)$ is decidable. $\mathcal{A}$ is *extensional* if $\alpha \equiv \beta$ implies that $\alpha = \beta$, e.g., a BDD algebra is typically extensional. If $\mathcal{A}$ is extensional then it is clearly decidable.

2.2 Derivatives

Let $\mathfrak{D}$ be either $\mathbb{D}^*$ or $\mathbb{D}^\omega$ (ω-words over $\mathbb{D}$), with the associated definition of complement $\complement(L) \stackrel{\text{DEF}}{=} \mathfrak{D} \setminus L$ for $L \subseteq \mathfrak{D}$. The *derivative of L for $a \in \mathbb{D}$* is $\mathbf{D}_a(L) \stackrel{\text{DEF}}{=} \{v \mid av \in L\}$. Observe that $\mathbf{D}_a(\complement(L)) = \complement(\mathbf{D}_a(L))$ and $\mathbf{D}_a(L_1 \cap L_2) = \mathbf{D}_a(L_1) \cap \mathbf{D}_a(L_2)$. For a nonempty word w let w_0 denote the first element of w and $w_{(1..)}$ the rest of w.

2.3 Transition Terms

Let $\mathcal{A} = (\mathbb{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ be a given EBA and let Φ be some given set called *leaves*. We work with *transition regexes* [13] in a generalized and normalized form that can be applied both, to represent derivatives over $\mathbb{D}^*$, as well as $\mathbb{D}^\omega$, but here we will primarily use $\mathbb{D}^*$. We use here a *lifted* view of transition terms to avoid *lift rules* [13, Section 4.1] – we consider it an *implementation choice* here whether to apply the operations (1) and (2) below eagerly or lazily, but the simplified view here is that those operations are eager.

The set of *transition terms* $\mathcal{T}_\mathcal{A}\langle\Phi\rangle$ ($\mathcal{T}\langle\Phi\rangle$ or just $\mathcal{T}$ when both $\mathcal{A}$ and Φ are clear from the context) is defined by the following abstract grammar where $f \in \mathcal{T}$, $\alpha \in \Psi$, and $\ell \in \Phi$:

$$f \quad ::= \quad \ell \quad | \quad \text{ITE}(\alpha, f_1, f_2)$$

where α is called the *condition*, f_1 the *true-case* and f_2 the *false-case* of $\text{ITE}(\alpha, f_1, f_2)$. All binary operations $\diamond : \Phi \times \Phi \to \Phi$, and unary operations $\blacklozenge : \Phi \to \Phi$ are *lifted* to $\mathcal{T}$:

$$\blacklozenge\text{ITE}(\alpha, f, g) \stackrel{\text{DEF}}{=} \text{ITE}(\alpha, \blacklozenge f, \blacklozenge g) \tag{1}$$

$$\begin{aligned} \text{ITE}(\alpha, f_1, f_2) \diamond g &\stackrel{\text{DEF}}{=} \text{ITE}(\alpha, f_1 \diamond g, f_2 \diamond g) \\ \ell \diamond \text{ITE}(\alpha, f_1, f_2) &\stackrel{\text{DEF}}{=} \text{ITE}(\alpha, \ell \diamond f_2, \ell \diamond f_2) \end{aligned} \tag{2}$$

Given $a \in \mathbb{D}$ and $f \in \mathcal{T}$, the *leaf (value) of f for a*, denoted by $f[a]$, is defined as follows.

$$\ell[a] \stackrel{\text{DEF}}{=} \ell, \qquad \text{ITE}(\alpha, f_1, f_2)[a] \stackrel{\text{DEF}}{=} \begin{cases} f_1[a], \textbf{ if } a \in [\![\alpha]\!]; \\ f_2[a], \textbf{ otherwise}. \end{cases}$$

The following facts hold for all the lifted operations:

$$(f \diamond g)[a] = f[a] \diamond g[a] \textbf{ and } (\blacklozenge f)[a] = \blacklozenge(f[a]) \tag{3}$$

Functional equivalence of $f, g \in \mathcal{T}$ is defined by

$$f \cong g \stackrel{\text{DEF}}{=} \forall a \in \mathbb{D} : f[a] = g[a]$$

The set of all leaves of f, $Lvs(f)$, and the set of all conditions that occur in f, $Conds(f)$, are defined inductively. A transition term f is *clean* when all of its branches are feasible. Many rewrites that preserve $\cong$, such as *cleaning* [13], can be applied incrementally during operations on $\mathcal{T}$ by incorporating *satisfiability* checking modulo $\mathcal{A}$ to eliminate unreachable subterms. The trivial rewrites $\text{ITE}(\_, f, f) \cong f$, $\text{ITE}(\top, f, \_) \cong f$, and $\text{ITE}(\bot, \_, f) \cong f$ should be seen as built-in simplifications that are always applied.

A further normalization can be applied to (2) by using a total order '$<$' over Ψ if such an order exists. Then (2) has the case:

$$\underbrace{\text{ITE}(\alpha, f_1, f_2)}_{f} \diamond \underbrace{\text{ITE}(\beta, g_1, g_2)}_{g} \stackrel{\text{DEF}}{=} \begin{cases} \text{ITE}(\alpha, f_1 \diamond g_1, f_2 \diamond g_2), & \textbf{if } \alpha = \beta; \\ \text{ITE}(\alpha, f_1 \diamond g, f_2 \diamond g), & \textbf{else if } \alpha < \beta; \\ \text{ITE}(\beta, f \diamond g_1, f \diamond g_2), & \textbf{otherwise}. \end{cases}$$

This definition is analogous to how BDD operations are implemented. Observe also that $\text{ITE}(\alpha, f, g) \cong \text{ITE}(\neg\alpha, g, f)$. A further rule that can be used is to rewrite $\text{ITE}(\alpha, f, g)$ into $\text{ITE}(\neg\alpha, g, f)$ when for example $\neg\alpha < \alpha$.

Cleaning Through Restriction

Consider a decidable EBA $\mathcal{A}$ and let f be a transition term in $\mathcal{T}_{\mathcal{A}}\langle\Phi\rangle$. A straightforward cleaning procedure of f is achieved through *restriction*, $f\restriction\top$, where for $\beta \in \Psi$, $f\restriction\beta$ is defined as follows. For all leaves ℓ let $\ell\restriction\beta \stackrel{\text{DEF}}{=} \ell$ and for ITE-terms let

$$\text{ITE}(\alpha, f_1, f_2)\restriction\beta \stackrel{\text{DEF}}{=} \begin{cases} f_2\restriction\beta, & \textbf{if UNSAT}(\alpha \wedge \beta); \\ f_1\restriction\beta, & \textbf{else if UNSAT}(\neg\alpha \wedge \beta); \\ \text{ITE}(\alpha, f_1\restriction(\alpha \wedge \beta), f_2\restriction(\neg\alpha \wedge \beta)), & \textbf{otherwise}. \end{cases}$$

Observe the invariant that β is satisfiable above, when starting the cleaning from $f\restriction\top$. Note also that if $\mathcal{A}$ is extensional then $\alpha\wedge\beta = \bot$ iff $\textbf{UNSAT}(\alpha\wedge\beta)$. While in the context of SMT solvers, like Z3, interval constraints that arise in transition regexes are not extensional in general, the EBA used for character classes in the regex matchers in [8, 15] is not only extensional, but is a *K-bit bitvector algebra* [8, Section 5.1], say BV. In most practical cases $K \leq 64$ and BV represents predicates using unsigned 64-bit integers or `UInt64` where all the Boolean operations are essentially $O(1)$ operations: bitwise-AND, bitwise-OR, and bitwise-NOT, with $\bot = 0$.

When implementing the binary operations (2) it is beneficial to maintain cleanness of transition terms as a global invariant, in particular when working with $\mathcal{A} = BV$. Then the restriction operation is embedded into (2), as is the case with transition regexes in [13].

3 Extended Regular Expressions Modulo $\mathcal{A}$

We fix the alphabet algebra $\mathcal{A} = (\mathbb{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$. $\mathbf{ERE}_{\mathcal{A}}$ or **ERE** for short is defined by the following abstract grammar, where $\alpha \in \Psi$. We let $R \in \mathbf{ERE}$, R is called a *regex*. Let $\mathbf{RE}_{\mathcal{A}}$ stand for the *standard* fragment of $\mathbf{ERE}_{\mathcal{A}}$ without intersection (`&`) and complement (`~`).

$$R ::= \alpha \mid \varepsilon \mid R_1\,|\,R_2 \mid R_1\&R_2 \mid R_1{\cdot}R_2 \mid R* \mid \sim R$$

The operators are in order of precedence, where union (`|`) binds weakest and complement (`~`) binds strongest. This definition gives rise to the EBA $(\mathbb{D}^*, \mathbf{ERE}, \mathcal{L}, \bot, \top*, |, \&, \sim)$ where

$$\mathcal{L}(\alpha) = [\![\alpha]\!], \quad \mathcal{L}(\varepsilon) = \{\epsilon\}, \quad \mathcal{L}(R_1{\cdot}R_2) = \mathcal{L}(R_1){\cdot}\mathcal{L}(R_2), \quad \mathcal{L}(R*) = \mathcal{L}(R)^*.$$

Nullability of a regex R, $\mathit{Null}(R)$, means that $\epsilon \in \mathcal{L}(R)$ that is defined inductively and is associated with each AST node of regexes at construction time as a Boolean flag.

$$\mathit{Null}(\varepsilon) \stackrel{\text{DEF}}{=} \textbf{true} \tag{4}$$
$$\mathit{Null}(R*) \stackrel{\text{DEF}}{=} \textbf{true} \tag{5}$$
$$\mathit{Null}(\alpha) \stackrel{\text{DEF}}{=} \textbf{false} \tag{6}$$
$$\mathit{Null}(\sim R) \stackrel{\text{DEF}}{=} \textbf{not}\ \mathit{Null}(R) \tag{7}$$
$$\mathit{Null}(R_1\,|\,R_2) \stackrel{\text{DEF}}{=} \mathit{Null}(R_1)\ \textbf{or}\ \mathit{Null}(R_2) \tag{8}$$
$$\mathit{Null}(R_1\&R_2) \stackrel{\text{DEF}}{=} \mathit{Null}(R_1)\ \textbf{and}\ \mathit{Null}(R_2) \tag{9}$$
$$\mathit{Null}(R_1{\cdot}R_2) \stackrel{\text{DEF}}{=} \mathit{Null}(R_1)\ \textbf{and}\ \mathit{Null}(R_2) \tag{10}$$

The *(symbolic) derivative of* $R \in \mathbf{ERE}$ is denoted by $\delta(R)$ [13] and is a transition regex in $\mathcal{T}_{\mathcal{A}}\langle\mathbf{ERE}\rangle$, defined as follows, where $\alpha \in \Psi$. Recall that (1) and (2) lift the operators to $\mathcal{T}$.

$$\delta(\varepsilon) \stackrel{\text{DEF}}{=} \bot \tag{11}$$

$$\delta(\alpha) \stackrel{\text{DEF}}{=} \text{ITE}(\alpha, \varepsilon, \bot) \tag{12}$$

$$\delta(R_1{\cdot}R_2) \stackrel{\text{DEF}}{=} \begin{cases} \delta(R_1){\cdot}R_2 \,|\, \delta(R_2), & \textbf{if } \mathit{Null}(R_1); \\ \delta(R_1){\cdot}R_2, & \textbf{otherwise}. \end{cases} \tag{13}$$

$$\delta(R*) \stackrel{\text{DEF}}{=} \delta(R){\cdot}R* \tag{14}$$

$$\delta(R_1 \diamond R_2) \stackrel{\text{DEF}}{=} \delta(R_1) \diamond \delta(R_2) \quad (\text{for } \diamond \in \{\,|\,, \&\}) \tag{15}$$

$$\delta(\tilde{}R) \stackrel{\text{DEF}}{=} \tilde{}\delta(R) \tag{16}$$

The union operation | over **ERE** is treated as an ACI (associative, commutative, and idempotent) operation in [13] which implies that $\mathcal{M}_R$, as defined next, is well-defined as a *symbolic DFA* (modulo $\mathcal{A}$), by having a *finite* state space.

Definition 1. [$\mathcal{M}_R$ for $R \in \mathbf{ERE}_{\mathcal{A}}$] $\mathcal{M}_R = (\mathcal{A}, Q, R, \varrho, F)$ where $Q = \bigcup_{q \in Q} \mathit{Lvs}(\delta(q)) \cup \{R\}$ is a finite set of states with R as the initial state, $\varrho : Q \to \mathcal{T}\langle Q\rangle$ is the transition function such that $\varrho(q) = \delta(q)$, and $F = \{q \in Q \mid \mathit{Null}(q)\}$. ⊠

Many regex rewrites are built-in simplification rules, such as treating $\bot$ as the unit of |, $\top*$ as the unit of &, ε as the unit of $\cdot$, e.g., $R\,|\,\bot = R$ and $R{\cdot}\bot = \bot$. More advanced rewrite rules are also used as approximations of subsumption as illustrated in Example 1.

Let $\mathcal{M} = \mathcal{M}_R$. For all $u \in \mathbb{D}^*$ and $q \in Q$ let

$$\hat{\varrho}(q, u) \stackrel{\text{DEF}}{=} \textbf{if } u = \epsilon \textbf{ then } q \textbf{ else } \hat{\varrho}(\delta(q)[u_0], u_{(1..)})$$

For all $q \in Q$ let $\mathcal{L}_{\mathcal{M}}(q) \stackrel{\text{DEF}}{=} \{u \in \mathbb{D}^* \mid \hat{\varrho}(q, u) \in F\}$. Lemma 1 is implied by [13, Theorem 4.3] and [2, Theorem 3.1].

Lemma 1 ([13]). $\forall q \in Q, a \in \mathbb{D} : \mathcal{L}_{\mathcal{M}}(q) = \mathcal{L}(q)$ **and** $\mathcal{L}(\varrho(q)[a]) = \mathbf{D}_a(\mathcal{L}(q))$.

A state q is *alive* if $\mathcal{L}_{\mathcal{M}}(q) \neq \emptyset$ else *dead*. Thus, a state $\hat{\varrho}(R, u)$ is alive $\Leftrightarrow \exists x : ux \in \mathcal{L}(R)$. A new algorithm for dead state detection was recently presented in [12].

Symbolic NFA Construction

The definition of $\mathcal{M}_R$ is somewhat simplified above. In particular, it is often advantageous, as is the case in Z3, to represent $\mathcal{M}_R$ as a *symbolic NFA*: one can avoid incremental *determinization* by keeping $\delta(R)$ in DNF with each leaf S as a union $S_1 \,|\, \cdots \,|\, S_m$ and, instead of using S as a state, treating each S_i as a separate state, which introduces nondeterminism into the underlying automaton being constructed lazily, during further derivation steps. The symbolic NFA construction is related to Antimirov derivatives [1].

A classical example is $R = \top*{\cdot}\alpha{\cdot}\top^{(n)}$ where $\delta(R) = \text{ITE}(\alpha, R\,|\,\top^{(n)}, R)$ and $\delta(\top^{(n)}) = \top^{(n-1)}$ with R and $\top^{(i)}$ as separate states. So $\mathcal{M}_R$ would in this case be a symbolic NFA of with $O(n)$ states instead of a symbolic DFA with $O(2^n)$ states.

4 Counters

Let $k > 1$ and let R be a regex. A *counter* or *bounded loop* is a regex $R\{k\}$ with the language semantics $\mathcal{L}(R\{k\}) \stackrel{\text{DEF}}{=} \mathcal{L}(R)^{(k)}$. The *derivative* and *nullability* of $R\{k\}$ are defined as follows.

Let also $R\{1\} \stackrel{\text{DEF}}{=} R$ and $R\{0\} \stackrel{\text{DEF}}{=} \varepsilon$.

$$\delta(R\{k\}) \stackrel{\text{DEF}}{=} \delta(R)\cdot R\{k-1\} \tag{17}$$
$$\mathit{Null}(R\{k\}) \stackrel{\text{DEF}}{=} \mathit{Null}(R) \tag{18}$$

Counters often occur in regexes that are used in the context of security critical applications, for example in credential scanning [7] and in network intrusion detection [11]. Typically, counters are used as bounded repetitions over some character class, such as a base-64 alphabet.

Several important rewrite rules arise in derivatives of counters, that often kick in during matching [15], to reduce the state space of the automaton that has been constructed so far. The following example illustrates a typical case.

Example 1. Consider a regex $R = \texttt{.*a?\{1000\}}$ where `.` is $\top$ and `a?` abbreviates $\texttt{a}|\varepsilon$.[1]

$$\begin{aligned}\delta(R) &= \delta(\texttt{.*})\cdot\texttt{a?\{1000\}} \mid \delta(\texttt{a?\{1000\}})\\ &\cong R \mid \delta(\texttt{a?})\cdot\texttt{a?\{999\}}\\ &\cong R \mid \mathrm{ITE}(\texttt{a}, \varepsilon, \bot)\cdot\texttt{a?\{999\}}\\ &\cong R \mid \mathrm{ITE}(\texttt{a}, \texttt{a?\{999\}}, \bot)\\ &\cong \mathrm{ITE}(\texttt{a}, R|\texttt{a?\{999\}}, R)\end{aligned}$$

For the leaf $R|\texttt{a?\{999\}}$ of $\delta(R)$ we now get the derivative

$$\begin{aligned}\delta(R|\texttt{a?\{999\}}) &= \delta(R)|\delta(\texttt{a?\{999\}})\\ &\cong \mathrm{ITE}(\texttt{a}, R|\texttt{a?\{999\}}, R)|\mathrm{ITE}(\texttt{a}, \texttt{a?\{998\}}, \bot)\\ &\cong \mathrm{ITE}(\texttt{a}, R|\texttt{a?\{999\}}|\texttt{a?\{998\}}, R)\\ &\cong \mathrm{ITE}(\texttt{a}, R|\texttt{a?\{999\}}, R)\end{aligned}$$

In the last simplification, a *loop subsumption* rule is used based on the fact that, for all *nullable* regexes S, and for all n and m such that $n \leq m$, it holds that $\mathcal{L}(S\{n\}) \subseteq \mathcal{L}(S\{m\})$. The resulting symbolic DFA has therefore only *two* states as illustrated in Figure 1 ☒

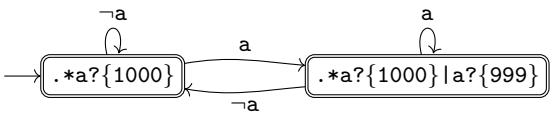

Figure 1: Symbolic DFA of `.*a?{1000}` in **RE** extended with counters.

Similar rewrites can be implemented in the context of SMT. Many of those rewrites, like the loop subsumption rule used in Example 1, would be hard to detect if the counter $\texttt{a?}\{k\}$ is first expanded into an equivalent repeated concatenation $\texttt{a?}^{(k)}$. Note also that the size of $\texttt{a?}^{(k)}$ is *exponential* in the size of $\texttt{a?}\{k\}$. A naive application of the derivative rule for concatenation (13) would cause further *quadratic* blowup of $\delta(\texttt{a?}^{(k)})$ because `a?` is *nullable*.

[1] The alternative notation for `.*a?{1000}` uses 0 as the *lower bound* in the bounded loop as `.*a{0,1000}`. In general, $S\{m,n\}$ where $m \leq n$ stands for $S\{m\}\cdot(S|\varepsilon)\{n-m\}$.

5 Fusion

Let $R, S \in \mathbf{ERE}$. *Fusion* of R with S, $R:S$, is a regex that denotes the language where the last and the first elements of the respective words must overlap, let $X, Y \subseteq \mathbb{D}^*$:

$$\mathcal{L}(R:S) \stackrel{\text{DEF}}{=} \mathcal{L}(R) : \mathcal{L}(S), \quad X : Y \stackrel{\text{DEF}}{=} \{xay \in \mathbb{D}^* \mid a \in \mathbb{D}, xa \in X, ay \in Y\}$$

Fusion is relatively unknown in the context of classical regexes, but plays an important role in the context of regexes (SERE) used in the IEEE 1850 Property Specification Language PSL of concurrent systems [6, 5]. In an ongoing line of work, we are investigating use of derivative-based unfolding of linear temporal logic enhanced with regular expressions, called $RLTL_{\mathcal{A}}$ [17], that also builds on transition terms and where fusion is a desired operator. In this setting the symbolic derivatives have language semantics over ω-words ($\mathbb{D}^\omega$) and are represented by transition terms in $\mathcal{T}_{\mathcal{A}}\langle RLTL_{\mathcal{A}}\rangle$. This framework combines regex derivatives with linear temporal logic (LTL) derivatives and thereby extends LTL into an ω-regular language modulo theories. The intended application domain for us in that work is trace analysis of cloud services, where the modulo theories aspect is crucial for dealing with JSON values represented by corresponding data types in Z3 [19]. Therefore, in that work, the ability to treat fusion through derivatives is fundamental. A variant of fusion is also present in other PSL operations, such as *existential suffix implication* $R \diamond\!\!\rightarrow \varphi$, that combines regexes R with temporal formulas φ.

5.1 One Step Lookahead

Let $R \in \mathbf{ERE}$. Let $\mathit{One}(R)$ be a predicate in Ψ that denotes $\mathcal{L}(R) \cap \mathbb{D}$. $\mathit{One}(R)$ plays a key role in the definition of the derivative for fusion below, essentially as a *one-step lookahead* predicate such that $a \in [\![\mathit{One}(R)]\!] \Leftrightarrow \mathit{Null}(\delta(R)[a])$.

$$\begin{aligned}
\mathit{One}(R) &\stackrel{\text{DEF}}{=} \mathit{One}(\top, \delta(R)) \\
\mathit{One}(\beta, R) &\stackrel{\text{DEF}}{=} \begin{cases} \beta, \textbf{ if } \mathit{Null}(R) \textbf{ and } \mathbf{SAT}(\beta); \\ \bot, \textbf{ otherwise}. \end{cases} \\
\mathit{One}(\beta, \mathrm{ITE}(\alpha, f, g)) &\stackrel{\text{DEF}}{=} \mathit{One}(\beta \wedge \alpha, f) \vee \mathit{One}(\beta \wedge \neg\alpha, g)
\end{aligned}$$

Thus (since $\bot$ is the unit of $\vee$), $\mathit{One}(R) = \bot \Leftrightarrow \mathcal{L}(R) \cap \mathbb{D} = \emptyset$. Note that if $\delta(R)$ is clean then the test $\mathbf{SAT}(\beta)$ is redundant. $\mathit{One}(R)$ is mostly used below when also $\delta(R)$ is needed, the calculation of the pair $(\mathit{One}(R), \delta(R))$ is therefore naturally combined into a single procedure.

5.2 Derivative of Fusion

The derivative of $R{:}S$ can be defined elegantly as the *union* of the derivative of S restricted with $\mathit{One}(R)$ (the condition that corresponds to the singleton words accepted by R) with the derivative of R whose leaves are fused with S:

$$\delta(R{:}S) \stackrel{\text{DEF}}{=} \mathrm{ITE}(\mathit{One}(R), \delta(S), \bot) \mid (\delta(R){:}S) \tag{19}$$

$$\mathit{Null}(R{:}S) \stackrel{\text{DEF}}{=} \textbf{false} \tag{20}$$

where again (2) lifts all the binary operations to transition regexes (extended with fusion). Many additional rewrite rules, such as $\varepsilon{:}S \cong \bot$, are then also used aggressively. It is also trivially clear from the formal semantics of fusion that $\mathit{Null}(R{:}S) = \textbf{false}$ because $\epsilon \notin \mathcal{L}(R{:}S)$.

Example 2. For $\alpha, \beta \in \Psi$ and $R = \alpha\texttt{*}\texttt{:}\beta\texttt{*}$, we get the following derivative, that we compare here side-by-side with the derivative of $S = \alpha\texttt{*}\cdot(\alpha\wedge\beta)\cdot\beta\texttt{*}$. Note that $\mathit{One}(\alpha\texttt{*}) = \alpha$.

$$\begin{array}{lcl}\delta(R) & \cong & \mathrm{ITE}(\alpha, \mathrm{ITE}(\beta, (\beta\texttt{*}\,|\,R), R), \bot)\\ \delta(S) & \cong & \mathrm{ITE}(\alpha, \mathrm{ITE}(\alpha\wedge\beta, (\beta\texttt{*}\,|\,S), S), \bot)\end{array}$$

where R and S are clearly equivalent regexes, which is also reflected in their derivatives. ☒

We now provide an informal argument showing that Lemma 1 holds for regexes with fusion, in particular that $\mathcal{L}_{\mathcal{M}}(R\texttt{:}S) = \mathcal{L}(R\texttt{:}S)$. Consider $X, Y \subseteq \mathbb{D}^*$ and let $av \in X\texttt{:}Y$. If $a \in X$ then either $av \in Y$ or $v \in \mathbf{D}_a(X)\texttt{:}Y$. If $a \notin X$ then $v \in \mathbf{D}_a(X)\texttt{:}Y$. This reformulation of $X\texttt{:}Y$ corresponds precisely to the definition of the symbolic derivative. For a formal argument, we can extend the inductive proof of [13, Theorem 4.3] with fusion as an additional construct.

6 Lookahead

Lookaheads (as well as lookbehinds) are in practice often used to define *boundary conditions* for matches when searching for matching subwords also known as *captures*. In some situations lookaheads are also used to overcome limitations of standard regexes (**RE**) where intersection is unavailable (by enabling a limited form of intersection). The abstract notation for lookaheads that we are using here is in line with the standard concrete notation. If $L \in \mathbf{RE}$ then the *lookahead of* L is $\texttt{(?=}L\texttt{)}$. While lookaheads can in general be combined freely in any context here we limit the discussion to when L itself *does not contain lookaheads* to simplify the reasoning.

Use of lookaheads is in some sense only meaningful as a prefix of a concatenation as in $\texttt{(?=}L\texttt{)}\cdot R$ where R can of course be ε as a trivial case but R can itself also contain lookaheads. With respect to language semantics $\texttt{(?=}L\texttt{)}\cdot R$ is equivalent with the regex $(L\cdot\top\texttt{*})\texttt{\&}R$ in **ERE**. In particular, observe that if L is *nullable* then

$$\mathcal{L}((L\cdot\top\texttt{*})\texttt{\&}R) = \mathcal{L}(L\cdot\top\texttt{*}) \cap \mathcal{L}(R) = \mathcal{L}(\top\texttt{*}) \cap \mathcal{L}(R) = \mathcal{L}(R)$$

The derivative of $\texttt{(?=}L\texttt{)}\cdot R$ propagates lookahead of the leaves of the derivative of L to the leaves of the derivative of R. Note that the lookahead operator $\texttt{(?=}f\texttt{)}$ below is lifted to transition terms f by (1), i.e.,

$$\texttt{(?=}\mathrm{ITE}(\alpha, f, g)\texttt{)} = \mathrm{ITE}(\alpha, \texttt{(?=}f\texttt{)}, \texttt{(?=}g\texttt{)})$$

The following is a form of synchronous product of transition terms that mimics intersection without explicitly introducing it.

$$\delta(\texttt{(?=}L\texttt{)}\cdot R) \overset{\text{DEF}}{=} \begin{cases}\delta(R), & \textbf{if } \mathit{Null}(L);\\ \texttt{(?=}\delta(L)\texttt{)}\cdot\delta(R), & \textbf{otherwise}.\end{cases} \tag{21}$$

$$\mathit{Null}(\texttt{(?=}L\texttt{)}) \overset{\text{DEF}}{=} \mathit{Null}(L) \tag{22}$$

Recall that concatenation $(\cdot)$ is lifted to transition terms by (2) that essentially constructs a "product" of the two transition terms whereby the leaves are concatenated as continued lookaheads – during which cleaning (modulo $\mathcal{A}$) is used to eliminate unreachable combined leaves. Note also that $\texttt{(?=}L\texttt{)}$ acts as ε if $\mathit{Null}(L)$. Use of the above rule also requires use of a right-associative normal form of concatenations of **RE** enhanced with lookaheads, say $\mathbf{RE}^{=}$, so that the *derivative rule (13) is not applied when* R_1 *is a lookahead*.

The following example shows a simple use case that illustrates a common pattern that avoids *factorial* explosion of regex size as the number of lookahead conditions is increased when expressed *without* lookaheads in **RE**. Such patterns are common for example in checking password conditions.

Example 3. The regex $R =$ `(?=.*a)(?=.*b)[a-z]*` matches a word in lowercase letters that contains at least one `a` and at least one `b`. In **ERE** R corresponds to `(.*a.*)&(.*b.*)&[a-z]*`. Next we construct the derivative of R by using the rules above and we apply cleaning to simplify, as indicated by $\cong$. Here '`.`' stands for $\top$. Any lookahead with nullable regex simplifies to ε, ε is the unit element of concatenation, and $\bot$ is the cancellation element of concatenation – we consider $\varepsilon{\cdot}R = R{\cdot}\varepsilon = R$ and $\bot{\cdot}R = R{\cdot}\bot = \bot$ as built-in equalities below.

We start by constructing the derivative of `(?=.*b)[a-z]*` in some detail to illustrate how the derivative operations, the propagation rules of regex operations, and the cleaning of transition terms work.

$$\begin{aligned}
\delta(\texttt{(?=.*b)[a-z]*}) &= \texttt{(?=}\delta(\texttt{.*b})\texttt{)}{\cdot}\delta(\texttt{[a-z]*}) \\
&= \texttt{(?=}\delta(\texttt{.*})\cdot\texttt{b|}\delta(\texttt{b})\texttt{)}{\cdot}\delta(\texttt{[a-z]*}) \\
&\cong \texttt{(?=.*b|}\mathrm{ITE}(\texttt{b},\varepsilon,\bot)\texttt{)}{\cdot}\delta(\texttt{[a-z]*}) \\
&\cong \texttt{(?=}\mathrm{ITE}(\texttt{b, .*b|}\varepsilon\texttt{, .*b)}){\cdot}\delta(\texttt{[a-z]*}) \\
&\cong \mathrm{ITE}(\texttt{b, (?=.*b|}\varepsilon\texttt{), (?=.*b)}){\cdot}\delta(\texttt{[a-z]*}) \\
&\cong \mathrm{ITE}(\texttt{b},\varepsilon,\texttt{(?=.*b)}){\cdot}\mathrm{ITE}(\texttt{[a-z]},\texttt{[a-z]*},\bot) \\
&= \mathrm{ITE}(\texttt{b},\varepsilon{\cdot}\mathrm{ITE}(\texttt{[a-z]},\texttt{[a-z]*},\bot),\texttt{(?=.*b)}{\cdot}\mathrm{ITE}(\texttt{[a-z]},\texttt{[a-z]*},\bot)) \\
&\cong \mathrm{ITE}(\texttt{b},\texttt{[a-z]*},\mathrm{ITE}(\texttt{[a-z]},\texttt{(?=.*b)[a-z]*},\bot))
\end{aligned}$$

We now construct the derivative of R but omit many intermediate rewrites.

$$\begin{aligned}
\delta(R) &= \texttt{(?=}\delta(\texttt{.*a})\texttt{)}{\cdot}\delta(\texttt{(?=.*b)[a-z]*}) \\
&\cong \mathrm{ITE}(\texttt{a},\varepsilon,\texttt{(?=.*a)}){\cdot}\mathrm{ITE}(\texttt{b},\texttt{[a-z]*},\mathrm{ITE}(\texttt{[a-z]},\texttt{(?=.*b)[a-z]*},\bot)) \\
&\cong \mathrm{ITE}(\texttt{a},\texttt{(?=.*b)[a-z]*},\mathrm{ITE}(\texttt{b},\texttt{(?=.*a)[a-z]*},\mathrm{ITE}(\texttt{[a-z]},R,\bot))) \\
&\cong \mathrm{ITE}(\texttt{a},\texttt{(?=.*b)[a-z]*},\mathrm{ITE}(\texttt{b},\texttt{(?=.*a)[a-z]*},R))
\end{aligned}$$

We get the derivative for the leaf `(?=.*a)[a-z]*` similarly to `(?=.*b)[a-z]*` above.

$$\delta(\texttt{(?=.*a)[a-z]*}) \cong \mathrm{ITE}(\texttt{a},\texttt{[a-z]*},\mathrm{ITE}(\texttt{[a-z]},\texttt{(?=.*a)[a-z]*},\bot))$$

We also have that $\delta(\texttt{[a-z]*}) = \mathrm{ITE}(\texttt{[a-z]},\texttt{[a-z]*},\bot)$. We have now reached a fixpoint. The resulting symbolic DFA accepting $\mathcal{L}(R)$ is illustrated in Figure 2. ⊠

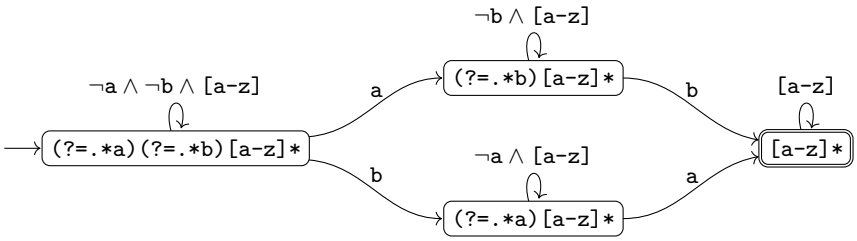

Figure 2: Symbolic DFA of `(?=.*a)(?=.*b)[a-z]*` in $\mathbf{RE}^{=}$.

So why bother about $\mathbf{RE}^{=}$ if one can already use intersection in **ERE**? The key point here is that $\mathbf{RE}^{=}$ is part of the standard use of regexes in practice, e.g., in .NET, while as of today, there *exists no industrial nonbacktracking matching engine that supports lookaheads*. Since the new nonbacktracking engine in .NET [8] is already built on derivatives, this technique can be

used to enhance its expressivity without requiring changes in the syntax of $\mathbf{RE}^{=}$, at least for `IsMatch`. From the perspective of SMT community one could consider a class of regexes that falls into the fragment of $\mathbf{RE}^{=}$ as a separate category.

7 Ongoing and Future Work

There are two lines of ongoing work where symbolic derivatives and transition terms are used as a core technology. We note that transition regexes for **ERE** were introduced in [13]. They are a symbolic generalization of *Brzozowski derivatives* [2] as well as *Antimirov derivatives* [1] and have been implemented in Z3 [4, 19].

In the first line of work the derivative based matching framework developed in the .NET library [8] is being generalized and lifted to **ERE** together with support also for *lookarounds* [15]. This work builds on a notion of derivatives that works with *spans* of words, that enables a formalization of both lookaheads and lookbehinds and allows to specify the POSIX semantics of matching. The correctness of this framework has recently also been formalized in Lean [20]. A future work item is to propose a similar extension to SMT that would enable to expand the support for regexes today to a much larger fragment.

The second line of work is about generalizing temporal logic beyond the classical (propositional) setting to modulo theories [17]. Here the motivation is related to ongoing work in cloud reliability, where we want to model, test, and verify behavioral properties of web services. The observable behavior of a web service at the top level can be seen as ω-words over JSON values that correspond to http request/response messages with the service. The $RLTL_{\mathcal{A}}$ framework that we have been working with is primarily intended for symbolic model checking of temporal properties. It first lifts classical LTL to modulo theories, $LTL_{\mathcal{A}}$, by using transition terms over $\mathbb{D}^{\omega}$. This construct lifts, what we call *Vardi derivatives* [16], to modulo $\mathcal{A}$. It has been widely recognized that full ω-regularity is fundamental for general applications of linear temporal properties [9], but is not expressible in LTL alone [18]. We were inspired by the core PSL primitives in SPOT [5], namely *existential suffix implication* ($R \diamond\!\!\rightarrow \varphi$) and *weak closure* ($\{R\}$), because both operators elegantly admit incremental derivative based unfolding and can be seamlessly integrated with transition regexes. While we have been working on a prototype in F#, we consider it fully feasible to also implement the framework directly in Z3, as then the sequence theory can be utilized more fully. Related open problems are to axiomatize ω-sequences in SMT and to standardize the PSL notation in SMT. Recently LTL modulo theories has also been studied in [10] for realizability of temporal specifications, where use of $RLTL_{\mathcal{A}}$ may also be relevant.

References

[1] Valentin Antimirov. Partial derivatives of regular expressions and finite automata constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[2] Janusz A. Brzozowski. Derivatives of regular expressions. *JACM*, 11:481–494, 1964.

[3] Loris D'Antoni and Margus Veanes. Automata modulo theories. *Communications of the ACM*, 64(5):86–95, May 2021.

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

[5] Alexandre Duret-Lutz. *Spot's Temporal Logic Formulas*, Dec 2022. Manual for Spot 2.11.3.

[6] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL*. Springer, 2006.

[7] Microsoft. CredScan, 2017. https://secdevtools.azurewebsites.net/helpcredscan.html.

[8] Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. Derivative based nonbacktracking real-world regex matching with backtracking semantics. In Nate Foster et al., editor, *PLDI '23: 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation, Florida, USA, June 17-21, 2023*, pages 1–2. ACM, 2023.

[9] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer, 1985.

[10] Andoni Rodríguez and César Sánchez. Boolean abstractions for realizability modulo theories. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, volume 9710 of *LNCS*, pages 305–328. Springer, 2023.

[11] Marty Roesch. SNORT. https://www.snort.org/.

[12] Caleb Stanford and Margus Veanes. Incremental dead state detection in logarithmic time. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, volume 13965 of *LNCS*, pages 241–264. Springer, 2023.

[13] Caleb Stanford, Margus Veanes, and Nikolaj S. Bjørner. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 620–635. ACM, 2021.

[14] Lenka Turoňová, Lukáš Holík, Ivan Homoliak, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. Counting in regexes considered harmful: Exposing redos vulnerability of nonbacktracking matchers. In *31st USENIX Security Symposium*, pages 4165–4182. USENIX, USENIX Association, 2022.

[15] Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. Derivative based extended regular expression matching supporting intersection, complement and lookarounds. In *arXiv*, 2023.

[16] Moshe Y. Vardi. *Alternating automata and program verification*, pages 471–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[17] Margus Veanes, Thomas Ball, Gabriel Ebner, and Olli Saarikivi. Symbolic automata: ω-regularity modulo theories. *arXiv*, (2310.02393), 2023.

[18] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1):72–99, 1983.

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

[20] Ekaterina Zhuchko, Margus Veanes, and Gabriel Ebner. Lean formalization of extended regular expression matching with lookarounds. In *CPP'24*, pages 118–131. ACM, 2024.