A Lazy Derivation-Based Regular Expression Membership Constraint Decision Procedure

Clemens Eisenhofer

Abstract. We present a lazy, derivative-based decision procedure for regular-expression membership constraints over a finite alphabet. The procedure works symbolically on extended regular expressions (intersection, complement), computes Brzozowski-style derivatives, and prevents infinite symbolic exploration by extracting left stabilizers via an annotated-history mechanism together with a widening operator. The method avoids explicit automaton construction and is designed for integration with SMT-style solvers where symbolic, incremental checks are important. We state soundness, discuss termination and complexity, and provide the core algorithm and motivating examples.

1 Introduction

We study satisfiability of regular-expression membership constraints over a finite, non-empty alphabet Σ, i.e. constraints of the form $u \in \mathcal{L}(r)$ where u is a symbolic string (concatenation of string variables and character constants) and r is an extended regular expression. Extended regular expressions allow intersection ($\sqcap$), complement ($\bar{\cdot}$) in addition to the usual constructs. Our goal is a compact, symbolic decision procedure that operates directly on expressions using derivatives rather than constructing explicit automata. The method is geared toward SMT-style integration: symbolic, incremental checks, and the ability to detect cycles in the symbolic exploration are essential.

1.1 Contributions

- A concise, symbolic decision procedure for membership constraints over extended regular expressions that avoids constructing full DFAs/NFAs; the core algorithm and pseudocode appear in Section 4.
- A practical mechanism to detect and record left stabilizers (prefix cycles) using annotated histories and a deterministic extraction policy, together with a widening operator Ω that merges repeating prefixes to guarantee a finite symbolic state-space under a finite alphabet.
- Proof sketches for soundness and completeness (Section 4.4), complexity remarks, and worked examples illustrating where naive exploration diverges and how stabilizer extraction prevents that.

2 Preliminaries

We fix a finite alphabet Σ and write $\mathcal{R}$ for the set of well-formed extended regular expressions over Σ. We use the following concrete syntax:

$$R ::= \varepsilon \mid \bot \mid a \mid RR \mid R^* \mid R \sqcup R \mid R \sqcap R \mid \overline{R}$$

for every $a \in \Sigma$. We write $\mathcal{L}(r)$ for the language denoted by r. Let X be the finite set of string variables occurring in the input and let $\mathcal{T}$ be the set of string terms, i.e., finite concatenations of alphabet symbols and variables from X. A primitive membership constraint has the form $x \in \mathcal{L}(r)$ with $x \in X$; in general, we consider membership constraints $u \in \mathcal{L}(r)$ with $u \in \mathcal{T}$. For simplicity, however, we just write $u \in r$ to denote $u \in \mathcal{L}(r)$. We write $r_1 \equiv r_2$ as an abbreviation for $\mathcal{L}(r_1) = \mathcal{L}(r_2)$ whereas $r_1 = r_2$ refers to syntactical equivalence. As usual, the union of empty sets $\bigsqcup$ is $\bot$, and the empty intersection is Σ^*. Further, we write $\bigsqcup S$ for some set S to denote $\bigsqcup_{s \in S} s$ and the analogous form for $\bigsqcap$. Also, we use Σ in the regular expression as an abbreviation for $\bigsqcup_{a \in \Sigma} a$.

For a variable $x \in X$ we write R^x for the current set of regular expressions r that occur in constraints of the form $x \in r$. We assume the alphabet Σ is finite throughout.

The core algorithmic ingredients are Brzozowski-style derivatives over Σ, a syntactic splitting function to reduce general constraints to primitive ones, and an annotated-history mechanism that detects and extracts left stabilizers to prevent infinite symbolic unfolding.

In practice, our implementation works on minterms. These are character classes such that, for any character from this class, the resulting derivative is the same. For simplicity, however, let us consider derivatives of single characters. Let $M(r)$ denote the set of "first characters" of a regular expression r in the following way:

$$\begin{aligned}
M(\bot) &:= \emptyset \\
M(\varepsilon) &:= \emptyset \\
M(a) &:= \{a\} \\
M(r_1^*) &:= M(r_1) \\
M(r_1 r_2) &:= \begin{cases} M(r_1) & \text{if } \neg\,\text{nullable}(r_1) \\ M(r_1) \cup M(r_2) & \text{otherwise} \end{cases} \\
M(r_1 \sqcup r_2) &:= M(r_1) \cup M(r_2) \\
M(r_1 \sqcap r_2) &:= M(r_1) \cup M(r_2) \\
M(\overline{r_1}) &:= \Sigma
\end{aligned}$$

Note that the intersection and complement cases are, in general, a coarse over-approximation. In practice, we would compute a more precise one based on min-terms, but for simplicity, let us go with this definition.

Nullable is the usual predicate with rules:

$$\begin{aligned}
\text{nullable}(\varepsilon) &= \mathsf{true},\\
\text{nullable}(\bot) &= \mathsf{false},\\
\text{nullable}(a) &= \mathsf{false},\\
\text{nullable}(r_1^*) &= \mathsf{true},\\
\text{nullable}(r_1 r_2) &= \text{nullable}(r_1) \land \text{nullable}(r_2),\\
\text{nullable}(r_1 \sqcup r_2) &= \text{nullable}(r_1) \lor \text{nullable}(r_2),\\
\text{nullable}(r_1 \sqcap r_2) &= \text{nullable}(r_1) \land \text{nullable}(r_2),\\
\text{nullable}(\overline{r_1}) &= \neg\, \text{nullable}(r_1).
\end{aligned}$$

Derivatives are defined for individual characters. For a character $a \in \Sigma$ and regular expression r we write $\delta_a(r)$:

$$\begin{aligned}
\delta_a(\varepsilon) &:= \bot,\\
\delta_a(\bot) &:= \bot,\\
\delta_a(a) &:= \varepsilon,\\
\delta_a(b) &:= \bot,\\
\delta_a(r_1^*) &:= \delta_a(r_1) r_1^*\\
\delta_a(r_1 r_2) &:= \delta_a(r_1) r_2 \sqcup \delta_a(r_2) && \text{if nullable}(r_1),\\
\delta_a(r_1 r_2) &:= \delta_a(r_1) r_2 && \text{if } \neg\, \text{nullable}(r_1),\\
\delta_a(r_1 \sqcup r_2) &:= \delta_a(r_1) \sqcup \delta_a(r_2),\\
\delta_a(r_1 \sqcap r_2) &:= \delta_a(r_1) \sqcap \delta_a(r_2),\\
\delta_a(\overline{r_1}) &:= \overline{\delta_a(r_1)}.
\end{aligned}$$

We use $\delta_w(r)$ with $w \in \Sigma^*$ as an abbreviation such that $\delta_\varepsilon(r) = r$ and $\delta_{aw'} = \delta_{w'}(\delta_a(r))$.

A regex r is non-empty ($\mathcal{L}(r) \neq \emptyset$) iff some derivative reachable by consuming a finite sequence of characters is nullable. This symbolic exploration corresponds to classical reachability for automata, but avoids explicit automata construction.

3 Solving by Splitting and Stabilizers

We transform general membership constraints into primitive ones (constraints where the left-hand side is a variable or concrete string) by splitting. We define a syntactic splitting function $\tau(r)$ for basic regular expressions r (i.e. expressions built from ε, $\bot$, a, concatenation, union, and Kleene star, without intersection or complement) that enumerates a set of factorization pairs $\langle \Delta, \nabla \rangle$ such that $\mathcal{L}(\Delta\nabla) \subseteq \mathcal{L}(r)$ for every pair, and conversely every word in $\mathcal{L}(r)$ is covered by some pair. The inductive definition is as follows:

$$\begin{aligned}
\tau(\varepsilon) &:= \{\langle \varepsilon, \varepsilon \rangle\}, \\
\tau(\bot) &:= \emptyset, \\
\tau(a) &:= \{\langle \varepsilon, a \rangle, \langle a, \varepsilon \rangle\}, \\
\tau(r_1 \sqcup r_2) &:= \tau(r_1) \cup \tau(r_2), \\
\tau(r_1 r_2) &:= \{\langle \varDelta_1, \nabla_1 r_2 \rangle \mid \langle \varDelta_1, \nabla_1 \rangle \in \tau(r_1)\} \cup \{\langle r_1 \varDelta_2, \nabla_2 \rangle \mid \langle \varDelta_2, \nabla_2 \rangle \in \tau(r_2)\} \\
\tau(r_1^*) &:= \{\langle r_1^*, r_1^* \rangle\} \cup \{\langle r_1^* \varDelta, \nabla r_1^* \rangle \mid \langle \varDelta, \nabla \rangle \in \tau(r_1)\}
\end{aligned}$$

Lemma 1 links this decomposition to solving regular membership constraints.

Lemma 1. *Let r be a basic regular expression (i.e. without intersection or complement). A membership constraint $uv \in r$ has a solution iff there exists $\langle \varDelta, \nabla \rangle \in \tau(r)$ such that $u \in \mathcal{L}(\varDelta)$ and $v \in \mathcal{L}(\nabla)$.*

Proof. By structural induction on r.

Base cases

$r = \varepsilon$ $uv \in \mathcal{L}(\varepsilon)$ iff $u = v = \varepsilon$; $\tau(\varepsilon) = \{\langle \varepsilon, \varepsilon \rangle\}$ and $\varepsilon \in \mathcal{L}(\varepsilon)$.
$r = \bot$ $\mathcal{L}(\bot) = \emptyset$ and $\tau(\bot) = \emptyset$, so both sides are false.
$r = a$ $uv = a$ iff $(u = \varepsilon, v = a)$ or $(u = a, v = \varepsilon)$; these are exactly the two pairs in $\tau(a)$.

Inductive cases

$r = r_1 \sqcup r_2$ $uv \in \mathcal{L}(r_1 \sqcup r_2)$ iff $uv \in \mathcal{L}(r_1)$ or $uv \in \mathcal{L}(r_2)$. By the induction hypothesis, this is equivalent to a matching pair in $\tau(r_1) \cup \tau(r_2) = \tau(r_1 \sqcup r_2)$.
$r = r_1 r_2$ Every word in $\mathcal{L}(r_1 r_2)$ can be written as $w_1 w_2$ with $w_1 \in \mathcal{L}(r_1)$, $w_2 \in \mathcal{L}(r_2)$. The split point of uv either falls inside w_1, on the boundary between w_1 and w_2, or inside w_2 (the boundary case is subsumed by either of the other two with $v_1 = \varepsilon$ or $u_2 = \varepsilon$, respectively). In the first case $u = u_1$, $v = v_1 w_2$ with $u_1 v_1 = w_1$; by induction there exists $\langle \varDelta_1, \nabla_1 \rangle \in \tau(r_1)$ with $u_1 \in \mathcal{L}(\varDelta_1)$ and $v_1 \in \mathcal{L}(\nabla_1)$, giving $\langle \varDelta_1, \nabla_1 r_2 \rangle$ with $v \in \mathcal{L}(\nabla_1 r_2)$. In the second case $u = w_1 u_2$, $v = v_2$ with $u_2 v_2 = w_2$; by induction there exists $\langle \varDelta_2, \nabla_2 \rangle \in \tau(r_2)$ giving $\langle r_1 \varDelta_2, \nabla_2 \rangle$. The converse directions follow analogously.
$r = r_1^*$

(⇒) Let $uv \in \mathcal{L}(r_1^*)$, so $uv = w_1 \cdots w_k$ with each $w_i \in \mathcal{L}(r_1)$. If the split point of uv falls on a boundary between repetitions, i.e. $u = w_1 \cdots w_j$ and $v = w_{j+1} \cdots w_k$, then $u \in \mathcal{L}(r_1^*)$ and $v \in \mathcal{L}(r_1^*)$, which is covered by the pair $\langle r_1^*, r_1^* \rangle \in \tau(r_1^*)$. Otherwise, the split falls inside some copy w_j, say $w_j = w_j' w_j''$ with $u = w_1 \cdots w_{j-1} w_j'$ and $v = w_j'' w_{j+1} \cdots w_k$. By the induction hypothesis on r_1, there exists $\langle \varDelta, \nabla \rangle \in \tau(r_1)$ with $w_j' \in \mathcal{L}(\varDelta)$ and $w_j'' \in \mathcal{L}(\nabla)$. Then $u \in \mathcal{L}(r_1^* \varDelta)$ and $v \in \mathcal{L}(\nabla r_1^*)$, matching the pair $\langle r_1^* \varDelta, \nabla r_1^* \rangle \in \tau(r_1^*)$.

(⇐) For the pair $\langle r_1^*, r_1^* \rangle$ the claim is immediate. For a pair $\langle r_1^* \Delta, \nabla r_1^* \rangle$ with $\langle \Delta, \nabla \rangle \in \tau(r_1)$: by the induction hypothesis $\mathcal{L}(\Delta\nabla) \subseteq \mathcal{L}(r_1)$, so $u \in \mathcal{L}(r_1^* \Delta)$ and $v \in \mathcal{L}(\nabla r_1^*)$ give $uv \in \mathcal{L}(r_1^* \cdot r_1 \cdot r_1^*) \subseteq \mathcal{L}(r_1^*)$.

Once all membership constraints are primitive, we can use a well-known lemma used by most complete decision procedures for regular membership constraints to actually compute the solution set.

Lemma 2. *Given a set of primitive membership constraints R^x for each variable x (i.e. each constraint has the form $x \in r$ for a single variable x), we can compute the solution set of each x independently by considering $\bigsqcap R^x$. This language describes exactly the set of all possible values for x.*

Proof. Since each constraint is primitive, the left-hand side of every constraint contains exactly one variable. Hence the constraints on distinct variables are independent: a global assignment σ satisfies all constraints iff for every $x \in X$ and every $r \in R^x$ we have $\sigma(x) \in \mathcal{L}(r)$. This is equivalent to $\sigma(x) \in \bigcap_{r \in R^x} \mathcal{L}(r) = \mathcal{L}(\bigsqcap R^x)$ for every x. Therefore, the set of satisfying assignments for x is exactly $\mathcal{L}(\bigsqcap R^x)$, and the overall system is satisfiable iff each such language is non-empty.

Hence, once we reduce all membership constraints to primitive ones, we can check

$$\bigwedge_{x \in X} (\mathcal{L}(\bigsqcap_{r \in R^x} r) \neq \emptyset) \tag{1}$$

to decide whether the current set of primitive membership constraints has a solution. By exhaustively enumerating all possible splits, we thus obtain a decision procedure for regular membership constraints.

However, computing all factorizations can blow up easily and does not extend cleanly to complement/intersection. Instead, we apply eager splitting only when the regex does not exceed a certain complexity threshold; otherwise, we split lazily using derivatives we will explain next to guide exploration while detecting repeating left prefixes ("left stabilizers") that cause infinite unfolding.

3.1 Stabilizers

Definition 1 (Left Quotient). *For a regex r and a word w define the left quotient*

$$w^{-1}r := \{v \mid wv \in \mathcal{L}(r)\}.$$

For a language L

$$L^{-1}r := \bigcup_{w \in L} w^{-1}r,$$

and for a regex s write $s^{-1}r := \mathcal{L}(s)^{-1}r$.

It is worth noting that for some $w \in \Sigma^*$ we have $w^{-1}r \equiv \delta_w(r)$, so δ gives us a way of computing some special cases of left quotients.

Definition 2 (Left Stabilizer). *A regex c is a (left) stabilizer of a regex r iff for every word $u \in \mathcal{L}(c)$:*

$$u^{-1}\mathcal{L}(r) = \mathcal{L}(r).$$

That is, quotienting r by any individual word of c does not change the language. As we only consider derivations from the left, we will drop the "left" qualifier and simply call them stabilizers.

For example, $c = ab$ is a stabilizer of $r = (ab)^*$: the only word $u = ab$ satisfies $u^{-1}\mathcal{L}((ab)^*) = \mathcal{L}((ab)^*)$. The strongest stabilizer (in terms of language inclusion) is $c = (ab)^*$. Conversely, $c = a$ is *not* a stabilizer of $r = aa^*$, because $a^{-1}\mathcal{L}(aa^*) = \mathcal{L}(a^*) \neq \mathcal{L}(aa^*)$. Consider the strongly connected component of the deterministic automaton defined by the Brzozowski derivatives of r. Each stabilizer describes a subset of the words accepted by the language defined by a deterministic automaton starting and accepting in initial state of the deterministic automaton. Moreover, the strongest stabilizer of r describes exactly the language of the strongly connected component.

Two useful consequences of the definition are the following.

Lemma 3 (Prefix Stabilizer Property). *If c is a stabilizer of r, then $\mathcal{L}(cr) \subseteq \mathcal{L}(r)$.*

Proof. Let $u \in \mathcal{L}(c)$ and $v \in \mathcal{L}(r)$. By definition $u^{-1}\mathcal{L}(r) = \mathcal{L}(r)$, so in particular $v \in u^{-1}\mathcal{L}(r)$, i.e. $uv \in \mathcal{L}(r)$. Hence $\mathcal{L}(cr) \subseteq \mathcal{L}(r)$.

Lemma 4 (Quotient Stabilizer Property). *If c is a stabilizer of r and $\mathcal{L}(c) \neq \emptyset$, then $\mathcal{L}(c)^{-1}r = \mathcal{L}(r)$.*

Proof. $\mathcal{L}(c)^{-1}r = \bigcup_{u \in \mathcal{L}(c)} u^{-1}\mathcal{L}(r) = \bigcup_{u \in \mathcal{L}(c)} \mathcal{L}(r) = \mathcal{L}(r)$, where the last equality uses $\mathcal{L}(c) \neq \emptyset$.

The following closure properties are useful and easy to verify.

Lemma 5 (Stabilizer Closure). *If c is a stabilizer of r, then c^* is a stabilizer of r. Conversely, if c^* is a stabilizer of r, then c is a stabilizer of r. Similarly, if c_1 and c_2 are stabilizers of r, then $c_1 \sqcup c_2$ is a stabilizer of r. Conversely, if $c_1 \sqcup c_2$ is a stabilizer of r, then c_1 and c_2 are each stabilizers of r.*

Proof.

Kleene star

($\Rightarrow$) Let $w \in \mathcal{L}(c^*)$, so $w = u_1 \cdots u_k$ with each $u_i \in \mathcal{L}(c)$. By definition each $u_i^{-1}\mathcal{L}(r) = \mathcal{L}(r)$, hence $w^{-1}\mathcal{L}(r) = u_k^{-1}(\cdots(u_1^{-1}\mathcal{L}(r))\cdots) = \mathcal{L}(r)$.

($\Leftarrow$) Since $\mathcal{L}(c) \subseteq \mathcal{L}(c^*)$, every $u \in \mathcal{L}(c)$ satisfies $u^{-1}\mathcal{L}(r) = \mathcal{L}(r)$ by assumption.

Union

- (⇒) Every $u \in \mathcal{L}(c_1 \sqcup c_2) = \mathcal{L}(c_1) \cup \mathcal{L}(c_2)$ belongs to $\mathcal{L}(c_1)$ or $\mathcal{L}(c_2)$; in both cases $u^{-1}\mathcal{L}(r) = \mathcal{L}(r)$ by assumption.
- (⇐) Since $\mathcal{L}(c_i) \subseteq \mathcal{L}(c_1 \sqcup c_2)$ for $i \in \{1, 2\}$, every $u \in \mathcal{L}(c_i)$ satisfies $u^{-1}\mathcal{L}(r) = \mathcal{L}(r)$ by assumption.

Lemma 6 (Self-Stabilization). *c^* is a stabilizer of c^* if and only if $\mathcal{L}(c^*)$ is* left-unitary*, i.e. for all $u, v \in \Sigma^*$: $u \in \mathcal{L}(c^*)$ and $uv \in \mathcal{L}(c^*)$ implies $v \in \mathcal{L}(c^*)$.*

Proof.

(⇒) Assume c^* is a stabilizer of c^*, i.e. for every $u \in \mathcal{L}(c^*)$ we have $u^{-1}\mathcal{L}(c^*) = \mathcal{L}(c^*)$. Let $u, v \in \Sigma^*$ with $u \in \mathcal{L}(c^*)$ and $uv \in \mathcal{L}(c^*)$. Since $u \in \mathcal{L}(c^*)$, the stabilizer property gives $u^{-1}\mathcal{L}(c^*) = \mathcal{L}(c^*)$. From $uv \in \mathcal{L}(c^*)$ we obtain $v \in u^{-1}\mathcal{L}(c^*) = \mathcal{L}(c^*)$. Hence $\mathcal{L}(c^*)$ is left-unitary.

(⇐) Assume $\mathcal{L}(c^*)$ is left-unitary. We show $u^{-1}\mathcal{L}(c^*) = \mathcal{L}(c^*)$ for every $u \in \mathcal{L}(c^*)$.

- (⊇) Let $v \in \mathcal{L}(c^*)$. Since $u \in \mathcal{L}(c^*)$ and $v \in \mathcal{L}(c^*)$, concatenation closure of the Kleene star gives $uv \in \mathcal{L}(c^* \cdot c^*) = \mathcal{L}(c^*)$. Hence $v \in u^{-1}\mathcal{L}(c^*)$.
- (⊆) Let $v \in u^{-1}\mathcal{L}(c^*)$, i.e. $uv \in \mathcal{L}(c^*)$. Since $u \in \mathcal{L}(c^*)$ and $uv \in \mathcal{L}(c^*)$, left-unitarity gives $v \in \mathcal{L}(c^*)$.

Clearly, c^* is then also the strongest stabilizer of itself.

The key decomposition argument used by our algorithm is:

Lemma 7 (Stabilizer Decomposition). *Let $xu \in \mathcal{L}(r)$ and let c be a non-nullable stabilizer of r. Then any solution for x can be written as $x = x'x''$ with $x' \in \mathcal{L}(c^*)$ and $x'' \in \mathcal{L}(\overline{c\Sigma^*})$. Moreover $x''u \in \mathcal{L}(r)$.*

Proof. Let σ be a model for the membership constraint. Further, $w := \sigma(x)$ is any concrete value for x and $\hat{u} := \sigma(u)$, so that $w\hat{u} \in \mathcal{L}(r)$. We decompose w greedily from the left: set $w_0 := w$. While $w_i \in \mathcal{L}(c\Sigma^*)$, write $w_i = p_i w_{i+1}$ where p_i is the shortest prefix of w_i in $\mathcal{L}(c)$ which exists since w_i starts with a word in $\mathcal{L}(c)$. Since c is non-nullable, each p_i is non-empty, so $|w_{i+1}| < |w_i|$ and the process terminates after finitely many steps k. At termination, $w_k \notin \mathcal{L}(c\Sigma^*)$, i.e. $w_k \in \mathcal{L}(\overline{c\Sigma^*})$. Set $x' := p_0 p_1 \cdots p_{k-1} \in \mathcal{L}(c^k) \subseteq \mathcal{L}(c^*)$ and $x'' := w_k$. Then $w = x'x''$ as required.

It remains to show $x''\hat{u} \in \mathcal{L}(r)$. Since c is a stabilizer of r, by Lemma 5 c^* is also a stabilizer. As $x' \in \mathcal{L}(c^*)$, the definition of stabilizer gives $(x')^{-1}\mathcal{L}(r) = \mathcal{L}(r)$. Since $x'(x''\hat{u}) = w\hat{u} \in \mathcal{L}(r)$, we get $x''\hat{u} \in (x')^{-1}\mathcal{L}(r) = \mathcal{L}(r)$.

This motivates splitting x into the maximal c^* prefix and the remaining tail that is not in $c\Sigma^*$, ensuring progress. Note that the requirement that c is non-nullable is required, as otherwise $x'' \in \mathcal{L}(\overline{c\Sigma^*})$ would simplify to $\bot$.

4 Annotated Constraints, Overapproximation and Stabilizer Extraction

4.1 Annotations.

To detect cycles during symbolic exploration, we attach history annotations to all constraints. An annotated membership constraint has the form

$$\langle C : u \in r : h \rangle$$

where C is a unique identifier, $u \in \mathcal{T}$ is a string term that should evaluate to something in the language of the regular expression $r \in \mathcal{R}$ that is also used for cycle detection. The $h \in \mathcal{R}$ is a regex recording consumed prefixes.

We maintain a global map $\mathcal{S} : \mathcal{R} \to 2^{\mathcal{R}}$ that records stabilizers for each regex. When we refer to "adding a stabilizer s to $\mathcal{S}(r)$" we mean $\mathcal{S}(r) \leftarrow \mathcal{S}(r) \cup \{s\}$. It is an invariant that the language $\mathcal{L}(\bigsqcup \mathcal{S}(r))$ grows monotonically during our procedure. In particular, it means that backtracking in the search does not retract elements from $\mathcal{S}(r)$. It is important to state that for each regex r there is a maximal stabilizer r' such that for any other stabilizer r'' it holds $\mathcal{L}(r'') \subseteq \mathcal{L}(r')$. The language of r' is unique for r, and $\bigsqcup \mathcal{S}(r)$ will converge against a regular expression representing this language. Additionally, we maintain a flag $\mathcal{F} \subseteq 2^{\mathcal{R}}$ to denote that $\bigsqcup \mathcal{S}(r)$ is the strongest stabilizer of r. Initially, $\mathcal{S}(r) = \emptyset$ for all $r \in \mathcal{R}$ and $\mathcal{F} = \emptyset$.

4.2 The Algorithm

We will now discuss our procedure in detail. The overall approach is sketched in the pseudo-code Algorithm 4.2.

4.2.1 Conflict detection

Given an annotated constraint $\langle C : u \in r : h \rangle$, we are checking if the overapproximation of the constraint is inconsistent. Concretely, let $\Omega(u)$ replace each variable $x \in X$ in u by $\bigsqcap R^x$. That is the intersection of primitive constraints known for x at the current step. If there are none, this simplifies to Σ^*. If we reach the situation where

$$\mathcal{L}(\Omega(u) \sqcap r) = \emptyset,$$

we can backtrack immediately. Note that this trivially also covers cases like

$$\langle C : au \in br : h \rangle$$

for some distinct $a, b \in \Sigma$.

4.2.2 Derivative Step

For a concrete leading character a like in

$$\langle C : au \in r : h \rangle$$

we replace $au \in r$ by $u \in \delta_a(r)$ and update annotations accordingly: history h is extended by a:

$$\langle C : u \in \delta_a(r) : ha \rangle$$

In case $\mathcal{T}(r)$ we add $\delta_a(r)$ to $\mathcal{T}$ and set $\mathcal{S}(\delta_a(r)) \leftarrow \{\delta_a(\bigsqcup \mathcal{S}(\delta_a(r))\ \}$.

Algorithm 1 Main Solving Loop

```
 1: procedure SOLVE(constraints, $\mathcal{S}$)
 2:     for all cnstr in constraints do
 3:         cnstr ← TRYDERIVATIVE(cnstr)                          ▷ see Section 4.2.2
 4:         if cnstr changed then
 5:             $\mathcal{S} \leftarrow \mathcal{S} \cup$ TRYEXTRACTCYCLE(cnstr)              ▷ see Section 4.2.5
 6:             constraints ← STABDECOMP(constraints, $\mathcal{S}$)      ▷ see Section 4.2.3
 7:     for all cnstr in constraints do
 8:         cnstr ← TRYSUBSUMPTION(cnstr)                         ▷ see Section 4.2.4
 9:     if $\forall$cnstr $\in$ constraints primitive then
10:         return CHECKINTERSECTION(constraints)                  ▷ see Lemma 2
11:     for all cnstr in constraints do
12:         if ISOVERAPPROXCONFLICT(cnstr) then
13:             return false                                      ▷ see Section 4.2.1
14:     cnstr ← CHOOSENONPRIMITIVE(constraints)
15:     for all split in NIELSENSPLITS(cnstr) do                  ▷ see Section 4.2.6
16:         if SOLVE(constraints[split], $\mathcal{S}$) then
17:             return true
18:     return false
```

4.2.3 Cycle Decomposition Step Given $\langle C : xu \in r : h\rangle$ and $\mathcal{S}(r) \neq \emptyset$ we perform the following rewrite using $s := \bigsqcup \mathcal{S}(r)$:

1. Substitute $x \mapsto x'x''$ throughout all constraints using fresh x' and $x'' \in X$
2. Add primitive constraints for the new parts as described in Lemma 7:
   - $x' \in s^*$ and
   - $x'' \in \overline{(s \sqcap \overline{\varepsilon})\Sigma^*}$.

We claim that s^* is the strongest stabilizer of itself. Thus, we add $\mathcal{S}(s^*) := \{s^*\}$ to avoid the necessity to analyse again the new regex s^* and potentially diverge, as we introduce more and more regular expressions to the system for which we need to find stabilizers. It is worth pointing out that all solutions for x', which is now leading in the term of C, are stabilizing words of r which will allow us to eliminate it using the following step in Section 4.2.4. Further, the intersection with the complement of ε ensures that s is not nullable and would lead to trivial unsatisfiability.

4.2.4 Cycle Subsumption Step This rule allows us to eliminate leading variables that could otherwise be replicated indefinitely. Let us assume we have a constraint $\langle C : xu \in r : h\rangle$ and $\mathcal{L}(\bigsqcap R^x) \subseteq \mathcal{L}((\bigsqcup \mathcal{S}(r))^*)$. We simplify the constraint to $\langle C : u \in r : h\rangle$. We can do so because $(\bigsqcup \mathcal{S}(r))^*$ is a stabilizer of r (by Invariant 8 and Lemma 5) and $\mathcal{L}(\bigsqcap R^x) \subseteq \mathcal{L}((\bigsqcup \mathcal{S}(r))^*)$: for any model σ, $\sigma(x) \in \mathcal{L}((\bigsqcup \mathcal{S}(r))^*)$ implies $\sigma(x)^{-1}\mathcal{L}(r) = \mathcal{L}(r)$ by definition, so $\sigma(u) \in \mathcal{L}(r)$.

4.2.5 Cycle Detection Step If we encounter after a derivative step the same syntactic constraint identifier C and regex r in some constraint $\langle C : u \in r : h\rangle$

as before $\langle C : u' \in r : h' \rangle$. That means h' is a proper syntactic prefix of h. Let t be the suffix such that $h = h't$. Then t denotes a stabilizer of r. We could add t or t^* into $\mathcal{S}(r)$, but this might not be sufficient to eventually converge in case there are nested loops. We, therefore, construct a stronger stabilizer using the following procedure:

$$\begin{aligned} E_\varepsilon(r) &:= \varepsilon \\ E_{aw}(r) &:= \left(\bigsqcup \mathcal{S}_{\overline{a}}(r)\right)^* a E_w(\delta_a(r)) \\ E'_{aw}(r) &:= a E_w(\delta_a(r)) \end{aligned}$$

where $\mathcal{S}_{\overline{a}}(r)$ refers to the current elements of the set $\mathcal{S}(r)$ that cannot start with character a. We add $(E'_t(r))^*$, which is a stabilizer of r as well, to $\mathcal{S}(r)$. Note that we use the special case E'_t for the first element in order to simplify the constructed regex - there is no theoretical problem in using E_t instead. When $t = a_1 \dots a_n$ we end up with $E'_t(r) = a_1 \left(\bigsqcup \mathcal{S}_{\overline{a_1}}(\delta_{a_1})\right)^* a_2 \dots \left(\bigsqcup \mathcal{S}_{\overline{a_{n-1}}}(\delta_{a_1 \dots a_{n-1}})\right)^* a_n$ being added under a Kleene star to $\mathcal{S}(r)$. Finally, we reset the history value of the constraint: $\langle C : u \in r : h' \rangle$.

4.2.6 Nielsen Step For any membership constraint $\langle C : xu \in r : h \rangle$ with a leading string variable, we can perform a Nielsen-style single-character split using the characters in $M(r)$. We branch on substituting x/ε or x/ax for some $a \in M$. After applying this split, we either eliminate a variable or enable a derivation step, as described in Section 4.2.2.

4.3 Example

We illustrate the stabilizer extraction on a single example

$$\langle C : \ xbx \in (a(b \sqcup c))^* : \varepsilon \rangle, \qquad \Sigma = \{a, b, c\}.$$

Table 1 shows a minimal, deterministic trace that exposes the loop.

From step 3, it follows that the suffix $t = ab$ (and symmetrically $t = ac$) is a stabilizer of $(a(b \sqcup c))^*$: consuming ab restores the residual to the original regex. Therefore, on cycle detection we add t (or t^*, or the widened expression $E_t(r)$) to $\mathcal{S}((a(b \sqcup c))^*)$.

Applying the stabilizer decomposition (StabDecomp) we split x as

$$x = x'x'', \qquad x' \in (ab \sqcup ac)^*, \quad x'' \notin (ab \sqcup ac)\Sigma^*,$$

which collapses the infinite repetition of ab/ac into a finite symbolic representation and permits the exploration to converge.

Note: the table can be extended to show the symmetric trace for ac and to display the exact form of $E_t(r)$ if the stronger stabilizer construction is required.

4.4 Soundness & Completeness & Termination

Invariant 8. *Throughout the execution of the algorithm, every $c \in \mathcal{S}(r)$ is a stabilizer of r.*

Proof. By induction on the sequence of insertions into $\mathcal{S}$. See Appendix A.1 for the full proof.

Lemma 9. *The claim that s^* is the strongest stabilizer of itself made in Section 4.2.3 is correct given our particular definition of $E_t(r)$.*

Proof. TODO

Theorem 1 (Soundness). *The algorithm described in Section 4 is sound. That means, in case it returns **true** for a given set of (annotated) membership constraints, the original set of input constraints is satisfiable, and a model can be extracted from the primitive constraints at the terminal recursive call.*

Proof. Each transformation rule is solution-preserving: any model of the transformed constraint set lifts to a model of the original. The full case analysis is given in Appendix A.2.

Theorem 2 (Completeness). *The algorithm described in Section 4 is complete. That means, if the algorithm returns **false** for a given set of membership constraints, then the input constraints are unsatisfiable.*

Proof. By contrapositive: if the input is satisfiable, any model survives every transformation step and at least one branch returns **true**. The full case analysis is given in Appendix A.3.

Theorem 3 (Termination). *The algorithm described in Section 4 terminates on every input consisting of a finite set of membership constraints over extended regular expressions and a finite alphabet Σ.*

Proof. Finite branching (Nielsen splits), bounded derivative chains, and convergence of stabilizer sets via an SCC induction over the finite Brzozowski automaton together yield termination by König's lemma. The full argument is given in Appendix A.4.

4.5 Notes on Implementation Aspects

Clearly, the described algorithm can perform very poorly in case implemented naively. Branching over Σ or repeated checks for intersection can become very expensive. Hence, a couple of optimizations need to be considered. For example, most operations on the regular expressions are cached or evaluated lazily. String terms (and even some slices of sequences) are perfectly shared. Once an operation like derivation or emptiness checks is performed on this term, it gets cached and can be easily retrieved once required again. As discussed in the previous proofs, the main insight is that the set of relevant regular expressions for some input is finite and so are the relevant subterms. Hence, aggressive caching becomes an obvious choice. Another important optimization is, that we do not actually reason and split on single characters in general, but rather on character classes/minterms. A split on $r = (\Sigma \mid ab)$ on real world Σ, would require several thousands of cases, even though there are only two different choices: a and $\Sigma \setminus \{a\}$. Working only on minterms is a well-known trick in reasoning on regexes and is applicable in our setting as well by tracking minterms, rather than concrete characters in the history of the constraints. The last optimization, we want to mention is that we can often deduce large parts of $\mathcal{S}(r)$ even without detecting a cycle. For example, if $r = r_1^* r_2$ and $M(r_1) \cap M(r_2) = \emptyset$ we can directly add r_1^* to $\mathcal{S}(r)$.

4.6 Notes on Length Constraints

Length constraints are additional arithmetic constraints making use of the length of string terms. As homomorphically $|uv| = |u| + |v|$, $|\varepsilon| = 0$, and $|a| = 1$ for all $a \in \Sigma$, we only care about length terms over single string variables $x \in X$: $|x|$. We do not directly deal with those length constraints during reasoning, but make sure that a separate integer reasoner receives enough information to eventually find a consistent model modulo the additional length constraints. For that, we call a string variable x length-relevant in case it occurs in at least one length constraint or a word equation/regular membership constraint involving a length-relevant variable. Assume we have found a solution for all the membership constraints. That means, for each variable x we have $\neg\,\text{empty}(\bigsqcap_{r \in R^x} r)$. Further, R^x of all length-relevant variables does not contain any complement or intersection operation. We compute the semi-linear set of possible lengths $\ell(r)$ for each $r \in R^x$ and consider their conjunction.

In particular, this means the following.

$$
\begin{aligned}
\ell(\varepsilon) &:= \{0\}\\
\ell(\bot) &:= \{\}\\
\ell(a) &:= \{1\}\\
\ell(r_1 r_2) &:= \{l_1 + l_2 \mid l_1 \in \ell(r_1), l_2 \in \ell(r_2)\}\\
\ell(r_1^*) &:= \{k_1 l_1 + \ldots + k_n l_n \mid k_i \in \mathbb{N}, l_i \in \ell(r_1)\}\\
\ell(r_1 \sqcup r_2) &:= \ell(r_1) \cup \ell(r_2)
\end{aligned}
$$

We can compute a finite representation of $\ell(r)$. From an implementation point of view, we replace every b in r via some fixed a to optimize for caching.

One way of representing for this set is to represent it as a union of a finite set of base lengths plus a finite set of periodic extensions [6]: $\ell(r) = F \cup \bigcup_{i=1}^{n} (P_i + k_i\mathbb{N})$ where F is a finite set of base lengths, and each $d_i + k_i\mathbb{N}$ represents an infinite arithmetic progression of lengths starting at d_i and repeating every k_i.

That means, we add to our solution the side-constraint that $\bigwedge_{r \in R^x} \exists \vec{z} : |x| = \ell(r)$ where $\vec{z}$ to replace the $\mathbb{N}$ in the semi-linear set. In case some of the length-relevant string variables have a primitive constraint $r \in R^x$ that contains a complement or negation, we need to determinize the constraint. Depending on the context we can do this either by continued splitting despite the fact that they are primitive until all blocking operations are eliminated, or by now eagerly constructing an automaton from them.

5 Related Work

Classical foundations are Brzozowski's derivatives and Antimirov's partial derivatives [2,1]. Automata constructions and complexity are covered in standard textbooks [4,5]. The solver-oriented view, SMT integration, and practical handling of regular constraints are discussed in the string-solver literature [3] and related works.

6 Conclusion

We presented a compact, symbolic decision procedure for regular-expression membership constraints that combines symbolic derivatives with an annotated-history mechanism and a deterministic stabilizer extraction policy. The algorithm avoids explicit automata construction and provides a practical avenue for integration into SMT-style solvers. Future work includes empirical evaluation, heuristics for selecting stabilizers to improve convergence speed, and formalised termination/completeness proofs in an appendix or mechanised setting.

References

1. Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
2. Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
3. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
4. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.
5. Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.
6. Anthony Widjaja To. Parikh images of regular languages: Complexity and applications. *CoRR*, abs/1002.1464, 2010.

A Deferred Proofs

A.1 Proof of Invariant 8

Proof. Elements are added to $\mathcal{S}(r)$ only by the cycle detection step (Section 4.2.5). We proceed by induction on the sequence of insertions into $\mathcal{S}$: at each insertion, we assume that all previously inserted elements are stabilizers of their respective regexes (induction hypothesis) and show that the newly inserted element is a stabilizer as well. We further assume throughout that all syntactic simplifications applied to derivatives are language-preserving, so that syntactic equality of a regex after a derivative cycle implies semantic equality.

Suppose we detect a cycle for constraint identifier C with regex r: we have an earlier annotated constraint $\langle C : u' \in r : h' \rangle$ and a current one $\langle C : u \in r : h \rangle$ with $h = h't$. The history records that starting from r, consuming $t = a_1 \dots a_n$ via Brzozowski derivatives brings us back to r, i.e. $\delta_t(r) \equiv r$. The element added to $\mathcal{S}(r)$ is $E'_t(r)$ as constructed in Section 4.2.5, satisfying $\mathcal{L}(t) \subseteq \mathcal{L}(E'_t(r))$. At each intermediate derivative state $r_i := \delta_{a_1 \dots a_i}(r)$, the construction inserts $(\bigsqcup \mathcal{S}(r_i))^*$. By the induction hypothesis every element of $\mathcal{S}(r_i)$ is a stabilizer of r_i, and by Lemma 5 so is $(\bigsqcup \mathcal{S}(r_i))^*$.

We show $w^{-1}\mathcal{L}(r) = \mathcal{L}(r)$ for every $w \in \mathcal{L}(E'_t(r))$. Each such w has the form $a_1 u_1 a_2 u_2 \cdots u_{n-1} a_n$ with $u_i \in \mathcal{L}((\bigsqcup \mathcal{S}(r_i))^*)$. Since $(\bigsqcup \mathcal{S}(r_i))^*$ is a stabilizer of r_i, the definition gives $u_i^{-1}\mathcal{L}(r_i) = \mathcal{L}(r_i)$ for each u_i, so consuming u_i from r_i preserves the language. Consuming a_{i+1} then yields r_{i+1}, and after all of w we arrive at $w^{-1}\mathcal{L}(r) = \mathcal{L}(r)$.

Hence $w^{-1}\mathcal{L}(r) = \mathcal{L}(r)$ for every $w \in \mathcal{L}(E'_t(r))$, so $E'_t(r)$ is a stabilizer of r by definition. By Lemma 5, $E'_t(r)^*$ is a stabilizer as well.

A.2 Proof of Theorem 1 (Soundness)

Proof. The algorithm returns **true** only in line 12 via CheckIntersection when all constraints are primitive, or in line 20 when a recursive call on a split sub-problem returns **true**. We show that each transformation rule applied during the procedure is *solution-preserving*: any model σ of the transformed constraint set can be lifted to a model σ' of the constraint set before the transformation. Soundness then follows by induction on the sequence of rule applications from the terminal (primitive) state back to the initial constraints.

(1) Derivative step (Section 4.2.2). The constraint $\langle C : au \in r : h \rangle$ is replaced by $\langle C : u \in \delta_a(r) : ha \rangle$. By the fundamental property of Brzozowski derivatives, for any word w:

$$aw \in \mathcal{L}(r) \iff w \in \mathcal{L}(\delta_a(r)).$$

Hence $\sigma(u) \in \mathcal{L}(\delta_a(r))$ iff $a\,\sigma(u) \in \mathcal{L}(r)$. The transformation is an equivalence; in particular, every model of the new constraint is a model of the old one.

(2) Nielsen split (Section 4.2.6). For $\langle C : xu \in r : h \rangle$, we branch on $x \mapsto \varepsilon$ and $x \mapsto ax'$ for each $a \in M(r)$ (with x' fresh).

- Branch $x \mapsto \varepsilon$: If σ is a model of the substituted system, define $\sigma' := \sigma[x \mapsto \varepsilon]$, which satisfies $\sigma'(xu) = \sigma'(u) = \sigma(u) \in \mathcal{L}(r)$.
- Branch $x \mapsto ax'$: If σ is a model, define $\sigma' := \sigma[x \mapsto a\,\sigma(x')]$. Then $\sigma'(xu) = a\,\sigma(x')\sigma(u)$, and since σ satisfies the substituted constraint $ax'u \in r$, we have $\sigma'(xu) \in \mathcal{L}(r)$.

In both cases, the model lifts. Note that we do not need $M(r) = \Sigma$ for soundness; $M(r)$ merely determines which branches are explored. Any model found in a branch is valid regardless of whether other branches were omitted.

(3) Stabilizer decomposition (Section 4.2.3). Let $b := \bigsqcup \mathcal{S}(r)$. By Invariant 8, b is a stabilizer of r (using closure under union, Lemma 5). The constraint $\langle C : xu \in r : h \rangle$ is replaced by the system where $x = x'x''$ with fresh variables x', x'' subject to $x' \in b^*$ and $x'' \in \overline{(b \sqcap \overline{\varepsilon})\Sigma^*}$.

If σ is a model of the decomposed system, define σ' by $\sigma'(x) := \sigma(x')\sigma(x'')$ and $\sigma'(y) := \sigma(y)$ for all other variables. Since the decomposed system includes the constraint $x'x''u \in r$ (obtained by substituting $x \mapsto x'x''$) and σ satisfies it, we directly obtain $\sigma'(x)\sigma'(u) = \sigma(x')\sigma(x'')\sigma(u) \in \mathcal{L}(r)$.

(4) Subsumption (Section 4.2.4). The constraint $\langle C : xu \in r : h \rangle$ is simplified to $\langle C : u \in r : h \rangle$ when $\mathcal{L}(\bigsqcap R^x) \subseteq \mathcal{L}((\bigsqcup \mathcal{S}(r))^*)$.

Suppose σ is a model of the simplified system, so $\sigma(u) \in \mathcal{L}(r)$. In the original system, $\sigma(x) \in \mathcal{L}(\bigsqcap R^x)$ (since σ must satisfy all primitive constraints on x), hence $\sigma(x) \in \mathcal{L}((\bigsqcup \mathcal{S}(r))^*)$. By Invariant 8 and Lemma 5, $(\bigsqcup \mathcal{S}(r))^*$ is a stabilizer of r. By Lemma 3, $\mathcal{L}((\bigsqcup \mathcal{S}(r))^* \cdot r) \subseteq \mathcal{L}(r)$, so $\sigma(x)\sigma(u) \in \mathcal{L}(r)$. Thus σ is also a model of the original constraint $xu \in r$.

(5) Cycle detection (Section 4.2.5). This step adds elements to $\mathcal{S}(r)$ and resets the history annotation. Neither operation modifies the semantic content of the constraint set: $\mathcal{S}$ is auxiliary bookkeeping whose correctness is established by Invariant 8, and the history h is used only for cycle detection, not for defining membership. Hence, any model before the step is a model after, and vice versa.

(6) Conflict detection (Section 4.2.1). This step only returns **false** (causing backtracking); it never produces a model. Therefore, it does not affect soundness.

(7) CheckIntersection. When all constraints are primitive, the algorithm returns **true** iff $\mathcal{L}(\bigsqcap_{r \in R^x} r) \neq \emptyset$ for all $x \in X$ (Lemma 2). A model is obtained by selecting, for each x, any word from $\mathcal{L}(\bigsqcap_{r \in R^x} r)$. By Lemma 2, this assignment satisfies all primitive constraints.

Combining (1)–(7): if the algorithm returns **true**, there exists a model σ at the terminal primitive state. By applying the lifting arguments (1)–(5) in reverse along the derivation trace, σ can be extended to a model σ' of the original input constraints.

A.3 Proof of Theorem 2 (Completeness)

Proof. Equivalently, we show the contrapositive: if the input constraints are satisfiable, the algorithm does not return **false**. Let σ be a model of the original input constraints. We show that σ (or a suitable extension thereof) survives every step of the algorithm, i.e. at each point in the execution, there exists at least one branch whose constraint set is satisfied by a model derived from σ. Since the algorithm returns **false** only when all branches are exhausted, this implies that some branch must return **true**.

We argue for each transformation rule.

(1) Derivative step (Section 4.2.2). As shown in the soundness proof, this is an equivalence transformation: $a\,\sigma(u) \in \mathcal{L}(r) \iff \sigma(u) \in \mathcal{L}(\delta_a(r))$. Hence, if σ satisfies the constraint before the step, it satisfies the constraint after.

(2) Nielsen split (Section 4.2.6). For $\langle C : xu \in r : h\rangle$, the algorithm branches on $x \mapsto \varepsilon$ and $x \mapsto ax'$ for each $a \in M(r)$. Since σ satisfies $xu \in r$, we have $\sigma(x)\sigma(u) \in \mathcal{L}(r)$.

- If $\sigma(x) = \varepsilon$, then $\sigma(u) \in \mathcal{L}(r)$ and the branch $x \mapsto \varepsilon$ is satisfied by σ.
- If $\sigma(x) = aw$ for some $a \in \Sigma$ and $w \in \Sigma^*$, then $a\,w\,\sigma(u) \in \mathcal{L}(r)$, so $\mathcal{L}(\delta_a(r)) \neq \emptyset$. By structural induction on r one verifies $\{a \in \Sigma \mid \mathcal{L}(\delta_a(r)) \neq \emptyset\} \subseteq M(r)$, hence $a \in M(r)$ and the branch $x \mapsto ax'$ exists. The model $\sigma' := \sigma[x' \mapsto w]$ satisfies the substituted constraint $ax'u \in r$.

In either case, at least one branch is satisfiable, so the algorithm does not return **false** at this point.

(3) Stabilizer decomposition (Section 4.2.3). Let $b := \bigsqcup \mathcal{S}(r)$ and consider $\langle C : xu \in r : h\rangle$ with $\sigma(x)\sigma(u) \in \mathcal{L}(r)$. Let $c := b \sqcap \overline{\varepsilon}$. Then c is non-nullable by construction. Since $\mathcal{L}(c^*) = \mathcal{L}(b^*)$ (removing ε from the base does not change the Kleene closure) and b^* is a stabilizer of r (Lemma 5), c^* is a stabilizer of r as well. By the converse direction of Lemma 5, c is a stabilizer of r. By Lemma 7 (applied with this c), we can decompose $\sigma(x) = w'w''$ with $w' \in \mathcal{L}(c^*) = \mathcal{L}(b^*)$ and $w'' \in \mathcal{L}(\overline{c\Sigma^*}) = \mathcal{L}(\overline{(b \sqcap \overline{\varepsilon})\Sigma^*})$. Moreover, $w''\sigma(u) \in \mathcal{L}(r)$ (by the lemma).

Define $\sigma' := \sigma[x' \mapsto w',\ x'' \mapsto w'']$. Then:

- $\sigma'(x') = w' \in \mathcal{L}(b^*)$, satisfying the primitive constraint on x'.
- $\sigma'(x'') = w'' \in \mathcal{L}(\overline{(b \sqcap \overline{\varepsilon})\Sigma^*})$, satisfying the primitive constraint on x''.
- $\sigma'(x'x''u) = w'w''\sigma(u) = \sigma(x)\sigma(u) \in \mathcal{L}(r)$, satisfying the decomposed constraint.

All other constraints are satisfied since x is replaced by $x'x''$ everywhere and $\sigma'(x'x'') = \sigma(x)$. Hence the decomposed system is satisfiable.

(4) Subsumption (Section 4.2.4). The constraint $\langle C : xu \in r : h\rangle$ is simplified to $\langle C : u \in r : h\rangle$ when $\mathcal{L}(\bigsqcap R^x) \subseteq \mathcal{L}((\bigsqcup \mathcal{S}(r))^*)$. We must show that if σ satisfies the original system (including $\sigma(x)\sigma(u) \in \mathcal{L}(r)$), then σ also satisfies $\sigma(u) \in \mathcal{L}(r)$.

Since σ satisfies all primitive constraints on x, we have $\sigma(x) \in \mathcal{L}(\bigsqcap R^x) \subseteq \mathcal{L}((\bigsqcup \mathcal{S}(r))^*)$. By Invariant 8 and Lemma 5, $(\bigsqcup \mathcal{S}(r))^*$ is a stabilizer of r, so $\sigma(x)^{-1}\mathcal{L}(r) = \mathcal{L}(r)$ by definition. Since $\sigma(x)\sigma(u) \in \mathcal{L}(r)$, we obtain $\sigma(u) \in \sigma(x)^{-1}\mathcal{L}(r) = \mathcal{L}(r)$. Hence $\sigma(u) \in \mathcal{L}(r)$ and the simplified constraint is satisfied.

(5) Cycle detection (Section 4.2.5). This step only adds elements to $\mathcal{S}(r)$ and resets a history annotation. The semantic constraints remain unchanged, so σ continues to satisfy them.

(6) Conflict detection (Section 4.2.1). The algorithm returns **false** for a constraint $\langle C : u \in r : h\rangle$ only if $\mathcal{L}(\Omega(u) \sqcap r) = \emptyset$. Recall that $\Omega(u)$ replaces each variable x by $\bigsqcap R^x$, which is a superset of the actual solution space. If σ satisfies all constraints, then $\sigma(x) \in \mathcal{L}(\bigsqcap R^x)$ for every x, so $\sigma(u) \in \mathcal{L}(\Omega(u))$. Since also $\sigma(u) \in \mathcal{L}(r)$ (by assumption), we have $\sigma(u) \in \mathcal{L}(\Omega(u) \sqcap r) \neq \emptyset$. Hence the conflict check cannot trigger when a model exists; the algorithm does not erroneously return **false**.

(7) CheckIntersection. When all constraints are primitive, the algorithm returns $\bigwedge_{x \in X}(\mathcal{L}(\bigsqcap_{r \in R^x} r) \neq \emptyset)$ (Lemma 2). If a model σ exists, then $\sigma(x) \in \mathcal{L}(\bigsqcap_{r \in R^x} r)$ for every x, so each intersection is non-empty and CheckIntersection returns **true**.

Combining (1)–(7): if σ is a model of the original constraints, then at every step of the algorithm, there exists at least one branch whose constraint set is satisfied by a model derived from σ. The algorithm returns **false** only when *all* branches return **false**, which, by the above, cannot happen when a model exists. Therefore, if the algorithm returns **false**, the input constraints are unsatisfiable.

A.4 Proof of Theorem 3 (Termination)

Proof. We show: (i) the branching factor at each node is finite, and (ii) every path in the search tree is finite. Termination then follows by König's lemma.

Finite branching. The only branching point is the Nielsen split, which produces at most $|M(r)| + 1 \leq |\Sigma| + 1$ branches (one per character in $M(r) \subseteq \Sigma$, plus $x \mapsto \varepsilon$). This is finite since Σ is finite.

Finitely many derivative states. For a fixed extended regular expression r_0 over finite Σ, the Brzozowski derivatives yield finitely many equivalence classes $D(r_0)$ modulo language equality (this is a classical result; see [2]).[1] Write $Q := \bigcup_{r_0 \in R_0} D(r_0)$ for the finite set of all derivative states reachable from the input regexes R_0. The regex component of any annotated constraint always belongs to Q.

Bounded derivative chains between cycle detections. The history h in an annotated constraint $\langle C : u \in r : h \rangle$ grows by one character per derivative step. Since r ranges over the finite set Q, after at most $|Q|$ consecutive derivative steps (without an intervening history reset) the same regex must appear twice (by the pigeonhole principle), triggering cycle detection and a history reset. Hence between any two consecutive history resets (or from the start to the first reset), at most $|Q|$ derivative steps occur.

Convergence of stabilizer sets (fixpoint argument). We now give the central argument showing that the stabilizer sets $\mathcal{S}(q)$ converge after finitely

[1] Syntactically, derivatives may produce infinitely many expressions, but modulo ACI (associativity, commutativity, idempotency of $\sqcup/\sqcap$) and standard simplifications ($\bot r = \bot$, $\varepsilon r = r$, etc.), only finitely many equivalence classes arise. We assume throughout that the implementation applies these simplifications, so that syntactic equality coincides with language equality on the derivative state-space.

many cycle detections. Consider the Brzozowski automaton $\mathcal{A}_{r_0}$ with state set Q and transition function δ.

For each state $q \in Q$ define the *maximal stabilizer language*

$$L_q \ := \ \{w \in \Sigma^* \mid \delta_w(q) = q\},$$

i.e. the set of all words that label loops at q in $\mathcal{A}_{r_0}$. Each L_q is a regular language (it is accepted by $\mathcal{A}_{r_0}$ with q as both initial and sole accepting state).

For each state q define the *accumulated stabilizer language* $\widehat{\mathcal{S}}(q) := \mathcal{L}(\bigsqcup \mathcal{S}(q))$. By Invariant 8, $\widehat{\mathcal{S}}(q) \subseteq L_q$ throughout the execution. The algorithm only adds to $\mathcal{S}$ (never removes), so the tuple $(\widehat{\mathcal{S}}(q))_{q\in Q}$ grows monotonically with respect to componentwise language inclusion.

Claim: After finitely many cycle detections, $\widehat{\mathcal{S}}(q)^* = L_q$ for every $q \in Q$.

Step 1: Structure of L_q via path languages. For states $q, q' \in Q$ define the *path language*

$$T_{q,q'} \ := \ \{w \in \Sigma^* \mid \delta_w(q) = q'\},$$

i.e. the set of all words labelling a path from q to q' in $\mathcal{A}_{r_0}$. Then $L_q = T_{q,q}$. These path languages satisfy the system of equations

$$T_{q,q'} \ = \ [\![q = q']\!] \ \cup \bigcup_{\substack{a\in\Sigma \\ \delta_a(q)=q''}} \{a\} \cdot T_{q'',q'}, \qquad \text{for each } q, q' \in Q, \tag{2}$$

where $[\![q = q']\!] = \{\varepsilon\}$ if $q = q'$ and $\emptyset$ otherwise. This is a standard right-linear system over $|Q|^2$ variables whose unique solution is exactly the path languages of $\mathcal{A}_{r_0}$ (cf. [4], Chapter 3). In particular, $L_q = T_{q,q}$ is characterised by

$$L_q \ = \ \{\varepsilon\} \ \cup \bigcup_{\substack{a\in\Sigma \\ \delta_a(q)=q'}} \{a\} \cdot T_{q',q}. \tag{3}$$

Step 2: Widening and strict growth. When a cycle $t = a_1 \cdots a_m$ is detected at state q (meaning $\delta_t(q) = q$), the algorithm adds $E'_t(q)$ to $\mathcal{S}(q)$ where

$$\mathcal{L}(E'_t(q)) \ = \ \{a_1\} \cdot \widehat{\mathcal{S}}(q_1)^* \cdot \{a_2\} \cdot \widehat{\mathcal{S}}(q_2)^* \cdots \widehat{\mathcal{S}}(q_{m-1})^* \cdot \{a_m\}$$

with $q_i := \delta_{a_1\cdots a_i}(q)$. In particular, $t \in \mathcal{L}(E'_t(q))$, so $\widehat{\mathcal{S}}(q)$ strictly grows with each cycle detection (the detected cycle word t is new—otherwise the anti-stabilizer constraint $x'' \in \overline{(\bigsqcup \mathcal{S}(q) \sqcap \overline{\varepsilon})\Sigma^*}$ would have blocked exploration along the first character of t at state q, preventing t from being produced by the Nielsen/derivative sequence).

Step 3: Finitely many cycle detections. Every cycle detected at state q yields a word t with $\delta_t(q) = q$ and $|t| \leq |Q|$ (since cycles are detected via pigeonhole on the finite state set Q). As argued in Step 2, each detected cycle word t satisfies $t \notin \widehat{\mathcal{S}}_{\text{old}}(q)^*$. The set of candidate cycle words at any state q is contained in

$\{w \in \Sigma^* \mid \delta_w(q) = q,\ |w| \leq |Q|\}$, which is finite (bounded by $|\Sigma|^{|Q|}$). Since each cycle detection adds t to $\widehat{\mathcal{S}}_{\text{new}}(q)^*$ (via $E'_t(q)$, which contains t), and $\widehat{\mathcal{S}}(q)^*$ grows monotonically, the same word t can never be detected twice at the same state. Therefore, the total number of cycle detections across all states is bounded by $\sum_{q \in Q} |\{w \in \Sigma^* \mid \delta_w(q) = q,\ |w| \leq |Q|\}| \leq |Q| \cdot |\Sigma|^{|Q|}$, which is finite.

Step 4: Convergence implies $\widehat{\mathcal{S}}(q)^* = L_q$. After all cycle detections have ceased (the algorithm has reached a fixpoint with respect to $\mathcal{S}$), we show $\widehat{\mathcal{S}}(q)^* = L_q$ for all $q \in Q$. The proof proceeds by induction on the strongly connected component (SCC) structure of the Brzozowski automaton $\mathcal{A}_{r_0}$, processing SCCs in reverse topological order (leaves first).

Let G be the directed graph on Q induced by the transition function δ, and let $S_1, S_2, \ldots, S_n$ be its SCCs in reverse topological order (so that if there is an edge from S_i to S_j with $i \neq j$, then $j < i$).

Base case (singleton SCCs with no self-loops): If q is in a singleton SCC with no self-loop, then $L_q = \{\varepsilon\}$ and $\widehat{\mathcal{S}}(q)^* \supseteq \{\varepsilon\}$ trivially, so $\widehat{\mathcal{S}}(q)^* = L_q$.

Inductive case: Consider an SCC S_i and assume $\widehat{\mathcal{S}}(q')^* = L_{q'}$ for all q' in SCCs S_j with $j < i$ (i.e. all SCCs reachable from S_i except S_i itself). Let $q \in S_i$ and suppose for contradiction that $\widehat{\mathcal{S}}(q)^* \subsetneq L_q$. Then there exists a word $w \in L_q \setminus \widehat{\mathcal{S}}(q)^*$. Among all such words, choose one of minimal length, say $w = a_1 \cdots a_m$ with $\delta_w(q) = q$.

We claim that no proper non-empty prefix of w is a loop at q. Suppose otherwise: $w = w_1 w_2$ with $w_1 \in L_q \setminus \{\varepsilon\}$ and $w_2 \in L_q \setminus \{\varepsilon\}$. By minimality of $|w|$, both $w_1 \in \widehat{\mathcal{S}}(q)^*$ and $w_2 \in \widehat{\mathcal{S}}(q)^*$, hence $w = w_1 w_2 \in \widehat{\mathcal{S}}(q)^*$—contradicting $w \notin \widehat{\mathcal{S}}(q)^*$. So w visits q only at the beginning and end; in particular $|w| \leq |Q|$ (the intermediate states are pairwise distinct from q, and distinct from each other within S_i or belong to earlier SCCs).

Now consider the intermediate states $q_1 = \delta_{a_1}(q), \ldots, q_{m-1} = \delta_{a_1 \cdots a_{m-1}}(q)$, all distinct from q. For each q_k that lies in an SCC S_j with $j < i$, we have $\widehat{\mathcal{S}}(q_k)^* = L_{q_k}$ by the induction hypothesis. For each $q_k \in S_i \setminus \{q\}$, consider any word $v \in \widehat{\mathcal{S}}(q_k) \setminus \{\varepsilon\}$: by definition v is a loop at q_k, so v stays within the SCC S_i (and possibly visits earlier SCCs). These intermediate stabilizers are used by the widening construction.

Since $w \notin \widehat{\mathcal{S}}(q)^*$ and w does not begin with any non-empty word in $\widehat{\mathcal{S}}(q)$ (every non-empty word in $\widehat{\mathcal{S}}(q)$ is a loop at q, but w does not revisit q until the end), the anti-stabilizer constraint $x'' \in \overline{(\bigsqcup \mathcal{S}(q) \sqcap \overline{\varepsilon})\Sigma^*}$ does not block the exploration along w: the constraint blocks values starting with a word in $\widehat{\mathcal{S}}(q) \setminus \{\varepsilon\}$, but a_1 does not begin any such word (since such a word would be a loop at q, and the first return of the w-path to q is at position m). Hence the Nielsen branch $x'' \mapsto a_1 x'''$ is not blocked, and continuing along $a_2, \ldots, a_m$ via successive Nielsen splits and derivative steps, the exploration reaches q again after $m \leq |Q|$ steps. This triggers a cycle detection—contradicting the assumption that no further cycle detections occur.

We still need to verify that after this cycle detection, w would actually enter $\widehat{\mathcal{S}}(q)^*$. The widening construction adds $E'_w(q)$ to $\mathcal{S}(q)$, where

$$\mathcal{L}(E'_w(q)) \;=\; \{a_1\} \cdot \widehat{\mathcal{S}}(q_1)^* \cdot \{a_2\} \cdot \widehat{\mathcal{S}}(q_2)^* \cdots \widehat{\mathcal{S}}(q_{m-1})^* \cdot \{a_m\}.$$

Since $\varepsilon \in \widehat{\mathcal{S}}(q_k)^*$ for each intermediate state q_k, we have $w = a_1 \cdots a_m \in \mathcal{L}(E'_w(q))$. Hence after the insertion, $w \in \widehat{\mathcal{S}}_{\text{new}}(q) \subseteq \widehat{\mathcal{S}}_{\text{new}}(q)^*$, confirming that the cycle detection resolves the gap. This strengthens the contradiction: not only does a new cycle detection occur, but it necessarily covers w.

Since the contradiction holds for every state q in S_i under the induction hypothesis, we conclude $\widehat{\mathcal{S}}(q)^* = L_q$ for all $q \in S_i$. By induction over all SCCs, $\widehat{\mathcal{S}}(q)^* = L_q$ for all $q \in Q$.

Termination of each path after convergence. Once $\widehat{\mathcal{S}}(q)^* = L_q$ for all $q \in Q$, consider a constraint $\langle C : x''u \in q : h\rangle$ where x'' carries the anti-stabilizer constraint $x'' \in \overline{(\bigsqcup \mathcal{S}(q) \sqcap \overline{\varepsilon})\Sigma^*}$. Since $\widehat{\mathcal{S}}(q)^* = L_q$, the constraint $x'' \in \overline{(\bigsqcup \mathcal{S}(q) \sqcap \overline{\varepsilon})\Sigma^*}$ means that $\sigma(x'')$ cannot begin with a non-empty word in L_q, i.e. no non-empty prefix of $\sigma(x'')$ is a loop at q. Consider any Nielsen branch $x'' \mapsto ax'''$: the derivative moves to state $q' = \delta_a(q)$. Along any continuation from q', within at most $|Q| - 1$ further derivative steps the state either reaches $\bot$ (the branch closes by conflict detection) or revisits some state. If it revisits q, then the characters consumed so far form a loop at q, contradicting the anti-stabilizer constraint on x'' (via conflict detection, since the overapproximation Ω detects that the prefix of x'' consumed so far must lie in L_q, violating $x'' \in \overline{(\bigsqcup \mathcal{S}(q) \sqcap \overline{\varepsilon})\Sigma^*}$). If it revisits a state $q'' \neq q$, a cycle detection fires at q''; but since $\widehat{\mathcal{S}}(q'')^* = L_{q''}$, the resulting stabilizer decomposition and subsumption immediately eliminate the leading variable from the constraint at q''. Hence, on every branch, the exploration either closes (conflict/emptiness) or reduces the number of leading variable occurrences within $|Q|$ steps.

Progress via stabilizer decomposition and variable counting. We define the following well-founded measure on constraint sets to show that each path is finite. Let V denote the multiset of leading variable occurrences across all non-primitive constraints. The Nielsen branch $x \mapsto \varepsilon$ strictly decreases $|V|$ (removes a leading variable without adding one). The Nielsen branch $x \mapsto ax'$ replaces x by x' but enables a derivative step; the subsequent cycle-detection/decomposition/subsumption pipeline either:

1. closes the branch (conflict or $\bot$), or
2. performs a stabilizer decomposition splitting x' into x_1x_2 followed by subsumption of x_1, leaving x_2 as the new leading variable with a strictly stronger anti-stabilizer constraint (since $\widehat{\mathcal{S}}$ has grown).

After convergence of all $\widehat{\mathcal{S}}(q)$, case (2) always leads to termination within $|Q|$ steps as argued above. Before convergence, case (2) triggers a cycle detection, and by Step 3 there are at most $|Q| \cdot |\Sigma|^{|Q|}$ cycle detections in total. Hence only finitely many iterations of case (2) can occur before convergence is reached.

Once $|V| = 0$ (all constraints are primitive), the algorithm decides satisfiability via CheckIntersection in a single step.

Combining all parts: every path in the search tree is finite (bounded by the finite number of cycle detections and the decrease of $|V|$), and the branching factor at each node is finite ($\leq |\Sigma| + 1$). By König's lemma, the search tree is finite, so the algorithm terminates.

Branch	Constraint	Comment
$\square$	$\langle C_1 : xbx \in (a(b \sqcup c))^* : \varepsilon \rangle$	Initial state
1	$\langle C_1 : b \in (a(b \sqcup c))^* : \varepsilon \rangle$	Nielsen Step: $x \mapsto \varepsilon$
1	$\langle C_1 : b \in (a(b \sqcup c))^* : \varepsilon \rangle$	Conflict: $b \sqcap (a(b \sqcup c))^*$
2	$\langle C_1 : axbax \in (a(b \sqcup c))^* : \varepsilon \rangle$	Nielsen Step: $x \mapsto ax$
2	$\langle C_1 : xbax \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Derivative Step δ_a
2.1	$\langle C_1 : ba \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Nielsen Step: $x \mapsto \varepsilon$
2.1	$\langle C_1 : ba \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Conflict: $ba \sqcap ((b \sqcup c)(a(b \sqcup c))^*)$
2.2	$\langle C_1 : bxbabx \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Nielsen Step $x \mapsto bx$
2.2	$\langle C_1 : xbabx \in (a(b \sqcup c))^* : ab \rangle$	Derivative Step δ_b
2.2	$\langle C_1 : xbabx \in (a(b \sqcup c))^* : \varepsilon \rangle$	Adding $(ab)^*$ to $\mathcal{S}((a(b \sqcup c))^*)$
	$\langle C_1 : x'x''babx'x'' \in (a(b \sqcup c))^* : \varepsilon \rangle$	Applying $x/x'x''$
2.2	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	Applying prefix cnstr
	$\langle C_3 : x'' \in \overline{((ab)^* \sqcap \overline{\varepsilon})\Sigma^*} : \varepsilon \rangle$	Applying postfix cnstr
	$\langle C_1 : x''babx'x'' \in (a(b \sqcup c))^* : \varepsilon \rangle$	
2.2	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	Leading x' subsumed
	$\langle C_3 : x'' \in \overline{((ab)^* \sqcap \overline{\varepsilon})\Sigma^*} : \varepsilon \rangle$	
	$\langle C_1 : babx' \in (a(b \sqcup c))^* : \varepsilon \rangle$	
2.2.1	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	Nielsen Step $x'' \mapsto \varepsilon$
	$\langle C_3 : \varepsilon \in \overline{((ab)^* \sqcap \overline{\varepsilon})\Sigma^*} : \varepsilon \rangle$	
	$\langle C_1 : babx' \in (a(b \sqcup c))^* : \varepsilon \rangle$	
2.2.1	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	Conflict $(bab\Sigma^*) \sqcap (a(b \sqcup c))^*$
	$\langle C_3 : \varepsilon \in \overline{((ab)^* \sqcap \overline{\varepsilon})\Sigma^*} : \varepsilon \rangle$	
	$\langle C_1 : ax''babx'ax'' \in (a(b \sqcup c))^* : \varepsilon \rangle$	
2.2.2	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	Nielsen Step $x'' \mapsto ax''$
	$\langle C_3 : ax'' \in \overline{((ab)^* \sqcap \overline{\varepsilon})\Sigma^*} : \varepsilon \rangle$	
	$\langle C_1 : x''babx'ax'' \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Derivative Step δ_a
2.2.2	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	
	$\langle C_3 : x'' \in \overline{b(ab)^*\Sigma^*} : a \rangle$	Derivative Step δ_a
	$\langle C_1 : babx'a \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Nielsen Step $x'' \mapsto \varepsilon$
2.2.2.1	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	
	$\langle C_3 : \varepsilon \in \overline{b(ab)^*\Sigma^*} : a \rangle$	Derivative Step δ_a
	$\langle C_1 : babx'a \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Conflict: $(bab\Sigma^*a) \sqcap ((b \sqcup c)(a(b \sqcup c))^*)$
2.2.2.1	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	
	$\langle C_3 : \varepsilon \in \overline{b(ab)^*\Sigma^*} : a \rangle$	
	$\langle C_1 : bx''babx'abx'' \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Nielsen Step $x'' \mapsto bx''$
2.2.2.2	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	
	$\langle C_3 : bx'' \in \overline{b(ab)^*\Sigma^*} : a \rangle$	
	$\langle C_1 : bx''babx'abx'' \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	
2.2.2.2	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	
	$\langle C_3 : bx'' \in \overline{b(ab)^*\Sigma^*} : a \rangle$	Conflict: $(b\Sigma^*) \sqcap \overline{b(ab)^*\Sigma^*}$
	$\langle C_1 : cx''babx'acx'' \in (b \sqcup c)(a(b \sqcup c))^* : a \rangle$	Nielsen Step $x'' \mapsto cx''$
2.2.2.3	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	
	$\langle C_3 : cx'' \in \overline{b(ab)^*\Sigma^*} : a \rangle$	
	$\langle C_1 : x''babx'acx'' \in (a(b \sqcup c))^* : ac \rangle$	Derivative Step δ_c
2.2.2.3	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	
	$\langle C_3 : x'' \in \Sigma^* : ac \rangle$	Derivative Step δ_c
	$\langle C_1 : x'''x''''babx'acx'''x'''' \in (a(b \sqcup c))^* : ac \rangle$	Adding $(ab \sqcup ac)^*$ to $\mathcal{S}((a(b \sqcup c))^*)$
2.2.2.3	$\langle C_2 : x' \in (ab)^* : \varepsilon \rangle$	
	$\langle C_4 : x''' \in (ab \sqcup ac)^* : \varepsilon \rangle$	
	$\langle C_5 : x'''' \in \overline{((ab \sqcup ac)^* \sqcap \overline{\varepsilon})\Sigma^*} : \varepsilon \rangle$	
etc.	...	...

Table 1. Compact derivative trace for $xbx \in (a(b \sqcup c))^*$.