EREQ: Regular Expressions with Quantifiers and Incremental Quantifier Elimination

EKATERINA ZHUCHKO, Tallinn University of Technology, Estonia
IAN ERIK VARATALU, Tallinn University of Technology, Estonia
MARGUS VEANES, Microsoft Research, USA
NIKOLAJ BJØRNER, Microsoft Research, USA

Weak monadic second-order logic (**wMSO**) is a foundational tool for specifying regular properties. Traditional decision procedures for this logic typically translate **wMSO** formulas into finite automata. Although the logic is decidable, this approach incurs non-elementary complexity in the worst-case. Nearly thirty years ago, the state-of-the-art MONA tool showed that, despite these theoretical limits, **wMSO** can be decided efficiently in practice through carefully optimized automata constructions. We revisit **wMSO** from an algebraic perspective by introducing Extended Regular Expressions with Quantifiers (**EREQ**). Instead of relying on automata determinization, **EREQ** employs symbolic derivatives to perform incremental quantifier elimination, providing a compositional and symbolic alternative to classical automata-based approaches.

We present a linear-time translation of **wMSO** into **EREQ** and a derivative-based decision procedure for **EREQ**. We prove the correctness of the translation and of the derivative construction in the Lean proof assistant. We implement our approach in Rust and evaluate it on a set of established MONA benchmarks, demonstrating competitive performance with state-of-the-art tools. Our results demonstrate the potential of derivative-based methods, opening new avenues for efficient decision procedures in **EREQ**.

CCS Concepts: • **Theory of computation → Regular languages**; • **Computing methodologies → Boolean algebra algorithms**.

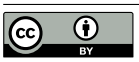
Additional Key Words and Phrases: regex, derivative, automata, MSO

ACM Reference Format:
Ekaterina Zhuchko, Ian Erik Varatalu, Margus Veanes, and Nikolaj Bjørner. 2026. EREQ: Regular Expressions with Quantifiers and Incremental Quantifier Elimination. *Proc. ACM Program. Lang.* 10, PLDI, Article 271 (June 2026), 24 pages. https://doi.org/10.1145/3808349

1 Introduction

We introduce a formal language of *regular expressions with quantifiers*, **EREQ**, over finite words with the formal expressivity of weak monadic second order logic or **wMSO** modulo alphabet theories $\mathcal{A}$ over an alphabet Σ. This work builds on the notion of *symbolic derivatives* [Stanford et al. 2021] $\delta(R)$ for regular expressions R in **ERE** that **EREQ** extends with quantifiers over Boolean variables or propositions. Crucially for our quest, *δ distributes over quantifiers*, formally $\delta(\exists R) = \exists\delta(R)$. Since δ distributes also over complement, it follows that $\delta(\neg\exists\neg R) = \neg\exists\neg\delta(R)$, i.e., $\delta(\forall R) = \forall\delta(R)$, where all operators propagate lazily. We use **EREQ** as a target language for translating all formulas φ in **wMSO** via a direct *linear* embedding $\ulcorner\varphi\urcorner$ into **EREQ**.

Authors' Contact Information: Ekaterina Zhuchko, Tallinn University of Technology, Tallinn, Estonia, ekzhuc@taltech.ee; Ian Erik Varatalu, Tallinn University of Technology, Tallinn, Estonia, ian.varatalu@taltech.ee; Margus Veanes, Microsoft Research, Redmond, USA, margus@microsoft.com; Nikolaj Bjørner, Microsoft Research, Redmond, USA, nbjorner@microsoft.com.



ACM 2475-1421/2026/6-ART271
https://doi.org/10.1145/3808349

Our motivation behind **EREQ** originates from a concrete (industrial) application related to trace analysis of web services where Σ is a service specific set of `http`-request/response pairs called *actions*. A typical example is to specify a *causal relationship* that some action specified by $\alpha \in \mathcal{A}$, is *necessary* before another action specified by $\beta \in \mathcal{A}$, can happen, e.g., in a *soft-delete* protocol of a storage service, a resource can be restored (β) only if it was previously deleted (α). In this case, the **wMSO** formula $\psi = \forall x_0(\beta(x_0) \Rightarrow \exists x_1(x_1 < x_0 \wedge \alpha(x_1)))$, where all the x_i range over positions of actions, is a direct formalization. At the same time, *concatenation* is a practical tool for expressing *sequential composition*, e.g., φ *happens before* ψ is the regex $\top*\cdot\ulcorner\varphi\urcorner\cdot\top*\cdot\ulcorner\psi\urcorner\cdot\top*$ In some cases it is more natural and efficient to express patterns directly in **ERE**. E.g., α *happens immediately before* β is more directly expressed by $\top*\cdot\alpha\cdot\beta\cdot\top*$ rather than $\ulcorner\exists x_0(\exists x_1(x_1 = x_0 + 1 \wedge \alpha(x_0) \wedge \beta(x_1)))\urcorner$.

The core advantages of working with **EREQ** rather than **wMSO** directly are summarized under the following main points.

Incremental Search: The *symbolic derivative function* δ of **EREQ** makes it possible to lazily unwind regexes without any explicit automata constructions, modulo any decidable theory over characters and propositions.

The symbolic derivative $\delta(R)$ is a *transition regex* (*t-regex*): a nested if-then-else term whose leaves are all the possible states (regexes) reachable by one step from R. Given $\delta^n(R)$ as the union of all states reachable by $\leq n$ steps from R, it follows that R accepts a word of length $\leq n$ iff $\delta^n(R)$ is nullable. Thus, *classical non-emptiness*: **wMSO** $\rightsquigarrow$ automaton-non-emptiness [Henriksen et al. 1995]; *our approach*: **wMSO** $\rightsquigarrow$ **EREQ** where φ is satisfiable iff $\delta^n(\ulcorner\varphi\urcorner)$ is nullable for some n.

Enhanced Expressivity: Concatenation and loops in **EREQ** are the most fundamental building blocks of regexes. They enable a direct way to express *sequential composition* $\ulcorner\varphi\urcorner\cdot\ulcorner\psi\urcorner$, *loops* $\ulcorner\varphi\urcorner*$, and *bounded loops* $\ulcorner\varphi\urcorner\{m\}$, not directly available in **wMSO**.

Use of a general translation of $R{\cdot}S$ into an equivalent formula $\varphi_{R\cdot S}$ in **wMSO** is a theoretical construction that requires several nested layers of quantifiers [Thomas 1996] already for standard regexes. As far as we know, $\varphi{\cdot}\psi$ is not supported in any variant of **wMSO**. However, by using our Theorem 1, concatenation (and loops) can be supported natively via the embedding into **EREQ** as $\ulcorner\varphi{\cdot}\psi\urcorner \stackrel{\text{DEF}}{=} \ulcorner\varphi\urcorner\cdot\ulcorner\psi\urcorner$ – at least for the case of *M2L-str semantics* of **wMSO** that corresponds to the built-in semantics of **EREQ** as the standard semantics of regular languages

Algebraic Laws: Working with symbolic derivatives, that propagate top-down, maintains the algebraic laws of **EREQ** in t-regexes. In contrast, automata-based algorithms that rely on building automata from formulas bottom-up, introduce a two-level representation obscuring uniform formula manipulation.

The key new laws of **EREQ** are related to $\exists$. The main law, that seems counterintuitive at first glance, is that $\exists$ distributes over derivatives, concatenation as well as $\vee$, i.e., for all $R, S \in$ **EREQ**, $\exists p(R{\cdot}S) \equiv \exists p(R){\cdot}\exists p(S)$ even though p can occur free in both R and S. This implies also the law $\exists p(R*) \equiv \exists p(R)*$. In that sense, $\cdot$ behaves similarly to $\vee$ in relation to $\exists$.

Search Strategies: A range of *normal forms* are available for regex simplifications to implement different *search strategies*.

Leaves of t-regexes can be maintained in *negation normal form* (NNF) as well as *anti-prenex normal form* where $\exists$ is distributed aggressively over $\cdot$ and $\vee$. One search strategy is to use (top-level) *disjunctive normal form* (DNF) that lifts $\vee$ to the top level in leaves L of t-regexes, say $L = (R_1 \vee R_2) \wedge S$ where, rather than treating L as a single state in the search space, $R_1 \wedge S$ and $R_2 \wedge S$ are treated as two separate states. In some situations this strategy may backfire, by hiding rewrites directly applicable to $R \wedge S$ in $\neg(R \wedge S)$ but not directly applicable to $\neg R \vee \neg S$.

In order to enable rewrites that can distribute $\exists$ it is favorable to apply rewrites that can eliminate top-level $\wedge$ and $\neg$ from R in $\exists(R)$. A typical situation is that $\neg(\top*\cdot p\cdot\top*)$ rewrites to $\overline{p}*$ (where $\overline{p}$ is the Boolean negation of p). Then $\exists p(\overline{p}*)$ rewrites to $\exists p(\overline{p})*$ and finally to $\top*$. Such rewrites often have a cascading effect of enabling further rewrites in the rest of the process.

Contributions. Our main contributions are as follows.

(1) **EREQ**: a novel logic that conservatively extends **ERE** with quantifiers, matching the expressive power of **wMSO** while retaining native support for concatenation, Kleene star, and bounded loops. We define a *symbolic derivative function* δ for **EREQ** and prove that it distributes over all operators, including $\exists$. This enables a derivative-based decision procedure for **EREQ** (and hence **wMSO**) via incremental quantifier elimination, without explicit automata construction.
(2) We provide a complete formalization in Lean 4, comprising approximately 4100 lines of code: the semantics of **wMSO** and **EREQ**, correctness of the **wMSO**-to-**EREQ** embedding (Theorem 1), correctness of the derivative rules (Theorem 2), and finiteness of the derivative closure (Theorem 3), the latter building on [Zhuchko et al. 2025].
(3) We implement the decision procedure in Rust and evaluate it against MONA on the full AutomatArk M2L-str benchmark suite (~10 000 instances), demonstrating competitive and in some cases superior performance, particularly on counter and loop families.
(4) Beyond **wMSO**, **EREQ** also enhances expressivity by allowing position-dependent properties to be combined directly with regex operators such as concatenation and loops. We demonstrate the practical benefits of this combination through a motivating industrial application in trace analysis (Section 3).

Synopsis. We begin with an overview of the related work in Section 2, followed by the motivating application from trace analysis of web services (Section 3) that drove the design of **EREQ**. We then present the necessary preliminaries (Section 4), the syntax and semantics of **EREQ** (Section 5), and the linear embedding of **wMSO** into **EREQ** with a proof of correctness in Lean (Section 6). In Section 6.2 we show also that **EREQ** differs fundamentally from **wMSO** from the point of view of computational complexity of its core fragments – while **wMSO** is directly embedded into **EREQ**, the opposite direction is nontrivial and does not preserve the same computational complexity. Next, we develop the derivative theory for **EREQ**, which is also formalized in Lean (Section 7). Section 8 describes the implementation in Rust and evaluation over an existing set of MONA benchmarks, along with additional micro benchmarks to illustrate the impact of search strategies. Finally, Section 9 outlines directions for future work.

2 Related Work

The "weak" in weak monadic second-order logic (**wMSO**) refers to the restriction that second-order variables range over *finite* sets of positions, rather than arbitrary sets as in full MSO. There are two standard interpretations of **wMSO** over strings. In *M2L-str* (*M*onadic *2*nd-order *L*ogic on *str*ings) semantics, formulas are interpreted over finite strings: variables denote sets of positions within the string, and positions beyond the end of the string do not exist. This interpretation coincides with the classical semantics of regular languages: the set of satisfying models of a formula is a regular language over the extended alphabet Σ_n, where n is the number of additional bits extending the core alphabet Σ. In *WS1S* (*W*eak monadic *S*econd-order theory of *1 S*uccessor) semantics, formulas are instead interpreted over the natural numbers $\mathbb{N}$, where second-order variables denote finite subsets of $\mathbb{N}$. Every model can be encoded as an infinite word over Σ_n that eventually stabilizes to the all-$\mathbf{0}$ padding symbol $\mathbf{0}$. As a consequence, WS1S requires that the language of every formula be closed under arbitrary right-padding, and complement must account for this closure. For example,

the formula $\forall x(a(x) \vee b(x))$ does not hold in WS1S over $\Sigma = \{a, b\}$ because it fails at positions beyond the string that carry no label in Σ, which contradicts classical derivative laws, where every position x is assumed to have a label in Σ. M2L-str avoids this issue: it aligns with standard regular language semantics, allows concatenation and Kleene star to retain their classical meaning, and supports clean lifting of derivative laws. **EREQ** is defined as a conservative extension of **ERE** and adopts M2L-str semantics for these reasons (see Section 5.3 for the formal definition).

[Owens et al. 2009] revisited Brzozowski's derivatives [Brzozowski 1964], showing that derivative-based unfolding of regular expressions enables practical simplifications using the algebraic laws of regular languages. The same idea has since appeared in several related domains, including linear temporal logic [Esparza et al. 2020] and ω-regular languages [Veanes et al. 2025].

[Traytel and Nipkow 2013] were the first to use derivatives to obtain a decision procedure for classical **wMSO** over finite Σ, formalizing their approach in the Isabelle proof assistant. They translate formulas into Π-extended regular expressions, where the projection operator Π encodes existential quantification. The language semantics is $\mathcal{L}_n(\Pi r) = (\text{map } \pi) \circ \mathcal{L}_{n+1}(r)$ and the derivative rule for projection is $D_b(\Pi r) = \Pi\big(\bigvee_{c \in \pi^{-1}(b)} D_c(r)\big)$ where $b \in \Sigma_n$ and $c \in \Sigma_{n+1}$. Handling existential quantification requires several modifications to the standard derivative construction. Because Π transforms the underlying language of the expression, the alphabet is parameterized by n, the number of free variables in the translated **wMSO** formula. The function $\pi : \Sigma_{n+1} \to \Sigma_n$ projects away the extra variable introduced by the quantifier and the inverse operation is defined as: $\pi^{-1}(b) = \{c \in \Sigma_{n+1} \mid \pi(c) = b\}$.

In the derivative rule, the union $\bigvee_{c \in \pi^{-1}(b)}$ iterates over *all* concrete characters in the preimage of b, computing a separate derivative $D_c(r)$ for each one and combining the results via alternation. As a concrete example, suppose $(\Sigma_0) = \Sigma = \{a, b\}$ and there is one quantified bit (n=1), so $\Sigma_1 = \{a, b\} \times \{0, 1\}$. Then $\pi^{-1}(a) = \{\langle a, 0\rangle, \langle a, 1\rangle\}$, and the derivative of $\Pi(r)$ with respect to a enumerates both elements: $D_a(\Pi r) = \Pi(D_{\langle a,0\rangle}(r) \vee D_{\langle a,1\rangle}(r))$. With k quantified bits, $|\pi^{-1}(b)| = 2^k$, so each derivative step requires 2^k concrete derivatives. For a symbolic alphabet $\mathcal{A}$ with $|\Sigma|$ potentially large or infinite, the enumeration becomes $|\Sigma| \times 2^k$. Since the union explicitly enumerates over a finite set, we cannot directly extend this to symbolic alphabets which may be infinite in general. While one could reduce this strategy to the classical case by enumerating all minterms, this approach incurs exponential blow-up in the worst case. In contrast, **EREQ** avoids all enumeration: the symbolic derivative $\delta(\exists(R)) = \exists(\delta(R))$ propagates $\exists$ through the derivative in a single step, regardless of alphabet size or number of bits. This illustrates a broader theme: in symbolic settings, it can be more effective to devise new techniques to solve the problem instead of relying on explicit enumeration over the alphabet [D'Antoni and Veanes 2021]. Subsequent work by [Traytel 2015] defined derivatives directly on **wMSO** formulas, avoiding the translation to regexes altogether.

Traytel and Nipkow present decision procedures for both M2L-str and WS1S semantics. They first introduce WS1S semantics, and then extend it to the M2L-str setting via relativization, which restricts quantified variables to finite domains. The authors argue in favor of WS1S semantics for its generality - any M2L-str formula can be translated to WS1S but not conversely. However, the derivative construction for WS1S is significantly more complex since under WS1S semantics, the language of a formula must be closed under arbitrary right paddings with $\mathbf{0}^n$. To account for this, they introduce an alternative definition of derivative that derives from the right of the string instead of the standard left-to-right direction.

In an orthogonal line of work, symbolic automata [Veanes et al. 2010], that lift classical algorithms to modulo theories, have also been studied in the context of **wMSO** [D'Antoni and Veanes 2017] based on M2L-str semantics, where symbolic automata algorithms, including product, complementation and minimization play a major role. The concept of symbolic derivatives [Stanford et al. 2021]

showed that it is possible to lazily propagate *complement*, which was a key breakthrough because it avoids up-front automata constructions for determinization. At the same time it fundamentally depends on the alphabet being viewed symbolically by predicates rather than as a finite alphabet. This view is grounded in *t-regexes* that are originally if-then-else (ITE) terms in SMT where any ITE term $\neg(\alpha\ ?\ R_{then} : R_{else})$ is equivalent to $(\alpha\ ?\ \neg R_{then} : \neg R_{else})$, where $\alpha \in \mathcal{A}$ is some condition and $\neg$ is regular expression complement. For this principle to hold, the language semantics of **EREQ** must obey classical Boolean laws where the state space consists of regexes. In contrast, MONA uses 3-valued logic internally, see [Klarlund et al. 2002, Section 4.6], for efficiency reasons that are also related to M2L-str support as a restriction of WS1S.

Although MONA's approach is rooted in using deterministic automata as the underlying representation, subsequent works have explored decision procedures for WS1S based on nondeterministic automata, leveraging antichain-based techniques to avoid explicit determinization [Fiedor et al. 2015, 2017a]. Further optimizations such as anti-prenexing, which pushes quantifiers deeper into formulas, have been shown to significantly improve performance [Fiedor et al. 2017a]. Another toolset for solving classic **wMSO** was discussed in [Kelb et al. 1997]. Finally, Gaston [Fiedor et al. 2017b] also targets WS1S/WS2S using an automata-based approach with antichains. Since its semantics differ from M2L-str, the two approaches are not directly comparable.

3 Motivating Application: Learning from Service Traces

This section describes the industrial application that originally motivated the development of **EREQ**. The work began with the use of **ERE** and symbolic derivatives [Stanford et al. 2021] for trace analysis of a major cloud storage service, whose correctness across multiple subprotocols is validated using model-based testing. However, **ERE** cannot naturally express *position-dependent* properties, such as when an action at one position causally depends on another action occurring earlier in the trace. Expressing such dependencies in **ERE** requires explicitly encoding position tracking, which quickly becomes impractical. This limitation motivated the introduction of quantifiers, leading to **EREQ**.

As input, we are given a log of concrete traces of a web service that is mapped into a set T of abstract traces over an abstract alphabet Σ. T itself is represented as a regex that summarizes all observed action sequences; it is typically large (thousands of nodes) but loop-free. The abstraction from concrete HTTP requests (and responses) to Σ is a crucial preprocessing step that is orthogonal to the learning algorithm itself. The aim is to learn patterns from T in the form of regexes that explain temporal dependencies between various HTTP requests (and responses), primarily in the form of *enabling* and *disabling* conditions. These learned patterns can in turn be unfolded into an automaton that reveals (often undocumented) abstract protocol behavior of the underlying service.

As a concrete example we illustrate the ideas using a small subset of the operations supported in a public cloud storage system. Here $\Sigma = \{\text{PUT}, \text{DEL}, \text{UNDEL}\}$ represents *actions* related to a fixed resource that can be updated with PUT, deleted with DEL, and undeleted with UNDEL. More generally, the actions have parameters that we omit here. In particular, every successful delete request is followed by a response containing an identifier subsequently needed as a parameter of UNDEL.

We work with **wMSO** formulas φ through the encoding $\ulcorner\varphi\urcorner$ in **EREQ**. To learn a condition φ in **wMSO** is to decide if $\ulcorner\varphi\urcorner$ *subsumes* T, i.e., if $\mathcal{L}(T) \subseteq \mathcal{L}(\ulcorner\varphi\urcorner)$, which reduces to emptiness of $\mathcal{L}(T \wedge \neg\ulcorner\varphi\urcorner)$. Properties φ are generated exhaustively for combinations of action predicates and then tested against T. Theoretically, such learning assumes that T is exhaustive (up to some depth). In practice, the process is necessarily best-effort since the ground-truth trace set T is incomplete.

Enabling Conditions. A key objective is to learn which *enabling* conditions hold. Let $\alpha, \beta \in \mathcal{A}$. Then α *enables* β, say $\alpha \rightsquigarrow \beta$, is defined via **wMSO** as $\ulcorner\forall x_0(\beta(x_0) \Rightarrow \exists x_1(x_1 < x_0 \wedge \alpha(x_1)))\urcorner$. For example, PUT$\rightsquigarrow$DEL and DEL$\rightsquigarrow$UNDEL. This property requires quantification over positions: stating that *every* occurrence of β is preceded by *some* occurrence of α cannot be expressed directly in

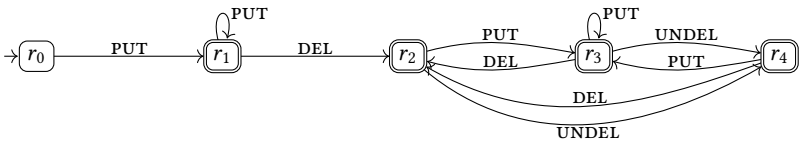


Fig. 1. Unfolding of $r_0 = \text{PUT}\cdot\top* \wedge \text{PUT}{\rightsquigarrow}\text{DEL} \wedge \text{DEL}{\rightsquigarrow}\text{UNDEL} \wedge \text{DEL}{\twoheadrightarrow}\text{DEL} \wedge \text{UNDEL}{\twoheadrightarrow}\text{UNDEL}$.

ERE without manually encoding position tracking. The formula $\alpha{\rightsquigarrow}\beta$ demonstrates why the step from **ERE** to **EREQ** was necessary in this application.

Disabling Conditions. Another objective is to learn which actions disable which other actions (in the next step). Let $\alpha, \gamma \in \mathcal{A}$. Then α *disables next* γ, say $\alpha{\twoheadrightarrow}\gamma$, can be defined via **wMSO** as $\ulcorner\forall x_0(\forall x_1((\alpha(x_0) \wedge x_1 = x_0 + 1) \Rightarrow \overline{\gamma}(x_1)))\urcorner$. In this case a direct equivalent formulation in **ERE** is more compact as the regex $\neg(\top*\cdot\alpha\cdot\gamma\cdot\top*)$. Above, $\text{DEL}{\twoheadrightarrow}\text{DEL}$, and $\text{UNDEL}{\twoheadrightarrow}\text{UNDEL}$.

Combined Conditions. Intersecting learned conditions into a single regex can be used to reveal various aspects of the underlying service as an automaton. Many patterns for concepts, such as *happens before* and *starts with*, like $\text{PUT}\cdot\top*$, use concatenation or are directly in **ERE**. See Figure 1. The automaton shows how the learned conditions constrain the set of valid action sequences. Note that the automaton is *underspecified* without additional conditions — for instance, without adding further enabling or ordering constraints, certain transitions may be overly permissive.

4 Preliminaries

An *Effective Boolean Algebra (EBA)* over an alphabet Σ is a tuple $(\Sigma, \mathcal{A}, \vDash, \bot, \top, \sqcup, \sqcap, \bar{\ })$ where $\mathcal{A}$ is a set of *predicates* that is closed under the Boolean connectives and contains $\bot$ and $\top$. For $a \in \Sigma$ and $\alpha \in \mathcal{A}$ the *models* relation $a \vDash \alpha$ with $[\![\alpha]\!] \stackrel{\text{DEF}}{=} \{a \in \Sigma \mid a \vDash \alpha\}$ obeys classical Tarski laws such that $[\![\bot]\!] = \emptyset$ and $[\![\top]\!] = \Sigma$. For $\alpha, \beta \in \mathcal{A}$ let $\alpha \equiv \beta \stackrel{\text{DEF}}{=} [\![\alpha]\!] = [\![\beta]\!]$. If $\alpha \not\equiv \bot$ then α is *satisfiable* (**SAT**(α)). All the connectives must be *computable* and $\vDash$ must be *decidable*.

For $\alpha, \beta \in \mathcal{A}$ let $\alpha \rightarrow \beta \stackrel{\text{DEF}}{=} \overline{\alpha} \sqcup \beta$ and $\alpha \leftrightarrow \beta \stackrel{\text{DEF}}{=} (\alpha \rightarrow \beta) \sqcap (\beta \rightarrow \alpha)$.

We overload $\mathcal{A}$ to also stand for the EBA $(\Sigma, \mathcal{A}, \vDash, \bot, \top, \sqcup, \sqcap, \bar{\ })$ and say that $\mathcal{A}$ is *decidable* if satisfiability in $\mathcal{A}$ is decidable. We use $\mathcal{A}$ as a given *label alphabet* EBA.

Let $\mathbb{N}$ be the set of natural numbers and let $\mathbb{B} = \{\mathbf{1}, \mathbf{0}\}$ denote basic Boolean truth values, where $\mathbf{1}$ stands for *true* and $\mathbf{0}$ for *false*. Given a word $w \in \Sigma^*$ and position $i \in \mathbb{N}$ such that $i < |w|$, then $w(i) \in \Sigma$ denotes the symbol or character in position i of w. We view $N \in \mathbb{N}$ as *ordinals*, i.e., $N = \{n \mid n < N\}$, e.g., 2^N is the power set of N. So $n < N$ and $n \in N$ have the same meaning.

Given a *rank* $N \in \mathbb{N}$, we extend $\mathcal{A}$ into an EBA $\mathcal{A}_N$ over the alphabet $\Sigma_N \stackrel{\text{DEF}}{=} \Sigma \times \mathbb{B}^N$, such that $\mathcal{A}_N$ contains *propositions* $P = \{\mathsf{p}_i \mid i < N\}$ in addition to all the predicates in $\mathcal{A}$. We let $\Sigma_0 \stackrel{\text{DEF}}{=} \Sigma$, i.e., we consider $\langle a, \epsilon\rangle = a$. A value $u \in \mathbb{B}^N$ represents a truth assignment for P. For completeness of definitions, so that $u(i)$ is well-defined for all $u \in \mathbb{B}^*$ and $i \in \mathbb{N}$, let $u(i) \stackrel{\text{DEF}}{=} \mathbf{0}$ for all $i \geq |u|$.

The models relation $\vDash$ in $\mathcal{A}_N$ is defined for all $\langle a, u\rangle \in \Sigma_N$, propositions p_i, and $\alpha \in \mathcal{A}$, and their Boolean combinations. We overload the operations of $\mathcal{A}$ in $\mathcal{A}_N$.

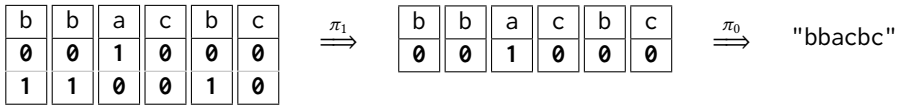
$$\langle a, u\rangle \vDash \mathsf{p}_i \stackrel{\text{DEF}}{=} u(i) = \mathbf{1} \qquad \langle a, u\rangle \vDash \alpha \stackrel{\text{DEF}}{=} a \vDash \alpha$$

For $c = \langle a, u\rangle \in \Sigma_N$ and $b \in \mathbb{B}$ let $c.b \stackrel{\text{DEF}}{=} \langle a, ub\rangle \in \Sigma_{N+1}$. In other words, $c.b$ appends b as bit N to c. Define the *projection* functions $\pi_N : \Sigma_{N+1} \rightarrow \Sigma_N$, $\pi_\Sigma : \Sigma \times \mathbb{B}^* \rightarrow \Sigma$, and $\pi_\mathbb{B} : \Sigma \times \mathbb{B}^* \rightarrow \mathbb{B}^*$:

$$\pi_N(c.b) \stackrel{\text{DEF}}{=} c \qquad \pi_\Sigma(\langle a, u\rangle) \stackrel{\text{DEF}}{=} a \qquad \pi_\mathbb{B}(\langle a, u\rangle) \stackrel{\text{DEF}}{=} u$$

and lift the definitions to words. We write π for π_N when N is clear from the context, i.e., if $c \in \Sigma_N$ and $b \in \mathbb{B}$ then $\pi(c.b)$ stands for $\pi_N(c.b)$. Characters in Σ_N are typically depicted as individual

columns in a table. For example,



4.1 The Implicit Pair Encodings in Σ_N^*

Each word in Σ_N^* uniquely encodes a pair (w, θ) where $w \in \Sigma^*$ and θ maps N to subsets of positions in w. This is essentially the standard mapping used between regular languages over the alphabet Σ_N and M2L-str semantics of **wMSO**. The encoding is denoted here by $\lceil w, \theta \rceil$ and is *bijective*.

$$\forall k < |w|\colon \lceil w, \theta \rceil(k) \stackrel{\text{DEF}}{=} \langle w(k), \lceil \theta, k \rceil \rangle \qquad \text{where} \qquad \forall i < N\colon \lceil \theta, k \rceil(i) \stackrel{\text{DEF}}{=} \begin{cases} \mathbf{1}, \textbf{ if } k \in \theta(i); \\ \mathbf{0}, \textbf{ otherwise}. \end{cases}$$

In that sense, $\lceil w, \theta \rceil$ is just an alternative equivalent representation of the pair (w, θ). In the formal semantics, the word $\lceil w, \theta \rceil$ in **EREQ** corresponds to the pair (w, θ) in **wMSO**. In the projection example depicted above, $\pi_1(\lceil \texttt{"bbacbc"}, [\{2\}, \{0, 1, 4\}] \rceil) = \lceil \texttt{"bbacbc"}, [\{2\}] \rceil$. The following lemma relates the encoding to π and is used in Theorem 1. It follows directly from the definitions.

Lemma 1. *Let $w \in \Sigma^*$, $\theta : N \to 2^{|w|}$, and $S \in 2^{|w|}$. Then $\pi(\lceil w, \theta{\cdot}S \rceil) = \lceil w, \theta \rceil$.*

4.2 Weak Monadic Second Order Logic

The full class of **wMSO** formulas (over finite strings without end marker) modulo $\mathcal{A}$ is defined by the following abstract grammar by φ. Here we formulate the logic without explicit use of first-order variables by instead using only second-order variables $\{X_i \mid i \in \mathbb{N}\}$ and the predicate $|X_i|{=}1$ as a primitive construct meaning that X_i denotes a *singleton* set. Let $\alpha \in \mathcal{A}$.

$$\varphi \quad ::= \quad \varphi_1 \vee \varphi_2 \quad | \quad \varphi_1 \wedge \varphi_2 \quad | \quad \neg\varphi \quad | \quad \exists X_i(\varphi) \quad | \quad X_i <_\exists X_j \quad | \quad X_i \subseteq X_j \quad | \quad \alpha(X_i) \quad | \quad |X_i|{=}1$$

The main additional defined constructs are *universal quantification* $\forall X_i(\varphi) \stackrel{\text{DEF}}{=} \neg\exists X_i(\neg\varphi)$, *language implication* $\varphi \Rightarrow \psi \stackrel{\text{DEF}}{=} \neg\varphi \vee \psi$, and *language equivalence* $\varphi \Leftrightarrow \psi \stackrel{\text{DEF}}{=} (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$. Moreover, $X_i = X_j$ stands for $X_i \subseteq X_j \wedge X_j \subseteq X_i$. A formula is *closed* if it has no free variables else *open*.

4.2.1 Semantics of **wMSO**. The semantics of φ is based on *judgements* of the form $w, \theta \models \varphi$ where $w \in \Sigma^*$ and, for all free variables X_i in φ, θ maps i to a subset of $|w|$. For $S \subseteq |w|$ let

$$\theta[i \mapsto S](j) \stackrel{\text{DEF}}{=} \textbf{if } i = j \textbf{ then } S \textbf{ else } \theta(j)$$

Limiting S to subsets of $|w|$ is commonly known as *M2L-str* semantics, compared to *WS1S* semantics where S may range over arbitrary finite subsets of $\mathbb{N}$. The rationale behind adopting M2L-str semantics here in favor of WS1S semantics is justified below in the context of **EREQ** whose semantics is defined as a conservative extension of the language semantics of **ERE**.

$$\begin{array}{rcl@{\qquad}rcl} w, \theta \models |X_i|{=}1 & \stackrel{\text{DEF}}{=} & |\theta(i)| = 1 & w, \theta \models X_i <_\exists X_j & \stackrel{\text{DEF}}{=} & \exists k \in \theta(i), l \in \theta(j) : k < l \\ w, \theta \models X_i \subseteq X_j & \stackrel{\text{DEF}}{=} & \theta(i) \subseteq \theta(j) & w, \theta \models \alpha(X_i) & \stackrel{\text{DEF}}{=} & \exists k \in \theta(i) : w(k) \vdash \alpha \\ w, \theta \models \neg\varphi & \stackrel{\text{DEF}}{=} & w, \theta \not\models \varphi & w, \theta \models \exists X_i(\varphi) & \stackrel{\text{DEF}}{=} & \exists S \subseteq |w| : w, \theta[i \mapsto S] \models \varphi \\ \text{(for closed } \varphi) \; w \models \varphi & \stackrel{\text{DEF}}{=} & w, \epsilon \models \varphi & w, \theta \models \varphi_1 \wedge \varphi_2 & \stackrel{\text{DEF}}{=} & w, \theta \models \varphi_1 \textbf{ ∧ } w, \theta \models \varphi_2 \\ & & & w, \theta \models \varphi_1 \vee \varphi_2 & \stackrel{\text{DEF}}{=} & w, \theta \models \varphi_1 \textbf{ ∨ } w, \theta \models \varphi_2 \end{array}$$

In the formulas $\alpha(X_i)$ and $X_i <_\exists X_j$ the assumption classically is that X_i and X_j are *first-order* (*always* denote singleton sets) but *we do not enforce this in the semantics*. In particular, $X_i <_\exists X_i$ intuitively means that $|X_i| > 1$. Note therefore that $\neg(X_i <_\exists X_j)$ and $X_i = X_j \vee X_j <_\exists X_i$ are equivalent *only when X_i and X_j are singletons*.

4.2.2 Additional Primitives. The formulas $X_i <_\forall X_j$ and $\alpha[X_i]$ are additional primitives that have *universal* interpretation in **wMSO** over positions. They can equivalently be used instead of $X_i <_\exists X_j$ and $\alpha(X_i)$ in any context where $|X_i|{=}1$ and $|X_j|{=}1$ hold. Their semantics is:

$$w, \theta \models X_i <_\forall X_j \stackrel{\text{DEF}}{=} \forall k \in \theta(i), l \in \theta(j) : k < l \qquad w, \theta \models \alpha[X_i] \stackrel{\text{DEF}}{=} \forall k \in \theta(i) : w(k) \vDash \alpha$$

They have slightly more succinct embeddings into **EREQ** than the existential ones.

4.2.3 Normalized Representation. In a *normalized* form of a closed formula φ, all variables X_i use *de Bruijn levels* as indices i (also known as *forward de Bruijn indices*). In this case, we sometimes write $\exists\varphi$ where the variable binding is implicit.

4.2.4 First-Order Variable Encoding. If X_i is intended to be *first-order* (*always* singleton) the **wMSO** formula has to explicitly ensure that using $|X_i|{=}1$ as an additional conjunct. We write x_i for implicit singleton set semantics of X_i. In particular this means that if $\varphi(x_i)$ contains x_i as a free variable, then $\varphi(x_i)$ stands for $\varphi(X_i) \wedge |X_i|{=}1$. Observe then that $\exists x_i(\varphi(x_i))$ stands for $\exists X_i(\varphi(X_i) \wedge |X_i|{=}1)$ and thus $\forall x_i(\varphi(x_i))$ stands for $\neg\exists X_i(\neg\varphi(X_i) \wedge |X_i|{=}1)$. Common first-order formulas include:

- $x_i \in X_j$ that stands for $X_i \subseteq X_j \wedge |X_i|{=}1$;
- $x_i < x_j$ that stands for $X_i <_\exists X_j \wedge |X_i|{=}1 \wedge |X_j|{=}1$.

These are orthogonal to **wMSO** itself that does not include first-order variables explicitly in our definition. However, the translation into **EREQ** treats first-order variables with specialized rules where the singleton semantics is embedded more directly, in order to improve efficiency of the representation of symbolic derivatives and enable more regex rewrite rules in **EREQ**.

5 Extended Regular Expressions with Quantifiers

We introduce extended regular expressions with *quantifiers* as the class **EREQ** defined by the grammar below, whose members are denoted by R. All operators appear in order of precedence where $\vee$ (union) binds weakest and $\neg$ (complement) strongest. Let $\beta \in \mathcal{A}_N$ and $m > 0$.

$$R \quad ::= \quad \beta \mid \varepsilon \mid R_1{\vee}R_2 \mid R_1{\wedge}R_2 \mid R_1{\cdot}R_2 \mid R\{m\} \mid R{*} \mid \neg R \mid \exists \mathsf{p}_i(R)$$

The intuitive meaning of p_i in terms of **wMSO** semantics is that, given the *current position*, say k, in the given word, then $k \in X_i \Leftrightarrow \mathsf{p}_i = \mathbf{1}$ at k. Dually, $\overline{\mathsf{p}_i}$ means that $k \notin X_i \Leftrightarrow \mathsf{p}_i = \mathbf{0}$ at k. For example, the regex $(\overline{\mathsf{p}_i} \vee \mathsf{p}_j){*}$ states that, for all positions k of the given word, either $k \notin X_i$ or $k \in X_j$, i.e., $X_i \subseteq X_j$. In contrast, $\neg$ is classical regex language complement. As we show later, $\overline{\mathsf{p}_i}$ is equivalent with $\neg\mathsf{p}_i \wedge \top$ in **EREQ**, where $\top \in \mathcal{A}$ denotes all words of length one. The regex denoting *nothing* is the predicate $\bot \in \mathcal{A}$. We also let $\bigstar \stackrel{\text{DEF}}{=} \top{*}$ that is the top element of **EREQ**. We write $R{+}$ for $R{\cdot}R{*}$. *Concatenation* $(\cdot)$ is often implicit by juxtaposition. Let $R\{0\} \stackrel{\text{DEF}}{=} \varepsilon$.

$\exists \mathsf{p}_i(R)$ is also called *projection* and intuitively means that p_i may evaluate to either $\mathbf{1}$ or $\mathbf{0}$. Let $\forall \mathsf{p}_i(R) \stackrel{\text{DEF}}{=} \neg\exists \mathsf{p}_i(\neg R)$. We also write $\exists^i$ as an abbreviation for $\exists \mathsf{p}_i$.

The *rank* of R, denoted by $\sharp R$, is $i + 1$ where i is the largest index of any free occurrence of p_i in R, or 0 if R is *closed* (contains no free variables).

A key result is that **EREQ** has a *symbolic derivative function* δ that maps R into a nested if-then-else term $\delta(R)$ called a *transition regex* such that the leaf of $\delta(R)$ for any element $c \in \Sigma_{\sharp R}$, $\delta(R)[c]$, is a regex in **EREQ** as the concrete derivative of R for c. That is proved below in Theorem 2.

5.1 Relationships Between Boolean Connectives in Predicates and Regexes

We work with predicates of $\mathcal{A}_N$ always in *negation normal form* or *NNF* (up to propositions). E.g., the NNF of $\overline{\mathsf{p}_i \sqcap (\mathsf{p}_j \rightarrow \alpha)}$ (where $\alpha \in \mathcal{A}$) in $\mathcal{A}_N$ is $\overline{\mathsf{p}_i} \sqcup (\mathsf{p}_j \sqcap \overline{\alpha})$, so the resulting predicate is a positive Boolean combination over proposition literals and predicates from $\mathcal{A}$. This enables a

uniform treatment of the positive Boolean connectives of $\mathcal{A}_N$ and **EREQ**. It follows from the regex language semantics $\mathcal{L}_N(R)$ below that, for all $\beta \in \mathcal{A} \cup \{\mathsf{p}_i, \overline{\mathsf{p}_i}\}_{i<N}$, $[\![\beta]\!]_{\mathcal{A}_N} = \mathcal{L}_N(\beta)$. Therefore,

$$[\![\beta_1 \sqcap \beta_2]\!]_{\mathcal{A}_N} = [\![\beta_1]\!]_{\mathcal{A}_N} \cap [\![\beta_2]\!]_{\mathcal{A}_N} = \mathcal{L}_N(\beta_1) \cap \mathcal{L}_N(\beta_2) = \mathcal{L}_N(\beta_1 \wedge \beta_2)$$
$$[\![\beta_1 \sqcup \beta_2]\!]_{\mathcal{A}_N} = [\![\beta_1]\!]_{\mathcal{A}_N} \cap [\![\beta_2]\!]_{\mathcal{A}_N} = \mathcal{L}_N(\beta_1) \cup \mathcal{L}_N(\beta_2) = \mathcal{L}_N(\beta_1 \vee \beta_2)$$

A typical case is that a proposition p_i is associated with a label predicate α using the regex $\alpha \wedge \mathsf{p}_i$ (or equivalently the predicate $\alpha \sqcap \mathsf{p}_i$). For example, $\overline{\mathsf{p}_0}^*\cdot(\alpha \wedge \mathsf{p}_0)\cdot\overline{\mathsf{p}_0}^*$ has rank 1 and accepts all words $w \in (\Sigma \times \mathbb{B})^*$ with exactly one position i such that $w(i) \vDash \mathsf{p}_0$, where moreover $w(i) \vDash \alpha$.

It is important though to keep in mind that the top element in **EREQ** is $\bigstar = \top^*$ while the top element in $\mathcal{A}_N$ is $\top$. In other words, $R \wedge \bigstar = R$ forall $R \in$ **EREQ** while $\beta \sqcap \top = \beta$ for all $\beta \in \mathcal{A}_N$. The bottom element $\bot$ behaves similarly in both. *Crucially, the semantics of negation of predicates in $\mathcal{A}_N$ and regex complement are different.* In particular $\overline{\bot} = \top$ while $\neg\bot = \bigstar$.

5.2 Normalized Representation

In a *normalized* form of a closed regex R, all propositions use *de Bruijn levels* as indices, same as in the case of **wMSO**, and we can similarly write $\exists(R)$ without ambiguity. E.g., $\ulcorner\exists(\neg\exists(X_0 <_\exists X_1))\urcorner$ implicitly binds the outer $\exists$ with X_0 and the inner $\exists$ with X_1 according to de Bruijn levels. The embedded **wMSO** formula states that there exists a set X_0 that contains the final position, i.e., there exists no set X_1 containing a larger position. The encoded regex is $\exists\mathsf{p}_0(\neg\exists\mathsf{p}_1(\bigstar\cdot\mathsf{p}_0\cdot\bigstar\cdot\mathsf{p}_1\cdot\bigstar))$, or using universal quantifiers and implicit variable naming, the regex is $\exists\forall\neg(\bigstar\cdot\mathsf{p}_0\cdot\bigstar\cdot\mathsf{p}_1\cdot\bigstar)$.

The technical reason behind using *de Bruijn levels* rather than *de Bruijn indices* (where the indices are calculated in the opposite direction) is related to incremental $\exists^i$ projection during construction of symbolic derivatives. De Bruijn *indices* can be used similarly with additional bookkeeping – *levels* are just technically more suitable. Eliminating p_i with the *largest* index first simplifies the projection rule for (symbolic) derivatives that are constructed bottom-up (as t-regexes) but projection is applied top-down, as detailed in Section 7. In the case of constructing the symbolic derivative of a regex such as $\exists^1(R_1(\mathsf{p}_0, \mathsf{p}_1) \wedge \exists^2(R_2(\mathsf{p}_0, \mathsf{p}_1, \mathsf{p}_2)))$ the projection $\exists^2$ is applied first, at which point the meaning of p_0 and p_1 remains stable, that aligns with the formal semantics that uses π_2.

5.3 Semantics of EREQ

As discussed in Section 2, **EREQ** adopts M2L-str semantics: it aligns with classical regular language semantics and cleanly supports concatenation, Kleene star, and complement without the right-padding closure required by WS1S. We define below the corresponding language interpretation.

The *language of rank N* of a *normalized* regex R with $\sharp R \leq N$, denoted by $\mathcal{L}_N(R)$, is defined as follows. We let also $\mathcal{L}(R) \stackrel{\text{DEF}}{=} \mathcal{L}_{\sharp R}(R)$. Let $\beta \in \mathcal{A}_N$, $m > 0$, and recall that $R\{0\} \stackrel{\text{DEF}}{=} \varepsilon$.

$$\begin{array}{rcl}
\mathcal{L}_N(\beta) & \stackrel{\text{DEF}}{=} & \{c \in \Sigma_N \mid c \vDash \beta\} \\
\mathcal{L}_N(\varepsilon) & \stackrel{\text{DEF}}{=} & \{\epsilon\} \\
\mathcal{L}_N(R_1 \vee R_2) & \stackrel{\text{DEF}}{=} & \mathcal{L}_N(R_1) \cup \mathcal{L}_N(R_2) \\
\mathcal{L}_N(R_1 \wedge R_2) & \stackrel{\text{DEF}}{=} & \mathcal{L}_N(R_1) \cap \mathcal{L}_N(R_2) \\
\mathcal{L}_N(R_1 \cdot R_2) & \stackrel{\text{DEF}}{=} & \mathcal{L}_N(R_1)\cdot\mathcal{L}_N(R_2)
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{L}_N(R\{m\}) & \stackrel{\text{DEF}}{=} & \mathcal{L}_N(R)\cdot\mathcal{L}_N(R\{m-1\}) \\
\mathcal{L}_N(R^*) & \stackrel{\text{DEF}}{=} & \bigcup_{n\geq 0} \mathcal{L}_N(R\{n\}) \\
\mathcal{L}_N(\neg R) & \stackrel{\text{DEF}}{=} & \Sigma_N^* \setminus \mathcal{L}_N(R) \\
\mathcal{L}_N(\exists(R)) & \stackrel{\text{DEF}}{=} & \{\pi(w) \mid w \in \mathcal{L}_{N+1}(R)\}
\end{array}$$

If we lift π to sets of words, then the definition of $\mathcal{L}_N(\exists(R))$ implies that $\mathcal{L}_N(\exists(R)) = \pi(\mathcal{L}_{N+1}(R))$. At the same time, for all $u, v \in \Sigma_{N+1}^*$, it holds that $\pi(uv) = \pi(u)\pi(v)$, and therefore for all $U, V \subseteq \Sigma_{N+1}^*$, that $\pi(U\cdot V) = \pi(U)\cdot\pi(V)$. It follows that $\exists$ *propagates over concatenation*:

$$\mathcal{L}_N(\exists(R_1\cdot R_2)) = \pi(\mathcal{L}_{N+1}(R_1\cdot R_2)) = \pi(\mathcal{L}_{N+1}(R_1))\cdot\pi(\mathcal{L}_{N+1}(R_2)) = \mathcal{L}_N(\exists(R_1))\cdot\mathcal{L}_N(\exists(R_2))$$

Starting from an initially closed regex R_0 in $\mathcal{L}(R_0)$, observe that the case of $\mathcal{L}_N(\exists(R))$ above corresponds to $\mathcal{L}_N(\exists^N(R))$, where N is the de Bruijn level N of p_N. So the truth assignment to any free occurrence of p_N in R is now bit number N (or N'th bit) in $\mathbb{B}^{N+1}$.

Recall that $\bigstar$ is $\top*$, so $\mathcal{L}_N(\bigstar) = \Sigma_N^*$ and $\mathcal{L}(\bigstar) = \Sigma^*$ because $\mathcal{L}_N(\top) = \Sigma_N$.

Match Relation. We define the *match* relation $w \Vdash R$ for $w \in \Sigma_N^*$ and $R \in \mathbf{EREQ}$ such that $\sharp R \leq N$, that is intended to be read as w *models* R or R *matches* w.

$$w \Vdash_N R \stackrel{\text{DEF}}{=} w \in \mathcal{L}_N(R)$$

The match relation is a more fundamental notion than language because $(\Sigma_N^*, \mathbf{EREQ}, \Vdash, \bot, \bigstar, \vee, \wedge, \neg)$ is an EBA where $\Vdash$ is decidable since $\vdash$ is decidable. Regarding $\exists^N$, note that for all $\lceil w, \theta \rceil \in \Sigma_N^*$,

$$\lceil w, \theta \rceil \Vdash_N \exists^N(R) \;\Leftrightarrow\; \exists v : \lceil w, \theta \rceil = \pi(v) \wedge v \Vdash_{N+1} R \overset{\text{Lma 1}}{\Leftrightarrow} \exists S \subseteq |w| : \lceil w, \theta{\cdot}S \rceil \Vdash_{N+1} R$$

which gives the projection operator $\exists^N$ a dual meaning as an existential quantifier over sets of positions, as in **wMSO**. Analogously

$$\lceil w, \theta \rceil \Vdash_N \forall \mathsf{p}_N(R) \quad\Leftrightarrow\quad \forall S \subseteq |w| : \lceil w, \theta{\cdot}S \rceil \Vdash_{N+1} R$$

Concatenation has similarly a dual meaning through the encoding. Let $N = |\theta| = |\sigma|$, and for $m \in \mathbb{N}$ and $i < N$ let $(\theta \uplus_m \sigma)(i) \stackrel{\text{DEF}}{=} \theta(i) \cup \{k + m \mid k \in \sigma(i)\}$. Then

$$\lceil v, \theta \rceil{\cdot}\lceil w, \sigma \rceil = \lceil vw, \theta \uplus_{|v|} \sigma \rceil$$

5.4 Laws of $\exists$ in EREQ

Foremost, $\exists$ *distributes over* $\cdot$ *and* $\vee$. Define $R \equiv_N S$ as $\mathcal{L}_N(R) = \mathcal{L}_N(S)$. Let $\exists$ stand for $\exists^N$.

$\exists(R{\cdot}S)$	$\equiv_N$	$\exists(R){\cdot}\exists(S)$	$\exists(R*)$	$\equiv_N$	$\exists(R)*$	
$\exists(R{\vee}S)$	$\equiv_N$	$\exists(R){\vee}\exists(S)$	$\exists(R{\wedge}S)$	$\equiv_N$	$R \wedge \exists(S)$	when p_N is not free in R
$\exists(R\{m\})$	$\equiv_N$	$\exists(R)\{m\}$	$\exists(p)$	$\equiv_N$	$\top$	when $p = \mathsf{p}_N$ or $p = \overline{\mathsf{p}_N}$

We showed that $\exists$ propagates over $\cdot$ immediately after the formal definition of semantics. Observe that $R\{m\}$ unfolds into m-time concatenation of R and the law above applies and $\exists$ distributes over $\vee$, e.g., $\exists(R \vee R{\cdot}R) \equiv_N \exists(R) \vee \exists(R){\cdot}\exists(R)$.

An important fact to keep in mind is that $\neg\mathsf{p}_N \not\equiv \overline{\mathsf{p}_N}$. E.g., $\exists^0(\neg\mathsf{p}_0) \equiv \bigstar$ but $\exists^0(\overline{\mathsf{p}_0}) \equiv \top$.

6 Embedding of wMSO into EREQ

The following rules embed *normalized* **wMSO** formulas φ as regexes $\ulcorner\varphi\urcorner$ in **EREQ**. We can therefore omit variable names from $\exists$ because the names are implicitly bound to corresponding $\exists$ quantifiers according to their de Bruijn levels. The encoding $\ulcorner\varphi\urcorner$ is a direct *linear* embedding.

$$\ulcorner\exists\varphi\urcorner \stackrel{\text{DEF}}{=} \exists(\ulcorner\varphi\urcorner) \qquad \ulcorner |X_i| = 1\urcorner \stackrel{\text{DEF}}{=} \overline{\mathsf{p}_i}*{\cdot}\mathsf{p}_i{\cdot}\overline{\mathsf{p}_i}*$$

$$\ulcorner\neg\varphi\urcorner \stackrel{\text{DEF}}{=} \neg\ulcorner\varphi\urcorner \qquad \ulcorner\alpha(X_i)\urcorner \stackrel{\text{DEF}}{=} \bigstar{\cdot}(\alpha \wedge \mathsf{p}_i){\cdot}\bigstar \qquad \ulcorner\alpha[X_i]\urcorner \stackrel{\text{DEF}}{=} (\overline{\mathsf{p}_i} \vee \alpha)*$$

$$\ulcorner\varphi_1 \wedge \varphi_2\urcorner \stackrel{\text{DEF}}{=} \ulcorner\varphi_1\urcorner \wedge \ulcorner\varphi_2\urcorner \qquad \ulcorner X_i <_\exists X_j\urcorner \stackrel{\text{DEF}}{=} \bigstar{\cdot}\mathsf{p}_i{\cdot}\bigstar{\cdot}\mathsf{p}_j{\cdot}\bigstar \qquad \ulcorner X_i <_\forall X_j\urcorner \stackrel{\text{DEF}}{=} \overline{\mathsf{p}_j}*{\cdot}\overline{\mathsf{p}_i}*$$

$$\ulcorner\varphi_1 \vee \varphi_2\urcorner \stackrel{\text{DEF}}{=} \ulcorner\varphi_1\urcorner \vee \ulcorner\varphi_2\urcorner \qquad \ulcorner X_i \subseteq X_j\urcorner \stackrel{\text{DEF}}{=} (\overline{\mathsf{p}_i} \vee \mathsf{p}_j)*$$

The *rank* of a *closed* normalized **wMSO** formula is 0. The *rank* of an *open* normalized **wMSO** formula φ is $i + 1$ where i is the largest index of a free variable X_i that occurs in φ. It is often convenient to assume that φ in $\exists X_i(\varphi)$ has rank $i + 1$, i.e., that φ actually contains X_i as a free variable.

The following example illustrates the embedding of a rather simple **wMSO** formula that states that a position exists, i.e., that the matched word is nonempty. It illustrates use of $\exists$ laws.

Example 6.1. Consider $\exists(|X_0|{=}1)$. So $\ulcorner\exists(|X_0|{=}1)\urcorner = \exists(\overline{\mathsf{p}_0}{*}{\cdot}\mathsf{p}_0{\cdot}\overline{\mathsf{p}_0}{*})$. This example illustrates laws of $\exists$ that are applied to propagate $\exists$ over $\cdot$ and $*$ and rewrite both $\exists(\mathsf{p}_0)$ and $\exists(\overline{\mathsf{p}_0})$ to $\top$, reducing $\exists(\overline{\mathsf{p}_0}{*}{\cdot}\mathsf{p}_0{\cdot}\overline{\mathsf{p}_0}{*})$ directly into $\bigstar{\cdot}\top{\cdot}\bigstar$ that further rewrites to $\top{+}$. ⊠

6.1 Correctness of the Embedding

The following theorem states the correctness of the embedding of **wMSO** into **EREQ**.

Theorem 1 (MSO). *Let φ be normalized and have rank N. Then $w, \theta \models \varphi \Leftrightarrow \lceil w, \theta \rceil \Vdash_N \ulcorner\varphi\urcorner$.*

We now outline the correctness proof of the embedding of **wMSO** formulas into **ERE** in Lean. The **ERE** formulas in Lean are represented by the inductive type `Formula α (n : Nat)`, where α is the type of the alphabet predicates. The type of formulas is indexed by the number n of free second-order variables in scope. In particular, the existential quantifier constructor increments n to account for the newly bound variable.

For a word of length `len`, an assignment maps each variable index (out of `n`) to the set of positions it denotes. This is encoded as a vector of booleans of length `len`, where the entry **1** marks inclusion in the set. Assignments are therefore functions from variable indices to boolean vectors:

```
abbrev assignment (len : Nat) : Nat → Type := fun (n : Nat) =>
  Fin n → Vector Bool len
```

Note that `Fin n` is the type of natural numbers strictly less than `n`.

Quantification introduces a fresh variable by extending the assignment with an additional vector:

```
def extAssignment (θ : assignment len i) (v : Vector Bool len) : assignment len (i + 1)
```

The semantics of formulas is defined as a function into **Prop** as a standard way to avoid the non-positivity issue which arises due to the negation constructor [Zhuchko et al. 2024]. We omit the Boolean operators as they are defined exactly as in Section 4.2.

```
def Formula.models : Formula α i → (xs : List σ) → assignment |xs| i → Prop
  | [| n |], xs, θ =>
    ∃ (i : Fin |xs|), (θ n)[i] ∧ ∀ (j : Fin |xs|), j ≠ i → ¬ (θ n)[j]
  | m ⊆ᶠ n, xs, θ => ∀ (i : Fin |xs|), (θ m).get i → (θ n).get i
  | m <ᶠ n, xs, θ => ∃ (i j : Fin |xs|), (θ m).get i ∧ (θ n).get j ∧ i < j
  | FPred (p : α) n, xs, θ => ∃ (i : Fin |xs|), (θ n).get i ∧ xs[i]?.any (models p)
  | ∃ᶠ φ, xs, θ => ∃ (v : Vector Bool |xs|), models φ xs (extAssignment θ v)
  | ...
```

The subscript f is used to refer to the corresponding operations on formulas.

We represent regular expressions as the inductive type `RE α n`, where α is the type of alphabet predicates and n is the number of free variables of the translated **wMSO** formula. To track variable assignments, each character of a word is paired with a boolean vector of length n, indicating which variables are active at that position:

```
abbrev Elem (σ : Type) (n : Nat) := σ × Vector Bool n
abbrev Word (σ : Type) (n : Nat) := List (Elem σ n)
```

Following [Traytel and Nipkow 2013], we encode assignments as bitvectors in Lean. The two main differences are that we do not have explicit first-order variables and, more importantly, the most recently bound variable is added at the end of the vector rather than at the beginning as in

[Traytel and Nipkow 2013]. This corresponds precisely to the distinction between de Bruijn indices and levels.

The language semantics of regular expressions is standard except for the case of the existential quantifier. Intuitively, the function `extend` adds a fresh variable to the word by appending a boolean to the end of each vector in the word, using `snoc`. Conversely, the projection operation removes the last boolean from each vector in the word i.e., the most recently bound variable.

```
def RE.models : RE α n → Word σ n → Prop
| ... | ∃ʳ R, w => ∃ (s : Vector Bool |w|), models R (extend w s)
```

wMSO formulas are translated into regular expressions using the function $\ulcorner \phi \urcorner$ which implements the translation from Section 6. Similarly, variable assignments are encoded as words using the translation $\lceil xs, \theta \rceil$ described in Section 5.3.

```
theorem toRE_correctness (ϕ : Formula α n) (xs : List σ) (θ : assignment |xs| n) :
  xs, θ ⊨ᶠ ϕ ⟺ ⌈xs, θ⌉ ⊨ ⌜ϕ⌝
```

The proof proceeds by induction on the structure of ϕ. One of the key cases is the existential quantifier, which relies on the correctness of translating between assignments and words - formalized in Lean as a variant of Lemma 1.

6.2 Note About Expressivity

The opposite direction of going from **EREQ** back to **wMSO** is nontrivial. In the following arguments we omit bounded loops $R\{m\}$ from **EREQ** and assume that satisfiability checking in $\mathcal{A}_N$ is in PSPACE. When considering the $\exists^*$-fragment of **EREQ** that is regexes in prenex normal form whose quantifier prefix is in $\exists^*$, then their nonemptiness problem reduces to nonemptiness of **ERE** that is in EXPSPACE. If considering the positive fragment of **EREQ**, i.e., **EREQ** without $\neg$, then its regexes in prenex normal form are regexes in **ERE** without $\neg$ whose quantifier prefix is also in $\exists^*$. In this case the nonemptiness problem is in PSPACE. These complexities follow from classical results [Meyer and Stockmeyer 1972] for extended regular expressions. If these fragments are translated back into **wMSO**, the resulting formulas require, in the general case, unbounded layers of quantifier alternations. Given that R is translated to $\phi_R(X)$ where X is a relative set of positions of a word accepted by R, then for $R{\cdot}S$ we get that $\phi_{R\cdot S}(X)$ can be represented in **wMSO** by

$$\exists Y, Z(\phi_R(Y) \wedge \phi_S(Z) \wedge X = Y \cup Z \wedge Y <_\forall Z \wedge \mathit{Interval}(X))$$

where $\mathit{Interval}(X) \stackrel{\text{DEF}}{=} \forall x, y, z((x < y < z \wedge x \in X \wedge z \in X) \Rightarrow y \in X)$, which means that X is a contiguous set of positions. This translation cannot be avoided in general. Even if multiple concatenations can be translated in one generalized step as above, quantifier alternations still arise when translating $*$-loops, or when nested under complement, e.g., in $\phi_{\neg(R_1\cdot R_2)\cdot S}$.

So the corresponding complexity characterizations no longer hold in **wMSO** that has non-elementary complexity in the case of unbounded quantifier alternations. In that sense **EREQ** is fundamentally different from **wMSO** as a formal language. Protocol-like specs of web-services that we have encountered so far, fall into a limited need of quantifiers for specifying certain concepts, and primarily use concatenation for *sequential composition* and *interleaving*, $*$-loops for expressing *repetition*, intersection for *parallel composition* and *restriction*, and complement for *exclusion*.

7 Symbolic Derivatives and Decision Procedure for EREQ

This section introduces the main algorithmic contribution that, in a nutshell, is incremental unfolding of regexes in **EREQ** into symbolic automata, including *incremental propagation of projection* as

the key new feature. In order to provide a self-contained description of the framework we include required background material: symbolic derivatives of **ERE** and t-regexes, originally introduced in [Stanford et al. 2021]. The new feature is *projection* in **EREQ** through the rule

$$\delta(\exists(R)) \stackrel{\text{DEF}}{=} \exists(\delta(R))$$

where $\delta(R)$ is the (symbolic) derivative of R represented as a t-regex. The true power of this rule resides in the fact that it is lazy, works incrementally and seamlessly together with complement ($\neg$) as well as intersection ($\wedge$) and union ($\vee$) while preserving algebraic laws that enable rewrites of (transition) regexes that would otherwise be *lost in translation* to automata.

In the context of the rest of the rules, propagation of projection is tantamount to *incremental quantifier elimination* of $\exists$. We start by introducing t-regexes and their normal form used as core representations in derivative computations.

7.1 Normalized Transition Regexes and Projection

A *t-regex* is either a *leaf* (R) where $R \in \mathbf{EREQ}$ or a *node* $f = (\gamma\ ?\ g : h)$ where $\gamma \in \mathcal{A}_N$ and g and h are t-regexes. In the case of a node $f = (\gamma\ ?\ g : h)$ we use the following accessors, where $cond(f)$ is the *condition*, $f_{\mathbf{1}}$ the *then-case*, and $f_{\mathbf{0}}$ the *else-case* of f.

$$cond(f) \stackrel{\text{DEF}}{=} \gamma \quad f_{\mathbf{1}} \stackrel{\text{DEF}}{=} g \quad f_{\mathbf{0}} \stackrel{\text{DEF}}{=} h$$

We let also $cond((R)) \stackrel{\text{DEF}}{=} \top$ for all $R \in \mathbf{EREQ}$. *Evaluation* of a t-regex f for $c \in \Sigma \times \mathbb{B}^*$, denoted by $f[\![c]\!]$, is defined as follows. In the following let R be a regex and let f be a node:

$$(R)[\![c]\!] \stackrel{\text{DEF}}{=} R \qquad f[\![c]\!] \stackrel{\text{DEF}}{=} \begin{cases} f_{\mathbf{1}}[\![c]\!], \textbf{ if } c \vDash cond(f); \\ f_{\mathbf{0}}[\![c]\!], \textbf{ otherwise}. \end{cases}$$

The immediate (trivial) rewrites are $(\gamma\ ?\ f : f) = f$, $(\top\ ?\ f : g) = f$, and $(\bot\ ?\ f : g) = g$. We define $Leaves(f) \stackrel{\text{DEF}}{=} Leaves_{\top}(f)$ as the set of *reachable* leaf regexes of f, where

$$Leaves_{\beta}((R)) \stackrel{\text{DEF}}{=} \begin{cases} \{R\}, \textbf{ if SAT}(\beta); \\ \emptyset, \quad \textbf{otherwise}. \end{cases} \qquad Leaves_{\beta}((\gamma\ ?\ f_{\mathbf{1}} : f_{\mathbf{0}})) \stackrel{\text{DEF}}{=} Leaves_{\beta \sqcap \gamma}(f_{\mathbf{1}}) \cup Leaves_{\beta \sqcap \overline{\gamma}}(f_{\mathbf{0}})$$

Going forward, to avoid wrapping and unwrapping of regexes as leaves, we do not distinguish between the leaf (R) and the regex R.

We use a *normalized* representation of nodes whose conditions are either propositions p_i or predicates $\alpha \in \mathcal{A}$ rather than arbitrary predicates in $\mathcal{A}_N$. Moreover, let $<$ be a total order such that $\mathsf{p}_i < \mathsf{p}_j \stackrel{\text{DEF}}{=} i < j$ and $\alpha < \mathsf{p}_i$ for all $\alpha \in \mathcal{A}$, all normalized nodes f maintain the invariant that $cond(f_{\mathbf{1}}) < cond(f)$ and $cond(f_{\mathbf{0}}) < cond(f)$, where $\mathcal{A}$ is also totally ordered by $<$.

All binary operations $\diamond$ over **EREQ** are lifted to normalized t-regexes as follows where at least one of f or g is a node. Thus, $f \diamond g$ is also normalized.

$$f \diamond g \quad \stackrel{\text{DEF}}{=} \quad \begin{cases} (cond(f)\ ?\ f_{\mathbf{1}} \diamond g_{\mathbf{1}} : f_{\mathbf{0}} \diamond g_{\mathbf{0}}), & \textbf{if } cond(f) = cond(g); \\ (cond(f)\ ?\ f_{\mathbf{1}} \diamond g : f_{\mathbf{0}} \diamond g), & \textbf{else if } cond(g) < cond(f); \\ (cond(g)\ ?\ f \diamond g_{\mathbf{1}} : f \diamond g_{\mathbf{0}}), & \textbf{otherwise}. \end{cases}$$

All unary operations $\circ$ over **EREQ** *except projection* are lifted as follows to all nodes f.

$$\circ f \quad \stackrel{\text{DEF}}{=} \quad (cond(f)\ ?\ \circ f_{\mathbf{1}} : \circ f_{\mathbf{0}})$$

It follows that for all unary operators $\circ$, all binary operators $\diamond$, and all t-regexes f and g:

$$(\circ f)[\![c]\!] = \circ(f[\![c]\!]) \qquad (f \diamond g)[\![c]\!] = f[\![c]\!] \diamond g[\![c]\!]$$

Due to normalization, if $cond(f) = \mathsf{p}_i$ then p_i does not occur as a condition anywhere in $f_{\mathbf{1}}$ or $f_{\mathbf{0}}$, i.e., any condition p_j in either $f_{\mathbf{1}}$ or $f_{\mathbf{0}}$ has index $j < i$. Projection is handled separately as follows for all nodes f, where $\circ_{\exists^i}$ is a new unary operator such that $\circ_{\exists^i}(R) \stackrel{\text{DEF}}{=} \exists^i(R)$ for $R \in \mathbf{EREQ}$.

$$\exists^i(f) \stackrel{\text{DEF}}{=} \begin{cases} \circ_{\exists^i}(f_{\mathbf{1}}) \vee \circ_{\exists^i}(f_{\mathbf{0}}), & \textbf{if } cond(f) = \mathsf{p}_i; \\ \circ_{\exists^i}(f), & \textbf{otherwise}. \end{cases}$$

When $cond(f) = \mathsf{p}_i$ the projection $\exists^i$ gets propagated to all the leaves of $f_{\mathbf{1}}$ and $f_{\mathbf{0}}$ as a unary operator because p_i cannot occur as a condition in either of them. This is the key construction that eliminates dependency on p_i and ultimately creates more nondeterminism via the resulting union $\circ_{\exists^i}(f_{\mathbf{1}}) \vee \circ_{\exists^i}(f_{\mathbf{0}})$ of t-regexes that is again a normalized t-regex.

When $cond(f) \neq \mathsf{p}_i$ the case $\exists^i(f)$ with $\mathsf{p}_i < cond(f)$ cannot arise because the derivative rules below guarantee, due to de Bruijn levels, that $\mathsf{p}_i \geq cond(f)$. So in this case $\exists^i(f) = \circ_{\exists^i}(f)$, i.e., the projection $\exists^i$ propagates to all the leaves of f as any other unary operator.

7.2 Symbolic Derivatives of EREQ

For literals p_i and $\overline{\mathsf{p}_i}$ their (symbolic) derivatives are defined as follows. The key new rule is the (symbolic) derivative of $\exists(R)$

$$\delta(\mathsf{p}_i) \stackrel{\text{DEF}}{=} (\mathsf{p}_i \;?\; \varepsilon : \bot) \qquad \delta(\overline{\mathsf{p}_i}) \stackrel{\text{DEF}}{=} (\mathsf{p}_i \;?\; \bot : \varepsilon) \qquad \delta(\exists^i(R)) \stackrel{\text{DEF}}{=} \exists^i(\delta(R))$$

where, by construction of normalized (transition) regexes it holds that $\mathsf{p}_i \geq cond(\delta(R))$. Due to immediate rewrites such as $(\mathsf{p}_i \;?\; f : f) = f$, or if p_i does not occur in R, it can be the case that $\mathsf{p}_i > cond(\delta(R))$, in which case $\delta(R) \equiv (\mathsf{p}_i \;?\; \delta(R) : \delta(R))$. Therefore, one can always assume that $cond(\delta(R)) = \mathsf{p}_i$ in the case of $\exists^i(R)$.

Nullability of R ($Null(R)$) uses the standard definition of nullability in **ERE** extended with the case for projection. Nullability is maintained as a flag of each regex (among several other meta information). The complete definition is as follows, where $\beta \in \mathcal{A}_N$ and $m > 0$.

$$\begin{array}{lll} Null(\beta) \stackrel{\text{DEF}}{=} \mathbf{0} & Null(R_1 \vee R_2) \stackrel{\text{DEF}}{=} Null(R_1) \vee Null(R_2) & Null(R\{m\}) \stackrel{\text{DEF}}{=} Null(R) \\ Null(\varepsilon) \stackrel{\text{DEF}}{=} \mathbf{1} & Null(R_1 \wedge R_2) \stackrel{\text{DEF}}{=} Null(R_1) \wedge Null(R_2) & Null(\neg R) \stackrel{\text{DEF}}{=} \neg Null(R) \\ Null(R*) \stackrel{\text{DEF}}{=} \mathbf{1} & Null(R_1 \cdot R_2) \stackrel{\text{DEF}}{=} Null(R_1) \wedge Null(R_2) & Null(\exists(R)) \stackrel{\text{DEF}}{=} Null(R) \end{array}$$

So $Null(R) \Leftrightarrow \epsilon \Vdash R$. In particular, $\epsilon \nVdash \overline{\mathsf{p}_i}$ but $\epsilon \Vdash \neg\mathsf{p}_i$. It follows also that $Null(\forall(R)) \Leftrightarrow Null(R)$.

All the remaining (symbolic) derivative rules remain identical to those in **ERE** [Stanford et al. 2021] that we include here for completeness. Let $\alpha \in \mathcal{A}$, $m > 0$, and recall that $R\{0\} \stackrel{\text{DEF}}{=} \varepsilon$ and also $R\{1\} = R$ is an immediate rewrite.

$$\begin{array}{rcl} \delta(\alpha) & \stackrel{\text{DEF}}{=} & (\alpha \;?\; \varepsilon : \bot) \\ \delta(R_1 \vee R_2) & \stackrel{\text{DEF}}{=} & \delta(R_1) \vee \delta(R_2) \\ \delta(R_1 \wedge R_2) & \stackrel{\text{DEF}}{=} & \delta(R_1) \wedge \delta(R_2) \\ \delta(\neg R) & \stackrel{\text{DEF}}{=} & \neg\delta(R) \end{array} \qquad \begin{array}{rcl} \delta(R\{m\}) & \stackrel{\text{DEF}}{=} & \delta(R)\cdot R\{m-1\} \\ \delta(R*) & \stackrel{\text{DEF}}{=} & \delta(R)\cdot R* \\ \delta(R_1 \cdot R_2) & \stackrel{\text{DEF}}{=} & \begin{cases} \delta(R_1)\cdot R_2 \vee \delta(R_2), & \textbf{if } Null(R_1); \\ \delta(R_1)\cdot R_2, & \textbf{otherwise}. \end{cases} \end{array}$$

Note that $\neg\mathsf{p}_i$ has very different semantics from $\overline{\mathsf{p}_i}$, namely $\delta(\neg\mathsf{p}_i) = \neg\delta(\mathsf{p}_i) = \neg(\mathsf{p}_i \;?\; \varepsilon : \bot) = (\mathsf{p}_i \;?\; \neg\varepsilon : \neg\bot) \equiv (\mathsf{p}_i \;?\; \top+ : \bigstar)$. In fact, $\overline{\mathsf{p}_i} \equiv \neg\mathsf{p}_i \wedge \top$ in **EREQ** because $\neg\mathsf{p}_i \wedge \top$ is not nullable and

$$\delta(\neg\mathsf{p}_i \wedge \top) \equiv (\mathsf{p}_i \;?\; \top+ : \bigstar) \wedge \delta(\top) = (\mathsf{p}_i \;?\; \top+ : \bigstar) \wedge \varepsilon = (\mathsf{p}_i \;?\; \top+ \wedge \varepsilon : \bigstar \wedge \varepsilon) \equiv (\mathsf{p}_i \;?\; \bot : \varepsilon)$$

7.3 Derivation Examples

We consider some examples that illustrate, step by step, how the derivative rules work in **EREQ**. The main objective is to show how the various aspects of derivatives, regex rewrites, and t-regex rewrites interact. The formal definitions above hide most of the actual work done during derivative construction, because all operations are lifted to t-regexes, where the actual work takes place. The

focus here is mostly on the new rules involving projection. Showing full details of operations over t-regexes is very verbose, so we have to considerably condense the steps, otherwise the steps become unreadable although being automatic and straightforward in the solver.

Example 7.1 illustrates that certain rewrites can be applied at the level of $\mathcal{A}_N$ inside regexes. It shows that $\ulcorner\forall x_0(\alpha(x_0) \vee \overline{\alpha}(x_0))\urcorner \equiv \bigstar$. Intuitively $\alpha(x_0) \vee \overline{\alpha}(x_0)$ should be equivalent with $(\alpha \sqcup \overline{\alpha})(x_0)$ that is $\top(x_0)$. Observe that $\ulcorner\forall x_0(\alpha(x_0) \vee \neg\alpha(x_0))\urcorner$ also reduces to $\bigstar$ – but for a different reason: $R \vee \neg R \equiv \bigstar$ for all regexes R.

Example 7.1. Recall that $\forall x_0(\alpha(x_0) \vee \overline{\alpha}(x_0))$ stands for $\varphi = \neg\exists X_0(\neg(\alpha(X_0) \vee \overline{\alpha}(X_0)) \wedge |X_0|{=}1)$. Let $S_0 = \overline{\mathsf{p}_0}{*}\mathsf{p}_0\overline{\mathsf{p}_0}{*}$, $A = \bigstar(\alpha \wedge \mathsf{p}_0)\bigstar$ and $B = \bigstar(\overline{\alpha} \wedge \mathsf{p}_0)\bigstar$ below. So $\ulcorner\varphi\urcorner = \neg\exists^0(\neg(A \vee B) \wedge S_0)$. Many rewrites below are implicit, but it is instructive to see some rewrites explicitly.

$$\begin{aligned}
\delta(\ulcorner\varphi\urcorner) &= \delta(\neg\exists^0(\neg(A \vee B) \wedge S_0)) = \neg\exists^0(\neg(\delta(A) \vee \delta(B)) \wedge \delta(S_0)) \\
&= \neg\exists^0(\neg((\mathsf{p}_0 \;?\; (\alpha \;?\; \bigstar : A) : A) \vee (\mathsf{p}_0 \;?\; (\alpha \;?\; B : \bigstar) : B)) \wedge (\mathsf{p}_0 \;?\; \overline{\mathsf{p}_0}{*} : S_0)) \\
&= \neg\exists^0(\neg(\mathsf{p}_0 \;?\; (\alpha \;?\; \bigstar \vee B : A \vee \bigstar) : A \vee B) \wedge (\mathsf{p}_0 \;?\; \overline{\mathsf{p}_0}{*} : S_0)) \\
&= \neg\exists^0(\neg(\mathsf{p}_0 \;?\; \bigstar : A \vee B) \wedge (\mathsf{p}_0 \;?\; \overline{\mathsf{p}_0}{*} : S_0)) \\
&= \neg\exists^0((\mathsf{p}_0 \;?\; \bot : \neg(A \vee B)) \wedge (\mathsf{p}_0 \;?\; \overline{\mathsf{p}_0}{*} : S_0)) = \neg\exists^0((\mathsf{p}_0 \;?\; \bot : \neg(A \vee B) \wedge S_0)) \\
&= \neg(\exists^0(\bot) \vee \exists^0(\neg(A \vee B) \wedge S_0)) = \neg\exists^0(\neg(A \vee B) \wedge S_0) = \ulcorner\varphi\urcorner
\end{aligned}$$

It follows that $\ulcorner\varphi\urcorner \equiv \bigstar$ because $\ulcorner\varphi\urcorner$ is nullable and $\delta(\ulcorner\varphi\urcorner) = \ulcorner\varphi\urcorner$. ⊠

For first-order variables in general, the encoding from **wMSO** to **EREQ** can inline the singleton constraints directly to maintain more compact representations of the resulting t-regexes, although the equivalent formal definition adds the singleton constraints as separate conjuncts. The following example uses such rewrites and illustrates one of the more generally used simplification tactics that if $Null(R) = Null(S)$ and $\delta(R) = \delta(S)$ then $R \equiv S$, and if S is a simpler(smaller) regex then R rewrites to S. In the following example that simpler regex S happens to be $\top$.

Example 7.2. Consider the **wMSO** formula $\psi = \exists x_0(\nexists x_1(x_1 = x_0 + 1) \wedge \nexists x_1(x_1 < x_0))$ that states that there exists a position without any immediate successor and without any predecessor, i.e., the matched word must have length one. Let S_0 stand for the singleton regex $\overline{\mathsf{p}_0}{*}{\cdot}\mathsf{p}_0{\cdot}\overline{\mathsf{p}_0}{*}$. After a series of regex rewrites that utilize, among several other ones, that $R \wedge \bigstar = R$, $R \wedge \neg R \equiv \bot$, $R \wedge \bot = \bot$, $\neg(R{+}) \wedge R{*} \equiv \varepsilon$, and $R \vee \bot = R$. It follows that

$$\begin{aligned}
\delta(\ulcorner\psi\urcorner) &= \overset{\text{distribute } \exists^1}{\cdots} = \delta(\exists^0(\neg(\overline{\mathsf{p}_0}{*}\mathsf{p}_0\overline{\mathsf{p}_0}{+}) \wedge \neg(\overline{\mathsf{p}_0}{+}\mathsf{p}_0\overline{\mathsf{p}_0}{*}) \wedge S_0)) \\
&= \cdots = \exists^0((\mathsf{p}_0 \;?\; \neg(\overline{\mathsf{p}_0}{+}) \wedge \bigstar \wedge \overline{\mathsf{p}_0}{*} : \neg(\overline{\mathsf{p}_0}{*}\mathsf{p}_0\overline{\mathsf{p}_0}{+}) \wedge \neg S_0 \wedge S_0)) \\
&= \exists^0((\mathsf{p}_0 \;?\; \varepsilon : \bot)) = \exists^0(\varepsilon) \vee \exists^0(\bot) = \varepsilon \vee \bot = \varepsilon
\end{aligned}$$

Since $\ulcorner\psi\urcorner$ is not nullable and $\delta(\ulcorner\psi\urcorner) = \varepsilon$, it follows that $\ulcorner\psi\urcorner \equiv \top$. Observe that the conjunct S_0 in $\ulcorner\psi\urcorner$ is necessary above, or else the singleton semantics of p_0 would not be preserved because it occurs as a free variable inside the complemented $\exists^1$ sub-expressions. ⊠

Any universally quantified formula $\forall x_0(\varphi(x_0))$ is, by definition, short for $\neg\exists x_0(\neg\varphi(x_0))$ that itself stands for $\neg\exists X_0(|X_0|{=}1 \wedge \neg\varphi(X_0))$ in **wMSO** and therefore always holds for ϵ, independent of φ, since ϵ *has no positions*. In terms of **EREQ**, $\ulcorner\neg\exists X_0(|X_0|{=}1 \wedge \neg\varphi(X_0))\urcorner$ is always nullable independently of φ because $\ulcorner|X_0|{=}1\urcorner$ is not nullable. To enforce *nonempty* word semantics, $\forall x_0(\varphi)$ can be strengthened to $\forall x_0(\varphi) \wedge \exists(|X_0|{=}1)$. Recall from earlier that $\ulcorner\exists(|X_0|{=}1)\urcorner \equiv \top{+}$.

Example 7.3. Consider the **wMSO** formula $\varphi = \forall x_0(\exists x_1(x_1 = x_0 + 1)) \wedge \exists X_0(|X_0|{=}1)$, i.e., $\varphi = \nexists x_0(\nexists x_1(x_1 = x_0 + 1)) \wedge \exists X_0(|X_0|{=}1)$, that is unsatisfiable in M2L-str semantics because the last position of any finite nonempty word has no successor. The corresponding regex in **EREQ** is as follows, where $\ulcorner\exists X_0(|X_0|{=}1)\urcorner$ is represented equivalently as $\top{+}$, and where we reuse the

fact from above that $\ulcorner \nexists x_1 (x_1 = x_0 + 1) \urcorner$ rewrites to the regex $\neg(\overline{\mathsf{p}_0}{*}\mathsf{p}_0\overline{\mathsf{p}_0}{+}) \wedge \overline{\mathsf{p}_0}{*}\mathsf{p}_0\overline{\mathsf{p}_0}{*}$. Now, any regex $\neg(\overline{\gamma}{*}\gamma R) \wedge \overline{\gamma}{*}\gamma S$ where γ is a predicate, rewrites to $\overline{\gamma}{*}\gamma(\neg R \wedge S)$. So $\neg(\overline{\mathsf{p}_0}{*}\mathsf{p}_0\overline{\mathsf{p}_0}{+}) \wedge \overline{\mathsf{p}_0}{*}\mathsf{p}_0\overline{\mathsf{p}_0}{*}$ rewrites to $\overline{\mathsf{p}_0}{*}\mathsf{p}_0(\neg(\overline{\mathsf{p}_0}{+}) \wedge \overline{\mathsf{p}_0}{*})$. Since $\neg(R{+}) \wedge R{*}$ rewrites to ε, $\overline{\mathsf{p}_0}{*}\mathsf{p}_0(\neg(\overline{\mathsf{p}_0}{+}) \wedge \overline{\mathsf{p}_0}{*})$ rewrites to $\overline{\mathsf{p}_0}{*}\mathsf{p}_0$. Note also that $\delta(\top{+}) = (\top\ ?\ \bigstar : \bot) = \bigstar$. Thus, by the rewrites above,

$$\begin{aligned}\delta(\ulcorner\varphi\urcorner) &= \delta(\neg\exists^0(\overline{\mathsf{p}_0}{*}\mathsf{p}_0) \wedge \top{+}) = \neg\exists^0(\delta(\overline{\mathsf{p}_0}{*}\mathsf{p}_0)) \wedge \bigstar \\ &= \neg\exists^0((\mathsf{p}_0\ ?\ \varepsilon : \overline{\mathsf{p}_0}{*}\mathsf{p}_0)) = \neg(\exists^0(\varepsilon) \vee \exists^0(\overline{\mathsf{p}_0}{*}\mathsf{p}_0)) = \neg(\varepsilon \vee \top{+}) = \neg\bigstar = \bot\end{aligned}$$

So $\ulcorner\varphi\urcorner \equiv \bot$ because $\ulcorner\varphi\urcorner$ is not nullable (due its conjunct $\top{+}$) and its derivative is $\bot$. ⊠

7.4 Derivative Correctness Theorem

Our main result is:

Theorem 2 (Derivative). *Let R be a normalized regex in* **EREQ**, $c \in \Sigma_{\sharp R}$ *and* $w \in \Sigma^*_{\sharp R}$. *Then* $cw \Vdash R \Leftrightarrow w \Vdash \delta(R)[c]$.

The correctness of the derivative operations relies on several normalization steps captured and proven correct in the formalization: these normalizations operate at the level of EBA predicates, at the level of regular expressions, and at the level of t-regexes.

Predicates are normalized using the NNF from Section 5.1 in the intuitive way. Regular expressions are normalized via a function `normalize` which eliminates intersections and unions at the level of predicates by expressing them using the intersection and union operations of regexes.

```
def normalize : RE α n → RE_norm α n
```

The type of normalized regexes is `RE_norm`, and differs from `RE` in that predicates are restricted to three canonical forms: positive predicates p_i, negative predicates $\overline{\mathsf{p}_i}$, and atomic predicates α from the alphabet $\mathcal{A}$. We then prove that the transformations are semantics preserving.

As discussed in Section 7.1, t-regexes are ordered in the sense that predicates of the type p_i and $\overline{\mathsf{p}_i}$ are always higher up in the tree than those of type α. This is implemented by the following type of ordered transition terms:

```
abbrev DerTree α n := TTerm (Fin n) (TTerm α (RE_norm α n))
```

The key idea is to reuse transition terms `TTerm α β`, which are simply binary trees where α is the condition type at binary nodes and β is the type contained at the leaves. Hence, we encode an ordered transition term by first having only conditions of type `Fin n` at the top level, whose leaves are themselves transition trees where now predicates are just elements α from the EBA.

To ensure correctness of the derivative construction, we impose two additional invariants on `DerTree`, which are crucial to later prove lemmas about pushing projection down transition terms:

```
inductive IsLessThan : DerTree α n → Fin n → Prop where
  | base : IsLessThan (.Leaf a) bound
  | ind : pred < bound → IsLessThan l bound → IsLessThan r bound
                       → IsLessThan (.Node pred l r) bound

inductive IsNormalizedTree : DerTree α n → Prop where
  | base : IsNormalizedTree (.Leaf a)
  | ind : IsLessThan l pred → IsLessThan r pred
      → IsNormalizedTree l → IsNormalizedTree r → IsNormalizedTree (.Node pred l r)
```

The first invariant, `IsLessThan f b`, ensures that all the predicate conditions (which are represented as de Bruijn levels) of a given term `f` are strictly less than some bound `b : Fin n`. The second invariant, `IsNormalizedTree f`, simply ensures that the above definition is applied for each predicate in the tree: together, these invariants enforce a stratification of terms where each predicate must not be larger than the previous one in the tree.

We can now introduce the derivative function δ which takes a normalized regular expression and computes a normalized transition term representing the derivative.

```
def derivative : RE_norm α n → DerTree α n
```

The main lemma establishing the invariant for derivatives is as follows:

```
lemma der_invariant {r : RE_norm α n} : IsNormalizedTree (δ r)
```

A key technical lemma used in the proof of derivative correctness (which crucially uses the `IsNormalizedTree` condition) is the following:

```
lemma liftU_proj_norm {x : Elem σ n} (f : DerTree α (n + 1)) (h : IsNormalizedTree f) :
       xs ⊨ₙ (liftU_proj (∃ₙ ·) f)[ x ]ₙ
  ⟺ xs ⊨ₙ ∃ₙ (f[ ⟨x.1, x.2.snoc 1⟩ ]ₙ ⩏ₙ f[ ⟨x.1, x.2.snoc 0⟩ ]ₙ)
```

Whenever present, the subscript n is used to refer to the corresponding operations on *normalized* structures (e.g., regular expressions, terms, etc.)

The above lemma states that, given a normalized transition term `f`, applying projection to it is equivalent to taking the union of two branches-one where the newly introduced bit is set to `1`, and another where it is set to `0`. Note that we are adding the fresh variable at the end of the list, which corresponds to the highest index. Intuitively, this lemma allows us to push projections into transition terms, propagating it to the leaves of the derivative tree.

We then establish the main correctness theorem for derivatives:

```
theorem derivative_correctness (R : RE α n) (w : Word σ n) :
  (x::xs) ⊨ R ⟺ xs ⊨ₙ (δ (normalize R))[ x ]ₙ
```

The proof is by induction on `R`. Only two cases are not straightforward: The predicate case requires minor bookkeeping due to the multiple layers of normalization. The projection case is where all previous lemmas about normalization are used. We resolve it using the invariant lemma `der_invariant` and the lifting lemma `liftU_proj_norm`, which together ensure that projection is correctly propagated through normalized transition terms.

7.5 Derivative Based Decision Procedure for EREQ and wMSO

The core decision problem for $R \in \mathbf{EREQ}$ is deciding $\mathcal{L}_{\sharp R}(R) \neq \emptyset$. For a set $\mathbf{S}$ of regexes let $Null(\mathbf{S}) \stackrel{\text{DEF}}{=} \exists r \in \mathbf{S}(Null(r))$. The δ-based algorithm for nonemptiness is checking $Null(\delta^*(R))$ where $\delta^*(R) \stackrel{\text{DEF}}{=} \bigcup_{n\geq 0} \delta^n(R)$ is a fixpoint procedure that constructs a *δ-reachable* set of states:

$$\delta^0(R) \stackrel{\text{DEF}}{=} \{R\} \qquad \delta^{n+1}(R) \stackrel{\text{DEF}}{=} \delta^n(R) \cup \bigcup\{States(\delta(q)) \mid q \in \delta^n(R)\}$$

In the most basic form of the algorithm, $States(f) = Leaves(f)$. The algorithm terminates when for some $n \geq 0$, either $Null(\delta^n(R))$ holds in which case R is *satisfiable*, or else $\delta^{n+1}(R) = \delta^n(R)$ and $\neg Null(\delta^n(R))$ holds in which case $\neg Null(\delta^*(R))$ holds and R is *unsatisfiable*.

Termination or finiteness of $\delta^*(R)$, i.e., $\exists n : \delta^{n+1}(R) = \delta^n(R)$, is established by the following theorem, proven in Lean.

Theorem 3 (Finiteness of the Derivative Closure). *For all $R \in$ **EREQ**, the set $\delta^*(R)$ is finite (modulo ACI of $\vee$), i.e., $\exists n \geq 0 : \delta^{n+1}(R) = \delta^n(R)$.*

The proof for **ERE** (i.e., **EREQ** without $\exists$) is formalized in Lean in [Zhuchko et al. 2025], where ACI (associativity, commutativity, and idempotency) of $\vee$ is the key structural property. The argument is extended to full **EREQ** by observing that projection $\exists(f)$ of t-regexes cannot introduce an unbounded number of new regexes (up to ACI of $\vee$): the additional new case (not present in **ERE**) is $States(\circ_{\exists^i}(f_1) \vee \circ_{\exists^i}(f_0))$ that can introduce new disjunctions in leaves, but preserves rank. Observe that $\circ_{\exists^i}(f_1) \vee \circ_{\exists^i}(f_0) = \circ_{\exists^i}(f_1 \vee f_0)$ when $\exists$ is propagated over $\vee$ in regexes, which the implementation enforces as an invariant. This extension to **EREQ** is also formalized in Lean and included in the artifact.

7.5.1 Correctness. The statement is $Null(\delta^*(R)) \Leftrightarrow \mathcal{L}_{\sharp R}(R) \neq \emptyset$ that builds on Theorem 2 and termination. *Soundness* ($\Rightarrow$) requires that if $q \in Leaves(f)$ then $\exists a(f[a] = q)$, which is built-in into the definition of $Leaves(f)$. By keeping t-regexes *clean* as an invariant, *all* leaves are reachable. *So soundness requires $\mathcal{A}$ to be decidable. Completeness* ($\Leftarrow$) requires termination *but does not depend on $\mathcal{A}$ being decidable.* Theorem 1 then implies for all $\varphi \in$ **wMSO** that φ is satisfiable $\Leftrightarrow Null(\delta^*(\ulcorner\varphi\urcorner))$.

7.5.2 Negation Normal Form and $\exists$ Propagation. In the general case, leaves of all nodes are maintained in Negation Normal Form (NNF) by using de Morgan's laws and $\exists$ is (aggressively) propagated over disjunction (as well as concatenation and loops) and simplified according to the laws in Section 5.4. In particular, $\exists(R_1 \vee R_2)$ rewrites to $\exists(R_1) \vee \exists(R_2)$ if $R_1 \vee R_2$ is already simplified.

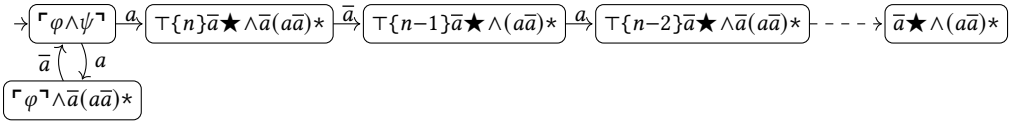
7.5.3 DNF-mode. In *DNF-mode* of the algorithm, $States(f) = \bigcup\{States_{\text{DNF}}(\ell) \mid \ell \in Leaves(f)\}$, where each leaf is translated into a top-level Disjunctive Normal Form, as a union of regexes where each member of the union becomes a separate state. DNF is applied at the top-level only.

$$States_{\text{DNF}}(\ell) \overset{\text{DEF}}{=} \begin{cases} States_{\text{DNF}}(R_1 \wedge R_3) \cup States_{\text{DNF}}(R_2 \wedge R_3), & \textbf{if } \ell = (R_1 \vee R_2) \wedge R_3; \\ States_{\text{DNF}}(R_1) \cup States_{\text{DNF}}(R_2), & \textbf{else if } \ell = R_1 \vee R_2; \\ \{\ell\}, & \textbf{otherwise}. \end{cases}$$

Example 7.4 (distance_n). This example illustrates DNF-mode when it matters. Consider φ and ψ in **wMSO** where $a \in \mathcal{A}$ and $n \geq 0$ is a parameter. Let $\overline{a}(x_0 + k)$ below stand for $\exists x_1(x_1 = x_0 + k \wedge \overline{a}(x_1))$. Assume that both a and $\overline{a}$ are satisfiable.

$$\begin{array}{lll} \psi = \forall x_0((a(x_0) \Leftrightarrow \overline{a}(x_0+1)) \wedge (x_0{=}0 \Rightarrow a(x_0))) & \rightsquigarrow & \ulcorner\psi\urcorner \equiv (a\overline{a})* \\ \varphi = \exists x_0(a(x_0) \wedge \overline{a}(x_0+n+1)) & \rightsquigarrow & \ulcorner\varphi\urcorner \equiv \bigstar a \top\{n\}\overline{a}\bigstar \end{array}$$

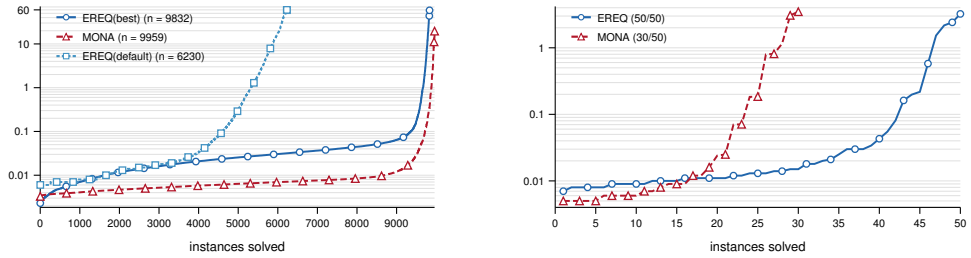
So in $\varphi \wedge \psi$, a and $\overline{a}$ must appear at distance $n+1$ from each other while alternating. Thus, $\varphi \wedge \psi$ is satisfiable iff n is even. This symbolic NFA depicts $\delta^*(\ulcorner\varphi \wedge \psi\urcorner)$ in DNF-mode when n is odd:



Note that $\delta(\overline{a}\bigstar \wedge (a\overline{a})*) = \bot$. If not run in DNF-mode then $|\delta^*(\ulcorner\varphi \wedge \psi\urcorner)| \approx 2^n$ for all odd n. ⊠

8 Implementation and Evaluation

We have developed a prototype implementation, **EREQ**-solver, in Rust, and we plan to make the implementation publicly available on GitHub. The implementation supports M2L-str formulas in the style of MONA [Henriksen et al. 1995], which we translate to **EREQ** using the translation from Section 6. MONA is the current state-of-the-art tool for deciding M2L-str formulas. We compare the performance of our implementation against MONA on the M2L-str formulas found in the



(a) 10048 AutomatArk M2L-str benchmarks (sec). (b) 50 LTL-finite benchmarks [De Wulf et al. 2008] (sec).

Fig. 2. Cactus plots comparing **EREQ** and MONA with a 60s timeout. Each point (x, y) means x instances were solved in $\leq y$ seconds. (b) combines `counter_n`, `lift_n`, and `szymanski_n` from AutomatArk.

AutomatArk [D'Antoni 2025] set of benchmarks, that consist of roughly 10000 M2L-str benchmarks. Since the benchmarks are targeting classical **wMSO**, they are not ideal for **EREQ** because they do not involve concatenation or loops, but they do provide a baseline. We evaluated the full set of benchmarks with a 60sec timeout, out of which **EREQ** timed out on 250 and MONA timed out on 89. Each tool times out on a different class of problems. In the current state, **EREQ**-solver lacks many optimizations and search strategies, but the evaluation results of some benchmark sets are very encouraging and indicate that the regex level rewrites do pay off, as we discuss below. For all benchmarks where both tools terminate, the answers (sat/unsat) match.

In Figure 2b, **EREQ** solves all 50 LTL-finite instances while MONA solves 30; **EREQ** outperforms MONA on every family except `szymanski_n`, where MONA is faster on small instances. Figure 2a shows the full AutomatArk M2L-str suite (10048 instances), where MONA—a mature and highly optimized tool—solves the vast majority. Performance of **EREQ** on these benchmarks is sensitive to the choice of solving strategy: the default configuration solves over half the instances, and the best configuration we found reduces timeouts to around 250. This highlights both the potential and the current limitations of our approach on randomly generated benchmarks.

The core of the implementation is the δ-construction for **EREQ** described in Section 7.2, which is used to build a symbolic automaton on-the-fly through lazy δ-unfolding. Both, propositions p_i and predicates in $\mathcal{A}$ are implemented using Binary Decision Diagrams (BDDs) for the benchmarks, where the top-level free second-order variables in benchmarks correspond to propositions in $\mathcal{A}$, e.g., for Σ=ASCII there would be seven such propositions (as bit-predicates). BDDs provide efficient logical operations needed for the derivative construction, and allow us to operate over an unbounded N in $\mathcal{A}_N$, which is crucial for M2L-str formulas, where the rank N constantly grows and shrinks as new bound variables are introduced and go out of scope through quantifiers.

Relationship Between Lean Formalization and Rust Implementation. The Lean formalization and the Rust implementation share the same foundational design: an effective Boolean algebra for character predicates and an ITE-tree (transition regex) structure for symbolic derivatives. The Lean development proves correctness of the derivative laws for all **EREQ** operators (Theorem 2), the **wMSO**-to-**EREQ** embedding (Theorem 1), and finiteness of the derivative closure [Zhuchko et al. 2025]. The Rust solver implements the same semantic definitions and derivative rules.

The gap lies in the implementation layer: the Rust solver additionally includes structural sharing (hash-consed nodes), reverse-derivative caching for equivalence detection, selectable normal forms (DNF/NNF), and metadata-driven heuristics for controlling state growth. These engineering refinements, critical for practical performance, are not mechanized in Lean. The full end-to-end

correctness of the decision procedure in the Rust implementation — including the interplay of rewrite rules, memoization, and search strategies — is therefore not formally verified. Extending the formalization to cover these aspects is possible in principle but would substantially enlarge the proof effort and is left for future work.

In comparison, [Traytel and Nipkow 2013] take an alternative approach: they formalize their decision procedure for **wMSO** entirely in Isabelle and use Isabelle's code generation facility to automatically extract a purely functional, verified program. This yields end-to-end correctness by construction but trades off implementation control: the extracted code inherits the functional style and data representations of the proof assistant, limiting the ability to apply low-level performance optimizations such as hash-consing, BDD-based predicate representations, or imperative graph structures. Our approach deliberately separates the verified core (Lean) from the optimized solver (Rust), gaining flexibility to employ implementation-level techniques that are essential for scaling to the full set of MONA benchmarks.

8.1 Optimizations for EREQ

Our implementation incorporates several optimizations to improve performance. Since many M2L-str problems have an enormous search space, it is crucial to keep the expressions as small as possible during the derivative construction. MONA addresses this issue by applying Myhill-Nerode minimization to the underlying automata after every product and projection operation [Klarlund et al. 2002]. In contrast, we apply simplifications based on logical equivalences, detected using three main rewrite strategies:

Syntactic Rewrites: Example: $(R \vee S) \wedge S \equiv S$. (Complexity: $O(1)$.)
Reverse Derivative Lookup: $\delta(R) = \delta(S) \wedge \mathit{Null}(R) = \mathit{Null}(S) \Rightarrow R \equiv S$.
Language Emptiness: $\mathcal{L}(R) = \emptyset \Rightarrow R \equiv \bot$. (Complexity: non-elementary.)

The need for these optimizations comes from the fact that any language in **EREQ** admits infinitely many equivalent syntactic representations. For example, $R \vee \bot \equiv R$ for all R, but if left unchecked, the derivative construction may produce expressions that grow without bound due to redundant sub-expressions. This growth can be mitigated in two ways: by maintaining normal forms for the underlying representations of $\vee$ and $\wedge$, and by checking full equivalence between states. The latter is prohibitively expensive for large expressions, so we instead focus on lightweight rewrite rules that can be applied frequently during derivative construction. Several such rewrites were shown in examples in Section 7.3.

Syntactic Rewrites. We implement a number of syntactic rewrites that are applied whenever a new expression is constructed. These include simple rules like eliminating redundant $\bigstar$ and $\bot$ expressions, maintaining various normal forms, and eliminating double negations. We also implement more complex rewrites that use logical equivalences. The most notable reason for syntactic rewrites is that they can be applied in constant time using reference equality, regardless of the size of the expressions involved, and thus can be used to simplify expressions, that would otherwise require exploring a state graph consisting of millions of nodes.

Reverse Derivative Lookup. We implement another relatively cheap equivalence checking mechanism that can determine if two expressions are equivalent. This is done by maintaining a lookup from derivative results to their source expressions. Whenever a derivative is computed, we check if the result has been seen before. If so, and if the nullability of the two expressions also matches, we can conclude that the two expressions are equivalent, and we can rewrite the new expression to the existing one. This strategy is very effective in practice, and detects many common equivalences bottom-up during the derivative construction, often hundreds of thousands throughout a single benchmark experiment.

Table 1. Benchmark comparison of lift_n

lift_n for n :	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
EREQ	: 9ms	11ms	19ms	49ms	310ms	317ms	461ms	1512ms
MONA	: 23ms	26ms	1195ms	**Out of memory at 128 GB**				

Language Emptiness. We also implement a more thorough rewrite procedure for **EREQ** expressions described in Section 7.5, which can be used to detect if an expression is equivalent to $\bot$. This can be used to detect many useful properties through emptiness of syntactic constructions, e.g., whether $\mathcal{L}(S)$ contains $\mathcal{L}(R)$ by checking the emptiness of $R \wedge \neg S$, which, if empty, allows us rewrite $R \wedge S$ to R. This procedure is more expensive than the previous two strategies, so we only apply it up to a fixed syntactic size limit, or during the initial construction of the **wMSO** to **EREQ** translation, where it can eliminate large unsatisfiable sub-expressions.

Specialized Rewrites. There are several rewrites that enable elimination of $\wedge$ and $\neg$ in many common instances. Some such laws are listed below. Let $\alpha, \beta, \gamma \in \mathcal{A}_N$.

(1) $\neg(\bigstar \cdot \alpha \cdot \bigstar) \equiv \overline{\alpha}*$ because if no element may model α then all elements must model $\overline{\alpha}$
(2) $\neg(\alpha \cdot \bigstar) \equiv \overline{\alpha} \cdot \bigstar \vee \varepsilon$ because if the word does not start with α then it is either empty or it starts with $\overline{\alpha}$. In particular, $\neg(\alpha \cdot \bigstar) \wedge R \equiv \overline{\alpha} \cdot \bigstar \wedge R$ if R is not nullable.
(3) $\overline{\gamma}* \cdot \gamma \cdot \overline{\gamma}* \wedge R_1 \cdot (\gamma \sqcap \beta) \cdot R_2 \equiv (R_1 \wedge \overline{\gamma}*) \cdot (\gamma \sqcap \beta) \cdot (R_2 \wedge \overline{\gamma}*)$ because γ is only allowed in a single position by the first conjunct and then β must also hold in that position.

Laws (1) and (2) reflect that it is useful to propagate complement into $\mathcal{A}$. Law (3) is used in the case when $\gamma = \mathsf{p}_i$ to reduce (and often eliminate) use of $\wedge$ that arises from singleton constraints.

8.2 Experiments

Our derivative-based approach exposes multiple solving strategies (normal forms, DNF-mode, rewrite rules) that can be tuned to handle formulas where MONA runs out of memory or hits BDD limits. The AutomatArk benchmark suite [D'Antoni 2025] contains several M2L-str benchmark families that illustrate this. We include all 50 benchmarks from the LTL-finite category [De Wulf et al. 2008] (following the benchmark selection in [D'Antoni and Veanes 2017]), comprising the three families `lift_n`, `counter_n`, and `szymanski_n`. Our implementation is able to solve all of these benchmarks. Figure 2b shows a combined cactus plot for all 50 LTL-finite instances, where **EREQ** solves all 50 while MONA solves only 30 out of 50.

The `lift_n` benchmarks encode the problem of lifting a binary relation to sets of size n [D'Antoni 2025]. The state space of these benchmarks grows very quickly with n, and MONA is only able to solve up to `lift_4` without running out of memory. In contrast, our **EREQ** implementation is able to solve up to `lift_9`, with a significant performance advantage over MONA (see Table 1). MONA runs out of memory at 128GBs on `lift_5` and beyond, while **EREQ** is able to solve `lift_9` in just over 1.5 seconds, using only 0.17GBs of memory.

The `counter_n` benchmarks encode the problem of counting up to 2^n using n bits [D'Antoni 2025]. These benchmarks also have a large state space that grows exponentially with n. MONA is able to solve up to `counter_11` before running into memory issues, while our **EREQ** implementation is able to solve up to `counter_16` (see Table 2). **EREQ** shows a significant performance advantage over MONA on these benchmarks as well, with MONA taking over 3 seconds to solve `counter_11`, while **EREQ** solves it in just 15ms. Here, MONA does not explicitly run out of memory, but reaches a BDD size limit of 16,777,216 nodes at `counter_12` and beyond.

DNF-mode. We consider the scenario described in Section 7.5.3, where the traditional DFA-based approach to deciding M2L-str formulas suffers from an exponential blowup. The benchmark encodes two simple properties: (1): the bits must alternate between 0 and 1 for each position, and (2): the

Table 2. Benchmark comparison of counter_n where $\lightning$ = BDD too large (>16,777,216 nodes).

counter_n for n :	**2**	**4**	**6**	**8**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
EREQ	: 6ms	7ms	7ms	10ms	14ms	15ms	17ms	21ms	23ms	27ms	31ms
MONA	: 23ms	25ms	31ms	88ms	864ms	3293ms	$\lightning$				

Table 3. Benchmark comparison of distance_n (Section 7.5.3) where $\lightning$ = BDD too large (>16,777,216 nodes).

distance_n for n :	**5**	**6**	**9**	**10**	**15**	**16**	**17**	**18**	**19**	**20**
EREQ	: 2ms	2ms	10ms	10ms	205ms	261ms	360ms	522ms	638ms	722ms
MONA	: 20ms	20ms	23ms	24ms	226ms	469ms	1227ms	3154ms	8753ms	$\lightning$

distance between two positions is exactly n. This benchmark is interesting because the underlying NFA for this formula is small and only has $O(n)$ states, while the corresponding DFA has $O(2^n)$ states. As a result, MONA struggles with this benchmark for large n, while **EREQ**-solver is able to solve it more efficiently using the DNF-mode described in Section 7.5.3. We compare the time of **EREQ** and MONA for n ranging from 2 to 20, as shown in Table 3, where MONA reaches its BDD size limit at $n = 20$, while **EREQ** is able to solve the benchmark in under a second.

Normal Forms. A key aspect of **EREQ**-solver is that it can use different normal forms in the leaves of t-regexes, which can have a significant impact on performance. We have experimented with both Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF) representations for leaves, as well as de Morgan's laws to permute negations in or out of leaves.

We found many randomly generated benchmarks that take over a minute to solve in one normal form can be solved in under a second using another normal form. For example, the `lift_n` and `counter_n` experiments perform much worse using DNF-mode. `lift_4` takes around 200ms using DNF, compared to just 19ms without DNF. One reason for this difference is that we have not defined syntactic rewrites for large spread out expressions, which leads to suboptimal results. We have not found a single *best* normal form for all M2L-str benchmarks, and we leave a more thorough investigation of this aspect to future work. The ability to easily swap normal forms is a key advantage of our approach, as it allows us to experiment with different solving strategies, as well as running multiple strategies in parallel without changing the core derivative construction.

9 Future Work

In our formalization we aim to establish correctness of the key rewrite rules used in the Rust implementation and to connect the Lean and Rust implementations for cross-validation. There are many search strategies yet to be investigated and evaluated, one such is use of a normal form where the DNF-mode is extened so that, for any leaf $R{\cdot}S$ or $\exists(R{\cdot}S)$, right-concatenation with S is distributed over the top-level DNF of R, that corresponds to exposing partial derivatives [Antimirov 1996]. We also need to develop a comprehensive benchmark suite for **EREQ** since existing benchmarks for **wMSO** are too limited – concatenation and loops, and modulo $\mathcal{A}$, are not expressible there. MONA additionally implements a decision procedure for WS2S – a regular logic over trees. How to extend symbolic derivatives to handle tree structures is an open problem. There is prior work on derivative-based constructions for tree languages [Attou et al. 2021], indicating that such generalizations are possible in principle. Another non-trivial extension is to extend the framework to infinite words (S1S semantics) and ω-regularity [Veanes et al. 2025] as well as infinite trees (S2S semantics). There is a long list of applications of MONA such as hardware verification, program synthesis and protocol verification [Henriksen et al. 1995]. We plan to adapt some of these applications to **EREQ** and also explore applications of **EREQ** in the context of modern AI with the focus on use of **EREQ** in agent level reasoning in agentic workflows.

Acknowledgements

E. Zhuchko was supported by the Estonian Research Council grant PRG1210. I. E. Varatalu interned at Microsoft Research, Redmond, during summer 2025 while carrying out the work discussed in Section 3. This work was conducted in close collaboration with Immad Naseer (Azure Storage), whose help in providing real log data was crucial to the research. We also thank all the anonymous reviewers for their detailed and valuable comments and numerous helpful suggestions.

Artifact Availability

The artifact [Varatalu 2026] covers all the evaluation results reported in Section 8. The artifact [Zhuchko 2026] includes all the correctness proofs developed in Lean, in particular including Theorem 1, Theorem 2, and Theorem 3.

References

Valentin Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science* 155 (1996), 291–319. doi:10.1007/3-540-59042-0_96

Samira Attou, Ludovic Mignot, and Djelloul Ziadi. 2021. Bottom-Up derivatives of tree expressions. *RAIRO-Theor. Inf. Appl.* 55, 4 (2021), 21 pages. doi:10.1051/ita/2021008

Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *JACM* 11 (1964), 481–494. doi:10.1145/321239.321249

Loris D'Antoni. 2025. *AutomatArk*. `https://github.com/lorisdanto/automatark`.

Loris D'Antoni and Margus Veanes. 2017. Monadic second-order logic on finite sequences. *ACM SIGPLAN Notices – POPL'17* 52, 1 (2017), 232–245.

Loris D'Antoni and Margus Veanes. 2021. Automata Modulo Theories. *Commun. ACM* 64, 5 (May 2021), 86–95. doi:10.1145/3419404

Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. 2008. Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008) (LNCS, Vol. 4963)*. Springer, 63–77.

Javier Esparza, Jan Křetínský, and Salomon Sickert. 2020. A Unified Translation of Linear Temporal Logic to ω-Automata. *JACM* 67, 6, Article 33 (Oct. 2020), 61 pages. doi:10.1145/3417995

Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. 2017a. Lazy Automata Techniques for WS1S. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–425.

Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. 2015. Nested Antichains for WS1S. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 658–674.

Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. 2017b. *Gaston: WS1S/WS2S solver*. `https://www.fit.vut.cz/research/group/verifit/tools/gaston/`.

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

Peter Kelb, Tiziana Margaria, Michael Mendier, and Claudia Gsottberger. 1997. Mosel: A sound and efficient tool for M2L(Str). In *Computer Aided Verification*, Orna Grumberg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 448–451. doi:10.1007/3-540-63166-6_45

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

A. R. Meyer and L. J. Stockmeyer. 1972. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT'72)*. IEEE, Piscataway, NJ, USA, 125–129.

Scott Owens, John H. Reppy, and Aaron Turon. 2009. Regular-expression Derivatives Re-examined. *J. Funct. Program.* 19, 2 (2009), 173–190. doi:10.1017/S0956796808007090

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In *PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual Event). ACM, New York, NY, USA, 620–635. doi:10.1145/3453483.3454066

Wolfgang Thomas. 1996. Languages, Automata, and Logic. In *Handbook of Formal Languages*. Springer, 389–455.

Dmitriy Traytel. 2015. A Coalgebraic Decision Procedure for WS1S. In *24th EACSL Annual Conference on Computer Science Logic (CSL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 41)*, Stephan Kreutzer (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 487–503. doi:10.4230/LIPIcs.CSL.2015.487

Dmitriy Traytel and Tobias Nipkow. 2013. Verified decision procedures for MSO on words based on derivatives of regular expressions. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 3–12. doi:10.1145/2500365.2500612

Ian Erik Varatalu. 2026. *ieviev/pldi26-ereq: PLDI26 version.* doi:10.5281/zenodo.19072417

Margus Veanes, Thomas Ball, Gabriel Ebner, and Ekaterina Zhuchko. 2025. Symbolic Automata: Omega-Regularity Modulo Theories. *Proc. ACM Program. Lang.* 9, POPL, Article 2 (Jan. 2025), 34 pages. doi:10.1145/3704838

Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. 2010. Symbolic Automata Constraint Solving. In *LPAR-17 (LNCS)*. 640–654.

Ekaterina Zhuchko. 2026. *ezhuchko/ereq-derivatives: PLDI26 version.* doi:10.5281/zenodo.19067931

Ekaterina Zhuchko, Hendrik Maarand, Margus Veanes, and Gabriel Ebner. 2025. Finiteness of Symbolic Derivatives in Lean. In *16th International Conference on Interactive Theorem Proving (ITP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 352)*, Yannick Forster and Chantal Keller (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:19. doi:10.4230/LIPIcs.ITP.2025.16

Ekaterina Zhuchko, Margus Veanes, and Gabriel Ebner. 2024. Lean Formalization of Extended Regular Expression Matching with Lookarounds. In *13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'24)* (London, UK). ACM, New York, NY, USA, 118–131. doi:10.1145/3636501.3636959