Beyond Classical Regexes: Symbolic-Derivative-Based Decision Procedures for Extended Regular Expressions

Margus Veanes

Microsoft Research

TU Wien Seminar • Vienna 2026, March 23

Outline

Part I: Foundations — Symbolic Derivatives & Transition Terms

Part II: ERE# (CAV'25) — Boolean Closed Extended Regexes

- Intersection, complement, lookarounds
- Decision procedures via symbolic derivatives
- SMT benchmark results

Part III: EREQ (PLDI'26) — Regexes with Quantifiers

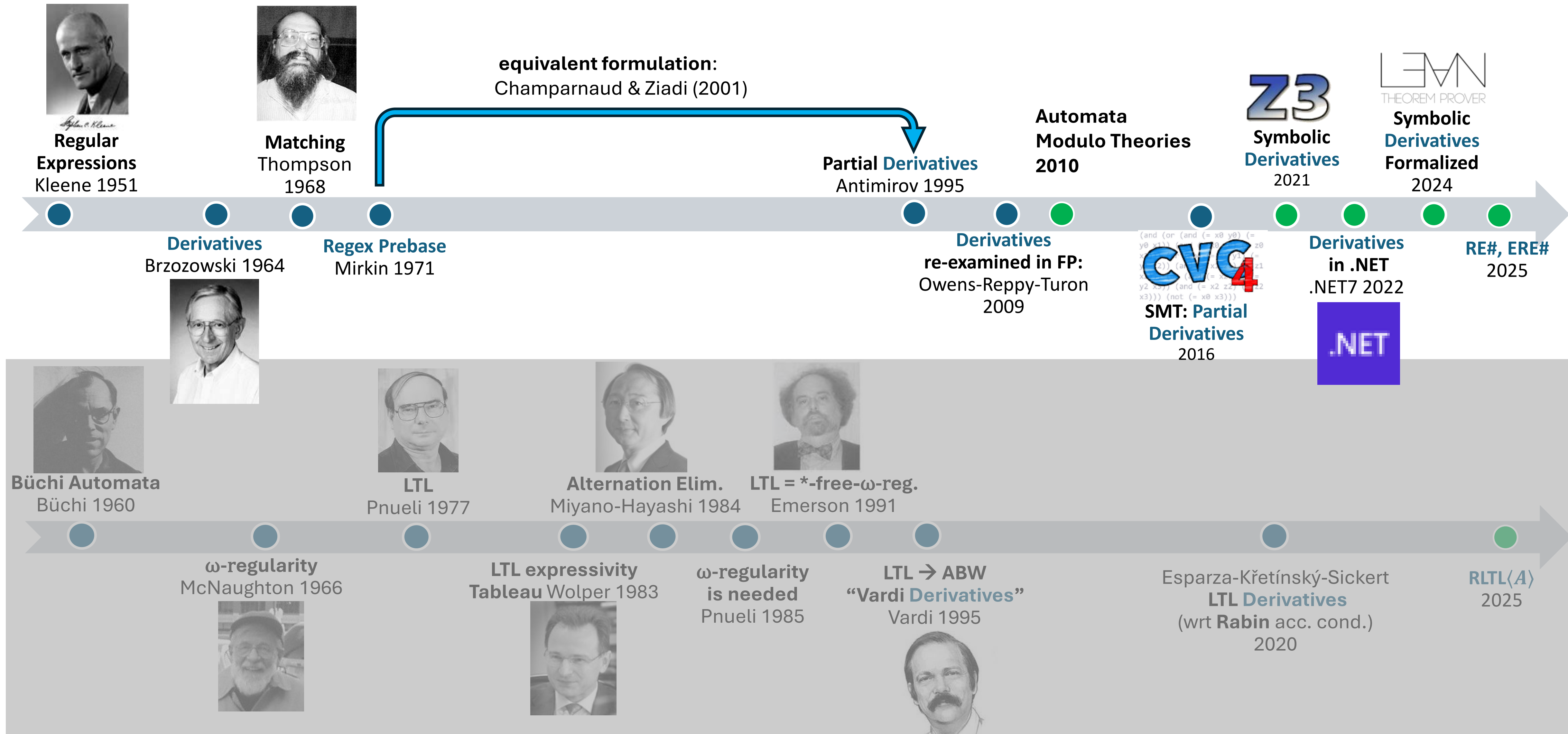
- Embedding wMSO into EREQ
- Derivatives distribute over quantifiers
- Incremental quantifier elimination

Part IV: Unified Perspective & Future Directions

Part I: Foundations

Symbolic Derivatives & Transition Terms

Regular Expressions and Derivatives



What are Symbolic Derivatives?

Classical Brzozowski derivative: $\boldsymbol{L}(D_{\mathtt{a}}(R)) = \{ w \mid \mathtt{a}w \in \boldsymbol{L}(R) \}$

- Computes one derivative per concrete character `a`

Symbolic derivative: $\delta(R)$ = transition term (or transition regex -- t-regex)

- A single computation captures ALL possible one-step transitions
- Result is an if-then-else term: $t = ite(\alpha, t_{then}, t_{else})$
- Conditions α are *predicates*, e.g., lower case letter: `[a-z]`
- Leaves are regexes — the possible "next states"

Key insight: $\delta(R)[a] = D_a(R)$ for any concrete character a

- One symbolic step replaces $|\Sigma|$ concrete steps

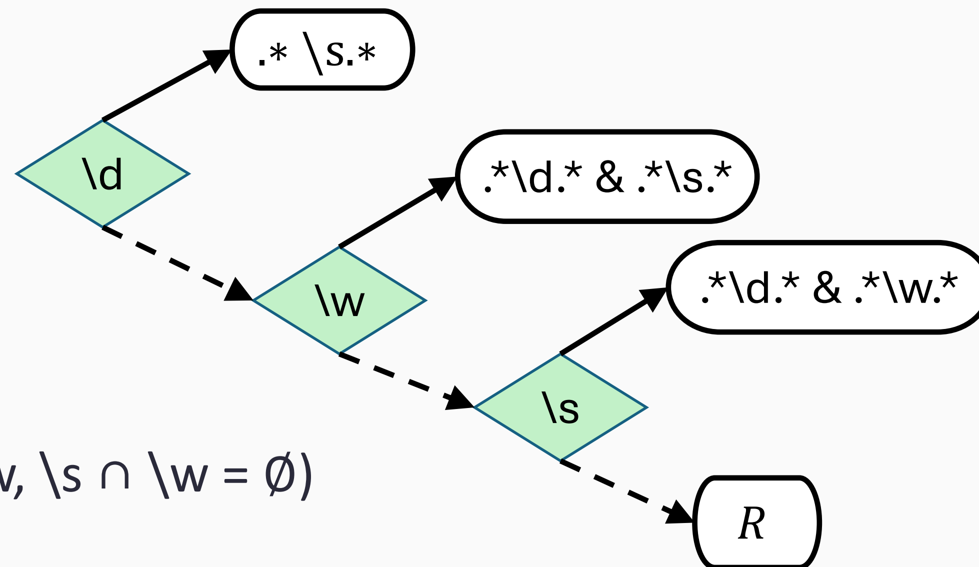
Transition Terms: The Key Data Structure

A transition term is a binary decision tree over predicates:

ite(α, R_{then}, R_{else}) —
"if character satisfies α then go to R_{then} else R_{else}"

Example: R = .*\d.* & .*\w.* & .*\s.*

δ(R) = ite(\d, .*\s.*,
ite(\w, .*\d.* & .*\s.*,
ite(\s, .*\d.* & .*\w.*, R)))



Cleaning uses the predicate algebra (\d ⊆ \w, \s ∩ \w = ∅)
to prune unreachable branches

Core Principle: Operations Propagate Through Transition Terms

Boolean operations propagate to the LEAVES of transition terms:

$$\neg\, ite(\alpha, t_1, t_2) \;=\; ite(\alpha, \neg t_1, \neg t_2)$$

$$ite(\alpha, t_1, t_2) \,\&\, ite(\alpha, s_1, s_2) \;=\; ite(\alpha, t_1 \,\&\, s_1, t_2 \,\&\, s_2)$$

This enables LAZY propagation of derivatives:

$\delta(\neg R) = \neg\delta(R)$ — complement propagates

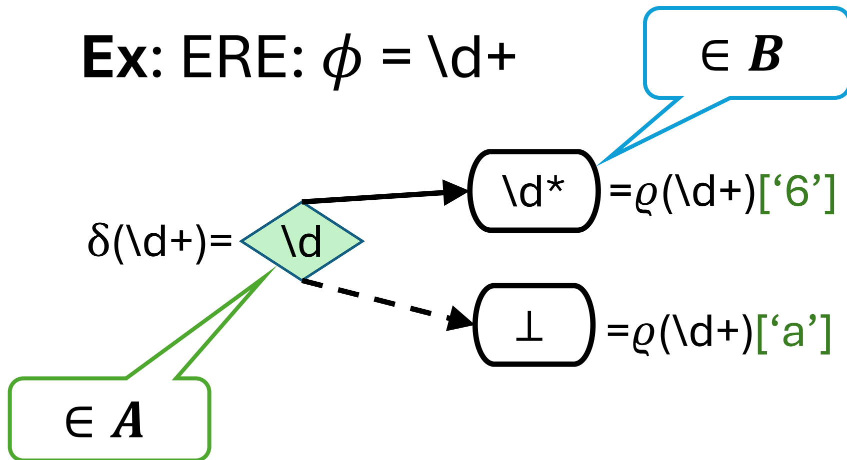
$\delta(R \,\&\, S) = \delta(R) \,\&\, \delta(S)$ — intersection propagates

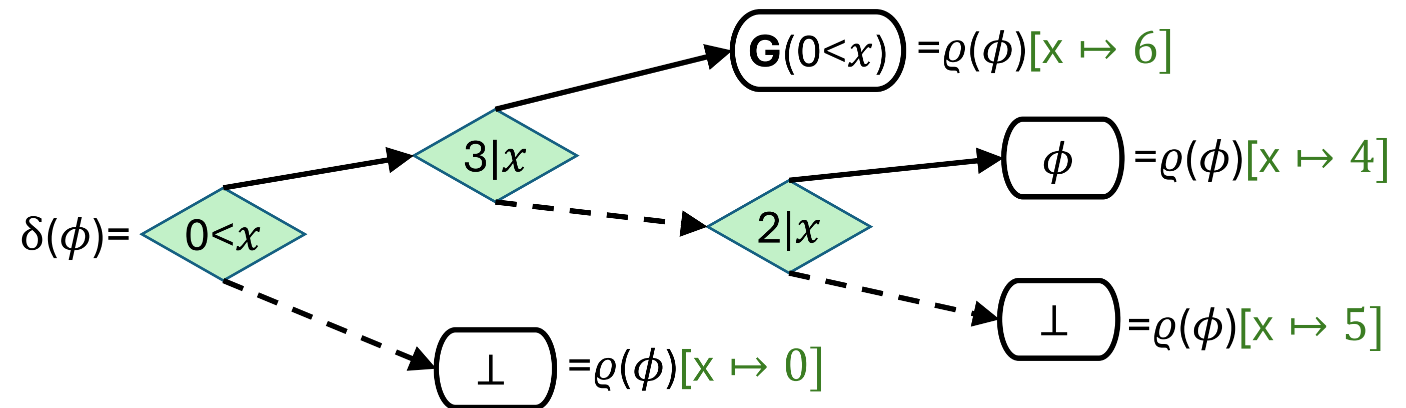
→ No up-front automaton determinization needed!

Any operation that commutes with ite can be added to the derivative framework

Symbolic Derivative Function $\boldsymbol{\delta}$ for a Logic $\boldsymbol{B}_{\boldsymbol{A}}$

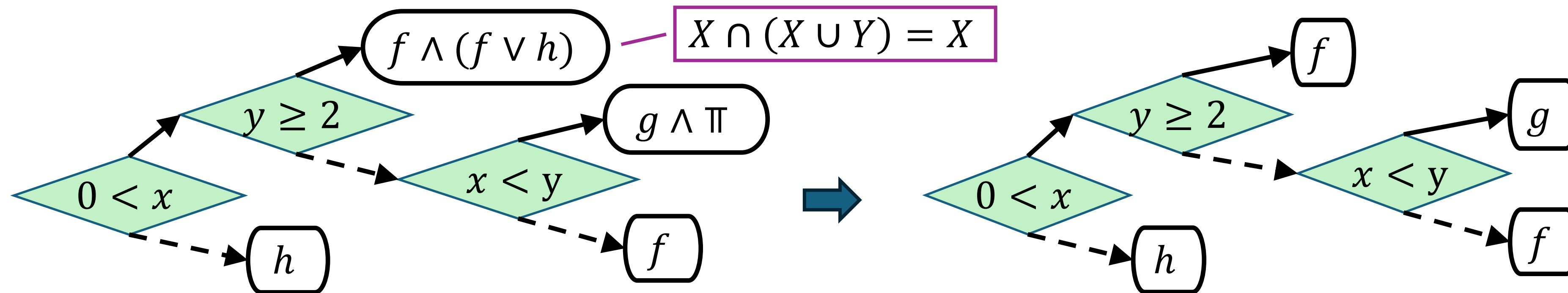
Symbolic derivative $\boldsymbol{\delta}(\phi)$ is a ***transition term s.t.*** $\boldsymbol{\delta}(\phi)[a] = D_a(\phi)$



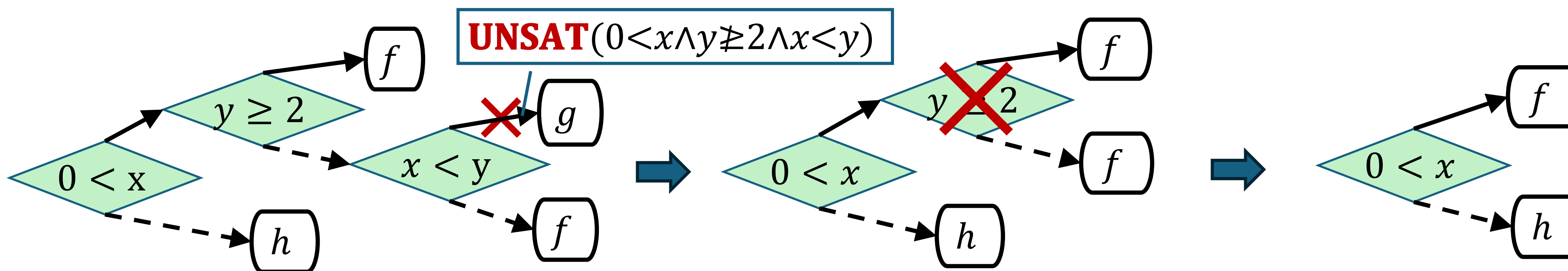


Transition Term $\mathbf{TTerm}\langle A, B_A \rangle$ Simplifications

- **Boolean Laws** of $\boldsymbol{B}$ (bot ⊥, top ⊤ and Boolean operators ~,∨,∧)
  - If *top* of $\boldsymbol{A}$ is ⊤ and $\boldsymbol{B}$ = **ERE** then ⊤ = ⊤*
  - **ACI** of ∧ and ∨, **distributivity**, ⊤ is **unit** of ∧, **de Morgan**, etc...

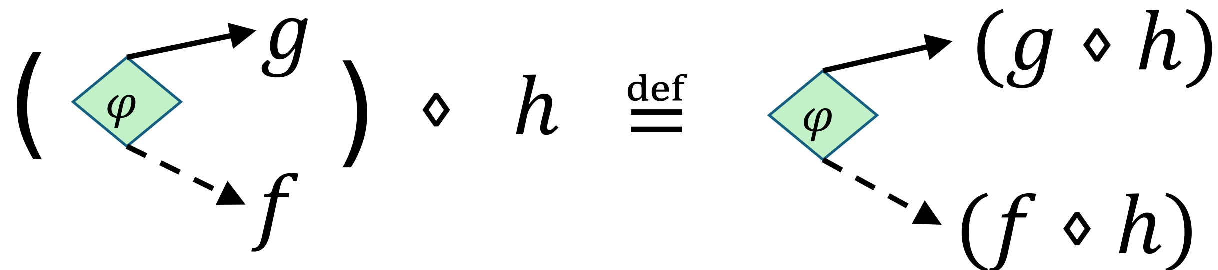


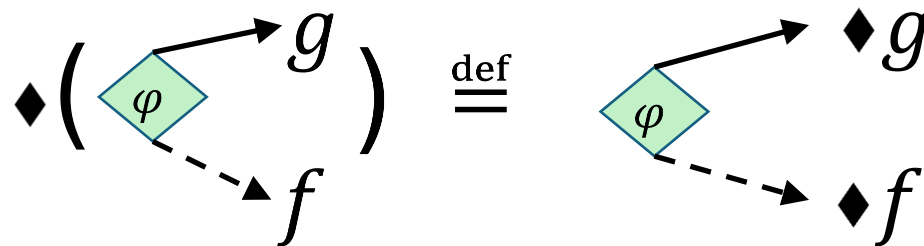
- **Cleaning** modulo $\boldsymbol{A}$, say x, y: $\mathbb{Z}$



Transition Term Operations

All operations $\diamond : \boldsymbol{B} \times \boldsymbol{B} \to \boldsymbol{B}$ and $\blacklozenge : \boldsymbol{B} \to \boldsymbol{B}$ are **lifted** to $\mathbf{TTerm}\langle \boldsymbol{A}, \boldsymbol{B} \rangle$





Part II: ERE# (CAV'25)

Boolean Closed Extended Regular Expressions

with Ian Erik Varatalu, Ekaterina Zhuchko, Juhan Ernits

RE#: Extending Regex Matcher with Boolean Operations

RE# (POPL'25) is a nonbacktracking regex matcher (.NET 7+) supporting:

- Intersection (&), Complement (~), Restricted lookarounds
- Currently world's fastest regex matcher
  - front page of dotnet reddit: https://www.reddit.com/r/dotnet/comments/1rf5jk2/re_how_we_built_the_worlds_fastest_regex_engine/
  - blog post: https://iev.ee/blog/symbolic-derivatives-and-the-rust-rewrite-of-resharp/

ERE#: Extending Regex Solver

ERE# = Boolean closure of RE# with **span** semantics

- Span (u, v, s) in word w = uvs: v = match, u/s = context
- $(?<=R_1)R_2(?=R_3)$ — lookbehind + match + lookahead

Normal Form Theorem (proved in Lean):

- Every ERE# expression reduces to (a union of)

$$(?<=R_1)R_2(?=R_3)$$

- Enables emptiness, subsumption, and equivalence checking

Symbolic Derivatives for ERE (w/o lookarounds)

$$R ::= \psi \mid \varepsilon \mid R_1 \diamond R_2 \mid R_1 \cdot R_2 \mid R\{m\} \mid R* \mid {\sim}R$$

$\diamond \in \{|, \&, \oplus\}$

$$\delta(\varepsilon) \stackrel{\text{DEF}}{=} \bot \qquad \delta(R*) \stackrel{\text{DEF}}{=} \delta(R) \cdot R*$$

$$\delta(\psi) \stackrel{\text{DEF}}{=} \mathbf{ite}(\psi, \varepsilon, \bot) \qquad \delta(R\{m\}) \stackrel{\text{DEF}}{=} \delta(R) \cdot R\{m-1\}$$

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

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

$$\delta(R \cdot S) \stackrel{\text{DEF}}{=} \begin{cases} \delta(R) \cdot S \mid \delta(S), & \textbf{if } Null(R); \\ \delta(R) \cdot S, & \textbf{otherwise}. \end{cases}$$

$$Null(\varepsilon) \stackrel{\text{DEF}}{=} \mathbf{true}$$

$$Null(R*) \stackrel{\text{DEF}}{=} \mathbf{true}$$

$$Null(\psi) \stackrel{\text{DEF}}{=} \mathbf{false}$$

$$Null(R\{m\}) \stackrel{\text{DEF}}{=} Null(R)$$

$$Null({\sim}R) \stackrel{\text{DEF}}{=} \neg Null(R)$$

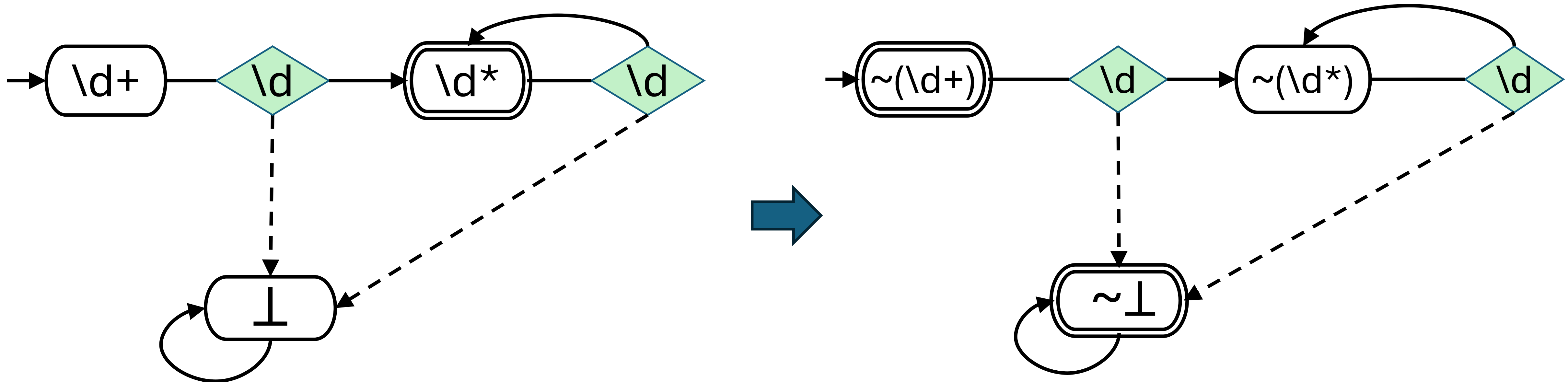
$$Null(R \mid S) \stackrel{\text{DEF}}{=} Null(R) \vee Null(S)$$

$$Null(R \,\&\, S) \stackrel{\text{DEF}}{=} Null(R) \wedge Null(S)$$

$$Null(R \oplus S) \stackrel{\text{DEF}}{=} Null(R) \neq Null(S)$$

$$Null(R \cdot S) \stackrel{\text{DEF}}{=} Null(R) \wedge Null(S)$$

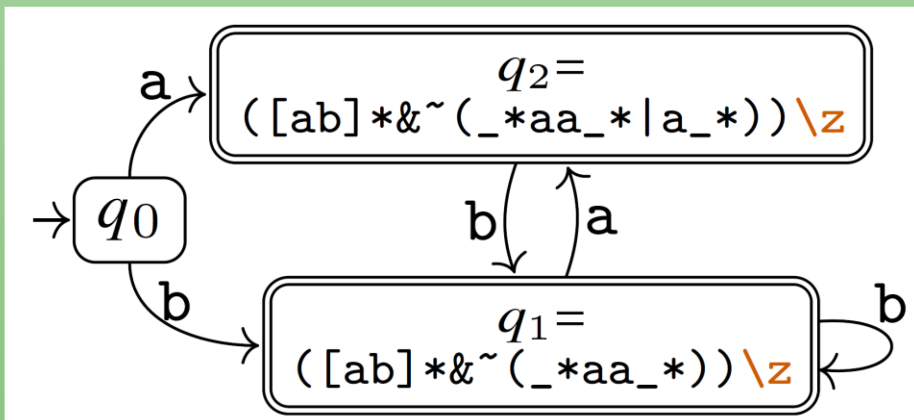
(Incremental) Propagation of Complement



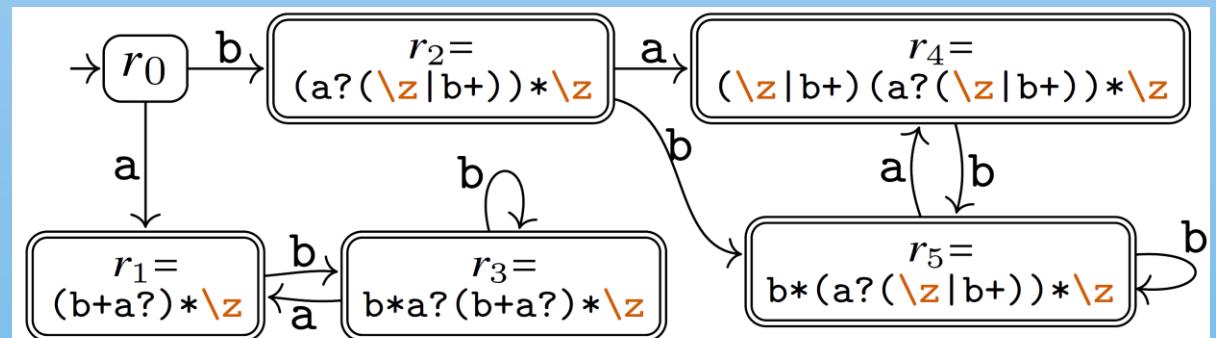
EXAMPLE OF ERE vs RE (with anchors)

Match all nonempty strings over [ab] but without two a’s in a row:

ERE: q_0= \A([ab]+**&~**(_*aa_*))\z



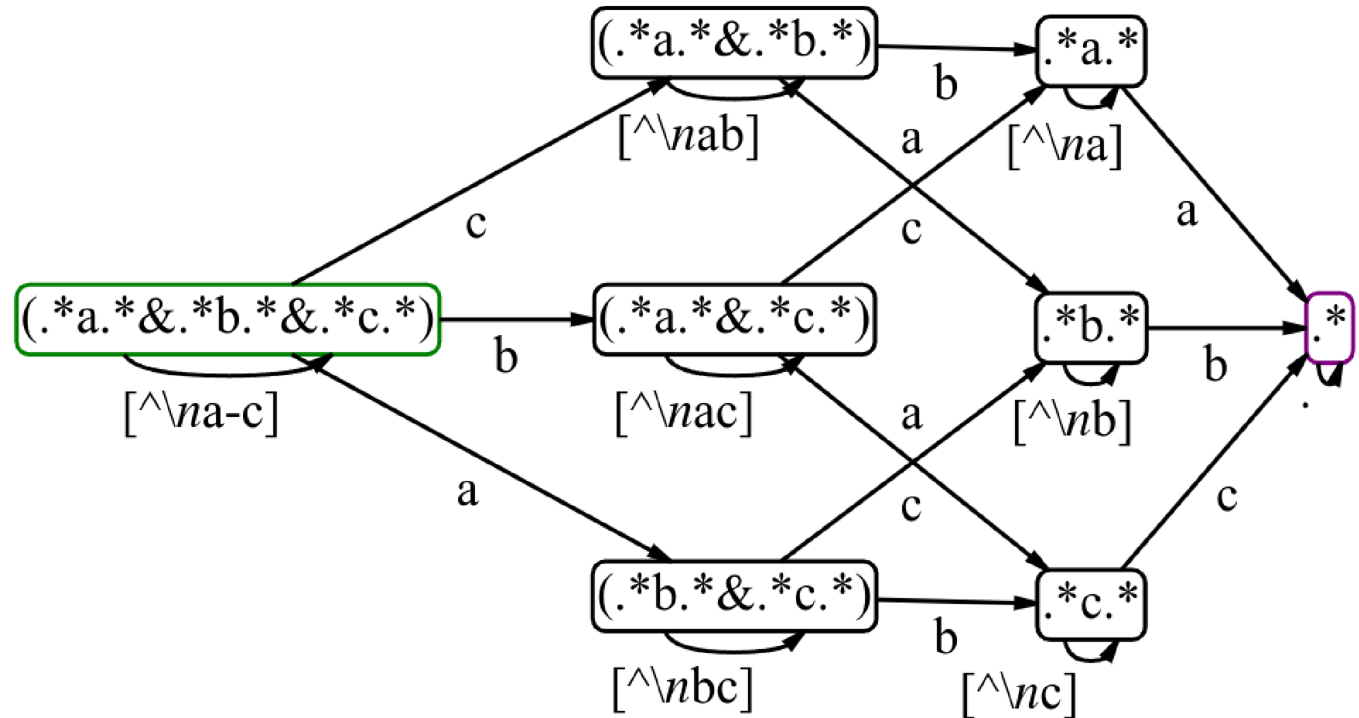
RE: r_0= \A(a(b+a?)*|b(a?(\z|b+))*)\z



UNFOLDING INTO DFA

.*a.***&**.*b.***&**.*c.* →

Lazily compiled



can be compiled into a program that runs in $O(n)$ time
solving all constraints in parallel with one pass over the input string

From matching to **analysis**: **ERE# solver** (in Rust)

- Is R **empty**?
  - .*\d.*&~(.*\w.*) ≡⊥
- Is R **universal**?
  - .*a?{k} ≡ _*
- Does R have "dead code" (in PCRE) ?
  - "may|mayo" $\equiv_{PCRE}$ "may"
- Does R have the same meaning in both POSIX and PCRE?
  - $R_1|R_2 \equiv R_1|R_2$&~(R_1·_*).
- Does one regex **subsume** another regex?
  - (lexer slicer optimizer) [^"\\]+ ⊆ ([^"\\]|\\(["\\/bfnrt]|u[a-fA-F0-9]{4}))+
- Are two regexes **equivalent**?
  - \A([ab]+**&~**(_*aa_*))\z ≡ \A(a(b+a?)*|b(a?(\z|b+))*)\z

Practical Applications of ERE#

Subsumption: Is $L(R_2) \subseteq L(R_1)$?

- Critical for DFA compilation: reuse states, avoid explosion
- Without it: state space can blow up to 2^k

Equivalence:

- Verify regex behavior across PCRE vs POSIX semantics
- Detect unreachable alternatives: may|mayo never matches "mayo" in PCRE

Emptiness:

- Debug impossible patterns: .*\d.* & ~(.*\w.*) is empty

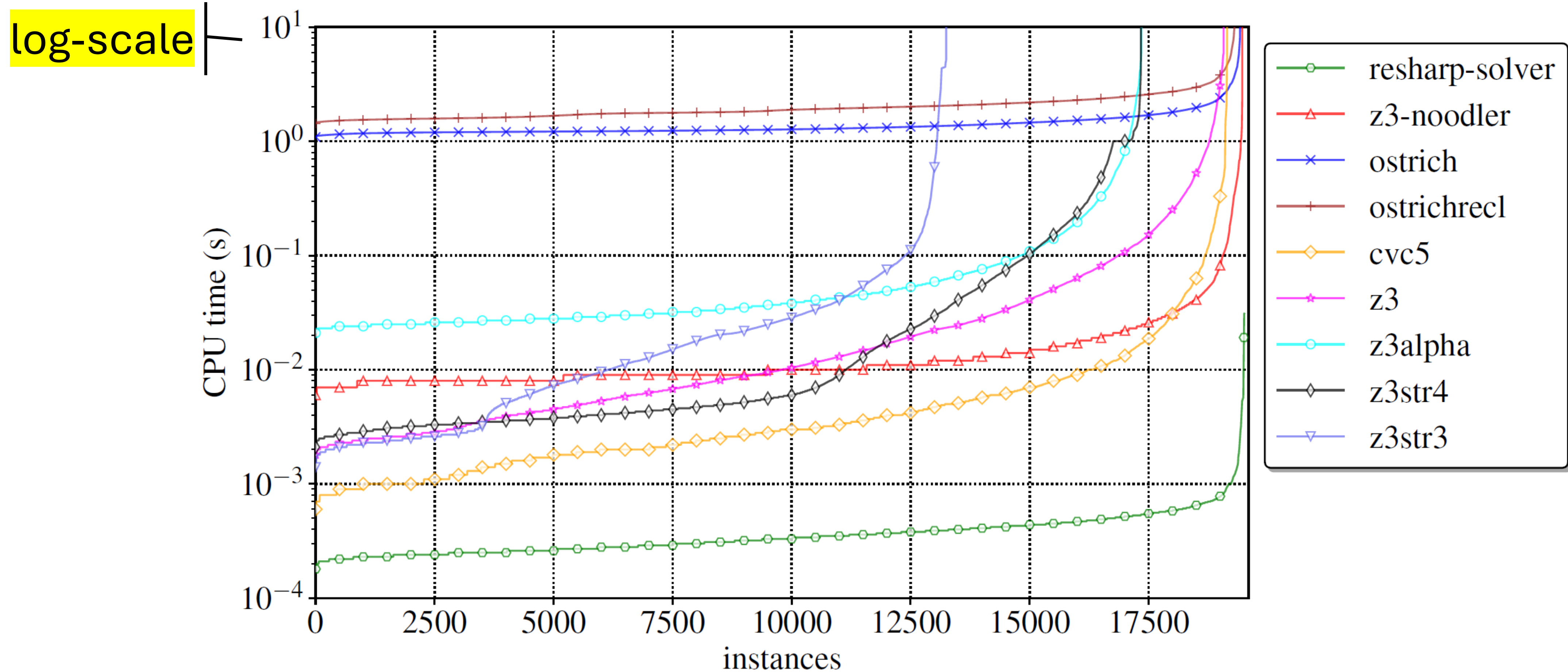
Pre-processing in SMT solvers (Z3)

Industrial application (by **Michał Moskal**) in **LLM Inference with Guidance**

- **Not vaporware: deployed at industry scale**
  - **Supported now in most LLM engines**
- Custom regex engine for lexer (**symbolic derivative-based**)
- For most grammars (e.g., JSON) ~**99.9%** of operations is **lexer-only**
- Slicer optimization: **1ms -> 50μs (avg)**
  - Ex: If [^”\\]+ is subset of prefixes of strings allowed by lexer,
    the whole inside-string slice can be allowed without walking its trie

ℒ([^”\\]+) ⊆ ℒ([^”\\]|\\(["\\/bfnrt]|u[a-fA-F0-9]{4}))+)

SMT RegLan Benchmarks Cactus plot



Total solving time for ERE# of **all** 19509 benchmarks was < 3sec

ERE# Performance Summary

19,509 SMT-LIB benchmarks for regular expressions

ERE# solver (Rust):

- Total time for ALL benchmarks: < 3 seconds
- No single benchmark needed > 0.1 seconds
- Orders of magnitude faster than state-of-the-art

Why so fast?

1. Specialization to regex + alphabet compression to bit-vectors
2. Aggressive rewrite rules via symbolic derivatives
3. Lazy propagation of intersection and complement
4. Reversal theorem for nonemptiness
5. XOR as a built-in Boolean operator

Part III: EREQ (PLDI’26)

Regular Expressions with Quantifiers

with Nikolaj Bjørner, Ian Erik Varatalu, Ekaterina Zhuchko

Motivation: From Regexes to Logic

Regular expressions cannot express position-dependent properties:

- "Every occurrence of b is preceded by some a" — needs quantifiers

Weak Monadic Second-Order Logic (wMSO):

- Variables range over positions / finite sets of positions in a word
- Predicates: $P_a(x)$, order $x < y$, subset $X \subseteq Y$, singleton $|X|=1$

Büchi-Elgot-Trakhtenbrot: wMSO ≡ Regular Languages

Classical approach: wMSO → finite automaton → check emptiness

- Non-elementary complexity (tower of exponentials)

The MONA Legacy

MONA (1995–): state-of-the-art wMSO solver

- "One really must be an optimist to implement these decision procedures"
- Despite worst-case blowup, practical with careful optimizations

Approach:

- Bottom-up: build automaton for each subformula
- $\exists$ = projection (nondeterminism), requires subsequent determinization
- $\neg$ = complement, requires deterministic automaton
- Each quantifier alternation → exponential blowup

Our alternative:

- Lazy top-down exploration via symbolic derivatives
- Avoid explicit automata construction entirely

Application: Learning from Service Traces

Trace analysis of a major cloud storage service

Actions: put (update), del (delete), undel (undelete)

Enabling conditions (requires quantification):

"del enables undel": $\forall x_0(\text{undel}(x_0) \rightarrow \exists x_1(x_1 < x_0 \land \text{del}(x_1)))$

Cannot be expressed in **ERE** without manual position tracking!

Disabling conditions (expressible in **ERE**):

"del disables next del": $\neg(\top^* \cdot \text{del} \cdot \text{del} \cdot \top^*)$

EREQ bridges the gap: quantifiers when needed,

regex operators (concatenation, loops) when more natural

EREQ: Extended Regular Expressions with Quantifiers

EREQ extends ERE with one new operator: $\exists R$ (existential quantifier)

$$R ::= \beta \mid \varepsilon \mid R_1 \mid R_2 \mid R_1 \& R_2 \mid R_1 \cdot R_2$$
$$\mid R\{m\} \mid R^* \mid \neg R \mid \exists R \quad \leftarrow \text{NEW}$$

$\forall R$ is derived as $\neg\exists\neg R$

Semantics of $\exists$:

$L(\exists R) = \{ \pi(w) \mid w \in L(R) \}$

where π projects away the quantified bit from the extended alphabet

Alphabet encoding: $\Sigma \times \{0,1\}^n$ (n = number of free variables)

Each position carries a character + bit-vector of variable assignments

Alphabet Encoding: From Positions to Bits

Example: w = "bbaab", $\theta = [\{2,3\}, \{0,1,4\}]$

Position:	0	1	2	3	4	
Character:	b	b	a	a	b	
Bit p_0:	0	0	1	1	0	← $X_0 = \{2,3\}$
Bit p_1:	1	1	0	0	1	← $X_1 = \{0,1,4\}$

Each position becomes a tuple (character, bit-vector)

Predicates:

- p_i tests "is bit i set?" (membership in set variable)
- $\alpha \in A$ tests the character component

Embedding wMSO into EREQ

Linear-time translation $\ulcorner \phi \urcorner$ from wMSO to EREQ:

$\ulcorner \exists\phi \urcorner = \exists\ulcorner \phi \urcorner$ $\quad$ $\ulcorner \neg\phi \urcorner = \neg\ulcorner \phi \urcorner$

$\ulcorner \phi_1 \wedge \phi_2 \urcorner = \ulcorner \phi_1 \urcorner \& \ulcorner \phi_2 \urcorner$ $\quad$ $\ulcorner \phi_1 \vee \phi_2 \urcorner = \ulcorner \phi_1 \urcorner | \ulcorner \phi_2 \urcorner$

Atomic formulas become regexes:

$\ulcorner \alpha(X_i) \urcorner = \top^* \cdot (\alpha \,\&\, p_i) \cdot \top^*$ "some position in X_i has label α"

$\ulcorner |X_i|=1 \urcorner = \neg p_i^* \cdot p_i \cdot \neg p_i^*$ "X_i is a singleton"

$\ulcorner X_i <_\exists X_j \urcorner = \top^* \cdot p_i \cdot \top^* \cdot p_j \cdot \top^*$ "some element of X_i precedes some of X_j"

$\ulcorner X_i \subseteq X_j \urcorner = (\neg p_i \mid p_j)^*$ "X_i is a subset of X_j"

Theorem 1 (**Lean verified**): $w,\theta \vDash \phi \iff \text{encode}(w,\theta) \in L(\ulcorner \phi \urcorner)$

Derivatives: From Concrete to Symbolic

Brzozowski (concrete): $\delta_c(\alpha) = \text{if } c \vDash \alpha \text{ then } \varepsilon \text{ else } \bot$

Symbolic (character-free): $\delta(\alpha) = \text{ite}(\alpha, \varepsilon, \bot)$

- Remove c altogether — result is a transition term

Operations propagate to leaves:

- $\delta(\neg\alpha) = \text{ite}(\alpha, \neg\varepsilon, \neg\bot)$ complement pushes through
- $\delta(R^*) = \delta(R) \cdot R^*$ star unfolds one step
- $\delta(R_1 \cdot R_2) = \delta(R_1) \cdot R_2$ (+ $\delta(R_2)$ if R_1 nullable)

This engine powers both ERE# and EREQ

The Key New Rule: $\delta(\exists R) = \exists\delta(R)$

The symbolic derivative distributes over the quantifier:

$$\delta(\exists R) = \exists\, \delta(R)$$

$$\delta(\forall R) = \forall\, \delta(R)$$

This means:

- Quantifier elimination happens incrementally, one step at a time
- No full automaton construction for the quantified sub-expression

Algebraic laws that make this work:

- $\exists(R \cdot S) \equiv \exists R \cdot \exists S$
- $\exists(R^*) \equiv (\exists R)^*$
- $\exists(R \mid S) \equiv \exists R \mid \exists S$

Incremental quantifier elimination — no explicit automata construction needed!

Decision Procedure: Nonemptiness via Derivatives

To decide satisfiability of wMSO formula ϕ:

1. Translate: $\phi \to \ulcorner \phi \urcorner$ in EREQ
2. Unfold derivatives: explore reachable states of δ
3. Check: is any reachable state nullable?

Termination: derivative closure is finite (Theorem 3, **Lean-verified**)

Search strategies matter:

- Negation Normal Form (NNF)
- Anti-prenex: push ∃ inward over · and | and *
- DNF: split disjunctions into separate search branches

Key rewrites: $\neg(\top^* \cdot p \cdot \top^*) \to \bar{p}^*$, then $\exists(\bar{p}^*) \to \top^*$

Example: Büchi Addition via EREQ

Binary addition $x + y = z$ as a regex over $\Sigma = \{0,1\}^3$:

Use proposition p_0 as a carry bit

$add = (b_0 \leftrightarrow b_1) \leftrightarrow (b_2 \leftrightarrow p_0)$	column-correct
$carry = p_0 \wedge (b_0 \vee b_1) \vee b_0 \wedge b_1$	carry-forward
$\neg p_0 \cdot \top^*$	initial carry = 0
$\top^* \cdot \neg carry$	final carry = 0

R_{bad} matches words with incorrect carry

R_add = $\exists(\overline{p}_0 \cdot \top^* \;\&\; add^* \;\&\; \neg R_{bad} \;\&\; \top^* \cdot \neg carry)$

∃ eliminates the carry bit — **single expression captures addition**!

EREQ Evaluation: Baseline Comparison with MONA

AutomatArk M2L-str benchmark suite (~10,000 instances)

EREQ solver (Rust):

- Timeouts: 250 instances
- Competitive on most families, superior on counter/loop families

MONA (30+ years of optimization):

- Timeouts: 89 instances
- Excels at problems needing global automata optimization

EREQ excels where algebraic structure helps:

loops, bounded repetitions, cascading rewrites

First derivative-based wMSO solver — and already competitive!

Lean Formalization

Both papers include extensive formalization in Lean 4:

ERE# (CAV'25):

- Lookaround Normal Form Theorem

EREQ (PLDI'26) — ~4,100 lines of Lean:

- Semantics of wMSO and EREQ
- Correctness of wMSO → EREQ embedding (Theorem 1)
- Correctness of derivative rules (Theorem 2)
- Finiteness of derivative closure (Theorem 3)

Why Lean?

1. Algebraic laws are easy to get wrong; formalization catches errors
2. Plugin for LLMs: Agentic workflows with LLMs (e.g. **agency** with copilot-cli) can **automatically use and integrate existing math libraries into program verification**

Part IV: Unified Perspective

How to Extend Derivative-Based Methods

The Unifying Principle

Symbolic derivatives work because operations commute with ite:

Classical RE:	$\delta(R_1 \mid R_2) = \delta(R_1) \mid \delta(R_2)$	✓
	$\delta(R^*) = \delta(R) \cdot R^*$	✓
+ Complement:	$\delta(\neg R) = \neg\delta(R)$	✓
+ Intersection:	$\delta(R_1 \& R_2) = \delta(R_1) \& \delta(R_2)$	✓
+ Quantifiers:	$\delta(\exists R) = \exists\delta(R)$	✓

Each extension follows the **SAME** recipe:

1. Define the new operator on regexes
2. Show it commutes with δ (pushes to leaves)
3. Prove derivative closure remains finite (termination)
4. Develop rewrites to keep the state space manageable

ERE# vs EREQ: Two Extensions, One Framework

	ERE# (CAV'25)	EREQ (PLDI'26)
New operators	Intersection, Complement, Lookarounds	Existential Quantifier ∃
Semantics	Span semantics over words	M2L-str (positions + sets)
Key principle	$\delta(\neg R)=\neg\delta(R)$, $\delta(R\&S)=\delta(R)\&\delta(S)$	$\delta(\exists R)=\exists\delta(R)$
Expressiveness	Boolean closure of RE#	wMSO (equiv. regular langs)
Implementation	Rust	Rust
Benchmarks	19,509 SMT-LIB (< 3s)	~10,000 AutomatArk
Formalization	Normal Form (Lean)	~4,100 lines Lean

Future Directions

Framework extensions:

- S1S: infinite words (ω-regular) / S2S: trees
- Richer alphabet theories

Applications:

- Revisit MONA applications for EREQ (hardware/protocol verification)
- SMT-LIB: extend RegLan with (seq T) and RegLan modulo T
- Pre-processing in SMT solvers (Z3 integration)

Tooling:

- Better search strategies and rewrite rules
- Connect Lean proofs with Rust for cross-validation

Vision: symbolic derivatives as a universal framework
for decision procedures over regular properties

Thank You!

Papers & Artifacts:

ERE# (CAV'25):

ERE# solver: https://github.com/ieviev/cav25-resharp-smt

RE# NuGet: https://www.nuget.org/packages/Resharp

Webapp: https://ieviev.github.io/resharp-webapp/

EREQ (PLDI'26):

EREQ solver (Rust) + Lean formalization (~4,100 lines)

Collaborators:

Nikolaj Bjørner, Ian Erik Varatalu, Ekaterina Zhuchko, Juhan Ernits