Checking Regular Expressions in Cvc5 Proofs

Ofec Israel[1], Yoni Zohar[1(✉)], Andrew Reynolds[2], S. Hitarth[3], Bruno Dutertre[4], Clark Barrett[5], and Cesare Tinelli[2]

[1] Bar-Ilan University, Ramat Gan, Israel
yoni.zohar@biu.ac.il
[2] The University of Iowa, Iowa City, IA, USA
[3] Max Planck Institute for Software Systems, Saarbrücken, Germany
[4] Amazon Web Services, Santa Clara, CA, USA
[5] Stanford University, Stanford, CA, USA

Abstract. cvc5 is a state-of-the-art proof-producing SMT solver, capable of solving formulas over a myriad of theories, including Unicode strings. Matching regular expressions against concrete strings is done numerous times during the solving process, and forms a bottleneck in proof-checking for unsatisfiable formulas. We describe three approaches for checking regular expressions in the Eunoia proof-checking framework, and evaluate them on proofs produced by cvc5.

1 Introduction

This work is part of a research program to efficiently check unsatisfiability proofs produced by Satisfiability Modulo Theories (SMT) solvers [4]. We focus on the cvc5 [2] SMT solver and the Eunoia [10] proof framework. We implement checking procedures for a common reasoning step in the SMT-LIBtheory of strings: matching regular expressions. These are implemented as *side conditions*: Eunoiaprograms that are defined in proof rules executed during proof checking.

Solving string constraints involves matching many concrete strings and regular expressions, to the point that verifying these matching steps becomes a bottleneck in proof checking. Implementing standard algorithms for regular expression checking as side conditions in Eunoiais challenging, because Eunoiaoffers limited support for standard data structures and control flow commands.

We have implemented three approaches. The first and simplest is based on backtracking. However, this basic approach does not perform well in many cases. The second follows Thompson's construction, and the third adopts Brzozowski's derivatives [5]. Our experiments show that the derivatives-based approach performs best and is therefore now the default in cvc5's main branch.

Section 2 presents necessary background. Section 3 to 5 survey the three approaches. We evaluate them in Sect. 6,[1] and conclude in Sect. 7.

B. Dutertre—This work does not relate to the author's position at Amazon.

[1] An accompanying artifact is available at https://doi.org/10.5281/zenodo.18524263.

A. Biere et al. (Eds.): IJCAR 2026, LNAI 16688, pp. 267–275, 2026.
https://doi.org/10.1007/978-3-032-32589-1_16

Table 1. Fragment of SMT-LIB strings.

Op.	Sort	SMT-LIB
none	$\mathcal{R}$	`re.none`
to_re	$\mathcal{S} \to \mathcal{R}$	`str.to_re`
comp	$\mathcal{R} \to \mathcal{R}$	`re.comp`
star	$\mathcal{R} \to \mathcal{R}$	`re.*`
rcon	$\mathcal{R} \times \mathcal{R} \to \mathcal{R}$	`re.++`
union	$\mathcal{R} \times \mathcal{R} \to \mathcal{R}$	`re.union`
inter	$\mathcal{R} \times \mathcal{R} \to \mathcal{R}$	`re.inter`

Table 2. Builtins in Eunoia.

Op.	Eunoia	Desc.
lconcat	eo::list_concat	List concat
find	eo::list_find	List find
diff	eo::list_diff	List diff
sconcat	eo::concat	Str concat
extract	eo::extract	Substr
len	eo::len	Str Length
eq	eo::is_eq	Equality

Related Work. The collection of proof rules for checking cvc5 proofs is described in [9]. Recent work [12] provides a solver whose calculus is verified in Lean. We focus on independent *proof-checking* for unsatisfiability results.

2 Preliminaries

Table 1 shows a sub-signature of the theory of strings [3], focusing on regular expressions (regexes). The sorts of strings and regular languages are $\mathcal{S}$ and $\mathcal{R}$, respectively. Operators *none*and *to_re*represent the empty language and string literals (w.l.o.g., of length 1), respectively, while ϵ denotes the empty string. Other constructors include *comp* (complementation), *star* (Kleene closure), *rcon* (concatenation), *union* (union), and *inter* (intersection). The signature includes all string literals, denoted with double quotes. We often use standard regex notation, e.g., writing a^*b instead of $rcon(star(to\_re(''a'')), to\_re(''b''))$. The implementation supports additional SMT-LIB operators (e.g. character ranges).

Eunoiais a logical framework for proofs, equipped with the Ethos proof checker [8,10], which supports dagification of terms and memoization of programs. Variadic terms are called *lists*. Internally, lists are LISP-style expressions. Eunoiaincludes a minimal standard library for lists and strings, which implements common operations natively for efficiency, such as the ones listed in Table 2. Eunoiaallows defining *side conditions* in the form of functional programs. The most direct rule that relies on regex matching is `str-in-re-eval`, which matches a string against a regex. Other rules include, e.g., `re-inter-inclusion` and `str-replace-re-eval`, that use matching for complex regex queries.

A *nondeterministic finite automaton (NFA)* $(\mathcal{Q}, \Sigma, \Delta, q_s, q_a)$ consists of finite sets $\mathcal{Q}$ and Σ of states and letters, with $\epsilon \notin \Sigma$, $\Delta : \mathcal{Q} \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(\mathcal{Q})$, the *transition function*, and $q_s, q_a \in \mathcal{Q}$, the *start* and *accepting* states. The ϵ*-closure* of q is the set of states reachable from q by ϵ-transitions.

3 Backtracking-Based Approach

Backtracking works by analyzing the regular expression and applying sub-queries recursively. Pseudocode for this approach is given in Algorithm 1. Function BtRec takes a string s, index n, and two regular expressions r_1 and r_2. It

Algorithm 1. Backtracking algorithm for regex matching.

```
1: function BtRec(s, n, r1, r2)
2:     if r2 ≠ null then
3:         s1 ← extract(s, 0, n − 1) ; s2 ← extract(s, n, len(s) − 1)
4:         if BtRec(s1, 0, r1, null) ∧ BtRec(s2, 0, r2, null) then return true
5:         else if n = len(s) then return false
6:         else return BtRec(s, n + 1, r1, r2)
7:     match r1 with
8:         case to_re(s′): return s′ = s
9:         case none: return false
10:        case union(ra, rb): return BtRec(s, 0, ra, null) ∨ BtRec(s, 0, rb, null)
11:        case inter(ra, rb): return BtRec(s, 0, ra, null) ∧ BtRec(s, 0, rb, null)
12:        case comp(ra): return ¬BtRec(s, 0, ra, null)
13:        case rcon(ra, rb): return BtRec(s, 0, ra, rb)
14:        case star(ra):
15:            if s = ϵ then return true
16:            else return BtRec(s, 1, ra, r1)
```

```
(program $bt_rec (...)
  :signature (String Int RegLan RegLan) Bool
  (
   (($bt_rec s 0 (str.to_re s1) @re.null) (eo::is_eq s s1))
   (($bt_rec s 0 (re.++ r1 rr) @re.null) ($bt_rec s 0 r1 rr))
   ...
  )
)
```

Fig. 1. Eunoia code for BtRec.

determines whether s conforms to the pattern defined by regex r_1 followed immediately by regex r_2. Parameter n specifies the minimum size of the prefix of s that must be consumed by r_1. We use *null* as an internal marker for invalid regexes. When first calling BtRec, n is passed as 0 and r_2 as *null*. Some cases require considering every partition of the string (lines 3–6). If a match fails in the recursion, the algorithm backtracks, causing an exponential growth in execution time. We show a snippet from the Eunoiacode of BtRec in Figure 1.

Algorithm 2. Function BNR for constructing automata from regexes.

```
1: function BNR(r, q_s, q_a, δ, σ)
2:     match r with
3:         case none: return δ
4:         case to_re(c): return cons((q_s, c, q_a), δ)
5:         case rcon(r_1, r_2):
6:             q_mid ← sconcat(σ, "m")
7:             δ_1 ← BNR(r_1, q_s, q_mid, δ, sconcat(σ, "l"))
8:             return BNR(r_2, q_mid, q_a, δ_1, sconcat(σ, "r"))
9:         case union(r_1, r_2):
10:            δ_1 ← BNR(r_1, q_s, q_a, δ, sconcat(σ, "l"))
11:            return BNR(r_2, q_s, q_a, δ_1, sconcat(σ, "r"))
12:        case star(r_1):
13:            q_hub ← sconcat(σ, "h")
14:            δ_loop ← lconcat(δ, [(q_s, ϵ, q_hub), (q_hub, ϵ, q_a)])
15:            return BNR(r_1, q_hub, q_hub, δ_loop, sconcat(σ, "l"))
```

Example 1. Consider the execution of BtRec($"aab", 0, a^*b, null$) to determine whether "aab" matches regex a^*b. Since $r_2 = null$, we skip the conditional. Then, we recurse on line 13, calling BtRec($"aab", 0, a^*, b$). In the recursive call, $r_2 \neq null$, and so we enter the conditional. There, we consider all splits of $"aab"$ into two strings. When $i = 2$, the match succeeds.

4 NFA-Based Approach

The second approach is based on Thompson's construction of NFAs [11]. It requires the *construction* of an automaton and then the *simulation* of the automaton on the input string. We only use this approach on regexes that exclude *comp* and *inter*, as these operators require expensive transformations. For regexes including these operators, we fall back to backtracking.

Eunoiacannot naturally represent NFAs as directed graphs, due to its strict bottom-up evaluation that does not support cycles. Instead, we represent an NFA as a list of transitions of the form (q, α, q'), where q and q' are states and α is a character or ϵ. The states are represented by string identifiers. To ensure uniqueness, we use a *path-based* identifier scheme. Each recursive call carries a string that encodes its position.

The construction is described in Algorithm 2. Function BNR takes a regular expression r without *comp* and *inter*, and computes an equivalent NFA. For that, it takes additional arguments: (string identifiers for) start and accept states (q_s and q_a, respectively), a list δ of transitions, and a path context σ. It is called with a start state $"s"$, accepting state $"a"$, an empty transition list, and an empty path context. The result is a transition list that represents an NFA.

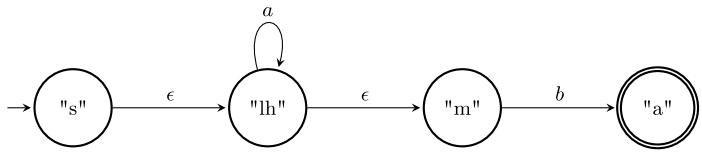

Fig. 2. The NFA for a^*b from the transition list generated by BNR.

Example 2. Running BNR (Algorithm 2) on regular expression $r = a^*b$ with $q_s :='' s''$, $q_a :='' a''$, $\delta := []$, and $\sigma :=''\ ''$, we identify expression a^*b as a concatenation and generate a unique intermediate identifier based on the current path σ: $q_{mid} = sconcat(\sigma, ''m'') ='' m''$. The problem is then split into two recursive calls. Eventually, we obtain four transitions: $(''s'', \epsilon, '' lh'')$, $(''lh'', '' a'', '' lh'')$, $(''lh'', \epsilon, '' m'')$, and $(''m'', '' b'', '' a'')$. A graphical representation is in Fig. 2.

After construction, we simulate the NFA on the input string. For each letter, we identify the list of reachable states by scanning the transition list, which can be costly. The algorithm updates the reachable states while scanning the input, and accepts if the final set contains the accepting state.

Example 3. Simulating the NFA of Example 2 with string $s ='' aab''$, the ϵ-closure of the initial state $''s''$ consists of $''s''$, $''lh''$, and $''m''$. The first character to match is "a". States $''s''$ and $''m''$ yield no transitions for "a". State $''lh''$ transitions to $''lh''$, so $''lh''$ is added as a reachable state. The algorithm continues until letter "b" is consumed, at which point q_a belongs to the reachable states.

4.1 Optimizations

To avoid quadratic simulation, we represent the automaton hierarchically, each letter points to a nested transition list. Due to term dagification in Ethos, this tree-shaped structure is stored as a DAG. Simulation shifts from global scanning to walking a decision tree. Since the underlying representation is acyclic, we implement cycles using an execution stack. The marks *LoopEntry* and *LoopExit* serve as control instructions that do not consume input and are used to simulate NFA loops. For a Kleene-star expression, the construction places a *LoopEntry* before the body and a matching *LoopExit* after it. When execution reaches a *LoopEntry*, it pushes a reference to this entry point onto the stack and proceeds into the loop body. When execution reaches the corresponding *LoopExit*, it pops this reference and branches nondeterministically.

Example 4. The optimized construction for a^*b results in the following hierarchical representation, which is illustrated in Fig. 3. Notice that the top level only contains two mappings, one for *LoopEntry* and one for b.

$$[(\mathit{LoopEntry}, [(a, [(\mathit{LoopExit}, [(b, [\textsc{Accept}])])])]), (b, [\textsc{Accept}])]$$

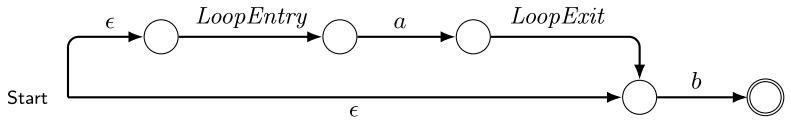

Fig. 3. Hierarchical representation for the optimized NFA construction of a^*b.

5 Derivatives-Based Approach

Our last approach is based on Brzozowski derivatives [5], as outlined in Algorithm 3. It uses function Nlbl, which checks if a regex accepts ϵ. Function Der computes the regex remaining after matching a character, which is called a *derivative*. For $\emptyset$ and ϵ, this is *none*. For single characters, the derivative is ϵ if the current character matches. Boolean operators and *star* are handled recursively. The derivative of a concatenation r_1r_2 splits according to whether r_1 accepts ϵ. The string is accepted if the final derivative accepts ϵ.

Example 5. For string $''aab''$ and regex a^*b, Algorithm 3 computes the derivative for "a", transforming a^*b to $\emptyset \cup (\epsilon \cdot a^*) \cdot b$, as a^* accepts ϵ. After the derivative for 'b' is computed, the string is accepted because the final expression is nullable. Notice that the final derivative is quite involved, even for this simple case, namely:

$$\emptyset \cup (\emptyset \cup (\epsilon \cup ((((\emptyset \cdot a^*) \cup (\emptyset \cdot a^*)) \cup (\emptyset \cdot a^*)) \cdot b)))$$

Algorithm 3. Function Der for computing derivatives.

```
1: function Der(c, r)
2:     match r with
3:         case none, to_re(ϵ): return none
4:         case to_re(c0): if c0 = c return to_re(ϵ) else return none
5:         case union(r1, r2): return union(Der(c, r1), Der(c, r2))
6:         case inter(r1, r2): return inter(Der(c, r1), Der(c, r2))
7:         case comp(r1): return comp(Der(c, r1))
8:         case star(r1): return rcon(Der(c, r1), star(r1))
9:         case rcon(r1, r2):
10:            if Nlbl(r1) then return union(Der(c, r2), rcon(Der(c, r1), r2))
11:            else return rcon(Der(c, r1), r2)
```

5.1 Optimizations

The naive implementation frequently involves redundant expressions (e.g., $\emptyset \cup r$, $r \cup r$, or $\epsilon \cdot r$), leading to an unnecessary growth in the size of the regular expression. To mitigate this, we enforce the algebraic simplification rules from [5,6] during the construction of the derivative, which flatten and normalize applications of *union*, *inter*and *rcon*, while also eliminating unnecessary occurrences of ϵ and *none*. Crucially, we keep terms normalized during construction, relying on this invariant and Eunoia's native list operators (e.g., *lconcat*and *diff*).

Example 6. With the simplification, the partial derivative for "a" remains a^*b, without any ϵ or *none*. The final derivative is simply ϵ, which is trivially nullable, and significantly shorter than the equivalent derivative computed in Example 5.

6 Evaluation

We use a cluster of 23 machines with Intel Xeon Gold 6348 CPUs. Proofs are generated with cvc5 1.3.2 using limits of 60 s and 8GB, and are checked separately with Ethos 0.2.2 using the same limits. There are 103,405 benchmarks in logics QF_S, QF_SLIA, and QF_SNIA of the 2024 release of the SMT-LIB benchmark library [7]. Within the given resources, cvc5 is able to determine that 38,765 are unsatisfiable. 3,118 unsatisfiability proofs apply rule `str-in-re-eval`. These proofs comprise our **Organic** benchmark set. The second set consists of 35,627 **Synthetic** proofs, obtained by logging a sequence of calls to the regex matching rule, applied to all regex checks performed during solving. These are not unsat proofs, but sequences of calls to rule `str-in-re-eval`.

Our configurations are listed on the left of Table 3, which summarizes the results. In **LB** the regex matching rule accepts each application immediately. It provides a **lower bound** for checking time. For the other configurations, we list the relevant section and the number of lines of code (LOC). **NFA**$^+$ has fewer LOCs than **NFA** because it does not need to maintain unique string identifiers.

The number of proofs commonly checked by all approaches is in parentheses. Total run-times (T) and memory usage (M) are given for the set of commonly checked proofs. Number of checked proofs (# S) is also given. On **Organic**, **DER**$^+$ is nearly optimal, taking just 8 s more than **LB**. The results also show that the optimizations of the NFA and derivatives approaches are crucial.

On **Synthetic**, **BT** checks slightly more proofs than the optimized approaches. These proofs originate from the sub-family `redos_attack_detection` of QF_SLIA and share a similar pattern with extremely long strings. On these, the default search order of **BT** guides it to an early termination of the loop in Algorithm 1, avoiding an exhaustive search. Thus, **BT** remains a viable alternative with performance complementary to the others on such benchmarks.

When considering commonly checked proofs, the results are similar to set **Organic**. Configurations **NFA** and **NFA**$^+$ fall back to **BT** on ∼5% of the checks. Finally, we remark that configuration **BT** is sensitive to the order in which the splits into two strings are considered in Algorithm 1. For example, when we reverse the order of the splits, its performance significantly degrades.

Table 3. Summary of configurations and results.

Config	Sec.	LOC	**Organic** (3,106 common)			**Synthetic** (34,050 common)		
			# S	T (s)	M (GB)	# S	T (s)	M (GB)
LB	–	–	3117	351	34	35620	1580	308
BT	3	40	3114	386	37	**35593**	3111	363
NFA	4	128	3112	517	42	35359	10214	743
NFA$^+$	4.1	91	**3117**	361	**34**	35580	2119	329
DER	5	46	3111	589	47	34204	12928	718
DER$^+$	5.1	66	**3117**	**359**	**34**	35556	**1743**	**314**

7 Conclusion

We have implemented and evaluated several approaches for checking regular expression membership in the Ethos checker for cvc5 proofs. The primary obstacle was reconciling efficiency with the strict architectural constraints of Eunoia. Future work includes further optimizing the derivatives and NFA approaches, and implementing similar support in Carcara [1].

Acknowledgments. This research was supported in part by the Stanford Center for Automated Reasoning, Defense Advanced Research Projects Agency (DARPA) contract FA875024-2-1001, Israel Science Foundation (ISF) grant number 209/25, Binational Science Foundation (BSF) grant number 2024049, and a gift from Amazon Web Services.

References

1. Andreotti, B., Lachnitt, H., Barbosa, H.: "Carcara: an efficient proof checker and elaborator for SMT proofs in the Alethe format. In: TACAS (1), vol. 13993. LNCS. Springer, pp. 367–386 (2023)
2. Barbosa, H., et al.: CVC5: a versatile and industrial-strength SMT solver. Presented at the (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
4. Barrett, C., et al.: "Satisfiability modulo theories". In: Handbook of Satisfiability, Second Edition. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press. Chap. 33, pp. 825–885 (2021)
5. Brzozowski, J.A.: "Derivatives of regular expressions". J. ACM **11**(4), 481–494 (1964)
6. Owens, S., Reppy, J.H., Turon, A.: "Regular-expression derivatives re-examined". In: J. Funct. Program. **19**(2), 173–190 (2009)
7. Preiner, M.: SMT-LIB release 2024 (non-incremental benchmarks) (2024)

8. Reynolds, A., et al.: “Ethos: a fast proof checker for the eunoia logical framework”. In: Automated Reasoning - 13th International Joint Conference, IJCAR 2026. Ed. by Armin Biere, Carsten Lutz, and Sara Negri. Lecture Notes in Computer Science. Springer (2026)
9. Reynolds, A., et al.: “The cooperating proof calculus: comprehensive proofs for an SMT solver”. In: CAV. LNCS. To appear. Springer (2026)
10. The Ethos Manual. https://github.com/cvc5/ethos/blob/main/usermanual.md (2025)
11. Thompson, K.: “Regular expression search algorithm”. Commun. ACM **11**(6), 419–422 (1968)
12. Varatalu, I.E., et al.: “Regex decision procedures in extended RE#”. In: CAV. Springer, pp. 106–129 (2025)