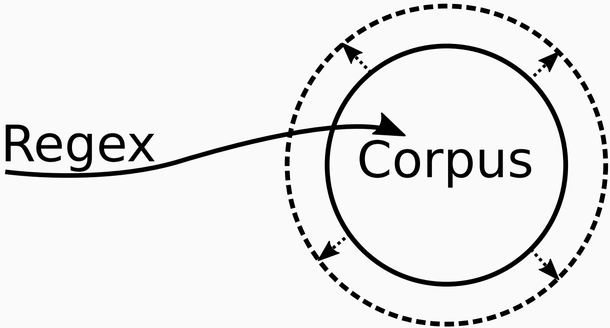
Model Checking Regular Expressions

Arlen Cox

5-9 May 2019

IDA – Center for Computing Sciences

Managing a corpus of regular expressions



Does the *language* of the corpus grow?

Managing a corpus of regular expressions

$$\exists s.\ s \in \mathcal{L}(R) \land s \notin \mathcal{L}(C)$$

How do different solvers perform on this problem?

Adapted from Hooimeijer, Weimer 2010

Managing a corpus of regular expressions

$$\exists s.\ s \in \mathcal{L}(R) \wedge s \notin \mathcal{L}(C)$$

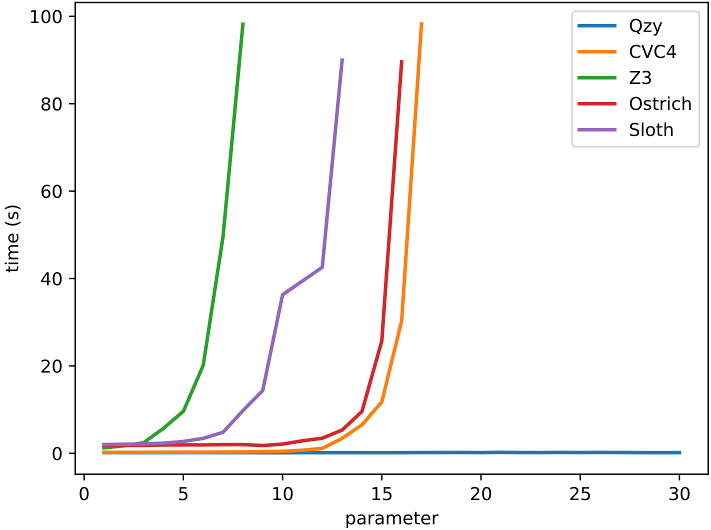
How do different solvers perform on this problem?

$$R = \texttt{\^{}[01]*1[01]\{}n\texttt{\}\$}$$

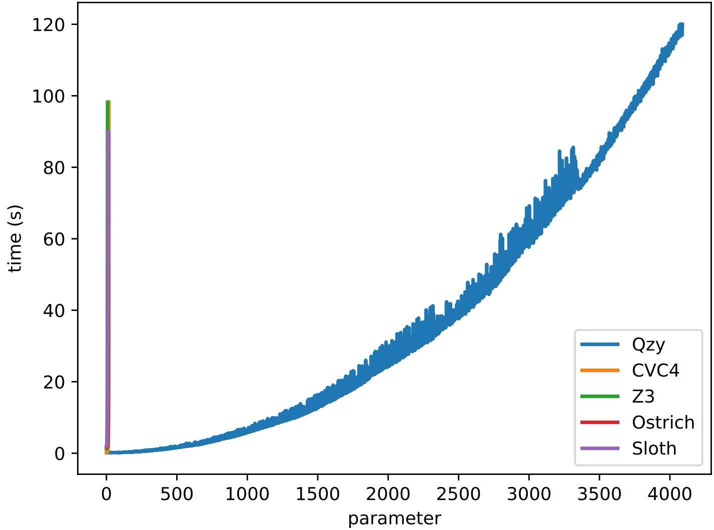
$$C = \texttt{\^{}[01]*0[01]\{}n-1\texttt{\}\$}$$

Adapted from Hooimeijer, Weimer 2010

Regular expression difference



Qzy has quadratic scaling in n



Existing solvers are too slow

C is really a corpus of regular expressions.

$$\exists s.\ s \in \mathcal{L}(R) \wedge s \notin \mathcal{L}(C_1) \wedge \cdots \wedge s \notin \mathcal{L}(C_n)$$

It only gets worse...

Existing solvers are too slow

C is really a corpus of regular expressions.

$$\exists s.\ s \in \mathcal{L}(R) \land s \notin \mathcal{L}(C_1) \land \cdots \land s \notin \mathcal{L}(C_n)$$

It only gets worse...

I built Qzy to solve this

Email address corpus

129 email address regular expressions from Regexlib

$$R = \text{one regular expression from corpus}$$
$$C = \text{remaining 128 regular expressions}$$

Email address corpus

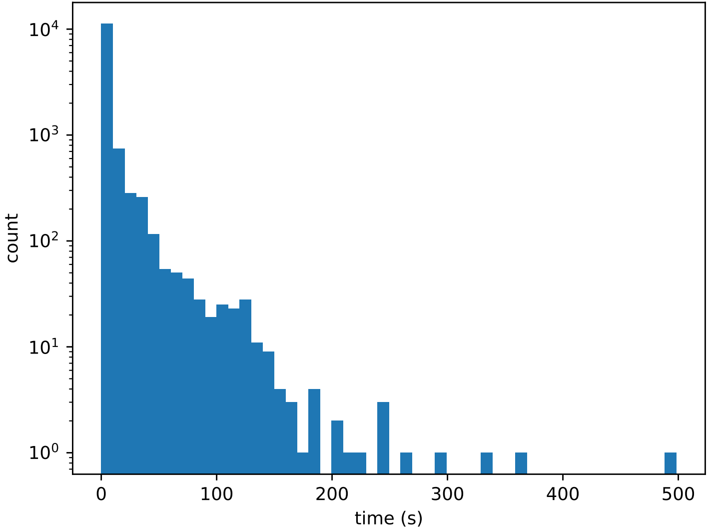
129 email address regular expressions from Regexlib

$$R = \text{one regular expression from corpus}$$

$$C = \text{remaining 128 regular expressions}$$

Solver	Result
CVC4	Can't encode (non-printable character ranges)
Z3	Time out after 24 hours (1 core)
Ostrich	Time out after 24 hours (44 cores!)
Sloth	Memory out (2G) after 10 minutes

Qzy is fast for email address corpus



Qzy is fast for email address corpus

Running the whole suite of 128 cases takes:

- 15m 2s using 1 core.
- 97s using 32 cores of a 36 core computer.

Overview

1. Encoding regular expression constraints for model checking
2. Implementation and optimization
3. Ongoing project: Capture groups

Encoding regular expression constraints for model checking

Tabakov/Vardi universality encoding[2]

$$\boxed{\text{Regex}} \longrightarrow \boxed{\text{NFA}} \longrightarrow \boxed{\text{TS}}$$

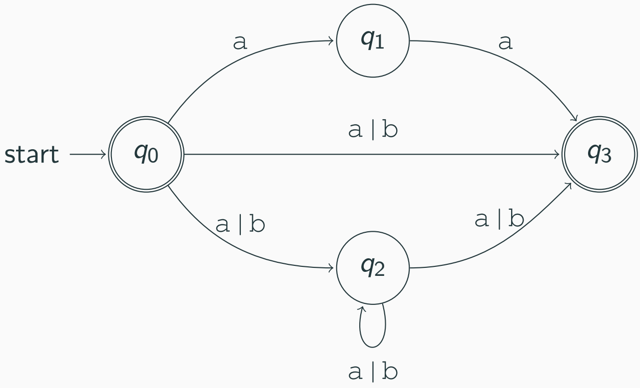
- Universality is encoded as a safety property of the transition system.
- Use an off-the-shelf model checker to check that property.
- Equivalent to a backward BFA encoding[1].

[1] Cox, Leasure. Model Checking Regular Language Constraints. 2017

[2] Tabakov, Vardi. Experimental Evaluation of Classical Automata Constructions. 2005

Tabakov/Vardi universality encoding example

Example regular expression: `aa|[ab]*`



One bit per NFA state transition system

$$I(q_0, q_1, q_2, q_3) = q_0 \land \neg q_1 \land \neg q_2 \land \neg q_3$$

$$T\left(\begin{array}{l} q_0, q_1, q_2, q_3, \\ q'_0, q'_1, q'_2, q'_3, x \end{array}\right) = \left(\begin{array}{l} \neg q'_0 \land \\ q'_1 = q_0 \land x \in \{\, \mathtt{a} \,\} \land \\ q'_2 = (q_0 \lor q_2) \land x \in \{\, \mathtt{a}, \mathtt{b} \,\} \land \\ q'_3 = \left(\begin{array}{l} q_1 \land x \in \{\, \mathtt{a} \,\} \lor \\ (q_0 \lor q_2) \land x \in \{\, \mathtt{a}, \mathtt{b} \,\} \end{array}\right) \end{array}\right)$$

$$P(q_0, q_1, q_2, q_3) = q_0 \lor q_3$$

Emptiness and universality

Emptiness can be checked with a model checker

- If P is satisfied with input string $\bar{x}$, $\bar{x}$ is in the language.
- If P is unsatisfiable for any input string, the language is empty.

T is really a transition *function*, so

- If $\neg P$ is satisfied with input string $\bar{x}$, $\bar{x}$ is *not* in the language.
- If $\neg P$ is unsatisfiable for any input string, the language is universal.

With determinism, language combinators follow

With a transition function, given an input, the set state bits (state set) are deterministic.

Consequently the following equivalences hold

$$\mathcal{L}_1 \setminus \mathcal{L}_2 \Leftrightarrow P_1 \wedge \neg P_2$$
$$\mathcal{L}_1 \cup \mathcal{L}_2 \Leftrightarrow P_1 \vee P_2$$
$$\mathcal{L}_1 \cap \mathcal{L}_2 \Leftrightarrow P_1 \wedge P_2$$

SMT solving with regular expressions

Using these Boolean combinators, I built Qzy, an SMT solver regular expressions.

Implementation and optimization

Implementation

Built as a C++ library with Python and C++ APIs.

API similar to SMT solvers:

- Multiple variables
- Arbitrary Boolean combinators

Goal: feature compatible with RE2:

- UTF-8 character classes
- Begin/end of string/line markers
- Word boundaries
- ~~Capture groups~~ (working on it – more later)
- ~~Back references~~ (not supported by RE2)
- ~~Look ahead~~ (not supported by RE2)

Start and end tags

Extend alphabet with special start and end characters

`^` is (*start* | `\n` | `\r` | `\r\n`) (depending on matching mode)

`$` is (*end* | `\n` | `\r` | `\r\n`) (depending on matching mode)

Enables:

- Unanchored regular expressions
- Begin/end of string/line markers
- Multiple variables

Multiple variables

Use a wide encoding: if a character is 8 bits wide, input for two variables is 16 bits.

Strings for different variables can have different lengths.

Start and end characters pad out strings so that all have the same length.

Start and end characters reveal the start and end of strings within counterexamples.

Optimizations

- Alphabet compression
- Regex structural hashing
- Transition system structural hashing
- SAT-simplification
- Preprocessing-free IC3

Ongoing project: Capture groups

Capture group example

Anchored regular expression: `(aa)|(([ab])*)`

Input	Group 1	Group 2	Group 3
`a`	–	`a`	`a`
`aa`	`aa`	–	–
`ba`	–	`ba`	`a`

Rules:

- Left gets priority
- Last gets priority

Capture group example

Anchored regular expression: `(aa)|(([ab])*)`

Input	Group 1	Group 2	Group 3
a	–	a	a
aa	aa	–	–
ba	–	ba	a

Rules:

- Left gets priority: prioritized state vector
- Last gets priority

Capture group example

Anchored regular expression: `(aa)|(([ab])*)`

Input	Group 1	Group 2	Group 3
a	–	a	a
aa	aa	–	–
ba	–	ba	a

Rules:

- Left gets priority: prioritized state vector
- Last gets priority: most-recent tag policy

Configuration is a prioritized state set

Almost identical encoding.

Before:

- Configuration is a set of states

Configuration is a prioritized state set

Almost identical encoding.

Before:

- Configuration is a set of states

After:

- Configuration is a *sequence* of states/tags
- Each group has a start/end tag
- Each tag is a bit encoding when the group starts/ends
- Sequence encodes priority of a particular state

Encoding is non-trivial in bits

Before n states uses n bits

Now n states and m groups uses $n^2 \cdot 2^m$ bits.

I plan on implementing this naive encoding.

It is likely that lazy instantiation of these bits will be required for efficiency.

This requires a more custom model checker.

Conclusions

Qzy is an efficient (in practice!) and complete procedure for Boolean combinations of regular expression constraints.

It supports all features of RE2 except for capture groups (for now):

UTF-8, case folding, complex character classes, anchors, word boundaries, etc.

It uses a linear time encoding to transition systems.

It uses IC3 to solve the resulting transition systems.

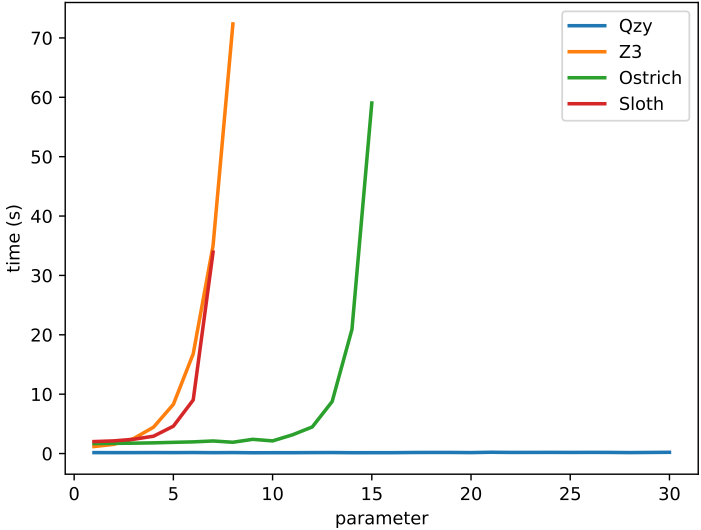
Extra Slides

Regular expression difference (unsat)

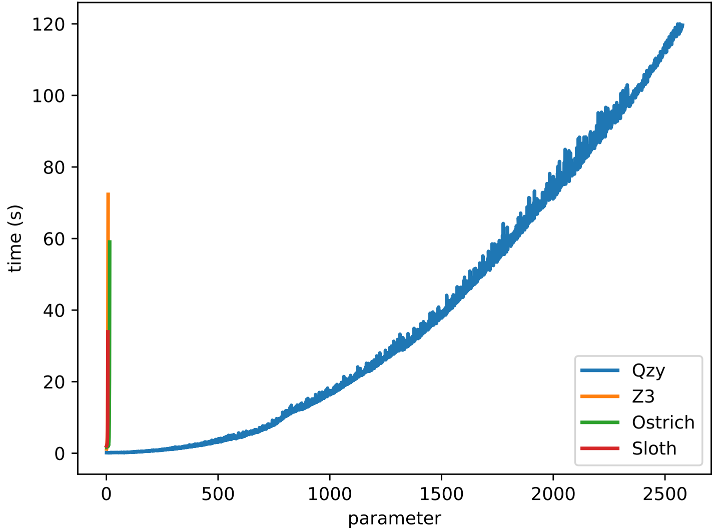
$$R = \texttt{\^{}[01]*11[01]\{}n\texttt{\}\$}$$

$$C = \texttt{\^{}[01]*1[01]\{}n+1\texttt{\}\$}$$

Regular expression difference (unsat)



Regular expression difference (unsat)



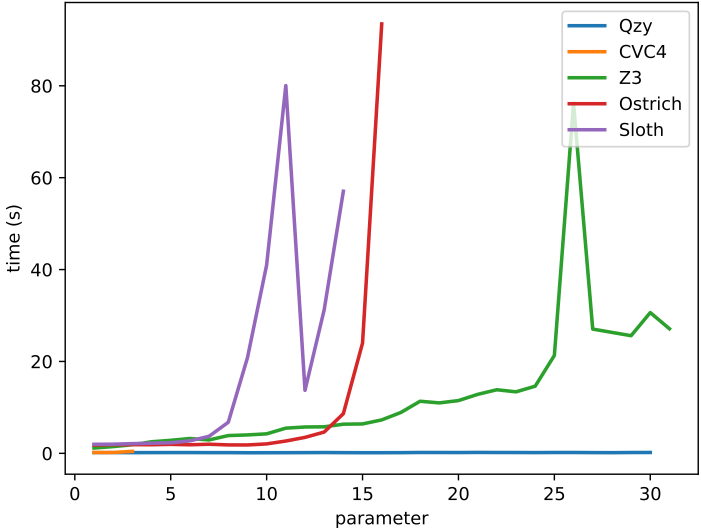
Regular expression intersection (sat)

$$\exists x.\ x \in \mathcal{L}(R) \land x \in \mathcal{L}(C)$$

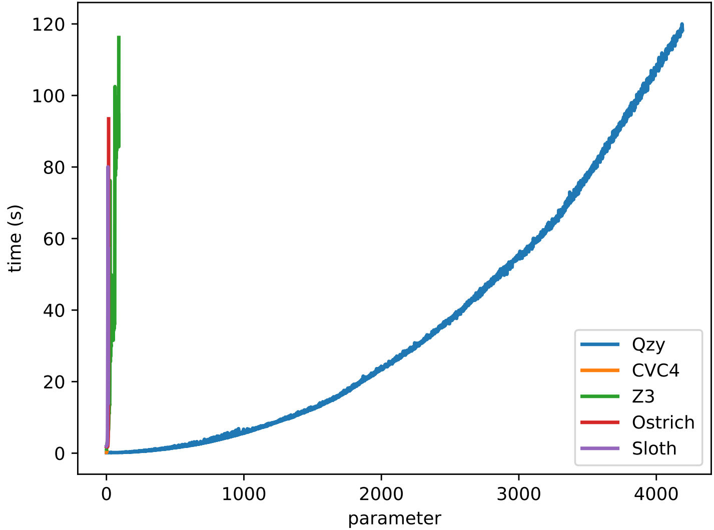
$$R = \texttt{\^{}[01]*1[01]\{}n\texttt{\}\$}$$

$$C = \texttt{\^{}[01]*0[01]\{}n-1\texttt{\}\$}$$

Regular expression intersection (sat)



Regular expression intersection (sat)



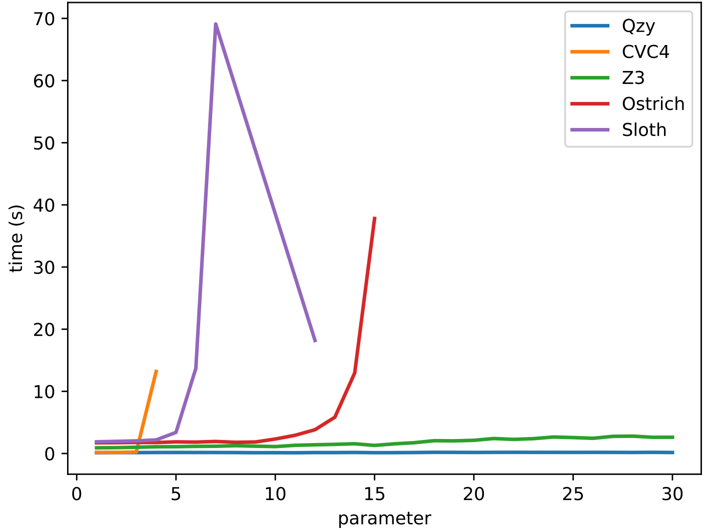
Regular expression intersection (unsat)

$$\exists x.\, x \in \mathcal{L}(R) \land x \in \mathcal{L}(C)$$

$$R = \texttt{\^{}[01]*1[01]\{}\boldsymbol{n}\texttt{\}\$}$$

$$C = \texttt{\^{}[01]*0[01]\{}\boldsymbol{n}\texttt{\}\$}$$

Regular expression intersection (unsat)



Regular expression intersection (unsat)