2010 Third International Conference on Software Testing, Verification and Validation

# Rex: Symbolic Regular Expression Explorer

Margus Veanes
Peli de Halleux
Nikolai Tillmann
*Microsoft Research*
*Redmond, WA, USA*
$\{$*margus,jhalleux,nikolait*$\}$*@microsoft.com*

***Abstract*—Constraints in form regular expressions over strings are ubiquitous. They occur often in programming languages like Perl and C#, in SQL in form of LIKE expressions, and in web applications. Providing support for regular expression constraints in program analysis and testing has several useful applications. We introduce a method and a tool called *Rex*, for symbolically expressing and analyzing regular expression constraints. Rex is implemented using the SMT solver Z3, and we provide experimental evaluation of Rex.**

***Keywords*-regular expressions; finite automata; satisfiability modulo theories; strings**

## I. Introduction

Regular expressions are used in a large variety of applications to express validity constraints on strings. The original motivation for this work comes from two particular applications. One is the support for regular expression constraints over strings in the context of program analysis and parameterized unit testing of code [1], [2]. The other one is the support for like-patterns in the context of symbolic analysis of database queries [3], where like-patterns are special kinds of regular expressions that are common in SQL select-statements.

Many languages such as C# and Java support strings as a built-in algebraic datatype: strings are treated as immutable *values* (unlike arrays for example), and are associated with purely functional operations over them. For analysis it is therefore useful to view strings as elements of a corresponding sort. Here we define strings as *lists of characters*, where a list of elements of a given sort is a built-in algebraic datatype supported by the SMT solver Z3 [4] that we are using as the underlying constraint solver. Characters are defined as $n$-bitvectors of a fixed $n \geq 1$, e.g. $n = 16$ for UTF-16 characters.

We translate (extended) regular expressions or *regexes* [5] into a symbolic representation of finite automata called *SFAs*. In an SFA, moves are labeled by formulas representing *sets* of characters rather than individual characters. An SFA $A$ is translated into a set of (recursive) axioms that describe the acceptance condition for the strings accepted by $A$ and build on the representation of strings as lists. This set of axioms is asserted to the SMT solver as the theory $Th(A)$ of $A$. The correctness of the axiomatization is stated in Theorem 1.

We revisit several classical algorithms for finite automata and describe the corresponding algorithms for SFAs. The key modification to the classical versions is the use of *satisfiability checking* of constraints over characters (bitvectors) in order to keep the SFAs "clean" (avoiding unsatisfiable formulas as labels on moves). We evaluate the performance of these algorithms based on our implementation called *Rex*. We compare different equivalent axiomatizations of language acceptors for a collection of sample regexes. In particular, when considering intersection constraints on regular expressions, it turns out that using the theory of the product of two SFAs is more efficient than using the conjunction of the individual theories.

All the algorithms and the translations in the paper are described formally and follow closely their implementation in Rex. Rex is evaluated on a set of benchmarks that shows an order of magnitude improvement compared to other approaches that have so far been used in Pex for supporting regex constraints [6].

The rest of the paper is structured as follows. In Section II we introduce some definitions and revisit some basic notions from logic that are used throughout the paper. Section III introduces SFAs and describes the variations of the classical algorithms on SFAs, that are used in Rex. Section IV explains how SFAs are translated into the corresponding axioms for the solver. Section V discusses a couple of key aspects of the implementation of Rex. Section VI provides some benchmarks regarding the implementation of Rex. Section VII describes related work. Section VIII provides some final remarks and some future work is mentioned in Section IX.

## II. Preliminaries

We assume that the reader is familiar with classical automata theory, we follow [7] in this regard. We also assume elementary knowledge about logic and model theory, our terminology is consistent with [8] in this regard.

We are working in a fixed multi-sorted universe $\mathcal{U}$ of values. For each sort $\sigma$, $\mathcal{U}^{\sigma}$ is a separate subuniverse of $\mathcal{U}$. The basic sorts needed in this paper are the Boolean sort $\mathbb{B}$,

978-0-7695-3990-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICST.2010.15

$\mathcal{U}^{\mathbb{B}} = \{\mathit{true}, \mathit{false}\}$, and the sort of *$n$-bitvectors*, for a given number $n \geq 1$; an $n$-bitvector is essentially a vector of $n$ Booleans. We also need other sorts but they are introduced at the point when they are used.

*Characters* are represented by $n$-bitvectors of a fixed length $n$, assuming that the alphabet of all characters has size $2^n$. For example, $n = 7$ ($n = 8$) for representing the standard (extended) ASCII character set, and $n = 16$ for representing the UTF-16 encoding.[1] We let $\mathbb{C}$ stand for a fixed character sort for some fixed $n$, and the complete alphabet is thus $\mathcal{U}^{\mathbb{C}}$. Without loss of generality, assume for example that $n = 7$ and that standard ASCII encoding is used to represent the characters. Keeping this intuition in mind, we write for example $\mathtt{a}$ to denote the character 'a'.

There is a *built-in* (predefined) *signature* of function symbols and a built-in theory (set of axioms) for those symbols. Each function symbol $f$ of arity $n \geq 0$ has a given domain sort $\sigma_0 \times \cdots \times \sigma_{n-1}$ and a given range sort $\sigma$, $f : \sigma_0 \times \cdots \times \sigma_{n-1} \to \sigma$. For example, there is a built-in *relation* or *predicate* (Boolean function) symbol $< : \mathbb{C} \times \mathbb{C} \to \mathbb{B}$ that provides a strict total order of all the characters. One can also declare *fresh* (new) *uninterpreted* function symbols $f$ of arity $n \geq 0$, for a given domain sort and a given range sort. Using model theoretic terminology, these new symbols *expand* the signature.

*Terms* and *formulas* (or Boolean terms) are defined by induction as usual and are assumed to be well-sorted. We write $FV(t)$ for the set of free variables in a term (or formula) $t$. A term or formula without free variables is *closed*. Let $\mathcal{F}_{\mathbb{C}}$ denote the set of all formulas without uninterpreted function symbols and at most one fixed free variable of sort $\mathbb{C}$. *Throughout the paper, we denote that variable by $\chi$.* Given a formula $\varphi \in \mathcal{F}_{\mathbb{C}}$, and a character or term $t$ of sort $\mathbb{C}$, we write $\varphi[t]$ for the formula where each occurrence of $\chi$ is replaced by $t$. For example, if $\varphi$ is $\mathtt{a} < \chi \wedge \chi < \mathtt{d}$ then $FV(\varphi) = \{\chi\}$ and $\varphi[\mathtt{b}]$ is the formula $\mathtt{a} < \mathtt{b} \wedge \mathtt{b} < \mathtt{d}$.

A *model* is a mapping from function symbols to their interpretations (values). The built-in function symbols have the same interpretation in all models, keeping that in mind, we may omit them from the model. A model $M$ *satisfies* a closed formula $\varphi$, written $M \models \varphi$, if $M$ provides an interpretation for all the uninterpreted function symbols in $\varphi$ that makes $\varphi$ true. For example, let $f : \mathbb{C} \to \mathbb{C}$ be an uninterpreted function symbol and $c : \mathbb{C}$ be an uninterpreted constant. Let $M$ be a model where $c^M$ (the interpretation of $c$ in $M$) is $\mathtt{a}$ and $f^M$ is a function that maps all characters to $\mathtt{b}$. Then $M \models \mathtt{a} < f(c)$ but $M \not\models \mathtt{a} < c$.

A closed formula $\varphi$ is *satisfiable* if it has a model. A formula $\varphi$ with $FV(\varphi) = \bar{x}$ is *satisfiable* if its existential closure $\exists\bar{x}\varphi$ is satisfiable. We write $\models_{\mathcal{U}} \varphi$, or $\models \varphi$, if $\varphi$ is *valid* (true in all models). Some examples: $\mathtt{a} < \mathtt{b} \wedge \mathtt{b} < \mathtt{d}$ is valid; $\mathtt{a} < \chi \wedge \chi < \mathtt{b}$ is unsatisfiable because there exists no character that is strictly greater that $\mathtt{a}$ and strictly smaller than $\mathtt{b}$; $\mathtt{0} < \chi \wedge \chi < \mathtt{4}$ is satisfiable, e.g., let $\chi = \mathtt{3}$.

## III. Symbolic finite automata

We use a representation of finite automata where several transitions from a source state to a target state are combined into a single symbolic move. Formally, a collection of transitions $(p, a_1, q), \ldots, (p, a_n, q)$ are represented by a single *(symbolic) move* $(p, \varphi, q)$ *from $p$ to $q$*, where $\varphi \in \mathcal{F}_{\mathbb{C}}$, such that

$$[\![\varphi]\!] = \{a_1, \ldots, a_n\},$$

where $[\![\varphi]\!] \stackrel{\text{def}}{=} \{a \mid a \in \mathcal{U}^{\mathbb{C}}, \models \varphi[a]\}$. Let also

$$[\![(p, \varphi, q)]\!] \stackrel{\text{def}}{=} \{(p, a, q) \mid a \in [\![\varphi]\!]\},$$

and, given a set $\Delta$ of moves, let

$$[\![\Delta]\!] \stackrel{\text{def}}{=} \{\tau \mid \delta \in \Delta, \tau \in [\![\delta]\!]\}.$$

Note that $[\![(p, \varphi, q)]\!] = \emptyset$ iff $\varphi$ is unsatisfiable. Define also

$$\begin{aligned} \mathit{Source}((p, \varphi, q)) &\stackrel{\text{def}}{=} p, \\ \mathit{Target}((p, \varphi, q)) &\stackrel{\text{def}}{=} q, \\ \mathit{Cond}((p, \varphi, q)) &\stackrel{\text{def}}{=} \varphi. \end{aligned}$$

For example, the move

$$(p, \mathtt{a} \leq \chi \wedge \chi \leq \mathtt{z}, q)$$

represents the set of all transitions $(p, c, q)$ where $c$ is a character between $\mathtt{a}$ and $\mathtt{z}$. Formally, we refer to such a representation of a finite automata (FA) as follows.

*Definition 1:* A *Symbolic Finite Automaton* or *SFA* $A$ is a tuple $(Q, q_0, F, \Delta)$, where $Q$ is a finite set of *states*, $q_0 \in Q$ the *initial state*, $F \subseteq Q$ is the set of *final states*, and $\Delta : Q \times \mathcal{F}_{\mathbb{C}} \times Q$ is the *move relation*.

We sometimes use $A$ as a subscript to identify its components. Just as with finite automata, it is often useful to add *epsilon moves* to an SFA. Consider a special symbol $\epsilon$ that is not in the background universe.

*Definition 2:* An *SFA with epsilon moves* or *$\epsilon$SFA* is a tuple $(Q, q_0, F, \Delta)$, where $Q$, $q_0$ and $F$ are as above, and $\Delta : Q \times (\mathcal{F}_{\mathbb{C}} \cup \{\epsilon\}) \times Q$.

The term SFA without the additional qualification allowing epsilon moves implies that epsilon moves do not occur. (Obviously, any SFA is also an $\epsilon$SFA.) Let $[\![(p, \epsilon, q)]\!] \stackrel{\text{def}}{=} (p, \epsilon, q)$. An $\epsilon$SFA $A = (Q, \Delta, q_0, F)$ denotes the finite automaton $[\![A]\!]$ with epsilon moves, where

$$[\![A]\!] \stackrel{\text{def}}{=} (Q, \mathcal{U}^{\mathbb{C}}, [\![\Delta]\!], q_0, F).$$

We write $\Delta_A^{\epsilon}$ for the set of all epsilon moves in $\Delta_A$ and $\Delta_A^{\not\epsilon}$ for $\Delta_A \setminus \Delta_A^{\epsilon}$.

*Definition 3:* An $\epsilon$SFA $A$ is *normalized* if there are no two distinct moves $(p, \varphi_1, q), (p, \varphi_2, q)$ in $\Delta_A^{\not\epsilon}$.

[1] Some Unicode encodings such as UTF-32, need more than 16 bits.

It is clear that for any $\epsilon$SFA $A$ there is a normalized SFA $A'$ such that $[\![A]\!] = [\![A']\!]$: for all states $p$ and $q$ in $Q_A$, make a disjunction $\varphi$ of all the conditions of the moves from $p$ to $q$ in $\Delta_A^{\not\epsilon}$ and let $(p, \varphi, q)$ be the single move in $\Delta_{A'}^{\not\epsilon}$ that goes from $p$ to $q$.

A move is *satisfiable* if its condition is satisfiable. Note that unsatisfiable moves are clearly superfluous and can always be omitted.

*Definition 4:* An $\epsilon$SFA $A$ is *clean* if all moves in $\Delta_A^{\not\epsilon}$ are satisfiable.

*Definition 5:* An SFA $A$ is *deterministic*, called *DSFA*, if $[\![A]\!]$ is deterministic.

The following proposition follows easily from the definitions and is used in characterizing DSFAs.

*Proposition 1:* The following statements are equivalent.

1) $A$ is deterministic.
2) For any two moves $(p, \varphi_1, q_1)$ and $(p, \varphi_2, q_2)$ in $\Delta_A$, if $q_1 \neq q_2$ then $\varphi_1 \wedge \varphi_2$ is unsatisfiable.

*Definition 6:* The language (set of strings) *accepted* by an SFA $A$, $L(A)$, is the language accepted by the finite automaton $[\![A]\!]$. Two SFAs are *equivalent* if they accept the same language.

*Definition 7:* A DSFA $A$ is *minimal* if $A$ is normalized, clean, and $[\![A]\!]$ is minimal.

Note that if a DSFA $A$ is minimal then it is unique up to logical equivalence of conditions and renaming of states.

### *A. From regular expressions to $\epsilon$SFAs*

We use [5] as the concrete language definition of regular expression patterns or *regexes* in this paper. Not all constructs are supported. Advanced regular expression languages offer features that go beyond classical regular expressions, e.g. with constructs such as "as few times as possible"-quantifiers (see also Section IX). Regarding the supported subset of regexes, besides a few extensions, the translation from a regex to an $\epsilon$SFA follows very closely the standard algorithm described in [7, Section 2.5] for converting a standard regular expression into a finite automaton with epsilon moves. For handling negations and character ranges, the translation creates a corresponding formula in $\mathcal{F}_{\mathbb{C}}$. A sample regex and $\epsilon$SFA are shown in Figure 1.

### *B. Algorithms on SFAs*

We revisit variations of standard algorithms on finite automata to perform equivalence preserving transformations on symbolic finite automata, and we also look at the product construction in order to encode intersection constraints:

1) *Epsilon elimination* from $\epsilon$SFAs;
2) *Determinization* of SFAs;
3) *Minimization* of DSFAs.
4) *Product* of SFAs.

In Section IV SFAs are encoded as inputs to the SMT solver in form of *language acceptors*. The above algorithms are used in Section VI to evaluate their effect on the performance of Rex under different equivalent encodings.

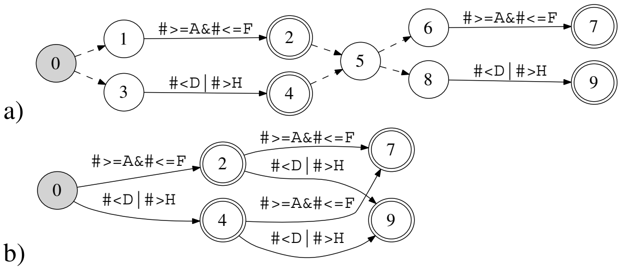


Figure 1. a) Sample $\epsilon$SFA generated by Rex from the regex `([A-F]|[^D-H]){1,2}`; b) equivalent SFA. The initial state is grey, the epsilon moves are dashed. The symbol `&` is used for conjunction and the symbol | is used for disjunction. The variable $\chi$ is denoted by `#`.

Note that all of the listed problems have a naive solution by using the underlying finite automata algorithms, but these algorithms often depend on the explicit (rather than symbolic) use of the characters, and are impractical when the alphabet is large, e.g., when it contains all the UTF-16 characters.

The algorithms are discussed next.

*1) Epsilon elimination:* The input to the algorithm is an $\epsilon$SFA $A$ and the output is an equivalent SFA $B$. We assume, without loss of generality, that $A$ is normalized. We reuse the notion of the *epsilon closure* [7, Section 2.4] of a state $q$ in $A$, denoted here by $\epsilon C(q)$.

(i) For all $q \in Q_A$ compute $\epsilon C(q)$ as the least subset of $Q_A$ such that $q \in \epsilon C(q)$, and if $q_1 \in \epsilon C(q)$ and $(q_1, \epsilon, q_2) \in \Delta_A$ then $q_2 \in \epsilon C(q)$.

(ii) Compute a partial map $E$ from $Q_A \times Q_A$ to $\mathcal{F}_{\mathbb{C}}$ such that, for all $(q, \_, r) \in \Delta_A^{\not\epsilon}$,

$$E(q, r) = \bigvee \{\varphi \mid \exists p\, (p \in \epsilon C(q), (p, \varphi, r) \in \Delta_A^{\not\epsilon})\}.$$

(iii) View $\mathrm{Dom}(E)$ as a directed graph and eliminate all edges and states that are not reachable from $q_{0A}$.

(iv) Let $B$ have the following components:
- $Q_B = \{p, q \mid (p, q) \in \mathrm{Dom}(E)\}$;
- $q_{0B} = q_{0A}$;
- $F_B = \{q \mid q \in Q_B, \epsilon C(q) \cap F_A \neq \emptyset\}$;
- $\Delta_B = \{(p, E(p, q), q) \mid (p, q) \in \mathrm{Dom}(E)\}$.

Step (iii) is not necessary but eliminates states and moves that are redundant; often half of the original states are redundant. The algorithm can be implemented in time linear in the size of $A$. For example the epsilon closures can be represented by shared linked lists. The result of applying the algorithm to the $\epsilon$SFA in Figure 1(a) is illustrated if Figure 1(b).

*2) Determinization:* The input to the algorithm is an SFA $A$ and the output is an equivalent DSFA $B$. We assume, without loss of generality, that $A$ is normalized. We use the following notations.

$$\begin{aligned}\Delta_A(q) &\overset{\text{def}}{=} \{t \mid t \in \Delta_A, \mathit{Source}(t) = q\}\\ \Delta_A(\mathbf{q}) &\overset{\text{def}}{=} \cup\{\Delta_A(q) \mid q \in \mathbf{q}\}\\ \mathit{Target}(\mathbf{t}) &\overset{\text{def}}{=} \cup\{\mathit{Target}(t) \mid t \in \mathbf{t}\}\end{aligned}$$

It is convenient to describe the algorithm as a depth-first-search algorithm using a stack $S$ of $B$ states as a frontier, a set $V$ of visited $B$ states, and a set $T$ of moves.

(i) Initially $S = (\{q_{0A}\})$, $V = \{\{q_{0A}\}\}$, and $T = \emptyset$.

(ii) If $S$ is empty proceed to (iv) else pop $\mathbf{q}$ from $S$.

(iii) For each nonempty subset $\mathbf{t}$ of $\Delta_A(\mathbf{q})$, let[2]

$$\varphi_{\mathbf{t}} = (\bigwedge_{t\in\mathbf{t}} \mathit{Cond}(t)) \wedge (\bigwedge_{t\in\Delta_A(\mathbf{q})\setminus\mathbf{t}} \neg\mathit{Cond}(t))$$

*If* $\varphi_{\mathbf{t}}$ *is satisfiable* then

- add $(\mathbf{q}, \varphi_{\mathbf{t}}, \mathit{Target}(\mathbf{t}))$ to $T$;
- if $\mathit{Target}(\mathbf{t})$ is not in $V$ then add $\mathit{Target}(\mathbf{t})$ to $V$ and push $\mathit{Target}(\mathbf{t})$ to $S$.

Proceed to (ii).

(iv) Let $B = (V, \{q_{0A}\}, \{\mathbf{q} \in V \mid \mathbf{q} \cap F_A \neq \emptyset\}, T)$.

The satisfiability check of $\varphi_{\mathbf{t}}$ is performed for example with an SMT solver and ensures that $B$ is clean. Without that check $B$ may get cluttered with unsatisfiable moves and states that are unreachable.

Given $(\mathbf{q}, \varphi_{\mathbf{t}_1}, \mathit{Target}(\mathbf{t}_1))$ and $(\mathbf{q}, \varphi_{\mathbf{t}_2}, \mathit{Target}(\mathbf{t}_2))$ in $T$ such that $\mathit{Target}(\mathbf{t}_1) \neq \mathit{Target}(\mathbf{t}_2)$, it follows immediately that $\mathbf{t}_1 \neq \mathbf{t}_2$ and thus $\varphi_{\mathbf{t}_1} \wedge \varphi_{\mathbf{t}_2}$ is unsatisfiable because there is at least one $t \in \Delta_A(\mathbf{q})$ such that both $\mathit{Cond}(t)$ and $\neg\mathit{Cond}(t)$ are conjuncts in $\varphi_{\mathbf{t}_1} \wedge \varphi_{\mathbf{t}_2}$. Thus $B$ is deterministic by Proposition 1.

*3) Minimization:* The input to the algorithm is a DSFA $A$ and the output is an equivalent minimal DSFA $B$. We assume, without loss of generality, that $A$ is normalized and clean. In order for the algorithm to work correctly we also need to assume that $A$ is *total*, meaning that for all $a \in \mathcal{U}^{\mathbb{C}}$ and all $q \in Q_A$ there is a transition $(q, a, p)$ in $[\![A]\!]$ for some $p \in Q_A$. To make $A$ total add a new "dead" state $d$ to it, add the move $(d, \mathit{true}, d)$, and from each state $q$ such that $\varphi = \bigwedge_{t\in\Delta_A(q)} \neg\mathit{Cond}(t)$ is satisfiable, add the move $(q, \varphi, d)$.

(i) Initialize $E$ to be the equivalence relation over $Q_A$ such that $E(p, q) \Leftrightarrow p, q \in F_A$.

(ii) If there exists $(p, q)$ in $E$ such that there are moves $(p, \varphi, p_1)$ and $(q, \psi, q_1)$ in $\Delta_A$ where $p_1 \neq q_1$ and $(p_1, q_1) \notin E$ and $\varphi \wedge \psi$ *is satisfiable*, then remove $(p, q)$ from $E$ and repeat (ii).

(iii) Let $B$ have the following components:

- $Q_B$ is the set of $E$-classes $\{[q] \mid q \in Q_A\}$;
- $q_{0B}$ is the $E$-class $[q_{0A}]$;
- $F_B$ is the set of $E$-classes $\{[q] \mid q \in F_A\}$;
- $\Delta_B$ is $\{([q], \varphi, [p]) \mid (q, \varphi, p) \in \Delta_A\}$.

(iv) Normalize $B$, and if $B$ has a dead state (a state from which no final state can be reached), eliminate all moves to the dead state and eliminate the dead state unless it is $q_{0B}$.[3]

*4) Product construction:* The input to the algorithm are two SFAs $A$ and $B$ and the output is an SFA $C$ that is the *product* of $A$ and $B$, such that $L(C) = L(A) \cap L(B)$. The algorithm is more or less standard, we are describing it to pinpoint some aspects of it that are important when the product construct is used below in Section VI.

As above, it is convenient to describe the algorithm as a depth-first-search algorithm using a stack $S$ of states of $C$ as a frontier, a set $V$ of visited states, and a set $T$ of moves.

(i) Initially $S = (\langle q_{0A}, q_{0B}\rangle)$, $V = \{\langle q_{0A}, q_{0B}\rangle\}$, $T = \emptyset$.

(ii) If $S$ is empty go to (iv) else pop $\langle q_1, q_2\rangle$ from $S$.

(iii) Iterate for each $t_1 \in \Delta_A(q_1)$ and $t_2 \in \Delta_B(q_2)$, let $\varphi = \mathit{Cond}(t_1) \wedge \mathit{Cond}(t_2)$, let $p_1 = \mathit{Target}(t_1)$, and let $p_2 = \mathit{Target}(t_2)$. *If* $\varphi$ *is satisfiable* then

- add $(\langle q_1, q_2\rangle, \varphi, \langle p_1, p_2\rangle)$ to $T$;
- if $\langle p_1, p_2\rangle$ is not in $V$ then add $\langle p_1, p_2\rangle$ to $V$ and push $\langle p_1, p_2\rangle$ to $S$.

Proceed to (ii).

(iv) Let $C = (\langle q_{0A}, q_{0B}\rangle, V, \{q \in V \mid q \in F_A \times F_B\}, T)$.

(v) Eliminate *dead states* from $C$ (states from which no final state is reachable).

Note that $|Q_C|$ is at most $|Q_A| * |Q_B|$. The satisfiability check in (iii) is important. It prevents unnecessary exploration of *unreachable* states, and may avoid a quadratic blowup of $Q_C$, whereas (v) avoids introduction of useless "dead end"-axioms in the symbolic language acceptor.

## IV. Symbolic language acceptors

In addition to a quantifier free *goal* formula $\psi$ that is provided to an SMT solver and for which proof of (or absence of) satisfiability is sought, one can also assert additional universally quantified *axioms* to the solver. We use axioms to encode *language acceptors* for $\epsilon$SFAs. We are using the programmatic API of the SMT solver Z3 [4], [9] in Rex. The description below follows closely the use of Z3 (although using a more mathematical notation) and is intended to be self-contained.

### A. On axioms in SMT solvers

During proof search, axioms are triggered by matching subexpressions in the goal. In Rex, we use particular kinds of axioms that are equivalences of the form

$$\forall\bar{x}(\varphi_{\text{lhs}} \Leftrightarrow \varphi_{\text{rhs}}) \qquad (1)$$

[2] Note that the empty conjunction $(\bigwedge_{t\in\emptyset} \ldots)$ is the same as $\mathit{true}$.

[3] Note that there can be at most one dead state, and if the language accepted by $A$ is empty then $B$ has a single state that is not final and $B$ has no moves.

where $FV(\varphi_{\rm lhs}) = \bar{x}$ and $FV(\varphi_{\rm rhs}) \subseteq \bar{x}$. The lhs $\varphi_{\rm lhs}$ of (1) is called the *pattern* of (1). In the axioms below, we underline the patterns.

The high-level view behind the use of axioms like (1) is as follows. The axiom (1) is *triggered* by the current goal $\psi$, if $\psi$ contains a subformula $\gamma$ and there exists a substitution $\theta$ such that $\gamma = \varphi_{\rm lhs}\theta$, i.e., $\gamma$ *matches the pattern* of the axiom. If (1) is triggered, then the current goal $\psi$ is replaced by the *logically equivalent* formula where $\gamma$ has been replaced by $\varphi_{\rm rhs}\theta$.

Thus, the axioms are used as "rewrite rules" in our case, and each application of an axiom preserves the logical equivalence to the original goal. As long as there exists an axiom that can be triggered, then triggering is guaranteed. Thus, termination is in general not guaranteed when (mutually) recursive axioms are being used.

### B. Representation of strings

For each sort $\sigma$ there is also a *list sort* $\mathbb{L}\langle\sigma\rangle$. Lists are provided as built-in algebraic datatypes and are accompanied with standard constructors and accessors in Z3. For a given element sort $\sigma$ there is an empty list *nil* (of sort $\mathbb{L}\langle\sigma\rangle$) and if $e$ is an element of sort $\sigma$ and $l$ is a list of sort $\mathbb{L}\langle\sigma\rangle$ then $cons(e, l)$ is a list of sort $\mathbb{L}\langle\sigma\rangle$. The accessors are, as usual, *hd* (head) and *tl* (tail). Strings are represented by lists of characters; we write $\mathbb{S}$ for the sort $\mathbb{L}\langle\mathbb{C}\rangle$. The empty string is abbreviated by `""` and a string $cons(\texttt{a}, cons(\texttt{b}, cons(\texttt{c}, nil)))$ is abbreviated by `"abc"`, e.g., $hd(\texttt{"abc"}) = \texttt{a}$ and $tl(\texttt{"abc"}) = \texttt{"bc"}$.

### C. Unary numbers

One can define other (arbitrarily nested) algebraic datatypes in Z3. We are using *unary natural numbers* as an algebraic datatype for reasons explained below. We declare $\mathbb{N}$ as the corresponding sort, and we declare the constructors

$$\overline{0} : \mathbb{N}, \quad \mathbf{s} : \mathbb{N} \to \mathbb{N}.$$

We write $\overline{k+1}$ for $\mathbf{s}(\overline{k})$.

### D. From $\epsilon$SFAs to axioms

Let $A$ be a given $\epsilon$SFA. Assume $A$ is normalized. For all $q \in Q_A$, declare the predicate symbol

$$Acc^A_q : \mathbb{S} \times \mathbb{N} \to \mathbb{B}$$

The idea behind the axioms defined below is that $Acc^A_q(s, \overline{k})$ holds for a string $s$ iff the length of $s$ is at most $k$ and there is a path from $q$ that reads $s$ and leads to a final state. In particular, if $Acc^A_{q_0A}(s, \overline{k})$ is true then $s$ is accepted by $[\![A]\!]$. Let

$$Acc^A \stackrel{\rm def}{=} Acc^A_{q_0A}$$

The role of the second argument of $Acc^A_q$ is to guarantee that the triggering process of axioms terminates. To this end, the sort $\mathbb{N}$ is used (rather than the built-in integer sort). This enables us to define the patterns below, since constructors of algebraic datatypes can be used effectively in patterns. Intuitively, when such an axiom is used then there is always something that strictly decreases and implies that there exists a well-ordering so that each time an axiom is applied the resulting goal is smaller with respect to that ordering. It is not possible to write axioms that use integers in this way because then one cannot associate a well-ordering with the axioms.

For $q \in Q_A$, assume $\Delta_A(q)$ is

$$\{(q, \varphi_1, q_1), \ldots, (q, \varphi_m, q_m), (q, \epsilon, p_1), \ldots, (q, \epsilon, p_n)\}$$

and define the axioms $ax0^A_q$ and $ax1^A_q$, where the patterns are underlined,[4]

$$\begin{array}{l} ax0^A_q \stackrel{\rm def}{=} \\ \quad \forall x\, (\underline{Acc^A_q(x, \overline{0})} \Leftrightarrow \bigvee_{i=1}^{n} Acc^A_{p_i}(x, \overline{0})\ \underbrace{\vee\ x = nil}_{\text{if } q \in F_A}) \end{array}$$

$$\begin{array}{l} ax1^A_q \stackrel{\rm def}{=} \\ \quad \forall x\, y\, (\underline{Acc^A_q(x, \mathbf{s}(y))} \Leftrightarrow \\ \qquad (x \neq nil \wedge (\bigvee_{i=1}^{m} (\varphi_i[hd(x)] \wedge Acc^A_{q_i}(tl(x), y)))) \\ \qquad \vee\ \bigvee_{i=1}^{n} Acc^A_{p_i}(x, \mathbf{s}(y))\ \underbrace{\vee\ x = nil}_{\text{if } q \in F_A}) \end{array}$$

Let $Th(A) \stackrel{\rm def}{=} \{ax0^A_q, ax1^A_q \mid q \in Q_A\}$. The set of formulas $Th(A)$ (or equivalently $\bigwedge Th(A)$) is asserted to the solver as the *axioms for* $A$.

*Definition 8:* An *epsilon loop* in an $\epsilon$SFA $A$ is a path of epsilon moves that starts and ends in the same state.

*Theorem 1:* Let $A$ be an $\epsilon$SFA without epsilon loops. Then $\bigwedge Th(A) \wedge Acc^A(s, \overline{k})$ is satisfiable iff $A$ accepts $s$ and the length of $s$ is at most $k$.

The proof of Theorem 1 is outlined in [10]. Let us consider a few examples.

*Example 1:* Consider $A = (\{q\}, q, \emptyset, \emptyset)$. The language accepted by $A$ is obviously empty. The axioms $Th(A)$ for $A$ are

$$\begin{array}{rcl} ax0^A_q & = & \forall x\, (\underline{Acc^A_q(x, \overline{0})} \Leftrightarrow false) \\ ax1^A_q & = & \forall x\, y\, (\underline{Acc^A_q(x, \mathbf{s}(y))} \Leftrightarrow false) \end{array}$$

Thus, if $M \models Th(A)$ then there is no string $s$ or number $k$ such that $M \models Acc^A_q(s, \overline{k})$. ☒

*Example 2:* Consider $A = (\{q\}, q, \{q\}, \{(q, true, q)\})$. The language accepted by $A$ is the set of all strings. The axioms $Th(A)$ for $A$ are

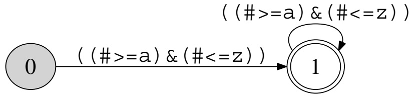
$$\begin{array}{rcl} ax0^A_q & = & \forall x\, (\underline{Acc^A_q(x, \overline{0})} \Leftrightarrow x = nil) \\ ax1^A_q & = & \forall x\, y\, (\underline{Acc^A_q(x, \mathbf{s}(y))} \Leftrightarrow \\ & & \quad (x \neq nil \wedge Acc^A_q(tl(x), y)) \vee x = nil) \end{array}$$

Assume $M \models Th(A)$. Since $M \models ax0^A_q$, we know that $Acc^A_q(\texttt{""}, \overline{0})^M = true$. By induction on $k$, it follows from

[4] Note that the empty disjunction is the same as *false*.

$M \models \mathit{ax1}^A_q$ that $\mathit{Acc}^A_q(s,\overline{k})^M = \mathit{true}$ for all strings $s$ of length at most $k$. ⊠

*Example 3:* Consider the regex `[a-z]+` and the following SFA $A$ for it (that also happens to be minimal):



The axioms $\mathit{Th}(A)$ for $A$ are as follows, where $\varphi[\chi]$ is the formula $(\chi \geq \mathtt{a}) \wedge (\chi \leq \mathtt{z})$,

$$\begin{aligned}
\mathit{ax0}^A_0 &= \forall x\,(\underline{\mathit{Acc}^A_0(x,\overline{0})} \Leftrightarrow \mathit{false})\\
\mathit{ax1}^A_0 &= \forall x\,y\,(\underline{\mathit{Acc}^A_0(x,\mathrm{s}(y))} \Leftrightarrow x \neq \mathit{nil} \wedge \varphi[\mathit{hd}(x)]\\
&\qquad\qquad \wedge\ \mathit{Acc}^A_1(\mathit{tl}(x),y))\\
\mathit{ax0}^A_1 &= \forall x\,(\underline{\mathit{Acc}^A_1(x,\overline{0})} \Leftrightarrow x = \mathit{nil})\\
\mathit{ax1}^A_1 &= \forall x\,y\,(\underline{\mathit{Acc}^A_1(x,\mathrm{s}(y))} \Leftrightarrow (x \neq \mathit{nil} \wedge \varphi[\mathit{hd}(x)]\\
&\qquad\qquad \wedge\ \mathit{Acc}^A_1(\mathit{tl}(x),y)) \vee x = \mathit{nil})
\end{aligned}$$

Declare a fresh (uninterpreted) constant $s\!:\!\mathbb{S}$ and assert the axioms $\mathit{Th}(A)$ and the goal $\mathit{Acc}^A_0(s,\overline{2})$ to the solver. We describe a plausible scenario for the resulting model generation process. First, the axioms are triggered:

$$\begin{aligned}
\mathit{Acc}^A_0(s,\overline{2}) \quad &\overset{\mathit{ax1}^A_0}{\rightsquigarrow} \quad s \neq \mathit{nil} \wedge \varphi[\mathit{hd}(s)] \ \wedge\\
&\qquad \mathit{Acc}^A_1(\mathit{tl}(s),\overline{1})\\
&\overset{\mathit{ax1}^A_1}{\rightsquigarrow} \quad s \neq \mathit{nil} \wedge \varphi[\mathit{hd}(s)] \ \wedge\\
&\qquad ((\mathit{tl}(s) \neq \mathit{nil} \wedge \varphi[\mathit{hd}(\mathit{tl}(s))] \ \wedge\\
&\qquad \mathit{Acc}^A_1(\mathit{tl}(\mathit{tl}(s)),\overline{0})) \vee \mathit{tl}(s) = \mathit{nil})\\
&\overset{\mathit{ax0}^A_1}{\rightsquigarrow} \quad s \neq \mathit{nil} \wedge \varphi[\mathit{hd}(s)] \ \wedge\\
&\qquad ((\mathit{tl}(s) \neq \mathit{nil} \wedge \varphi[\mathit{hd}(\mathit{tl}(s))] \ \wedge\\
&\qquad \mathit{tl}(\mathit{tl}(s)) = \mathit{nil}) \vee \mathit{tl}(s) = \mathit{nil})
\end{aligned}$$

Second, a model is generated for the resulting (quantifier free) formula (if a model exists) using the built-in theories. In this case a possible model $M$ is such that $s^M = \texttt{"ok"}$.

In Z3 the resulting model $M$ provides also an interpretation for all the predicate symbols $\mathit{Acc}^A_q$. For entries $(e,\overline{k})$ that did *not* occur in the derivation, the interpretation is arbitrary – typically a default value of the range sort, that is $\mathit{false}$ for $\mathbb{B}$. In other words, $M$ is not necessarily a model for $\mathit{Th}(A)$, since it may violate the axioms for entries that are irrelevant with respect to the goal. ⊠

The condition that $A$ has no epsilon loops is necessary in Theorem 1. The theorem would fail otherwise, as the following example illustrates.

*Example 4:* Consider $A = (\{q\}, q, \emptyset, \{(q,\epsilon,q)\})$. The language accepted by $A$ is obviously empty because there are no final states. The axioms $\mathit{Th}(A)$ for $A$ are

$$\begin{aligned}
\mathit{ax0}^A_q &= \forall x\,(\underline{\mathit{Acc}^A_q(x,\overline{0})} \Leftrightarrow \mathit{Acc}^A_q(x,\overline{0}))\\
\mathit{ax1}^A_q &= \forall x\,y\,(\underline{\mathit{Acc}^A_q(x,\mathrm{s}(y))} \Leftrightarrow \mathit{Acc}^A_q(x,\mathrm{s}(y)))
\end{aligned}$$

The axioms are simply useless logical tautologies. Consider for example a model $M$ with an interpretation for $\mathit{Acc}^A_q$ such that $M \models \mathit{Acc}^A_q(\texttt{""},\overline{0})$. Trivially, $M$ is also a model for $\mathit{Th}(A)$ but `""` is not accepted by $A$. Moreover, if the axioms were asserted to Z3, the resulting proof search for a goal such as $\mathit{Acc}^A_q(\texttt{""},\overline{0})$ would not terminate. ⊠

When an $\epsilon$SFA is created from a regex the property that there are no epsilon loops follows immediately from the constructions in [7]. So the case when epsilon loops are present is not relevant here.

## V. Implementation and use

The automata algorithms discussed in Section III and the axiom generation discussed in Section IV have been implemented in *Rex*. The SMT solver Z3 is used for satisfiability checking and model generation. Interaction with Z3 is implemented through its programmatic API rather than using a textual format, such as the smt-lib format [11]. The main reasons for this are:

- the API provides access to built-in datatypes, such as algebraic datatypes, and corresponding theories that are not (yet) part of the smt-lib standard;
- the API enables working within a given Z3 context.

The first point was illustrated clearly in Section IV. The second point is equally important, it allows Rex to be used as a decision procedure that is seamlessly integrated with Z3, and used by other tools such as Pex [12], [1] (for dealing with regular expression constraints in parameterized unit tests) and Qex [13], [3] (for dealing with LIKE expressions in database unit tests). In Rex, Z3 is also used for checking constraint satisfiability in the implementation of the algorithm steps described in Section III-B2(iii), Section III-B3(ii), and Section III-B4(iii).

### A. Working within a Z3 context

A (Z3) *context* includes declarations for a set of symbols, assertions for a set of formulas, and the status of the last satisfiability check (if any). There is a *current context* and a backtrack stack of previous contexts. Contexts can be saved through *pushing* and restored through *popping*.

The following is an actual code snippet from Rex illustrating how the satisfiability of a formula `f` is checked in the current context without "cluttering" the context.

```
z3.Push();
z3.AssertCnstr(f);
LBool isSat = z3.Check();
z3.Pop(); ...
```

### B. Model generation

Besides allowing to check satisfiability, perhaps the most important feature of SMT solvers is generating a *model* as a witness of the satisfiability check, i.e., a mapping of the uninterpreted function symbols in the current context to their interpretations. The Z3 API has a separate method for satisfiability checking with model generation. This code snippet illustrates the use of that functionality:

Table I
SAMPLE REGEXES.

<table>
<tr><td>#1</td><td>\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*([,;]\s*\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)*</td></tr>
<tr><td>#2</td><td>$?(\d{1,3},?(\d{3},?)*\d{3}(\.\d{0,2})?|\d{1,3}(\.\d{0,2})?|\.\d{1,2}?)</td></tr>
<tr><td>#3</td><td>([A-Z]{2}|[a-z]{2} \d{2}  [A-Z]{1,2}|[a-z]{1,2} \d{1,4})?([A-Z]{3}|[a-z]{3} \d{1,4})?</td></tr>
<tr><td>#4</td><td>[A-Za-z0-9](([ \.\-]?[a-zA-Z0-9]+)*)@([A-Za-z0-9]+)(([\.\-]?[a-zA-Z0-9]+)*)\. ([A-Za-z][A-Za-z]+)</td></tr>
<tr><td>#5</td><td>(\w|-)+@((\w|-)+\.)+(\w|-)+</td></tr>
<tr><td>#6</td><td>[+-]?([0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)([eE][+-]?[0-9]+)?</td></tr>
<tr><td>#7</td><td>((\w|\d|\-|\.)+)@{1}(((\w|\d|\-){1,67})|((\w|\d|\-)+\.(\w|\d|\-){1,67}))\.((([a-z]|[A-Z]|\d){2,4})(\.([a-z]|[AZ]|\d){2})?)</td></tr>
<tr><td>#8</td><td>(([A-Za-z0-9]+ +)|([A-Za-z0-9]+\-+)|([A-Za-z0-9]+\.+)|([A-Za-z0-9]+\++))*[A-Za-z0-9]+@((\w+\-+)|(\w+\.))*\w{1,63}\.[a-zA-Z]{2,6}</td></tr>
<tr><td>#9</td><td>(([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.([a-zA-Z]{2,5}){1,25})+([;.](([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.([a-zA-Z]{2,5}){1,25})+)*</td></tr>
<tr><td>#10</td><td>((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[,]{0,1}\s*)+</td></tr>
</table>

Table II
EVALUATION RESULTS FOR SAMPLE REGEXES.

| | $\epsilon$**SFA**$(r)$ | | **SFA**$(r)$ | | **DSFA**$(r)$ | | **mDSFA**$(r)$ | |
|---|---|---|---|---|---|---|---|---|
| $r$ | *size* | $t$ ms | *size* | $t$ ms | *size* | $t$ ms | *size* | $t$ ms |
| #1 | 91 | 100 | 73 | 40 | 81 | 70 | 20 | 140 |
| #2 | 90 | 10 | 64 | 10 | 71 | 30 | 29 | 40 |
| #3 | 83 | 10 | 70 | 10 | 104 | 30 | 69 | 100 |
| #4 | 45 | 40 | 35 | 60 | 53 | 70 | 26 | 70 |
| #5 | 98 | 100 | 71 | 10 | 74 | 30 | 15 | 40 |
| #6 | 31 | 0 | 12 | 0 | 16 | 10 | 10 | 10 |
| #7 | 2728 | 840 | 920 | 1800 | | | | |
| #8 | 281 | 40 | 269 | 60 | 380 | 170 | 296 | 870 |
| #9 | 1944 | 280 | 2128 | 260 | | | | |
| #10 | 112 | 30 | 104 | 30 | | | | |

```
Model m;
z3.AssertCnstr(f);
LBool sat = z3.CheckAndGetModel(out m);
Term v = m.Eval(s); ...
```

Suppose that f above is the conjunction of $Th(A)$ for a given SFA $A$ and the goal $Acc^A(s, \overline{|Q_A|})$. If $L(A)$ is nonempty then the value of v is a string in $L(A)$.

## VI. EXPERIMENTS

We evaluated the performance of Rex on a collection of sample regexes shown in Table I.[5] These are typical examples of concrete regexes appearing in various practical contexts. The regexes are taken from [6], *where the technique is not able to handle regexes #7 and #8*. Table I excludes some of the samples from [6] due to shortcomings of the regex parser that we use temporarily in Rex.

For each regex $r$ we conducted the following experiments, that are summarized in Table II. We constructed the $\epsilon$SFA, SFA, DSFA and minimal DSFA for $r$ using the algorithms described in Section III. For regexes #7, #9 and #10, determinization timed out (using a timeout of 10 seconds).

The size of each automaton, shown in column *size*, is its number of moves plus its number of states. The graph of each automaton is sparse, having, in average, at most twice as many moves as states; e.g., Figure 2 shows a typical $\epsilon$SFA generated for one of the regexes in Table I.

For every automaton $A$ we performed an independent *member generation* experiment as follows.

1) Declare a fresh constant $s\!:\!\mathbb{S}$ and assert $Th(A)$ and $Acc^A(s, \overline{|Q_A|})$.
2) Generate a model $M$ for the assertions.
3) Validate that $s^M$ indeed matches $r$ using the built-in .NET regex class.

The member generation time (in milliseconds) is shown in column $t$. The time contains the regex parsing time, the automaton construction time, the axiom construction time, and the model generation time with Z3. For all $\epsilon$SFAs and SFAs the construction time is negligible (a few milliseconds or less than a millisecond). In case of DSFAs and minimal DSFAs, the time spent in model generation in Z3 is in many cases only a small fraction of $t$. Thus, minimization could pay off if the automaton is created once, but used several times. (We have rounded the measurements and use 0 if the entries are less than a millisecond.)

In most cases $t$ is marginally better for SFAs, with case #7 being an exception, where the $\epsilon$SFA is 3 times *larger* than the SFA but $t$ is twice *smaller*. One possible explanation is that after epsilon elimination the conditions on the moves in the SFA, although fewer, are more complex.

In general, using a (minimal) DSFA eliminates choices during backtracking and could be preferable in a context where it is combined with other constraints. A general heuristic could be to try to determinize (and minimize) using a time limit and fall back to using the original SFA upon timeout. Some regexes are inherently hard to determinize, since the resulting (minimal) DSFA may be exponentially larger than the $\epsilon$SFA. The classical example is [a-c]*a[a-c]{n} where $n$ is a fixed positive number; the number of states in the $\epsilon$SFA is in this case $n+3$, e.g., the $\epsilon$SFA for [a-c]*a[a-c]{2} is

[5] The experiments were run on a Lenovo T61 laptop with Intel dual core T7500 2.2GHz processor.

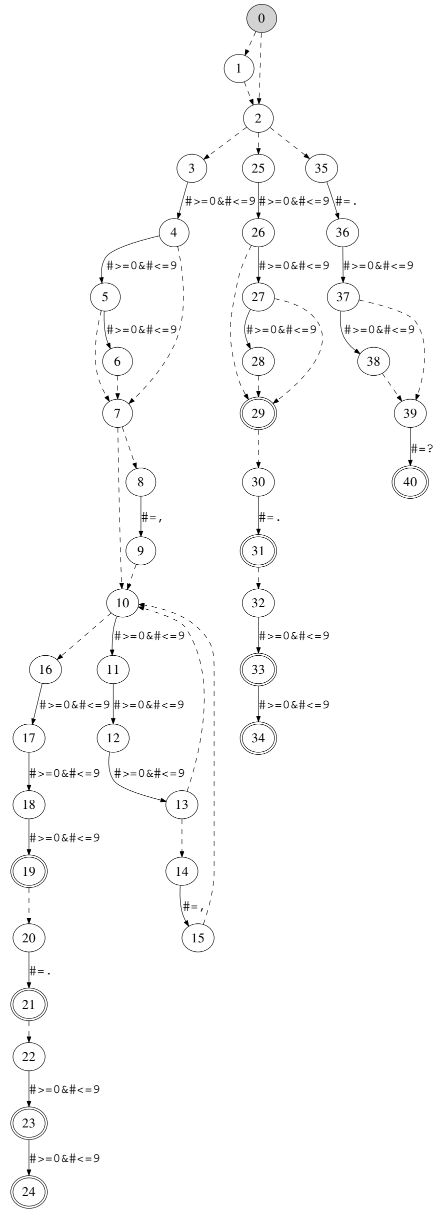
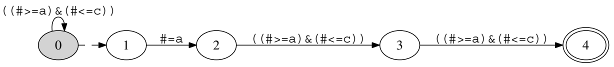
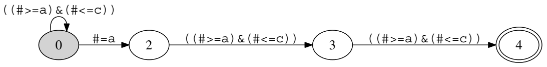


Figure 2. $\epsilon$SFA of regex #2 in Table I.



and the corresponding SFA is



but the number of states in the (minimal) DSFA is $2^{n+1}$. We conducted the above member generation experiment for this regex and $n = 1, \ldots, 11$. In all cases, using the $\epsilon$SFA or SFA, time was negligible (a few ms); whereas, by making the DSFA, time increased exponentially, see Figure 3.

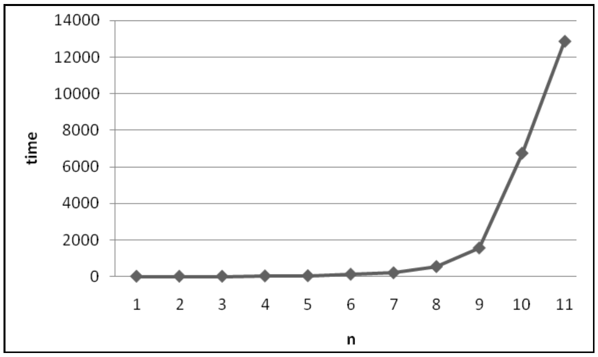


Figure 3. Determinization and member generation times (ms) with Rex for `[a-c]*a[a-c]{n}`, where $n = 1, \ldots, 11$.

The final experiment in this section shows scalability and the use of the product construction. We ran the following member generation check for $n$ up to 1000.

1) Construct $A$ as the product of the SFAs for `[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}`.
2) Declare a fresh $s{:}\mathbb{S}$ and assert $Th(A)$ and $Acc^A(s, \overline{|Q_A|})$.
3) Generate a model $M$ for the assertions (also validate that $s^M$ indeed matches both regexes).

The result of the experiment is shown as a chart in Figure 4. The trendline is also shown, that is polynomial in $n$. One can also assert the acceptors for the two regexes as a conjunction (without building the product), also in this case the time complexity scales reasonably well, e.g., for $n = 50$ it takes around 10 seconds, but which is still in the order of 100 slower than using the product construction. The main reason for this is that, during the product construction, *unsatisfiable* moves are eliminated, and *dead states* are eliminated, as explained in Section III-B4. This means that the corresponding axioms are never created and the search space that Z3 needs to cover during model generation is

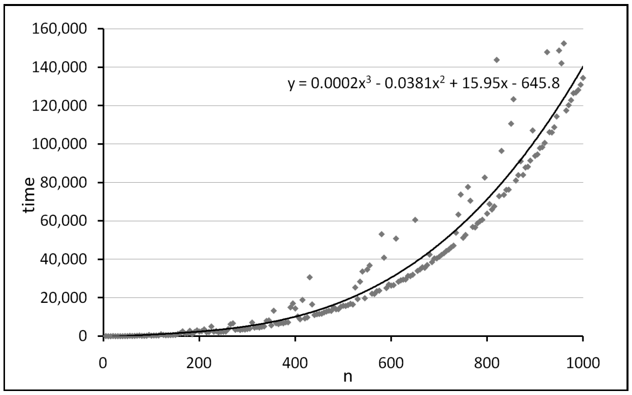


Figure 4. Member generation times (ms) for the intersection of the regexes `[a-c]*a[a-c]{n+1}` and `[a-c]*b[a-c]{n}` for $n$ up to 1000.

dramatically reduced. The satisfiability checks during the product construction are also done with Z3, but these checks are typically very fast because they are "local" to the moves.

## VII. Related work

It was suggested earlier [14] to annotate transitions of an FA with predicates. Van Noord et.al. [15] later formalized the basic idea, and it was implemented [15] as an extension to a Prolog-based automata library [16]. However, they [15] merely suggest in a footnote that "an implementation might choose to ignore transitions for which the corresponding predicate is not satisfiable". We formally introduce the notion of "clean" SFAs, which we construct and maintain by systematically pruning infeasible transitions in our determinization, minimization and product construction algorithms using an SMT solver to prove unsatisfiability. This is the key to efficiency, which we evaluated as well.

A connection between logic and automata has been discovered already fifty years ago [17], [18], and revived about a decade ago [19] in the context of symbolic reasoning with Binary Decision Diagrams (BDDs) [20]. With BDDs, rather dense automata over large alphabets can be represented compactly and reasoned about efficiently. However, with BDDs all characters must be encoded as strings over Boolean variables, while our approach allows transition predicates over variables that belong to any theory support by the underlying (SMT) solver.

Several program analysis techniques for programs with strings [21], [22], [23], [24] build on automata libraries [19], [25] that efficiently handle transitions over sets of characters as BDDs and interval constraints. Most of those program analysis approaches suffer from the separation of the decision procedures, as constraints over strings are decided by one solver, while constraints over other domains are decided by other solvers, and the specialized solver usually cannot be combined in a sound or complete fashion. Our approach avoids this problem by building on top of an SMT solver which has decision procedures for a variety of theories. We discussed symbolic analysis of SQL queries with an SMT solver in earlier work [3]. Another instance is the analysis .NET programs, which use a rich set of string operations; we developed a framework to reason about such string operations using an SMT solver by separating index and length constraints from character constraints [2], and in earlier work we discussed how regular expression matching could be handled by a general-purpose program analysis framework for .NET [1] after translating the queries to a .NET program [6].

Hooimeijer et.al. give a decision procedure for subset constraints over regular language variables [26]. They do so in a self-contained way, reasoning over dependency graphs. In contrast, we showed how finite automata can be generalized by making transitions symbolic, and how a decision procedure can be embedded into a logic of an SMT solver.

HAMPI [27] is another string solver, closely related to ours, which turns string constraints over fixed-size string variables into a query to STP [28], a solver for bit-vectors and arrays. As their solver neither supports lazily instantiated quantifiers nor the theory of data types (or variable-length lists), they can only handle fixed-size inputs. They use "templates" that resemble our quantifiers, but their templates get instantiated eagerly upfront. Also, they do not formally generalize finite automata to symbolic finite automata.

Yu et.al. [29] describe the derivation of unary length automata from finite automata, in order to analyze the relationship among string and integer variables in programs. Their unary length automata are related to our unary encoding of string lengths. They verify program properties by an over-approximating fixpoint computation, while we are concerned with exact decidability.

## VIII. Conclusion

The scalability of the approach taken in Rex was highly surprising to us. Rex is able to handle large regexes, for which the automata often have hundreds or even thousands of states, often under a second. It shows also how effective the underlying SMT solver Z3 is in handling large collections of axioms that are created on-the-fly, since the number of axioms is proportional to the size of the automaton. Moreover, the initial experiments show that it does not seem to pay off to determinize (and minimize) SFAs for this application, since the performance is very unpredictable then. The integration of Rex in Pex and Qex looks really promising. Moreover, there are still a lot of opportunities for further improving the performance, we mention some below.

## IX. Future work

There are several possible optimizations on the set of axioms $Th(A)$ for a given $\epsilon$SFA $A$, that have not been tried out or implemented yet. One such optimization is to use individual characters in patterns of axioms when the condition on the corresponding move is the matching of a single character.

In the context of integrating Rex in Pex one technical challenge is the conversion between different representations of strings. In Pex strings are currently represented by arrays of bitvectors rather than lists. We can define axioms that provide such conversions (the direction from lists to arrays is easy), in order to combine regex constraints with other constraints on strings that arise in path conditions.

A more thorough evaluation of Rex is also needed in order to better understand the role of the list sorts and the lazy instantiation of the axioms, in particular, comparing the performance with related approaches like HAMPI [27]. Moreover, evaluation of Rex integrated into Pex will enable its evaluation when regex constraints arise in program verification and testing in combination with other constraints (such as string length constraints). In this context, it is also interesting to compare the approach to existing techniques based on constraint reasoning using regular sets that can also be combined with other string constraints [30].

The current implementation of Rex uses an incomplete regex parser. In order to expand the technique presented in the paper to a more expressive class of regexes, the presented axiomatization needs to be extended in a nontrivial way. Modern regular expression languages have features such as laziness and as-few-times-as-possible quantifiers that go beyond classical regular languages. It is possible to define symbolic language acceptors for more complex languages (for example for CFGs), by using additional parameters. However, the price to pay is that the prerequisites for the correctness of the axiomatizaton will be more involved, and the variations of the classical algorithms that can be used for SFAs and enable the use of automata theoretic methods will be unclear.

On the theoretical side, we have not formally studied the computational complexity of the SFA algorithms, i.e., how much harder are they than the classical versions? Note that it is not reasonable to view the size of the alphabet as a constant in this case, that would for example be $2^{32}$ for UTF-32 character encoding.

Furthermore, none of the algorithms really depends on the actual sort of the characters, which could be unbounded, such as integers or reals. We have not investigated how changing the character sort from bitvectors to some other sort affects the performance of Rex.

*Acknowledgment*: We thank the anonymous reviewers for very useful and constructive comments.

## References

[1] N. Tillmann and J. de Halleux, "Pex - white box test generation for .NET," in *Proc. of Tests and Proofs (TAP'08)*, ser. LNCS, vol. 4966. Prato, Italy: Springer, April 2008, pp. 134–153.

[2] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, ser. LNCS, vol. 5505. Springer, 2009, pp. 307–321.

[3] M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann, "Symbolic query exploration," in *ICFEM'09*, ser. LNCS, K. Breitman and A. Cavalcanti, Eds., vol. 5885. Springer, 2009, pp. 49–68.

[4] Z3, http://research.microsoft.com/projects/z3.

[5] MSDN, ".NET Framework Regular Expressions," 2009, http://msdn.microsoft.com/en-us/library/hs600312.aspx.

[6] N. Li, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Reggae: Automated test generation for programs using complex regular expressions," in *ASE'09*. IEEE, 2009.

[7] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[8] W. Hodges, *Model theory*. Cambridge Univ. Press, 1995.

[9] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08)*, ser. LNCS. Springer, 2008.

[10] M. Veanes, P. de Halleux, and N. Tillmann, "Rex: Symbolic regular expression explorer," Microsoft Research, Tech. Rep. MSR-TR-2009-137, October 2009.

[11] S. Ranise and C. Tinelli, "The SMT-LIB Standard: Version 1.2," Department of Computer Science, The University of Iowa, Tech. Rep., 2006, available at www.SMT-LIB.org.

[12] Pex, http://research.microsoft.com/projects/pex.

[13] Qex, http://research.microsoft.com/projects/qex.

[14] B. W. Watson, "Implementing and using finite automata toolkits," pp. 19–36, 1999.

[15] G. V. Noord and D. Gerdemann, "Finite state transducers with predicates and identities," *Grammars*, vol. 4, p. 2001, 2001.

[16] G. V. Noord and R. Groningen, "Fsa utilities: A toolbox to manipulate finite-state automata," in *Automata Implementation*. Springer Verlag, 1997, pp. 87–108.

[17] J. Buchi, "Weak second-order arithmetic and finite automata," *Zeit. Math. Logik und Grundl. Math.*, vol. 6, pp. 66–92, 1960.

[18] C. Elgot, "Decision problems of finite automata design and related arithmetics," *Trans. Amer. Math. Soc.*, vol. 98, pp. 21–52, 1961.

[19] N. Klarlund, "Mona & fido: The logic-automaton connection in practice," in *CSL*, 1997, pp. 311–326.

[20] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*. New York, NY, USA: ACM Press, 1990, pp. 40–45.

[21] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra, "Symbolic string verification: An automata-based approach," in *SPIN*, 2008, pp. 306–324.

[22] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *SAS*, 2003, pp. 1–18.

[23] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid, "Abstracting symbolic execution with string analysis," in *TAICPART-MUTATION '07*. IEEE, 2007, pp. 13–22.

[24] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, p. 14, 2007.

[25] "Brics finite state automata utilities," www.brics.dk/automaton/.

[26] P. Hooimeijer and W. Weimer, "A decision procedure for subset constraints over regular languages," in *PLDI*, 2009, pp. 188–198.

[27] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *ISSTA '09*. New York, NY, USA: ACM, 2009, pp. 105–116.

[28] V. Ganesh and D. L. Dill, "A decision procedure for bitvectors and arrays," in *CAV*, 2007, pp. 519–531.

[29] F. Yu, T. Bultan, and O. H. Ibarra, "Symbolic string verification: Combining string analysis and size analysis," in *TACAS*, 2009, pp. 322–336.

[30] K. Golden and W. Pang, "Constraint reasoning over strings," in *CP 2003*, ser. LNCS, F. Rossi, Ed., vol. 2833. Springer, 2003, pp. 377–391.