Solving Extended Regular Constraints Symbolically

Microsoft Research Technical Report MSR-TR-2009-177

Margus Veanes
Nikolaj Bjørner
Leonardo de Moura
Microsoft Research
Redmond, WA, USA
{*margus,nbjorner,leonardo*}*@microsoft.com*

***Abstract*—Constraints over regular expressions are common in programming languages, often in combination with other constraints involving strings. Efficient solving of such constraints has many useful applications in program analysis and testing. We introduce a method for symbolically expressing and analyzing regular constraints using state of the art SMT solving techniques. The method is implemented using the SMT solver Z3 and is evaluated over a collection of benchmarks.**

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

I. Introduction

Regular expressions are used in different applications to express validity constraints over strings. In our case, the original motivation for supporting regular expression constraints comes from two particular applications: program analysis [1], [2], and database query analysis [3]. In the latter case, *like-patterns* are special kinds of regular expressions that are common in SQL select-statements, consider for example the query:

$$\begin{array}{l}\texttt{SELECT} * \texttt{FROM}\ T \\ \texttt{WHERE}\ C\ \texttt{LIKE}\ r_1\ \texttt{AND NOT}\ C\ \texttt{LIKE}\ r_2\ \texttt{AND LEN}(C) < D + E\end{array} \quad (1)$$

that selects all rows from a table T having columns C, D and E, where the C-value matches the like-pattern r_1, does not match the like-pattern r_2 and whose length is less than the sum of D-value and E-value. The analysis discussed in [3] aims at generating tables that satisfy a test condition, e.g., that the result of (1) is nonempty. A core part of that analysis is to find solutions to select-conditions of the above form.

We introduce a technique that allows conditions such as the select condition of (1) to be expressed and analyzed using satisfiability modulo theories (SMT) solving in a way that is extensible with other constraints and theories. The central idea behind the technique is the notion of a *symbolic (language) acceptor* for a language (set of strings) L, as a binary predicate $Acc^L(w,k)$ that is true modulo a theory $Th(L)$ iff $w \in L$ and k is the length $|w|$ of w. For a regular expression r the symbolic acceptor for $L(r)$ is constructed from a symbolic finite automaton A_r that accepts $L(r)$; the symbolic acceptor is denoted by Acc^{A_r} and the theory is denoted by $Th(A_r)$. The automaton A_r is itself symbolic in the sense that its moves are labeled by formulas rather than individual characters, which provides a more succinct and convenient way to represent automata for like-patterns or more generally for *regexes* [4] involving complex character ranges.

In particular, solving the select condition in (1) corresponds to solving the formula,

$$Acc^{A_{r_1}}(c,k) \wedge \neg Acc^{A_{r_2}}(c,k) \wedge k < d + e \quad (2)$$

modulo the theories $Th(A_{r_1})$, $Th(A_{r_2})$ and linear arithmetic. A *solution* of (2) is a mapping of particular values for c, k, d, and e which makes (2) true (modulo the given theories).

In applications such as [1], [2], [3], that build on the SMT technology, a fundamental aspect is that new theories that become available for expanding the scope of the analysis, can be added seamlessly and work in combination with existing theories. In other words, *extensibility* is a must.

The construction of symbolic language acceptors uses the generic theory of *algebraic data types*, in particular *lists*. Algebraic data types are supported by the SMT solver Z3 [5], [6] that we use as the underlying SMT solver in our implementation. Strings are represented by lists of *characters*. Characters are represented by n-bit-vectors of a fixed $n \geq 1$, provided that the size of the vocabulary is 2^n. For example $n = 16$ for representing Unicode characters. The representation of strings as lists is convenient for the purpose of our encoding of language acceptors.

The construction of $Th(A)$ builds on automata theory that offers a choice between various logically equivalent forms of axiomatization and composition techniques for performance considerations. For example, an encoding for (1) that is equivalent to the direct encoding (2) has the form

$$Acc^{A_{r_1} \times \overline{A_{r_2}}}(c,k) \wedge k < d + e \quad (3)$$

where $\overline{A}$ denotes the complement of A and $A \times B$ denotes the product of A and B. Since complementation may cause exponential blowup in the size of an automaton, it may be useful to use an encoding that combines product with complementation as *difference*:

$$Acc^{A_{r_1} \setminus A_{r_2}}(c,k) \wedge k < d + e \quad (4)$$

Note that if $L(r_1) \subseteq L(r_2)$ in (1), i.e., $L(r_1) \setminus L(r_2) = \emptyset$ then the query (1) is *infeasible*. Since our use of the SMT solver is optimized for satisfiability checking rather than unsatisfiability checking, it can beneficial to use (4) instead of (2). Independently, difference checking provides a way to check *subset* constraints, that has other useful applications.

Although the current application context of the paper and the experiments we show in the paper focus on regular languages, all the definitions for symbolic language acceptors and algorithms are

given for *symbolic push down automata* as a generalization of symbolic finite automata. We prove a theorem about the completeness and the soundness of the axiomatization.

Combination of regular constraints on strings with quantifier free linear arithmetic and length constraints is known to be decidable [7], [8]. One can effectively compute an upper bound on the length of all strings. We recall how this is done in Appendix A. By using these bounds to restrict the maximum length of strings in solutions of acceptor formulas, one obtains a *complete* decision procedure for solving linear arithmetic with regular constraints and length constraints with the approach described in this paper. For context free languages the approach gives a complete semi-decision procedure.

We describe a specialized algorithm for constructing the difference $A \setminus B$ between a symbolic PDA A and a symbolic FA A. This algorithm is of interest independently from the main application context; one use of the algorithm is for checking subset constraints of the form $L_1 \subseteq L_2$ where L_1 is context free and L_2 is regular.

We evaluate the performance of the different approaches that have been implemented in a prototype tool by the authors.

The rest of the paper is structured as follows. Section II introduces some background material to keep the paper self-contained and to explain the notational conventions. Section III defines symbolic automata. Section IV introduces symbolic language acceptors. Section V describes an algorithm for difference construction. Section VI discusses some aspects of the implementation and Section VII provides some benchmarks. Finally, Section VIII describes related work, and Section IX states some concluding remarks.

II. Preliminaries

We assume that the reader is familiar with classical automata theory, we follow [9] in this regard. We also assume elementary knowledge about logic and model theory, our terminology is consistent with [10] in this regard.

We are working in a fixed multi-sorted universe $\mathcal{U}$ of values. For each sort σ, $\mathcal{U}^\sigma$ is a separate sub-universe of $\mathcal{U}$. The basic sorts needed in this paper are the Boolean sort $\mathbb{B}$, $\mathcal{U}^{\mathbb{B}} = \{true, false\}$, and the sort of *$n$-bit-vectors*, for a given number $n \geq 1$; an n-bit-vector 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-bit-vectors 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 full range of Unicode characters. 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 `'a'` to denote a character.

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 $\sigma_0 \times \cdots \times \sigma_{n-1}$, when $n > 0$, and a given range sort σ, $f : \sigma_0 \times \cdots \times \sigma_{n-1} \to \sigma$. 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. A *constant* is a nullary function symbol.

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 quantifier-free formulas with at most one fixed free variable of sort $\mathbb{C}$. *Throughout the paper, we denote that variable by χ.* Given $\varphi \in \mathcal{F}_{\mathbb{C}}$, and a character or term t of sort $\mathbb{C}$, we write $\varphi[t]$ for the formula where each occurrence of χ is replaced by t.

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 for a formula* φ provides an interpretation for all the uninterpreted symbols in φ. A model M for a closed formula φ *satisfies* φ, $M \models \varphi$, if the interpretations provided by M make φ true. A closed formula φ is *satisfiable* if it has a model. A formula φ with $FV(\varphi) = \bar{x}$ is *satisfiable* if its existential closure $\exists \bar{x} \varphi$ is satisfiable. We write $\models \varphi$, if φ is *valid* (true in all models for φ).

For example, the character range set `[a-z\d]` in a regex is translated into the formula $\psi = (\texttt{'a'} \leq \chi \wedge \chi \leq \texttt{'z'}) \vee (\texttt{'0'} \leq \chi \wedge \chi \leq \texttt{'9'})$ with χ as the single free variable in ψ. The formula ψ is satisfiable; $\psi[\texttt{'b'}]$ is true; $\psi[\texttt{'a'}]$ is false. Note that `'a'`, `'z'`, `'0'` and `'9'` in ψ stand for terms that use only built-in function symbols and denote the bit-vector encodings of the corresponding characters and digits.

III. Symbolic 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, φ, 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 Δ of moves, let

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

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

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

For example, the move

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

represents the set of all transitions (p, c, q) where c is a character between `'a'` and `'z'`. The symbolic representation of conditions is convenient and fits well with the encoding of the symbolic acceptors for automata discussed below. Formally, we refer to such a representation of a finite automata (FA) as follows.

Definition 1: A *Finite Symbolic Automaton* or *SFA* A is a tuple (Q, q_0, F, Δ), 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. We also use the following notations.

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

Just as with finite automata, it is useful to add *epsilon moves* to an SFA. Consider a special symbol ϵ that is not in the background universe.

Definition 2: An *SFA with epsilon moves* or *ϵSFA* is a tuple (Q, q_0, F, Δ), 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 ϵSFA.) Let $[\![(p, \epsilon, q)]\!] \stackrel{\text{def}}{=} (p, \epsilon, q)$. An ϵSFA $A = (Q, \Delta, q_0, F)$ denotes the finite automaton $[\![A]\!]$ with epsilon moves [9], where

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

We write Δ_A^{ϵ} for the set of all epsilon moves in Δ_A and $\Delta_A^{\not\epsilon}$ for $\Delta_A \setminus \Delta_A^{\epsilon}$. Epsilon elimination for ϵSFAs is a straightforward extension of epsilon elimination of finite automata and has linear time complexity in the size of the automata, see [11]. If the input ϵSFA is clean then so is the resulting SFA, since the only combination of move conditions performed during epsilon elimination is disjunction.

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

It is clear that for any ϵSFA A there is a normalized SFA A' such that $[\![A]\!] = [\![A']\!]$: for all states p and q in Q_A, make a disjunction φ of all the conditions of the moves from p to q in $\Delta_A^{\not\epsilon}$ and let (p, φ, 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 ϵSFA 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, φ_1, q_1) and (p, φ_2, q_2) in Δ_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.

We use [4] as the concrete language definition of regular expression patterns or *regexes* in this paper. The translation from a regex to an ϵSFA follows very closely the standard algorithm, see e.g., [9, Section 2.5], for converting a standard regular expression into a finite automaton with epsilon moves. A sample regex and corresponding ϵSFA are illustrated in Figure 1.

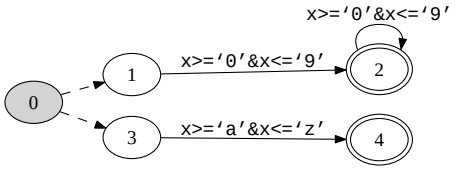

Figure 1. Sample ϵSFA generated from the regex `\d+|[a-z]`. The initial state is gray, the epsilon moves are dashed. The symbol `&` is used for conjunction.

A. Symbolic Push Down Automata

Similar to SFAs we use a representation of PDAs where moves are labeled by formulas from $\mathcal{F}_{\mathbb{C}}$ that represent sets of characters rather than individual characters.

Definition 7: A *Symbolic Push Down Automaton* or *SPDA* A is a tuple $(Q, Z, q_0, z_0, F, \Delta)$, where Q is a finite set of *states*, Z is a finite set of *stack symbols*, $q_0 \in Q$ is the *initial state*, $z_0 \in Z$ is the *initial stack symbol*, $F \subseteq Q$ is the set of *final states* and $\Delta : Q \times Z \times \mathcal{F}_{\mathbb{C}} \times Q \times Z^*$ is the *move relation*.

Similar to ϵSFAs an ϵSPDA may have moves where the condition is ϵ. Each move $t = (q, z, \alpha, p, \vec{z})$ of an ϵSPDA, where $\alpha \in \mathcal{F}_{\mathbb{C}} \cup \{\epsilon\}$, denotes the set of transitions $[\![t]\!] \stackrel{\text{def}}{=} \{(q, z, a, p, \vec{z}) \mid a \in [\![\alpha]\!]\}$. The underlying PDA is denoted by $[\![A]\!]$ (with the alphabet $\mathcal{U}^{\mathbb{C}}$). The language $L(A)$ accepted by A is the language $L([\![A]\!])$ accepted by the PDA $[\![A]\!]$. We reuse the definitions of $Source(t)$, $Target(t)$, and $Cond(t)$ also for moves t of an ϵSPDAs and introduce the additional definitions

$$\begin{aligned} Pop((q, z, \alpha, p, \vec{z})) &\stackrel{\text{def}}{=} z, \\ Push((q, z, \alpha, p, \vec{z})) &\stackrel{\text{def}}{=} \vec{z}. \end{aligned}$$

An ϵSPDA A is *clean* if all moves in Δ_A have satisfiable conditions, and *normalized* if there are no two moves t_1 and t_2 in Δ_A that differ only with respect to $Cond(t)$.

Elimination of epsilon moves from an ϵSPDA corresponds to transforming the corresponding context free grammar into Greibach Normal Form (GNF), which can be done in polynomial time. Move conditions play no active role in the algorithm. (Terminals are in general treated as black boxes in normal form transformations of grammars.) We use a variation of the Blum-Koch algorithm [12] for GNF transformation that has worst case time complexity n^4.

Note that every ϵSFA A can be trivially transformed into an ϵSPDA: let $Z_A = \{z\}$, $z_{0A} = z$, and for each move $t \in \Delta_A$ let $Pop(t) = z$ and $Push(t) = z$. We say that the resulting ϵSPDA *represents* an ϵSFA.

1) Product with SFA: The classical construction of the product of a PDA with an FA has a corresponding version for a product of an SPDA with an SFA. The key modification is the use of satisfiability checking of conditions in order to keep the construction clean. The input to the algorithm is an SPDA A and an SFA B and the output is an SPDA C that is the *product* of A and B, also denoted by $A \times B$, such that $L(C) = L(A) \cap L(B)$. It is convenient to describe the algorithm as a depth-first-exploration algorithm using a stack S as a frontier, a set $V \subseteq (Q_A \times Q_B) \times Z_A$ of already visited elements, and a set T of moves. Initially, let $T = \emptyset$, $S = (\langle\langle q_{0A}, q_{0B}\rangle, z_{0A}\rangle)$, and $V = \{\langle\langle q_{0A}, q_{0B}\rangle, z_{0A}\rangle\}$.

(i) If S is empty go to (iii) else pop $\langle\langle p, q\rangle, z\rangle$ from S.

(ii) For each $(p, z, \varphi, p', \vec{z}) \in \Delta_A$ and $(q, \psi, q') \in \Delta_B$. *If* $\varphi \wedge \psi$ *is satisfiable* then
- add $(\langle p, q\rangle, z, \varphi \wedge \psi, \langle p', q'\rangle, \vec{z})$ to T;
- for each z' in $\vec{z}$, if $v = \langle\langle p', q'\rangle, z'\rangle \notin V$ then add v to V, and if there is $t \in \Delta_A$ with $Source(t) = p'$ and $Pop(t) = z'$ then push v to S.

Go to (i).

(iii) Let $C = (\pi_1(V), \pi_2(V), \langle q_{0A}, q_{0B}\rangle, z_{0A}, \pi_1(V) \cap (F_A \times F_B), T)$.

(iv) 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 (ii) is important. It prevents unnecessary exploration of *unreachable* states, and may avoid a quadratic blowup of Q_C, whereas (iv) avoids introduction of useless "dead end"-axioms in the symbolic language acceptor for C.

Note also that if A represents an SFA then so does C.

IV. Symbolic language acceptors

To encode language acceptors, we use particular kinds of axioms, all of which are equations of the form

$$\forall \bar{x}(t_{\rm lhs} = t_{\rm rhs}) \tag{5}$$

where $FV(t_{\rm lhs}) = \bar{x}$ and $FV(t_{\rm rhs}) \subseteq \bar{x}$. When $t_{\rm lhs}$ and $t_{\rm rhs}$ are formulas, we often write '$\Leftrightarrow$' instead of '$=$'. The left-hand-side $t_{\rm lhs}$ of (5) is called the *pattern* of (5). *While SMT solvers support various kinds of patterns in general, in this paper we use the convention that the pattern of an equational axiom is always the left-hand-side of the equation.*

Axioms are asserted to the SMT solver as macros that are expanded during proof search. It is not easy to expand the initial goal formula outside the solver and to assert only quantifier free formulas to the solver. The reason is that it is hard to know when to expand a pattern and when not to expand a pattern. Moreover, axioms that are introduced for automata are typically mutually recursive and a naive a priori exhaustive expansion of patterns would in most cases not terminate.

The overall idea behind the axioms is as follows. For a given ϵSPDA A we construct a theory $Th(A)$ that includes a particular axiom with lhs $Acc^A(w, k)$. The main property of $Th(A)$ is that it precisely characterizes the language accepted by A:

$$L(A) = \{w^M \mid M \models Th(A) \wedge Acc^A(w, k)\}$$

and if $M \models Acc^A(w, k)$ then $k^M = |w^M|$, where w is a list of characters and $|w|$ denotes the length of w, as explained below.

A. Lists

Lists are built-in algebraic data-types and are accompanied with standard constructors and accessors. For each sort σ, $\mathbb{L}\langle\sigma\rangle$ is the *list sort* with element sort σ. For a given element sort σ there is an empty list nil (of sort $\mathbb{L}\langle\sigma\rangle$) and if e is an element of sort σ 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 $\mathbb{W}$ for the sort $\mathbb{L}\langle\mathbb{C}\rangle$. The empty string is abbreviated by `""` and a string $cons($`'a'`$, cons($`'b'`$, cons($`'c'`$, nil)))$ is abbreviated by `"abc"`, e.g., $hd($`"abc"`$) =$ `'a'` and $tl($`"abc"`$) =$ `"bc"`.

B. Construction of $Th(A)$

Let A be a given ϵSPDA. Assume A is normalized. Let $\mathbb{N}$ be a built-in non-negative numeral sort such as a bit-vector or integer sort restricted to non-negative integers. We use $\mathbb{N}$ for representing the length $|l|$ of a list l, i.e., the number of elements in l. Let $\mathbb{Z}$ be a sort for representing Z_A and assume that $Z_A \subseteq \mathcal{U}^{\mathbb{Z}}$. With slight abuse of notation, we also use stack symbols as terms. We may assume, without loss of generality that $\mathbb{Z}$ is a fixed numeral sort as well. We represent a *stack* as an element of sort $\mathbb{S} = \mathbb{L}\langle\mathbb{Z}\rangle$. A stack is denoted by $(z_1, \ldots, z_k)$, $k \geq 0$, where z_1 is the top element on the stack. We write $(z_1, \ldots, z_k) \cdot (z_{k+1}, \ldots, z_l)$ for $(z_1, \ldots, z_l)$. The empty stack is denoted by ε.

For all $q \in Q_A$, declare the predicate symbol

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

An *ID* of $[\![A]\!]$ is a triple (q, w, s) where $q \in Q_{[\![A]\!]}$, w is a string and s is a stack [9]. For defining the axioms it is more convenient to use *acceptance by the empty stack* rather than final states, the language accepted by the empty stack is denoted by $N([\![A]\!])$ in [9, page 112]. The transformation of A to an equivalent ϵSPDA A' such that $L([\![A]\!]) = N([\![A']\!])$ is straightforward. We therefore assume that $F_A = \emptyset$.

The idea behind the axioms defined below is that the formula $Acc^A_q(w, n, s)$ holds iff $|w| = n$ and $(q, w, s) \vdash^*_{[\![A]\!]} (p, \texttt{""}, \varepsilon)$ for some $p \in Q_{[\![A]\!]}$, where $\vdash_{[\![A]\!]}$ is the step relation of $[\![A]\!]$ as defined in [9, page 112]. Declare also

$$Acc^A : \mathbb{W} \times \mathbb{N} \to \mathbb{B}$$

The intuition is that $Acc^A(w, n)$ holds iff $|w| = n$ and $w \in L(A)$. For $q \in Q_A$ and $z \in Z_A$ let $\Delta_A(q, z)$ be

$$\{(q, z, \varphi_i, q_i, \vec{z}_i) \mid 1 \leq i \leq m\} \cup \{(q, z, \epsilon, q_i, \vec{z}_i) \mid m < i \leq k\}.$$

Define the following axioms.

$$\begin{array}{rcl}
ax^A & \stackrel{\rm def}{=} & \forall w\, n\, (Acc^A(w, n) \Leftrightarrow Acc^A_{q_{0A}}(w, n, cons(z_{0A}, \varepsilon))) \\
ax^A_q & \stackrel{\rm def}{=} & \forall w\, n\, (Acc^A_q(w, n, \varepsilon) \Leftrightarrow (w = \texttt{""} \wedge n = 0)) \\
ax^A_{q,z} & \stackrel{\rm def}{=} & \forall w\, n\, s\, (Acc^A_q(w, n, cons(z, s)) \Leftrightarrow \\
 & & ((w \neq \texttt{""} \wedge n > 0 \wedge \\
 & & (\bigvee_{i=1}^{m} (\varphi_i[hd(w)] \wedge Acc^A_{q_i}(tl(w), n{-}1, \vec{z}_i \cdot s)))) \\
 & & \vee \bigvee_{j=m+1}^{k} Acc^A_{q_j}(w, n, \vec{z}_j \cdot s))) \\
Th(A) & \stackrel{\rm def}{=} & \{ax^A\} \cup \{ax^A_q, ax^A_{q,z} \mid q \in Q_A, z \in Z_A\}.
\end{array}$$

The set of formulas $Th(A)$ (or equivalently $\bigwedge Th(A)$) is asserted to the solver as the *theory of* A.

The following theorem can be generalized to a class of well-behaved ϵSPDAs but does not hold for all ϵSPDAs, i.e., when arbitrary epsilon moves are allowed.

Theorem 1: Let A be an SPDA or an epsilon-loop-free ϵSFA. Let w:$\mathbb{W}$, n:$\mathbb{N}$. For all M, $M \models \bigwedge Th(A) \wedge Acc^A(w, n)$ iff $w^M \in L(A)$ and $|w^M| = n^M$.

Proof outline: For the case when A is an epsilon-loop-free ϵSFA the proof is similar to the proof of [11, Theorem 1]. Assume A is an SPDA. Since epsilon moves are not present, the

following statement follows by induction over the length of $\vdash_{[\![A]\!]}$-computations. For all IDs (q, w, s) of $[\![A]\!]$:

$$\exists p \in Q_A((q, w, s) \vdash^*_{[\![A]\!]} (p, \texttt{""}, \varepsilon)) \iff Th(A) \models Acc^A_q(w, |w|, s)$$

The theorem follows, by letting $q = q_{0A}$ and $s = (z_{0A})$. ■

The theorem fails in general when epsilon moves are allowed as illustrated by the following example.

Example 1: Let $A = (\{q\}, \{z\}, q, z, \emptyset, \{(q, z, \epsilon, q, (z))\})$. For example $(q, \texttt{""}, (z)) \vdash_{[\![A]\!]} (q, \texttt{""}, (z))$. The language accepted by A is empty. The theory $Th(A)$ for A includes the axiom $ax^A_{q,z}$:

$$\forall w\, n\, s\, (Acc^A_q(w, n, cons(z, s)) \Leftrightarrow Acc^A_q(w, n, cons(z, s)))$$

This axiom is a useless tautology. Consider for example a model M with an interpretation for Acc^A_q such that $M \models Acc^A_q(\texttt{""}, 0, (z))$ and expand M so that $M \models ax^A \wedge ax^A_q$. Then $M \models Acc^A(\texttt{""}, 0)$ but $\texttt{""} \notin L(A)$. ⊠

For all ϵSPDAs there is an equivalent loop-free ϵSPDA that can be computed effectively, e.g., the GNF normal form for CFGs implies that.

For an ϵSFA (or an ϵSPDA that represents an ϵSFA) the axioms can be simplified, by omitting the stack variable. We illustrate this with an example that also shows the application of the axioms.

Example 2: Let A be the ϵSFA in Figure 1. The axioms $Th(A)$ for A are as follows, where $\varphi = (\chi \geq \texttt{'a'}) \wedge (\chi \leq \texttt{'z'})$ and $\psi = (\chi \geq \texttt{'0'}) \wedge (\chi \leq \texttt{'9'})$ and $q \in Q_A$:

$$\begin{aligned}
ax_0&: \forall w\, n\, (Acc(w, n) \Leftrightarrow (Acc_1(w, n) \vee Acc_3(w, n)))\\
ax_1&: \forall w\, n\, (Acc_1(w, n) \Leftrightarrow (w \neq \texttt{""} \wedge n > 0 \wedge\\
&\qquad \psi[hd(w)] \wedge Acc_2(tl(w), n-1)))\\
ax_2&: \forall w\, n\, (Acc_2(w, n) \Leftrightarrow ((w \neq \texttt{""} \wedge n > 0 \wedge \psi[hd(w)] \wedge\\
&\qquad Acc_2(tl(w), n-1)) \vee (w = \texttt{""} \wedge n = 0)))\\
ax_3&: \forall w\, n\, (Acc_3(w, n) \Leftrightarrow (w \neq \texttt{""} \wedge n > 0 \wedge \varphi[hd(w)] \wedge\\
&\qquad Acc_4(tl(w), n-1)))\\
ax_4&: \forall w\, n\, (Acc_4(w, n) \Leftrightarrow (w = \texttt{""} \wedge n = 0))
\end{aligned}$$

Declare fresh constants $s : \mathbb{W}$, $k : \mathbb{N}$ and assert the axioms $Th(A)$ and the goal $Acc(s, k)$ to the solver. We describe a plausible scenario for the resulting model generation process. First, the axioms are triggered, we indicate the selected sub-term of the goal by underlying it:

$$\begin{aligned}
\underline{Acc(s,k)} \quad &\overset{ax_0}{\rightsquigarrow} \quad Acc_1(s,k) \vee \underline{Acc_3(s,k)}\\
&\overset{ax_3}{\rightsquigarrow} \quad Acc_1(s,k) \vee (s \neq \texttt{""} \wedge k > 0\\
&\qquad \wedge \varphi[hd(s)] \wedge \underline{Acc_4(tl(s), k-1)})\\
&\overset{ax_4}{\rightsquigarrow} \quad Acc_1(s,k) \vee (s \neq \texttt{""} \wedge k > 0\\
&\qquad \wedge \varphi[hd(s)] \wedge tl(s) = \texttt{""} \wedge k - 1 = 0)
\end{aligned}$$

The triggering process may continue, but the conjunct of the goal that does not include patterns enables a concrete model to be generated using built-in theories. For example, there is a model M such that $s^M = \texttt{"a"}$ and $k^M = 1$. ⊠

C. E-matching

During proof search in an SMT solver, axioms are triggered by matching sub-expressions in the goal. The high-level idea of how an SMT solver uses an axiom such as (5) is as follows. The axiom (5) is *triggered* by the current goal ψ of the solver, if ψ contains a sub-term u and there exists a substitution θ such that $u = t_{\text{lhs}}\theta$, i.e., u *matches the pattern* of the axiom. If (5) is triggered, then the current goal is replaced by the *logically equivalent* formula where u has been replaced by $t_{\text{rhs}}\theta$.

Thus, equational axioms can be viewed as "rewrite rules", and each application of an axiom preserves the logical equivalence to the original goal. Termination is in general not guaranteed in the presence of (mutually) recursive axioms. Note that, unlike in term rewrite systems, there is no notion of term orderings or well-defined customizable strategies (at least not in the current version of Z3) that could be used to guide the triggering process of the axioms.

V. Difference construction

We describe an algorithm that is used below for encoding difference constraints. The input to the algorithm consists of a clean SPDA A and a clean SFA B, and the output of the algorithm is a clean SPDA C that is equivalent to $A \times \overline{B}$, i.e., $L(C) = L(A) \setminus L(B)$. Thus, $L(C) = \emptyset$ iff $L(A) \subseteq L(B)$.

The general idea behind the algorithm is to incrementally determinize and complement B, and simultaneously compose it with A, while keeping the construction clean. During this process the SMT solver is used to generate all solutions to *cube* formulas that represent satisfiable combinations of move conditions for all moves from subsets of states of B that arise during determinization of B. Given a finite sequence of formulas $\vec{\varphi} = (\varphi_i)_{i<n}$ from $\mathcal{F}_{\mathbb{C}}$, and distinct Boolean constants $\vec{b} = (b_i)_{i<n}$ define

$$Cube(\vec{\varphi}, \vec{b}) \overset{\text{def}}{=} \bigwedge_{i<n} \varphi_i \Leftrightarrow b_i.$$

Recall that the variable χ is shared in all the φ_i. A *solution* of $Cube(\vec{\varphi}, \vec{b})$ is a model M such that $M \models \exists \chi\, Cube(\vec{\varphi}, \vec{b})$. In particular, M provides a truth assignment to all the b_i's. Given a set G of formulas we write $\bigvee G$ for the formula $\bigvee_{\varphi \in G} \varphi$, similarly for $\bigwedge G$. The following property follows by using basic model theory.

Proposition 2: If M is a solution of $Cube(\vec{\varphi}, \vec{b})$ then $\bigwedge\{\varphi_i \mid i < n, M \models b_i\}$ is satisfiable.

Given a solution M of $Cube(\vec{\varphi}, \vec{b})$, let φ_M denote the formula

$$\bigwedge(\{b_i \mid M \models b_i\} \cup \{\neg b_i \mid M \models \neg b_i\})$$

We use the following iterative model generation procedure to generate the set $Solutions(Cube(\vec{\varphi}, \vec{b}))$ of all solutions of $Cube(\vec{\varphi}, \vec{b})$.

1) Initially let $\mathbf{M} = \emptyset$.
2) Keep adding solutions of $Cube(\vec{\varphi}, \vec{b})$ to $\mathbf{M}$ until $Cube(\vec{\varphi}, \vec{b}) \wedge \bigwedge_{M \in \mathbf{M}} \neg\varphi_M$ is unsatisfiable.
3) Let $Solutions(Cube(\vec{\varphi}, \vec{b})) = \mathbf{M}$.

The procedure is still exponential in the *worst case*, but seems to work well in practice. It is also better than creating all subsets of $\vec{\varphi}$ and filtering out all combinations that are unsatisfiable, which is *always* exponential.

The following property is used in the difference construction algorithm for generating all satisfiable subsets of move conditions for a given set of moves.

Proposition 3: Let $\vec{\varphi}$ and $\vec{b}$ be as above. For all subsets G of $\vec{\varphi}$, $\bigwedge G$ is satisfiable if and only if there exists $M \in Solutions(Cube(\vec{\varphi}, \vec{b}))$ such that $M \models b_i$ for all $\varphi_i \in G$.

Proof: The direction $\Longleftarrow$ follows from Proposition 2. For the direction $\Longrightarrow$ assume $\bigwedge G$ is satisfiable and let M be such that

$M \models b_i$ for all φ_i such that φ_i is a logical consequence of G. Let $\mathbf{M} = Solutions(Cube(\vec{\varphi}, \vec{b}))$. Then $M \in \mathbf{M}$ or else $Cube(\vec{\varphi}, \vec{b}) \wedge \bigwedge_{N \in \mathbf{M}} \neg\varphi_N$ would be satisfiable, contradicting the construction of $\mathbf{M}$. ∎

Definition 8: An SFA A is *total* if for all $q \in Q_A$, the formula $\forall\chi \bigvee\{Cond(t) \mid t \in \Delta_A(q)\}$ is valid.

In order to make an SFA that is not total into an equivalent total SFA, one can add a new *dead state* d to it with the move $(d, true, d)$, and a new move (q, φ, d) from each state q where φ is satisfiable and $\varphi = \bigwedge\{\neg Cond(t) \mid t \in \Delta(q)\}$. Clearly, determinism is preserved by this transformation.

Definition 9: Given a total DSFA A, the *complement* $\overline{A}$ of A is the DSFA $(Q_A, q_{0A}, Q_A \setminus F_A, \Delta_A)$.

It is easy to see that for a total DSFA A, $\overline{L(A)} = L(\overline{A})$.

We use the following property of regular languages to speed up the difference construction in some cases, with a low initial overhead. For regular languages it is a well-known fact that reversing the language preserves regularity.

Definition 10: Given an ϵSFA A with nonempty $L(A)$ and a state $q \notin Q_A$, the *reverse* A^r of A with initial state q is the ϵSFA

$$\begin{array}{l}(Q_A \cup \{q\}, q, \{q_{0A}\},\\ \{(Target(t), Cond(t), Source(t)) \mid t \in \Delta_A\}\\ \cup\{(q, \epsilon, p) \mid p \in F_A\})\end{array}$$

Given a string s let s^r denote the string that is s in reverse and let L^r denote the language $\{s^r \mid s \in L\}$. (Note that $L = (L^r)^r$.) It follows that $L(A^r) = L(A)^r$. We make use of the following property.

Proposition 4: Let A be an ϵSFA, $\overline{L(A)} = L(\overline{A^r})^r$.

The point of reversing is that complementation of an SFA A requires determinization that may cause exponential blowup in the size of the automaton, which can be avoided if A^r is deterministic. A classical example is the SFA A for the regex `[ab]*a[ab]{n}` where n is a positive integer. A has $n + 2$ states and the size of the minimum DSFA for this regex has 2^{n+1} states, whereas A^r is deterministic.

We are now ready to describe the algorithm. Let A be an SPDA and B an SFA. Assume that A is clean and B is normalized, clean, and total.

Check the special cases first:

- If B is deterministic let $C = A \times \overline{B}$.
- Else, if A represents an SFA and B^r is deterministic let $C = (A^r \times \overline{B^r})^r$.

General case. We describe the algorithm as a depth-first-exploration algorithm using a stack S as a frontier, a set $V : (Q_A \times 2^{Q_B}) \times Z_A$ of visited elements, and a set T of moves. Initially, let $q_{0C} = \langle q_{0A}, \{q_{0B}\}\rangle$, $S = (\langle q_{0C}, z_{0A}\rangle)$, $V = \{\langle q_{0C}, z_{0A}\rangle\}$, and $T = \emptyset$.

(i) If S is empty go to (iv) else pop $\langle\langle p, \mathbf{q}\rangle, z\rangle$ from S.

(ii) Let $\Delta_A(p, z) = (p, z, \varphi_i, p_i, \vec{z_i})_{i<m}$, $\Delta_B(\mathbf{q}) = (\_, \psi_i, q_i)_{i<n}$. Let $\vec{a} = (a_i)_{i<m}$ and $\vec{b} = (b_i)_{i<n}$ be fresh Boolean constants. Compute

$$\mathbf{M} = Solutions(Cube((\varphi_i)_{i<m} \cdot (\psi_i)_{i<n}, \vec{a} \cdot \vec{b}))$$

with the additional constraint that $\bigvee \vec{a}$ is true. For each move $(p, z, \varphi_i, p_i, \vec{z_i})$ of A and for each solution M in $\mathbf{M}$ such that $M \models a_i$ do the following. Let

$$\begin{array}{l}\gamma = \varphi_i \wedge \bigwedge(\{\psi_j \mid M \models b_j\} \cup \{\neg\psi_j \mid M \models \neg b_j\}),\\ \mathbf{q}' = \{q_j \mid M \models b_j\}.\end{array}$$

Add the move $(\langle p, \mathbf{q}\rangle, z, \gamma, \langle p_i, \mathbf{q}'\rangle, \vec{z_i})$ to T. Foreach $z' \in \vec{z_i}$, if $v = \langle\langle p_i, \mathbf{q}'\rangle, z'\rangle \notin V$ then add v to V and if there exists $t \in \Delta_A$ such that $Source(t) = p'$ and $Pop(t) = z'$ then push v to S.

(iii) Go to (i).

(iv) Compute the set of final states F:

$$F = \{\langle p, \mathbf{q}\rangle \mid \langle p, \mathbf{q}\rangle \in \pi_1(V), p \in F_A, \mathbf{q} \cap F_B = \emptyset\}.$$

If $F = \emptyset$ let $C = (\{q_{0C}\}, \{z_{0A}\}, q_{0C}, z_{0A}, \emptyset, \emptyset)$, else let $C = (\pi_1(V), \pi_2(V), q_{0C}, z_{0A}, F, T)$.

The complementation of B in the algorithm is reflected in the computation of F where a state of C is final if its first component is a final A-state and its second component, that is a set of B-states, *includes no final B state*.

The totality of B is assumed in the computation of $\mathbf{M}$, where each solution will make at least one a_i and at least one b_j true. The totality assumption can be avoided by representing a "dead state" implicitly in the algorithm. The presentation of the algorithm gets technically more involved in this case.

To see that B is indeed incrementally determinized, consider any two moves

$$\begin{array}{rcl}t_1 & = & (\mathbf{q}, \bigwedge_{M_1 \models b_j} \psi_j \wedge \bigwedge_{M_1 \models \neg b_j} \neg\psi_j, \{q_j \mid M_1 \models b_j\})\\ t_2 & = & (\mathbf{q}, \bigwedge_{M_2 \models b_j} \psi_j \wedge \bigwedge_{M_2 \models \neg b_j} \neg\psi_j, \{q_j \mid M_2 \models b_j\})\end{array}$$

that are composed with moves of A and added to T in Step (ii), where $M_1, M_2 \in \mathbf{M}$. By using Proposition 1, we need to show that if $Target(t_1) \neq Target(t_2)$ (i.e., for some b_j, $M_1 \models b_j$ and $M_2 \models \neg b_j$), then $Cond(t_1) \wedge Cond(t_2)$ is unsatisfiable, which holds because there is at least one ψ_j such that ψ_j is a conjunct of $Cond(t_1)$ and $\neg\psi_j$ is a conjunct of $Cond(t_2)$.

The property that all possible satisfiable combinations of B-moves are considered in Step (ii) and that the composition with A-moves preserves satisfiability of the conditions of the moves added to T, follows from Proposition 3 and the added constraint that $\bigvee \vec{a}$ is true in the computation of $\mathbf{M}$.

Finally, note that if A represents an SFA then so does C.

A. Difference checking

The above algorithm has also a more efficient version in the case when A represents an SFA and the purpose is to decide (in isolation) if $L(C) = L(A) \setminus L(B)$ is empty, and to provide a single witness in $L(C)$ otherwise. For this version, the explicit construction of the moves of C is not needed. The checking of final states can be done when an element is popped from S and a "witness tree" can be incrementally updated (instead of T) that records links backwards from newly found target states to their source states.

VI. Implementation

The algorithms discussed in Section III and Section V and the axiom generation discussed in Section IV have been implemented in a prototype tool for analyzing regular expressions and context

free grammars. The SMT solver Z3 is used for satisfiability checking and model generation. We use some features that are specific to Z3, including the integrated combination of decision procedures for algebraic data-types, integer linear arithmetic, bit-vectors and quantifier instantiation. We also make use of incremental features so that we can manipulate logical contexts while exploring different combinations of constraints. Use of algebraic data-types is central in the construction of the language acceptors, as was illustrated in Section IV. The definitions of the axioms match very closely with the actual implementation.

Working within a context enables *incremental* use of the solver. A 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 use of contexts is illustrated in Figure 2 that shows a simplified code snippet from Rex responsible for computing $Solutions(Cube(\vec{\varphi}, \vec{b}))$ in the difference construction algorithm in Section V, where the solutions are generated incrementally using a context, and the *model generation* feature is used to extract the actual solutions from Z3.

```
...
z3.Push();  //push a new context for collecting solutions
Term[] b = ...    //fresh Boolean constants for B-moves
Term[] a = ...    //fresh Boolean constants for A-moves
Term[] cube = ... //corresponding cube equations

z3.AssertCnstr(z3.MkAnd(cube)); //assert the cube formula
z3.AssertCnstr(z3.MkOr(a)); //at least one a[i] must hold
Model M;
while (z3.CheckAndGetModel(out M) != LBool.False) //get M
{
  AddToSolutions(M);                    //record M
  z3.AssertCnstr(Negate(M,a,b));        //exclude M
}
z3.Pop(); //return to the previous context
...
```

Figure 2. Computation of solutions for cubes.

VII. Experiments

We conducted several experiments where we evaluated the performance of the difference algorithms and the axiomatization approach on a collection of sample regexes shown in Table I.[1] These are typical examples of concrete regexes appearing in various practical contexts. The regexes are taken from [13], where the analysis is not able to handle several of the regexes for member generation. In all cases $L(r_i) \not\subseteq L(r_j)$ for regexes i and j, $i \neq j$.

A. Experiment 1

Table II shows the time t it took with the difference construction algorithm (Section V) to construct the automaton for $L(r_i) \setminus L(r_j)$ and to generate a witness using the axioms of the automaton, where i is the row number and j is the column number in the table. The table also shows the number of states of the constructed automata. For every (nonempty) automaton A that was constructed for $L(r_i) \setminus L(r_j)$ we performed a *member generation* check as follows.

1) Declare fresh constants $s : \mathbb{W}$, $k : \mathbb{N}$, and assert $Th(A)$ and $Acc^A(s, k)$.
2) Generate a model M for the assertions and extract a witness in $L(A)$.

The experiment illustrates that the use of axioms scales. For example, the automaton constructed for $\#7 \setminus \#8$ has almost 5k sates (and around 14k moves, the automata are typically sparse) and the size of the theory is proportional to the size of the automaton that has in the order of 10k axioms that are created on-the-fly. For checking $L(r_9) \subseteq L(r_9)$ the algorithm did not terminate (using a timeout of > 10 min). Overall, the whole experiment took around 1 minute to complete (if the case $i = j = 9$ is excluded).

B. Experiment 2

What is the payoff (if any) in using the difference construction algorithm as opposed to a direct encoding using a Boolean combination of the acceptor formulas? We conducted the following experiment for each pair of sample regexes r_i and r_j.

1) Construct the automata A and B for r_i and r_j, respectively and assert the theories $Th(A)$ and $Th(B)$. (For the case when $i = j$ we modify B slightly but keep it equivalent to A, so the theories for A and B are disjoint. If we use the same theory and acceptor predicate, the problem becomes trivial).
2) Assert the goal $Acc^A(s, k) \wedge \neg Acc^B(s, k)$ with fresh $s : \mathbb{W}$ and $k : \mathbb{N}$.
3) Check and generate a model for the goal.

Table III
Results of experiment 2.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
#1	?	109	47	124	78	156	78	141	171	2746
#2	16	?	15	16	15	16	31	31	110	31
#3	<1	16	**78**	16	<1	15	47	31	109	16
#4	156	296	110	?	78	156	202	156	172	171
#5	31	32	15	63	?	62	94	47	140	47
#6	16	31	16	<1	<1	?	47	31	109	16
#7	94	78	62	140	78	62	?	141	203	93
#8	171	141	140	219	249	375	670	?	1061	671
#9	172	171	156	187	188	172	327	281	?	203
#10	172	46	141	78	327	78	297	171	203	?

The result of the experiment is somewhat surprising. See Table III. In the case when the difference is nonempty, the direct encoding clearly outperforms the difference construction in many cases. In contrast, the experiments done in [11] indicated that it is beneficial to candider $Th(A \times B)$ rather than $Th(A) \cup Th(B)$. However, for $i = j$, the direct encoding only terminates when the automaton has no loops (which is the case when $i = j = 3$). Overall, the experiment took around 15 seconds to complete (if the non-terminating cases are excluded).

C. Experiment 3

For each pair of regexes r_i and r_j, $1 \leq i, j \leq 10$, Table IV shows the time it took with the difference checking algorithm (Section V-A) to decide if $L(r_i) \subseteq L(r_j)$ and to generate a witness in $L(r_i) \setminus L(r_j)$ if $L(r_i) \not\subseteq L(r_j)$, where i is the row number and j is the column number in the table.

As expected, the full construction of the difference automata and member generation for it takes more time than generating a single witness. The total time of the experiments (excluding the

[1] All experiments were executed on a Lenovo T61 laptop with Intel dual core T7500 2.2GHz processor.

Table I
SAMPLE REGEXES.

#1	`\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*([,;]\s*\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)*`
#2	`$?(\d{1,3},?(\d{3},?)*\d{3}(\.\d{0,2})?|\d{1,3}(\.\d{0,2})?|\.\d{1,2}?)`
#3	`([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})?`
#4	`[A-Za-z0-9](([ \.\-]?[a-zA-Z0-9]+)*)@([A-Za-z0-9]+)(([\.\-]?[a-zA-Z0-9]+)*)\. ([A-Za-z][A-Za-z]+)`
#5	`(\w|-)+@((\w|-)+\.)+(\w|-)+`
#6	`[+-]?([0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)([eE][+-]?[0-9]+)?`
#7	`((\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})?)`
#8	`(([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}`
#9	`(([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})+)*`
#10	`((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[,]{0,1}\s*)+`

Table II
RESULTS OF EXPERIMENT 1.

	#1		#2		#3		#4		#5		#6		#7		#8		#9		#10	
	t	states	t	states	t	states	t	states	t	states	t	states	t	states	t	states	t	states	t	states
#1	**93**	1	125	33	62	24	110	42	327	45	63	26	764	604	515	283	452	40	2091	103
#2	46	34	**32**	1	15	25	47	34	31	31	31	34	47	34	63	34	405	34	31	34
#3	32	29	31	29	**15**	1	31	35	16	29	16	29	31	29	94	35	374	35	31	29
#4	500	29	156	22	156	22	**31**	1	125	27	249	19	905	373	437	154	3432	279	265	39
#5	63	24	31	11	31	7	47	23	**15**	1	16	9	234	237	296	208	406	8	125	41
#6	31	12	31	25	16	8	31	16	16	11	**15**	1	31	12	47	18	375	15	15	12
#7	687	564	156	160	156	147	686	499	312	298	140	153	**172**	1	6583	4784	640	170	1419	760
#8	515	430	749	127	1030	260	1201	307	2215	294	640	108	4804	5023	**203**	1	4275	241	3198	573
#9	1061	532	998	530	983	527	1295	668	1029	526	967	529	1248	697	1170	623	∞	?	999	542
#10	483	78	78	38	63	29	140	51	406	57	140	31	1311	823	1029	418	453	49	**265**	1

Table IV
RESULTS OF EXPERIMENT 3.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
#1	**78**	15	<1	31	<1	16	31	16	359	124
#2	<1	**32**	15	<1	16	<1	31	16	405	16
#3	<1	15	**32**	15	<1	16	15	16	374	<1
#4	16	16	15	**31**	<1	<1	31	16	359	15
#5	16	<1	15	16	**16**	<1	15	31	359	<1
#6	<1	15	<1	16	<1	**16**	15	16	359	15
#7	16	15	16	31	16	31	**172**	62	390	16
#8	16	<1	<1	16	15	16	31	**187**	483	16
#9	374	359	359	359	359	343	359	390	∞	359
#10	16	15	<1	16	16	<1	31	16	374	**265**

case for $i = j = 9$ that did not terminate) was around 10 sec for this experiment and around 1 min for the first experiment. The performance is the same in the case when the difference is empty. The immediate advantage over the direct encoding used in the second experiment is the case when the difference is empty.

D. Experiment 4

We consider an experiment that combines regex constraints with length constraints on strings. Let r_1 and r_2 be the regexes "`.{n}a.*`" and "`.*a.{n}`", respectively, where n is a positive integer. Thus, r_1 requires that the $(n+1)$'st character from the beginning is `a`, and r_2 requires that the $(n+1)$'st character from the end is `a`. Consider an SQL select condition of the form[2]

$$s \texttt{ LIKE } r_1 \texttt{ AND NOT } s \texttt{ LIKE } r_2 \texttt{ AND } 3 * \texttt{LEN}(s) > 2 * n \quad (6)$$

Let A and B be ϵSFAs such that $L(A) = L(r_1)$ and $B = L(r_2)$. The sample was chosen for several reasons. Complementation of B would require a DFA with 2^{n+1} states. Unlike in the previous experiments, the difference algorithm reduces in this case to the second special case where B^r (accepting $L(r_2)^r$) in the algorithm is deterministic. We can systematically increase n and compare the performance of a *direct encoding* of (6):

$$Acc^A(s,k) \wedge \neg Acc^B(s,k) \wedge 3k > 2n,$$

against the *difference encoding* of (6):

$$Acc^{A \setminus B}(s,k) \wedge 3k > 2n,$$

where $s : \mathbb{W}$ and $k : \mathbb{N}$ are fresh constants. For both encodings, we measured model generation time for $n = 1 \ldots 100$. The result of the experiment is shown Figure 3. In both cases the trend-line is polynomial ($O(n^3)$ for the direct encoding and $O(n^2)$ for the difference encoding). Thus, even though the acceptor for B occurs negatively in the direct encoding of (6), this does not cause exponential behavior during model generation.

E. Comparison with Hampi

To our knowledge, a system that comes closest to the scope of ours is the open source string constraint solver Hampi [14]. We

[2]The `LIKE`-pattern corresponding to r_2 is "`%a_..._`", and for r_1 is "`_..._a%`" with n occurrences of '_'.

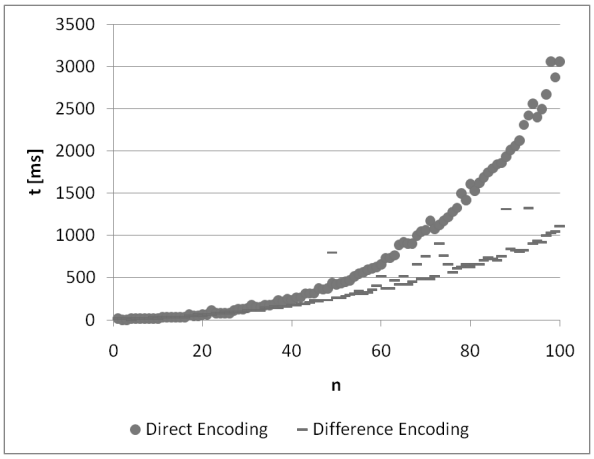

Figure 3. Experiment 4 model generation times.

conducted an experiment similar to Experiment 1 using Hampi. The following experiments were run on a desktop with an Intel dual core E8300 2.8GHz processor. Given the regexes r_i, r_j, $i \neq j$, from Table I, Hampi input corresponding to the membership constraint $x \in L(r_i) \setminus L(r_j)$ is:

$$\texttt{var}\ x : l;\ \texttt{reg}\ a := R_i;\ \texttt{reg}\ b := R_2;$$
$$\texttt{assert}\ x\ \texttt{in}\ a;\ \texttt{assert}\ x\ \texttt{not in}\ b;$$

where R_i is a Hampi representation of the regex r_i. The declaration $\texttt{var}\ x : l$ constrains the length of x to be l. Altough Hampi supports length ranges $\texttt{var}\ x : l_{lower} .. l_{upper}$ the range declaration caused segmentation faults in the underlying STP [15] solver, so we resorted to using the more restricted case. The experiment with using $l = 10$ took a toal of 2min to complete for the 90 cases. By setting $l = 15$, the experiment took 4min 30sec to complete. For values of $l < 10$, several of the membership constraints become unsatisfiable and fail to detect nonemptiness of $L(r_i) \setminus L(r_j)$. For example, for $l = 3$, the experiment took 1min and 30sec, but for most of the constraints the result was unsat.

VIII. Related work

The work presented here is a nontrivial extension of the work started in [11] where different ϵSFA algorithms and their effect on language acceptors for ϵSFAs (including minimization and determinization) are studied. The experiments in [11] failed in determinization, which needed the idea of solving *cube* formulas. Moreover, the approach of language acceptors presented in [11] does not support precise length constraints, and the axioms were not studied for ϵSPDAs. Theorem 1 strengthens a similar statement for ϵSFAs in [11].

Although, an extension of FAs with predicates has been suggested earlier [16], and later formalized and implemented in Prolog as an automata library [17], we are not aware of similar results for PDAs that make the difference algorithm possible. We are also not aware of symbolic analysis with SMT being studied, based on such extensions.

A tool developed in [18] is used to compute a context free grammar G as a conservative approximation of possible string values of variables of a given PHP Web application, and to check if $L(G) \cap L(R)$ is empty for a given regex R representing "bad" strings (strings that may cause a security risk). This technique is used to check for SQL injection vulnerabilities of Web applications. The HAMPI [14] tool, that is string constraint solver, has an additional advantage that it can produce a witness in $L(G) \cap L(R)$ if it is nonempty, provided that G is first finitized. HAMPI turns string constraints over fixed-size string variables into a query to STP [15] that is a solver for bit-vectors and arrays. The input size needs to be fixed, since STP neither supports lazily instantiated quantifiers nor the theory of algebraic data types. The approach described here is capable of performing the same task without requiring G to be finitized first, and can moreover be combined with other constraints.

A connection between logic and automata has been studied for over fifty years ago [19], [20], and revived about decade ago [21] in the context of symbolic reasoning with Binary Decision Diagrams (BDDs) [22]. 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 supported by the underlying (SMT) solver.

Several program analysis techniques for programs with strings [7], [23], [24], [25] build on automata libraries [21], [26] 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. In particular, symbolic analysis of SQL queries with an SMT solver is discussed in [3]. Another instance is the analysis .NET programs [1], which use a rich set of string operations [2], [13].

A decision procedure for subset constraints over regular language variables is introduced in [27] by reasoning over dependency graphs. In contrast, we showed how finite pushdown automata can be generalized by making transitions symbolic, and how a decision procedures can be embedded into a logic of an SMT solver.

In [28] several decision problems related to CFGs are studied and depth-bounded versions thereof are mapped to SAT solving. In particular, an algorithm is provided for checking bounded version of ambiguity (whether a string has more than one parse tree) of CFGs. A particular advantage of the approach in [28] over the algorithm in [29] is that a *witness* can be produced when a grammar G is ambiguous. As an interesting direction for future work, we can approach the same problem by extending symbolic acceptors with an argument that captures a parse tree of a string; where a parse tree can be represented with an algebraic data-type based on the productions of G. This approach avoids the need to provide a priori depth-bounds.

IX. Conclusion

We believe that the use of symbolic language acceptors as a purely logical description of formal languages and their mapping to state of the art SMT solving techniques opens up a new approach to analyzing and solving language theoretic problems. We have

demonstrated the scalability of the technique on solving extended regular constraints, that have direct applications in static analysis, testing, and database query analysis. We have also experimented with symbolic language acceptors for CFGs. In this context it is not clear if the normal forms that are important for efficient implementation, play the same role for efficient symbolic analysis. For example, epsilon elimination from ϵSFAs or unit production elimination from CFGs may eliminate sharing and increase the complexity of symbolic analysis.

ACKNOWLEDGEMENT

The experiments in Section VII-E would not have been possible without the help of *Pieter Hooimeijer* who set up the whole environment for the experiments and provided scripts for converting the regex expressions to Hampi format.

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

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

[6] 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.

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

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

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

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

[11] M. Veanes, P. de Halleux, and N. Tillmann, "Rex: Symbolic Regular Expression Explorer," in *ICST'10*. IEEE, 2010.

[12] N. Blum and R. Koch, "Greibach normal form transformation revisited," *Inf. Comput.*, vol. 150, no. 1, pp. 112–118, 1999.

[13] N. Li, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Reggae: Automated test generation for programs using complex regular expressions," in *Proceedings of the 24th IEEE International Conference on Automated Software Engineering*, 2009.

[14] 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.

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

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

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

[18] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *PLDI'07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2007, pp. 32–41.

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

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

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

[22] 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.

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

[24] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid, "Abstracting symbolic execution with string analysis," in *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–22.

[25] 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.

[26] "Brics finite state automata utilities," http://www.brics.dk/automaton/.

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

[28] R. Axelsson, K. Heljanko, and M. Lange, "Analyzing context-free grammars using an incremental SAT solver," in *ICALP'08, Part II*, ser. LNCS, vol. 5126. Springer, 2008, pp. 410–422.

[29] S. Schmitz, "Conservative ambiguity detection in context-free grammars," in *ICALP'07*. Springer, 2007, pp. 692–703.

APPENDIX

To capture the essence of the combination of length and regular constraints we consider formulas of the form:

$$\left(\bigwedge_i s_i \in r_i\right) \wedge F(|s_1|, \ldots, |s_n|)$$

where each s_i, $i = 1, \ldots, n$ is a variable ranging over strings, each r_i is a regular expression, and F is a quantifier-free Presburger formula. We will show that the formula is equisatisfiable to

$$\left(\bigwedge_i s_i \in r_i\right) \wedge F(|s_1|, \ldots, |s_n|) \wedge \bigwedge_i |s_i| \leq |r_i| \cdot 2^{p(sz)}$$

where $p(sz)$ is a polynomial and sz is the size of formula F.

Let $\vec{s}$ denote the product string obtained from $s_1, \ldots, s_n$ by aligning the character positions from each s_i. We can also assume that there is a special end-of-string character, `\000`, so that the product construction does not need to worry about miss-aligned strings (string lengths are still computed up to, but excluding the end-of-string character). We will make use of properties of quantifier-free Presburger formulas and Hilbert bases. The properties have been previously used for checking satisfiability of multi-sets with cardinality bounds by Piskac and Kuncak in *(Linear Arithmetic with Stars, CAV 2008)*.

Lemma 1: Let $F(x_1, \ldots, x_n)$ be a quantifier-free Presburger formula. Then there is a collection J and , A_j, B_j for $j \in J$, where $|J| \leq 2^{p(sz)}$ and each A_j, B_j is a set of n-dimensional vectors, such that

$$F(x_1, \ldots, x_n) \;\Leftrightarrow\; (x_1, \ldots, x_n) \in \bigcup_{j \in J} (A_j + B_j^*)$$

where $+$ is extended to sets (it is also known as the Minkowski sum) and B_j^* is a linear combination of vectors from B_j.

Furthermore, there is a polynomial $p(sz)$, where sz is the size of F, such that $||A_j||_1 + ||B_j||_1 \leq 2^{p(sz)}$.
We can then derive the following equivalences:

$$
\begin{aligned}
& (\bigwedge_i s_i \in r_i) \wedge F(|s_1|, \ldots, |s_n|) \\
\equiv\ & \text{Let } R \text{ be the product automata of } r_i \\
& \vec{s} \in R \wedge F(|s_1|, \ldots, |s_n|) \\
\equiv\ & \text{Let } A_j \text{ and } B_j \text{ be as in lemma 1} \\
& \vec{s} \in R \wedge (|s_1|, \ldots, |s_n|) \in \bigcup_j (A_j + B_j^*) \\
\equiv\ & \text{Let } A_{F_j} \text{ the automata encoding of the semi-linear sets} \\
& \vec{s} \in R \wedge \bigvee_{j \in J} \vec{s} \in A_{F_j} \\
\equiv\ & \text{By basic properties of products} \\
& \bigvee_{j \in J} \vec{s} \in R \times A_{F_j} \\
\equiv\ & \text{By downwards pumping} \\
& \bigvee_{j \in J} \vec{s} \in R \times A_{F_j} \wedge |\vec{s}| \leq |R \times A_{F_j}| \\
\equiv\ & \text{By a size estimate on } A_{F_j} \\
& \bigvee_{j \in J} \vec{s} \in R \times A_{F_j} \wedge |\vec{s}| \leq |R| \cdot 2^{p(sz)} \\
\equiv\ & \text{By replacing } A_{F_j} \text{ by } F \\
& \vec{s} \in R \wedge F(|s_1|, \ldots, |s_n|) \wedge |\vec{s}| \leq |R| \cdot 2^{p(sz)}
\end{aligned}
$$