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:

\[
\text{SELECT * FROM } T \\
\text{WHERE } C \text{ LIKE } r_1 \text{ AND NOT } C \text{ LIKE } r_2 \text{ AND LEN}(C) < D + E
\]

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 \( \text{Acc}^L(w, k) \) that is true modulo a theory \( Th(L) \) if and only if \( w \in L \) 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 \( \text{Acc}^A_r \) 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,

\[
\text{Acc}^{A_{r_1}}(c, k) \land \neg \text{Acc}^{A_{r_2}}(c, k) \land k < d + e
\]

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 (3) has the form

\[
\text{Acc}^{A_{r_1} \times A_{r_2}}(c, k) \land k < d + e
\]

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:

\[
\text{Acc}^{A_{r_1} \setminus A_{r_2}}(c, k) \land k < d + e
\]

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 benefit 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], [3]. 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 \(\sigma\), \(\mathcal{U}^\sigma\) is a separate sub-universe of \(\mathcal{U}\). The basic sorts needed in this paper are the Boolean sort \(\mathbb{B}\), \(\mathcal{U}^{\mathbb{B}} = \{\text{true}, \text{false}\}\), 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 where 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 \(\mathcal{C}\) stand for a fixed character sort for some fixed \(n\), and the complete alphabet is thus \(\mathcal{U}^\mathcal{C}\). 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 \(\sigma\). \(f : \sigma_0 \times \cdots \times \sigma_{n-1} \rightarrow \sigma\). For example, there is a built-in relation or predicate (Boolean function) symbol \(\leq : \mathcal{C} \times \mathcal{C} \rightarrow \mathbb{B}\) that provides a strict total order of all the characters. One can also declare fresh (new) uninterpreted function symbols 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 \(F_C\) denote the set of all quantifier-free formulas with at most one fixed variable of sort \(\mathcal{C}\). Throughout the paper, we denote that variable by \(\chi\). Given \(\varphi \in F_C\), and a character or term \(t\) of sort \(\mathcal{C}\), we write \(\varphi[t]\) for the formula where each occurrence of \(\chi\) 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 \(\varphi\) provides an interpretation for all the uninterpreted symbols in \(\varphi\). A model \(M\) for a closed formula \(\varphi\) satisfies \(\varphi\), \(M \models \varphi\), if the interpretations provided by \(M\) make \(\varphi\) true. A closed formula \(\varphi\) is satisfiable if it has a model. A formula \(\varphi\) with \(FV(\varphi) = \emptyset\) is satisfiable if its existential closure \(\exists \chi \varphi\) is satisfiable. We write \(\models \varphi\), if \(\varphi\) is valid (true in all models for \(\varphi\)).

For example, the character range set \([a-z]\) in a regex is translated into the formula \(\psi = (\`a` \leq \chi \wedge \chi \leq \`z`) \vee (\`0` \leq \chi \wedge \chi \leq \`9`)\) with \(\chi\) as the single free variable in \(\psi\). The formula \(\psi\) is satisfiable; \(\psi[\`a`]\) is true; \(\psi[\`z`]\) is false. Note that \(\`a`\), \(\`z`\), \(\`0`\) and \(\`9`\) in \(\psi\) 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 \((p, \varphi, q)\) from \(p\) to \(q\), where \(\varphi \in F_C\), such that

\[
\{p\} = \{a_1, \ldots, a_n\},
\]

where \(\{p\} \overset{\text{def}}{=} \{a \mid a \in \mathcal{U}^\mathcal{C}, \models \varphi[a]\}\). Let also

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

and, given a set \(\Delta\) of moves, let

\[
\{\Delta\} \overset{\text{def}}{=} \{\{\tau \mid \delta \in \Delta, \tau \in \delta\}\}.
\]

Note that \(\{(p, \varphi, q)\} = \emptyset\) iff \(\varphi\) is unsatisfiable. Define also

\[
\begin{align*}
\text{Source}((p, \varphi, q)) & \overset{\text{def}}{=} p, \\
\text{Target}((p, \varphi, q)) & \overset{\text{def}}{=} q, \\
\text{Cond}((p, \varphi, q)) & \overset{\text{def}}{=} \varphi.
\end{align*}
\]

For example, the move

\[(p, \`a` \leq \chi \wedge \chi \leq \`z`, q)\]

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 SF A \(A\) is a tuple \((Q, q_0, F, \Delta)\), where \(Q\) is a finite set of states, \(q_0 \in Q\) the initial
We also use the following notations.

\[ \Delta_A(q) \triangleq \{ t \mid t \in \Delta_A, \text{Source}(t) = q \} \]

\[ \Delta_A(q) \triangleq \cup \{ \Delta_A(q) \mid q \in q \} \]

\[ \text{Target}(t) \triangleq \cup \{ \text{Target}(t) \mid t \in t \} \]

Just as with finite automata, it is 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 eSFA is a tuple \( (Q, q_0, F, \Delta) \), where \( Q, q_0 \) and \( F \) are as above, and \( \Delta : Q \times (\mathcal{F}_C \cup \{ \epsilon \}) \times Q \).

The term SFA without the additional qualification allowing epsilon moves implies that epsilon moves do not occur. (Obviously, any SFA is also an eSFA.) Let \([p, q, \epsilon] \triangleq (p, q, \epsilon)\). An eSFA \( A = (Q, \Delta, q_0, F) \) denotes the finite automaton \([A] \) with epsilon moves [9], where

\[ [A] \triangleq (Q, \mathcal{L}, [\Delta], q_0, F). \]

We write \( \Delta_A \) for the set of all epsilon moves in \( \Delta_A \) and \( \Delta'_A \) for \( \Delta_A \setminus \Delta' \). Epsilon elimination for eSFAs 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 eSFA is clean then so is the resulting SFA, since the only combination of move conditions performed during epsilon elimination is disjunction.

**Definition 3:** An eSFA \( A \) is normalized if there are no two distinct moves \( (p, \varphi_1, q), (p, \varphi_2, q) \) in \( \Delta'_A \).

It is clear that for any eSFA there is a normalized SFA \( A' \) such that \([A] = [A']\); for all states \( p \) and \( q \) in \( A \), make a disjunction \( \varphi \) of all the conditions of the moves from \( p \) to \( q \) in \( \Delta_A \) and let \((p, \varphi, q)\) be the single move in \( \Delta_A' \) 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 eSFA \( A \) is clean if all moves in \( \Delta_A \) 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 \( \varphi_1 \land \varphi_2 \) 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 eSFA 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 eSFA are illustrated in Figure 1.

---

**Figure 1.** Sample eSFA generated from the regex \( [d+] \{a-z\} \). The initial state is gray, the epsilon moves are dashed. The symbol 'a' 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 \( \mathcal{F}_C \) 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 \( \Delta : Q \times Z \times \mathcal{F}_C \times Q \times Z \) is the move relation.

Similar to eSFAs an SPDA may have moves where the condition is \( \epsilon \). Each move \( t = (q, z, \alpha, p, z') \) of an SPDA, where \( \alpha \in \mathcal{F}_C \cup \{ \epsilon \} \), denotes the set of transitions \([t] \triangleq \{(q, z, \alpha, p, z') \mid \alpha \in [\alpha]\}\). The underlying PDA is denoted by \([A]\) (with the alphabet \( \mathcal{L}_F \)). The language \( L(A) \) accepted by \( A \) is the language \( L([A]) \) accepted by the PDA \([A]\). We reuse the definitions of \( \text{Source}(t) \), \( \text{Target}(t) \), and \( \text{Cond}(t) \) also for moves \( t \) of an SPDA and introduce the additional definitions

\[ \text{Pop}(q, z, \alpha, p, z') \triangleq z, \]

\[ \text{Push}(q, z, \alpha, p, z') \triangleq z'. \]

An SPDA \( A \) is clean if all moves in \( \Delta_A \) have satisfiable conditions, and normalized if there are no two moves \( t_1 \) and \( t_2 \) in \( \Delta_A \) that differ only with respect to \( \text{Cond}(t) \).

Elimination of epsilon moves from an eSPDA 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 eSFA \( A \) can be trivially transformed into an eSPDA: let \( Z_A = \{ z \} \), \( z_0A = z \), and for each move \( t \in \Delta_A \) let \( \text{Pop}(t) = z \) and \( \text{Push}(t) = z \). We say that the resulting eSPDA represents an eSFA.

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 eSPDA with an eSFA. 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 eSPDA \( A \) and an SFA \( B \) and the output is an eSPDA \( C \) 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 \( T = \emptyset \), \( S = (\{(q_0A, q_0B), z_0A\}) \), and \( V = \{ \langle (q_0A, q_0B), z_0A \rangle \} \).

(i) If \( S \) is empty go to (ii) else pop \( \langle (p, q), z \rangle \) from \( S \).
(ii) For each \( (p, z, \varphi, p', \vec{z}) \in \Delta_A \) and \( (q, \psi, q') \in \Delta_B \). If \( \varphi \land \psi \) is satisfiable then:
- add \( (p, q, z, \varphi \land \psi, (p', q'), \vec{z}) \) to \( T \);
- for each \( z' \) in \( \vec{z} \), if \( v = \langle (p', q'), z' \rangle \notin V \) then add \( v \)
to \( V \), and if there is \( t \in \Delta_A \) with \( \text{Source}(t) = p' \) and
\( \text{Pop}(t) = z' \) then push \( v \) to \( S \).

Go to (i).

(iii) Let \( C = (\pi_1(V), \pi_2(V), \langle q_{0A}, q_{0B} \rangle, \pi_{0A}, \pi_{1}(V) \cap (F_A \times F_B), T) \).

(iv) Eliminate dead states from \( C \) (states from which no final state is reachable).

Note that \( |Q_C| \) is at most \( |Q_A| \ast |Q_B| \). 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. SYMBOLOGIC LANGUAGE ACCEPTORS

To encode language acceptors, we use particular kinds of axioms, all of which are equations of the form
\[
\forall z (t_{ha} = t_{rha}) \tag{5}
\]
where \( FV(t_{ha}) = \bar{x} \) and \( FV(t_{rha}) \subseteq \bar{x} \). When \( t_{ha} \) and \( t_{rha} \) are formulas, we often write \( \langle \varphi \rangle \) instead of \( \langle \bar{z} \rangle \). The left-hand-side \( t_{ha} \) 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 \( e \)SMPDA \( A \) 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 \( A \):
\[
L(A) = \{ w \in M \mid M \models Th(A) \land Acc^A(w, k) \}
\]
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 \( \sigma \), \( L(\sigma) \) is the
list sort with element sort \( \sigma \). For a given element sort \( \sigma \) there is an empty list \( \text{nil} \) (of sort \( L(\sigma) \)) and if \( e \) is an element of sort \( \sigma \) and \( l \) is a list of sort \( L(\sigma) \) then \( \text{cons}(e, l) \) is a list of sort \( L(\sigma) \). The accessors are, as usual, \( \text{hd} \) (head) and \( \text{tl} \) (tail).

Strings are represented by lists of characters; we write \( \mathbb{W} \) for the sort \( L(C) \). The empty string is abbreviated by \( "\) and a string
\( \text{cons}("a", \text{cons}("b", \text{cons}("c", \text{nil})) \) is abbreviated by \( "abc" \).

B. Construction of \( Th(A) \)

Let \( A \) be a given \( e \)SMPDA. 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 \( Z_A \subseteq \mathbb{N} \). 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 \( S = L(\mathbb{Z}) \). 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 \( (z_1, \ldots, z_k) \cdot (\overline{z}_{k+1}, \ldots, z_l) \) for \( (z_1, \ldots, z_l) \).

The empty stack is denoted by \( \varepsilon \).

For all \( q \in Q_A \), declare the predicate symbol
\[
Acc^A_q : \mathbb{W} \times \mathbb{N} \times \mathbb{S} \rightarrow \mathbb{B}
\]
An ID of \([A]\) is a triple \( (q, w, s) \) where \( q \in Q_{[A]} \), \( w \) is a string and \( s \) is a stack \( \mathbb{S} \). 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 \( e \)SMPDA \( A' \) 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 \( (q, w, s) \vdash_{[A]} (p, "\), \( \varepsilon \) for some \( p \in Q_{[A]} \), where \( \vdash_{[A]} \) is the step relation of \([A]\) as defined in [9] page 112]. Declare also
\[
Acc^A_q : \mathbb{W} \times \mathbb{N} \rightarrow \mathbb{B}
\]
For \( q \in Q_A \) and \( z \in Z_A \) let \( \Delta_A (q, z) \) be
\[
\{(q, z, \varphi, q_i, \vec{z}_i) \mid 1 \leq i \leq m \} \cup \{(q, z, \varepsilon, q_i, \vec{z}_i) \mid m < i \leq k \}.
\]
Define the following axioms.
\[
ax^A_q \equiv \forall w n \in Acc^A_q (w, n, cons(z_{0A}, \varepsilon))
\]
\[
ax^A_{q, n} \equiv \forall w n \in Acc^A_q (w, n, \varepsilon) \leftrightarrow (w = "\) \land n = 0)
\]
\[
ax^A_q \equiv \forall n s \in Acc^A_q (w, n, cons(z, s)) \leftrightarrow
\]
\[
\{(w \neq "\) \land n > 0 \land
\]
\[
\bigvee_{i=1}^m (\varphi_i [\text{hd} (w)] \land Acc^A_{q_i} (tl (w), n-1, z_i \cdot s))}
\]
\[
\bigvee_{j=m+1}^k Acc^A_{q_j} (w, n, z_j \cdot s)
\]
\[
Th(A) \equiv \{ ax^A_q \} \cup \{ ax^A_{q, n} \mid q \in Q_{[A]}, z \in Z_A \}.
\]
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 \( e \)SMPDAs but does not hold for all \( e \)SMPDAs, i.e., when arbitrary epsilon moves are allowed.

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

Proof outline: For the case when \( A \) is an epsilon-loop-free \( e \)SFA the proof is similar to the proof of [11] Theorem 1. Assume \( A \) is an \( e \)SMPDA. Since epsilon moves are not present, the
following statement follows by induction over the length of $\text{\textquoteleft\textquoteleft} A\text{\textquoteright\textquoteright}$-computations. For all $\text{\textids{q, w, s}}$ of $[A]$:  
\[ \exists p \in Q_{A}((q, w, s) \text{\textquoteleft\textquoteleft} A\text{\textquoteright\textquoteright} (p, \text{""}, e)) \iff \text{Th}(A) \models \text{Acc}_{A}^{\delta}(w, w|, s) \]

The theorem follows, by letting $q = q_{0_{A}}$ and $s = \delta_{0_{A}}$. \qed

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

Example 1: Let $A = (\{q\}, \{z\}, q, z, \emptyset, \{q, z, e, q(z)\})$. For example, $(q, \text{\text{""}}, \text{\text{""}}, z)$ is the language accepted by $A$ is empty. The theory $\text{Th}(A)$ for $A$ includes the axiom $\text{ax}_{A}^{\delta_{A}}$:  
\[ \forall w n s (\text{Acc}_{A}^{\delta}(w, n, \text{cons}(z, s)) \models \text{Acc}_{A}^{\delta}(w, n, \text{cons}(z, s))) \]

This axiom is a useless tautology. Consider for example a model $M$ with an interpretation for $\text{Acc}_{A}^{\delta}$ such that $M \models \text{Acc}_{A}^{\delta}(\text{""}, 0, (z))$ and expand $M$ so that $M \models \text{ax}_{A}^{\delta} \land \text{ax}_{A}^{\delta}$. Then $M \models \text{Acc}_{A}^{\delta}(\text{""}, 0)$ but $\text{""} \not\in L(A)$. \qed

For all $\text{sSPA}$s there is an equivalent loop-free $\text{sSPA}$ that can be computed effectively, e.g., the GNF normal form for CFGs implies that.

For an $\text{sFA}$ (or an $\text{sSPA}$ that represents an $\text{sFA}$) 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 $\text{sFA}$ in Figure 1. The axioms $\text{Th}(A)$ for $A$ are as follows, where $\varphi = (\chi \geq \text{\textquoteleft\textquoteleft} a\text{\textquoteright\textquoteright}) \land (\chi \leq \text{\textquoteleft\textquoteleft} z\text{\textquoteright\textquoteright})$ and $\psi = (\chi \geq \text{\textquoteleft\textquoteleft} 0\text{\textquoteright\textquoteright}) \land (\chi \leq \text{\textquoteleft\textquoteleft} 0\text{\textquoteright\textquoteright})$ and $q \in Q_{A}$:

\[ ax_{0}: \forall v w n (\text{Acc}(w, n) \iff (\text{Acc}(w, n) \lor \text{Acc}(w, n))) \]
\[ ax_{1}: \forall v w n (\text{Acc}(w, n) \iff (w \neq \text{\textquoteleft\textquoteleft} n\text{\textquoteright\textquoteright} \land \psi[h(w)] \land \text{Acc}_{A}(\text{Acc}(w, n - 1))) \]
\[ ax_{2}: \forall v w n (\text{Acc}(w, n) \iff ((w \neq \text{\textquoteleft\textquoteleft} n\text{\textquoteright\textquoteright} \land \psi[h(w)] \land \text{Acc}_{A}(\text{Acc}(w, n - 1))) \lor (w = \text{\textquoteleft\textquoteleft} n\text{\textquoteright\textquoteright} \land n = 0)) \]
\[ ax_{3}: \forall v w n (\text{Acc}(w, n) \iff (w \neq \text{\textquoteleft\textquoteleft} n\text{\textquoteright\textquoteright} \land \psi[h(w)] \land \text{Acc}_{A}(\text{Acc}(w, n - 1))) \]
\[ ax_{4}: \forall v w n (\text{Acc}(w, n) \iff (w = \text{\textquoteleft\textquoteleft} n\text{\textquoteright\textquoteright} \land n = 0)) \]

Declare fresh constants $s, t, k : \mathbb{N}$ and assert the axioms $\text{Th}(A)$ and the goal $\text{Acc}(s, k)$ 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 underlining it:

\[ \text{Acc}(s, k) \underset{ax_{2}}{\rightarrow} \text{Acc}(s, k) \lor \text{Acc}(s, k) \]
\[ \text{Acc}(s, k) \lor (s \neq \text{\textquoteleft\textquoteleft} n\text{\textquoteright\textquoteright} \land k > 0 \land \psi[h(s)] \land \text{Acc}_{A}(\text{Acc}(s, k - 1))) \]
\[ \text{Acc}(s, k) \lor (s \neq \text{\textquoteleft\textquoteleft} n\text{\textquoteright\textquoteright} \land k > 0 \land \psi[h(s)] \land \text{Acc}_{A}(\text{Acc}(s, k - 1))) \]

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 $s_{M} = \text{\textquoteleft\textquoteleft} a\text{\textquoteright\textquoteright}$ and $k_{M} = 1$. \qed

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 $\psi$ of the solver, if $\psi$ contains a sub-term $u$ and there exists a substitution $\theta$ such that $u = t_{0_{A}}\theta$, 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 $t_{0_{A}}\theta$.

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 SPCA $C$ 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 determine 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 $\mathcal{F}_{C}$, and distinct Boolean constants $\vec{b} = (b_{i})_{i<n}$ define
\[ \text{Cube}(\vec{\varphi}, \vec{b}) = \bigwedge_{i<n} \varphi_{i} \iff b_{i} \]

Recall that the variable $\chi$ is shared in all the $\varphi_{i}$. A solution of $\text{Cube}(\vec{\varphi}, \vec{b})$ is a model $M$ such that $M \models \exists \chi \text{Cube}(\vec{\varphi}, \vec{b})$. 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 $\text{Cube}(\vec{\varphi}, \vec{b})$ then $\bigwedge_{i<n} \{ \varphi_{i} \mid i < n, M \models b_{i} \}$ is satisfiable.

Given a solution $M$ of $\text{Cube}(\vec{\varphi}, \vec{b})$, let $\varphi_{M}$ denote the formula
\[ \bigwedge_{i<n} \{ b_{i} \mid M \models b_{i} \} \cup \{ \neg b_{i} \mid M \models \neg b_{i} \} \]

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

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

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 $M \in \text{Solutions}(\text{Cube}(\vec{\varphi}, \vec{b}))$ such that $M \models b_{i}$ for all $\varphi_{i} \in G$.

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

Definition 8: An SFA $A$ is total if for all $q \in Q_A$, the formula $\forall t \in \Delta_A(q) \exists \psi(q, t, d)$. 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 $(d, \text{true}, d)$, and a new move $(q, \varphi, d)$ from each state $q$ where $\varphi$ is satisfiable and $\varphi = \bigwedge \neg \Delta(q)$. Clearly, determinism is preserved by this transformation.

Definition 9: Given a total DSFA $A$, the complement $\overline{A}$ of $A$ is the DSFA $(Q_A, q_0A, Q_A \setminus F_A, \Delta_A)$.

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 adding the move $(d, \text{true}, d)$ to it with the move $(d, \text{true}, d)$, and a new move $(q, \varphi, d)$ from each state $q$ where $\varphi$ is satisfiable and $\varphi = \bigwedge \neg \Delta(q)$. Clearly, determinism is preserved by this transformation.

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

\[
\begin{align*}
&\{qA \cup \{q\}, q, \{qA\}, \\
&\{(\text{Target}(t), \text{Cond}(t), \text{Source}(t)) | t \in \Delta_A\} \\
&\cup \{(q, \epsilon, p) | p \in F_A\}\}
\end{align*}
\]

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

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

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 $A'$ is deterministic. A classical example is the SFA for the regex $\{ab\}^\star a\{ab\}^\star (n)$ 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 $A'$ 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 $B'$ is deterministic let $C = (A' \times \overline{B'})^\dagger$.

General case. We describe the algorithm as a depth-first-exploitation 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

\[
\begin{align*}
\overline{q}_{0A} &= \{q_{0A}\}, \{q_{0B}\}, \{q_{0C}\}, z_{0A}, z_{0C}, z_{0A}, 0\}, T = \emptyset,
\end{align*}
\]

(i) If $S$ is empty go to (iv) else pop $(p, q, z)$ from $S$.

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

\[
M = \text{Solutions}(\text{Cube}((\varphi_i)_{i < m}, (\psi_i)_{i < n}, \overline{a} \cdot \overline{b}))
\]

with the additional constraint that $\overline{a}$ true. For each move $(p, z, \varphi_i, p_i, z_i)$ of $A$ and for each solution $M$ in $M$ such that $M \models a_i$ do the following. Let $\gamma = \varphi_i \land \bigwedge \neg \psi_j \land (\neg \psi_j) \land (M \models b_j)$.

Add the move $(\langle p, q \rangle, z, \varphi_i, (p_i, q_i'), z_i')$ to $T$. Foreach $z' \in z_i$, if $w = \langle p_i, q_i', z' \rangle \notin V$ 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 $F$:

\[
F = \{\langle p, q \rangle | \langle p, q \rangle \in \pi_1(V), p \in F_A, q \in F_B = \emptyset\}
\]

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

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 $M$, where each solution will make at least one move, 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

\[
t_1 = (q_i \bigwedge \bigwedge \neg \psi_j \land (M_1 \models b_j))
\]

\[
t_2 = (q_i \bigwedge \bigwedge \neg \psi_j \land (M_2 \models b_j))
\]

that are composed with moves of $A$ and added to $T$ in Step (ii), where $M_1, M_2 \in M$. By using Proposition 4 we need to show that $\text{Target}(t_1) \neq \text{Target}(t_2)$ (i.e., for some $b_j, M_1 \models b_j$ and $M_2 \models \neg b_j$), then $\text{Cond}(t_1) \land \text{Cond}(t_2)$ is unsatisfiable, which holds because there is at least one $\psi_j$ such that $\psi_j$ is a conjunct of $\text{Cond}(t_1)$ and $\neg \psi_j$ is a conjunct of $\text{Cond}(t_2)$.

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 4 and the added constraint that $\overline{a}$ true in the computation of $M$.

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 IV and the axiom generation discussed in Section V 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 automata 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
\texttt{Solutions(Cube(\mathcal{Z},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.

...  
\texttt{z3.Push(); //push a new context for collecting solutions}
\texttt{Term[]} b = ... //fresh Boolean constants for B-moves
\texttt{Term[]} a = ... //fresh Boolean constants for A-moves
\texttt{Term[]} cube = ... //corresponding cube equations
\texttt{z3.AssertCnstr(z3.MkOr(a)); //at least one a[i] must hold}
\texttt{z3.AssertCnstr(z3.MkAnd(cube)); //assert the cube formula}
\texttt{z3.AssertCnstr(z3.MkOr(a)); //at least one a[i] must hold}
\texttt{Model M;}
\texttt{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].
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 \(L(r_i) \nsubseteq L(r_j)\) for regexes \(i\) and \(j, i \neq j\).

A. Experiment 1

Table [III] 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 automaton.
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
states (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 \(L(r_i) \subseteq L(r_j)\) 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 combi-
nation 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 \(Acc^A(s,k) \land \neg Acc^B(s,k)\) with fresh \(s: \mathbb{W}\)
and \(k: \mathbb{N}\).

3) Check and generate a model for the goal.

<table>
<thead>
<tr>
<th>(r_i)</th>
<th>(r_j)</th>
<th>(t)</th>
<th>#states</th>
<th>#moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>#2</td>
<td>0.19</td>
<td>74</td>
<td>124</td>
</tr>
<tr>
<td>#3</td>
<td>#4</td>
<td>0.75</td>
<td>68</td>
<td>156</td>
</tr>
<tr>
<td>#5</td>
<td>#6</td>
<td>0.51</td>
<td>61</td>
<td>31</td>
</tr>
<tr>
<td>#7</td>
<td>#8</td>
<td>0.35</td>
<td>40</td>
<td>141</td>
</tr>
<tr>
<td>#9</td>
<td>#10</td>
<td>0.20</td>
<td>210</td>
<td>249</td>
</tr>
</tbody>
</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

\footnote{All experiments were executed on a Lenovo T61 laptop with Intel dual
core T7500 2.2GHz processor.}
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 ".\{n\}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

\[ s \text{ LIKE } r_1 \text{ AND NOT } s \text{ LIKE } r_2 \text{ AND } 3 * \text{LEN}(s) > 2 * n \]  

Let $A$ and $B$ be $\varepsilon$-SFAs 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 $B^c$ (accepting $s$\text{ NOT LIKE } r_2$) in the algorithm is deterministic. We can systematically increase $n$ and compare the performance of a direct encoding of (6):

\[ Acc^A(s, k) \land \neg Acc^B(s, k) \land 3k > 2n, \]

against the difference encoding of (6):

\[ Acc^{A\Delta B}(s, k) \land 3k > 2n, \]

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 trendline is polynomial $O(n^2)$ 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
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:

\[
\begin{align*}
\text{var } x : l; \text{ reg a := } R_i; \text{ reg b := } R_j; \\
\text{assert x in a; assert x not in b;}
\end{align*}
\]

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 $l$. Although Hampi supports length ranges \texttt{var x : } \texttt{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 total 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 cSFA algorithms and their effect on language acceptors for cSFAs (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 cSPDAs. Theorem 1 strengthens a similar statement for cSFAs 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 VI would not have been possible without the help of Pieter Hooymeijer who set up the whole environment for the experiments and provided scripts for converting the regex expressions to Hampi format.

REFERENCES


APPENDIX

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

\[ \bigwedge_{i=1}^{n} s_i \in r_i \wedge F([s_1, \ldots, s_n]) \]

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

\[ \bigwedge_{i=1}^{n} s_i \in r_i \wedge F([s_1, \ldots, s_n]) \wedge \bigwedge_i |s_i| \leq |r_i| \cdot 2^{|s|} \]

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

Let \( \mathcal{S} \) 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, \( \perp \), 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 Piskau 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 \( |J| \leq 2^{|s|} \) 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:

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