Symbolic Automata Constraint Solving

Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura

Microsoft Research Redmond, WA, USA
{margus,nbjorner,leonardo}@microsoft.com

Abstract. Constraints over regular and context-free languages are common in the context of string-manipulating programs. Efficient solving of such constraints, often in combination with arithmetic and other theories, has many useful applications in program analysis and testing. We introduce and evaluate a method for symbolically expressing and solving constraints over automata, including subset constraints. Our method uses techniques present in the state-of-the-art SMT solver Z3.

1 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 [13], and database query analysis [19], where the application is to synthesize data-base tables from SQL queries. In the latter case, *like-patterns* are special kinds of regular expressions that are common in SQL queries, consider e.g.:

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

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 [19] 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 `SELECT` 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 succinct way to represent automata.

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

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

C. Fermüller and A. Voronkov (Eds.): LPAR-17, LNCS 6397, pp. 640–654, 2010.

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 [13,19], that build on the SMT technology, a fundamental aspect is that new theories can be added seamlessly and work in combination with existing theories.

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

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

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*:

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

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*. In contrast to (2), the encoding in (4) provides some benefits for the integration with SMT solving, since it can detect emptiness during the incremental difference construction. Independently, difference checking provides a way to check *subset* constraints, that have other useful applications [9].

Combination of regular constraints on strings with quantifier free linear arithmetic and length constraints is known to be decidable [22,23]. One can effectively compute an upper bound on the length of all strings, see [17]. 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 B. 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 without the need to provide fixed length bounds for the words. We evaluate the performance of the different approaches we implemented in a prototype tool and provide a comparison with the HAMPI tool [11].

2 Preliminaries

In the following, we assume familiarity with classical automata theory [10], logic and model theory [8]. 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. *Characters* are represented by n-bit-vectors of fixed length n, such that $n = 7$ (8) bits encode standard (extended) ASCII characters, and $n = 16$ encode Unicode characters. With n clear from the context, we write $\mathbb{C}$ for the character sort. The complete alphabet is $\mathcal{U}^{\mathbb{C}}$. Constant characters are for example written as `'a'`.

There is a *built-in* (predefined) *signature* of function symbols and a built-in theory (set of axioms) for those symbols. Each function symbol f of arity $n \geq 0$ has a given domain sort $\sigma_0 \times \cdots \times \sigma_{n-1}$, 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 Boolean function $< : \mathbb{C} \times \mathbb{C} \to \mathbb{B}$ 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. Well-sorted *terms* and *formulas* (Boolean terms) are defined as usual. 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 $\varphi \in \mathcal{F}_{\mathbb{C}}$, we write $[\![\varphi]\!]$ for the set of all $a \in \mathcal{U}^{\mathbb{C}}$ such that $\models \varphi[a]$.

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.

3 Symbolic Automata

We use a representation of automata where several transitions from a source state to a target state are combined into a single symbolic move. Symbolic moves are labeled by formulas from $\mathcal{F}_{\mathbb{C}}$ that represent *sets of characters* rather than individual characters. This representation has the advantage of being more succinct for symbolic analysis than an *explicit* representation. The following definition builds directly on the standard definition of PDAs.

Definition 1. A *Symbolic Push Down Automaton* or *SPDA* A is a tuple $(Q, \varphi_\Sigma, Z, \Delta, q_0, z_0, F)$, where Q is a finite set of *states*, φ_Σ is a formula in $\mathcal{F}_\mathbb{C}$ called *input predicate*, 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*.

An SPDA A denotes the PDA $[\![A]\!]$ whose input alphabet is $[\![\varphi_\Sigma]\!]$ and $[\![A]\!]$ has a transition $(q, z, a, p, \boldsymbol{z})$ for each $(q, z, \varphi, p, \boldsymbol{z}) \in \Delta$ and $a \in [\![\varphi \wedge \varphi_\Sigma]\!]$. The remaining components map directly to the corresponding components of a PDA. When φ_Σ is *true*, i.e., when the input alphabet is $U^\mathbb{C}$, we omit φ_Σ from the definition.

An ϵSPDA A may in addition have moves where the condition is $\epsilon \notin \mathcal{F}_\mathbb{C}$, denoting the corresponding ϵ-move in $[\![A]\!]$. Let $\rho = (q, z, \alpha, p, \boldsymbol{z})$ be a move of an ϵSPDA A.We define $Src(\rho) \stackrel{\text{def}}{=} q$, $Tgt(\rho) \stackrel{\text{def}}{=} p$, $Cnd(\rho) \stackrel{\text{def}}{=} \alpha$, $Pop(\rho) \stackrel{\text{def}}{=} z$, and $Push(\rho) \stackrel{\text{def}}{=} \boldsymbol{z}$. We also use the following notations

$$\Delta_A(q) \stackrel{\text{def}}{=} \{\rho \mid \rho \in \Delta_A, Src(\rho) = q\}$$
$$\Delta_A(q, z) \stackrel{\text{def}}{=} \{\rho \mid \rho \in \Delta_A(q), Pop(\rho) = z\}$$

and furthermore will allow lifting functions to sets. For example, $\Delta_A(Q) \stackrel{\text{def}}{=} \cup\{\Delta_A(q) \mid q \in Q\}$. We write Δ^ϵ_A for the set of all epsilon moves in Δ_A and $\Delta^{\not\epsilon}_A$ for $\Delta_A \setminus \Delta^\epsilon_A$.

An ϵSPDA A is *clean* if all moves in $\Delta^{\not\epsilon}_A$ have satisfiable conditions, and *normalized* if there are no two moves in $\Delta^{\not\epsilon}_A$ that differ only with respect to their condition.

The language $L(A)$ accepted by A is the language $L([\![A]\!])$ accepted by the PDA $[\![A]\!]$.

It is clear that for any ϵSPDA A there is a normalized ϵSPDA A' such that $[\![A]\!] = [\![A']\!]$: just combine all nonepsilon moves that only differ with respect to their conditions into a single move by making a disjunction of their conditions.

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.) In the prototype tool we use a variation of the Blum-Koch algorithm [3] for GNF transformation that has worst case time complexity $O(n^4)$.

Definition 2. An ϵSPDA A *represents a symbolic finite automaton with epsilon moves* or *ϵSFA* if for all $\rho \in \Delta_A$, $Pop(\rho) = Push(\rho) = z_{0A}$.

When considering an ϵSPDA A that represents an ϵSFA, we omit the stack symbols and denote A by the tuple $(Q_A, \varphi_{\Sigma A}, \Delta_A, q_{0A}, F_A)$. We use [14] as the concrete language definition of regular expressions or *regexes* in this paper. The translation from a regex to an ϵSFA follows very closely the standard algorithm, see e.g., [10, Section 2.5], for converting a standard regular expression into a

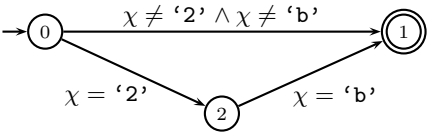

Fig. 1. Sample ϵSFA generated from the regex `^(2b|[^2b])$`

finite automaton with epsilon moves. A sample regex and corresponding ϵSFA are illustrated in Figure 1.

When we say SPDA or SFA we assume that epsilon moves are not present.

4 Symbolic Language Acceptors

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

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

where $FV(t_{\text{lhs}}) = \bar{x}$ and $FV(t_{\text{rhs}}) \subseteq \bar{x}$. When t_{lhs} and t_{rhs} are formulas, we often write '$\Leftrightarrow$' instead of '='. The left-hand-side t_{lhs} of (5) is called the *head* of (5) and the right-hand-side t_{rhs} of (5) is called the *body* of (5)

Many SMT solvers support various kinds of patterns for triggering axioms. Yet in this paper, we use the convention that the pattern of an equational axiom is always its head.

The same convention is used in [19]. Axioms are asserted as equations that are expanded during proof search. Expanding the formula up front is problematic since the equational axioms introduced for automata are in general mutually recursive (as shown below) and a naive a priori exhaustive expansion would in most cases not terminate. Straight-forward depth-bounded expansions are also not practical as the size of the bounded expansion is easily exponential in the depth.

The overall idea behind the axioms introduced below is as follows. For a given ϵSPDA A we construct a theory $Th(A)$ that includes a particular axiom with head $Acc^A(w,k)$. The main property of $Th(A)$ is that it precisely characterizes the language accepted by A as the set of solutions w and k for $Th(A) \wedge Acc^A(w,k)$, where k is the length of w.

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 ε (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 $[e|l]$ is a list of sort $\mathbb{L}\langle\sigma\rangle$. The accessors are, as usual, *hd* (head) and *tl* (tail). *Words* or *strings* are represented by lists of characters; we write $\mathbb{W}$ for the sort $\mathbb{L}\langle\mathbb{C}\rangle$. We adopt the common convention that $[a,b,c]$ stands for the list $[a|[b|[c|\varepsilon]]]$ and we use $l_1 \cdot l_2$ for concatenation of l_1 and l_2.

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 $Z_A \subseteq \mathcal{U}^{\mathbb{Z}}$. By slight abuse of notation, we also use elements in $\mathcal{U}$ 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 $\mathbb{S} = \mathbb{L}\langle\mathbb{Z}\rangle$.

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

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

Recall that an *ID* of $[\![A]\!]$ is a triple (q, w, s) where $q \in Q_{[\![A]\!]}$, w is a word and s is a stack [10]. 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 [10, page 112]. The transformation of A to an equivalent ϵSPDA A' such that $L([\![A]\!]) = N([\![A']\!])$ is straightforward. We therefore assume below that $F_A = \emptyset$.

The idea behind the axioms defined below is that the formula $Acc_q^A(w, n, s)$ holds iff $|w| = n$ and $(q, w, s) \vdash_{[\![A]\!]}^* (p, \varepsilon, \varepsilon)$ for some $p \in Q_{[\![A]\!]}$, where $\vdash_{[\![A]\!]}$ is the step relation of $[\![A]\!]$ as defined in [10, page 112]. We write $\vdash_A$ for $\vdash_{[\![A]\!]}$. 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 $w \in L(A)$.

Definition 3. Fix $q \in Q_A$ and $z \in Z_A$. Assume $\Delta_A(q, z)$ is

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

Define

$$\begin{aligned}
ax^A &\stackrel{\text{def}}{=} \forall w\, n\, (Acc^A(w, n) \Leftrightarrow Acc^A_{q_{0A}}(w, n, [z_{0A}])) \\
ax^A_q &\stackrel{\text{def}}{=} \forall w\, n\, (Acc^A_q(w, n, \varepsilon) \Leftrightarrow (w = \varepsilon \wedge n = 0)) \\
ax^A_{q,z} &\stackrel{\text{def}}{=} \forall w\, n\, s\, (Acc^A_q(w, n, [z|s]) \Leftrightarrow \\
&\qquad ((w \neq \varepsilon \wedge n > 0 \wedge (\bigvee_{i=1}^{m} (\varphi_i[hd(w)] \wedge Acc^A_{q_i}(tl(w), n{-}1, \boldsymbol{z}_i \cdot s)))) \\
&\qquad \vee \bigvee_{j=m+1}^{k} Acc^A_{q_j}(w, n, \boldsymbol{z}_j \cdot s))) \qquad (6) \\
Th(A) &\stackrel{\text{def}}{=} \{ax^A\} \cup \{ax^A_q, ax^A_{q,z} \mid q \in Q_A, z \in Z_A\}.
\end{aligned}$$

Definition 4. An *ϵ-loop* of A is a derivation $(q, w, s) \vdash_A^+ (q, w, s')$ s.t. $|s| \le |s'|$.

Intuitively, an ϵ-loop is a derivation that does not consume any characters from the input word and starts and ends in the same state for some stacks that do not decrease in size. Note that an ϵ-loop can only involve ϵ-moves, since any

nonepsilon move decreases the length of the input word by one. Define the binary relation $\models^{\epsilon}_{A}$: $((Q_A \times Z_A^*) \times (Q_A \times Z_A^*))$ called the *ϵ-step relation of A* as:

$$(q_1, s_1) \models^{\epsilon}_{A} (q_2, s_2) \stackrel{\text{def}}{=} (q_1, \varepsilon, s_1) \vdash_A (q_2, \varepsilon, s_2)$$

Lemma 1. *If A has no ϵ-loops then $\models^{\epsilon}_{A}$ is wellfounded.*

Proof. Assume A has no ϵ-loops and suppose, by contradiction, that there is an infinite chain $((q_i, s_i) \models^{\epsilon}_{A} (q_{i+1}, s_{i+1}))_{i<\omega}$. Since Q_A is finite, there is a fixed q and an infinite subset $I \subseteq \omega$ such that $q = q_i$ for $i \in I$ and $(q, \varepsilon, s_i) \vdash^{+}_{A} (q, \varepsilon, s_j)$ for $i, j \in I$ such that $i < j$. Since Z_A is finite and I is infinite, it follows that $|s_i| \leq |s_j|$ for some $i, j \in I$ where $i < j$, contradicting ϵ-loop-freeness of A. ⊠

Theorem 1. *Assume that $\models^{\epsilon}_{A}$ is wellfounded. For all $w \in \mathcal{U}^{\mathbb{W}}$ and $n \in \mathcal{U}^{\mathbb{N}}$:*

$$Th(A) \models Acc^A(w, n) \iff w \in L(A) \text{ and } |w| = n.$$

Proof. By using that $\models^{\epsilon}_{A}$ is wellfounded, define $(w_1, (q_1, s_1)) \succ (w_2, (q_2, s_2))$ as the *lexicographic order*: $|w_1| > |w_2|$ or, $|w_1| = |w_2|$ and $(q_1, s_1) \models^{\epsilon}_{A} (q_2, s_2)$.

For each axiom $ax_{q,z}$ in $Th(A)$ we show that each occurrence of Acc_p^A in the body of $ax_{q,z}$ is smaller wrt $\succ$ than the head of $ax_{q,z}$. For the cases when $w \neq \varepsilon$ we have that

$$(w, (q, [z|s])) \succ (tl(w), (q_i, \boldsymbol{z}_i \cdot s))$$

by using that $|w| > |tl(w)|$ according to the built-in theory of lists. For the case of the epsilon moves in (6) we have that

$$(w, (q, [z|s])) \succ (w, (q_j, \boldsymbol{z}_j \cdot s))$$

since $(q, \varepsilon, [z|s]) \vdash_A (q_j, \varepsilon, \boldsymbol{z}_j \cdot s)$ and thus $(q, [z|s]) \models^{\epsilon}_{A} (q_j, \boldsymbol{z}_j \cdot s)$.

It follows that the set of axioms is mathematically well-defined. We can now prove, by induction over the length of w that the following statement holds, which is also directly evident from the definitions. For all IDs (q, w, s) of $[\![A]\!]$:

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

Finally, let $q = q_{0A}$, $s = [z_{0A}]$ and use axiom ax^A. ⊠

In general, presence of ϵ-loops may imply that $\models^{\epsilon}_{A}$ is not wellfounded and the theorem fails, as illustrated by the following example. Moreover, for ϵSFAs, ϵ-loop-freeness is *equivalent* to $\models^{\epsilon}_{A}$ being wellfounded.

Example 1. Let $A = (\{q\}, \{z\}, q, z, \emptyset, \{(q, z, \epsilon, q, (z))\})$. For example $(q, \varepsilon, (z)) \vdash_A (q, \varepsilon, (z))$. The language accepted by A is empty. The theory $Th(A)$ for A includes the axiom $ax^A_{q,z}$: $\forall w\, n\, s\, (Acc^A_q(w, n, [z|s]) \Leftrightarrow Acc^A_q(w, n, [z|s]))$. This axiom is a useless tautology. Consider for example a model M with an interpretation for Acc^A_q such that $M \models Acc^A_q(\varepsilon, 0, (z))$ and expand M so that $M \models ax^A \wedge ax^A_q$. Then $M \models Acc^A(\varepsilon, 0)$ but $\varepsilon \notin L(A)$. ⊠

For ϵSFAs full epsilon elimination may cause quadratic increase in the number of moves, although the number of states may decrease. For ϵSPDAs the increase is even higher (although still polynomial) by using GNF transformation. For symbolic analysis this may create more complex axioms than needed and may reduce the performance considerably [18]. For ϵSFAs A we implemented *ϵ-loop-elimination* by using the following construction, that does not increse the number of moves. Recall the definition of *ϵ-closure*, denoted here by $\epsilon(q)$, as the closure of $\{q\}$ by ϵ-moves [10]. Similarly, define $\ni(q)$ as the closure of $\{q\}$ by ϵ-moves in reverse. Let $\widetilde{q} \stackrel{\text{def}}{=} \epsilon(q) \cap \ni(q)$ (note that $\{q\} \subseteq \widetilde{q}$) and let

$$\widetilde{A} \stackrel{\text{def}}{=} (\{\widetilde{q} \mid q \in Q_A\}, \varphi_{\Sigma A}, \{(\widetilde{q}, \varphi, \widetilde{p}) \mid (q, \varphi, p) \in \Delta_A\}, \widetilde{q_{0A}}, \{\widetilde{q} \mid q \in F_A\})$$

It is straightforward to show that $\widetilde{A}$ is ϵ-loop-free and equivalent to A. For ϵSPDAs we have not investigated specialized algorithms for ϵ-loop-elimination and resort to extended GNF (EGNF) transformation [3] that, translated into ϵSPDAs, allows some ϵ-moves but eliminates ϵ-loops.[1]

5 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 $\boldsymbol{\varphi} = (\varphi_i)_{i<n}$ from $\mathcal{F}_{\mathbb{C}}$, and distinct Boolean constants $\boldsymbol{b} = (b_i)_{i<n}$ define

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

Recall that the variable χ is shared in all the φ_i. A *solution* of $\mathit{Cube}(\boldsymbol{\varphi}, \boldsymbol{b})$ is a model M such that $M \models \exists \chi \mathit{Cube}(\boldsymbol{\varphi}, \boldsymbol{b})$. 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 1. *If M is a solution of $\mathit{Cube}(\boldsymbol{\varphi}, \boldsymbol{b})$ then $\bigwedge\{\varphi_i \mid i < n, M \models b_i\}$ is satisfiable.*

[1] Absence of ϵ-loops does not directly follow from the definition of EGNF that allows grammar productions of the form $A \to B$ where A and B are nonterminals, thus $A \to B$ and $B \to A$ would be allowed simultaneously. But the algorithm in [3] for converting a CFG to EGNF will not generate such circular productions.

Given a solution M of $Cube(\boldsymbol{\varphi}, \boldsymbol{b})$, 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 $Solutions(Cube(\boldsymbol{\varphi}, \boldsymbol{b}))$ of all solutions of $Cube(\boldsymbol{\varphi}, \boldsymbol{b})$.

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

The procedure is still exponential (in n) in the *worst case*, but seems to work well in practice. It is also better than creating all subsets of $\boldsymbol{\varphi}$ 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 2. *Let $\boldsymbol{\varphi}$ and $\boldsymbol{b}$ be as above. For all subsets G of $\boldsymbol{\varphi}$, $\bigwedge G$ is satisfiable if and only if there exists $M \in Solutions(Cube(\boldsymbol{\varphi}, \boldsymbol{b}))$ such that $M \models b_i$ for all $\varphi_i \in G$.*

An SFA A is *total* if for all $q \in Q_A$, the formula $\forall \chi \bigvee\{Cnd(t) \mid t \in \Delta_A(q)\}$ 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 $\varphi = \bigwedge\{\neg Cnd(t) \mid t \in \Delta(q)\}$. Clearly, determinism is preserved by this transformation. An SFA A is *determinisitic* if $[\![A]\!]$ is deterministic. Note that, it is easy to show that A is deterministic iff for all (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. Given a total deterministic SFA A, the *complement* $\overline{A}$ of A is the deterministic SFA $(Q_A, q_{0A}, Q_A \setminus F_A, \Delta_A)$.

It is easy to see that for a total deterministic SFA 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. 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

$$(Q_A \cup \{q\}, q, \{q_{0A}\}, \{(Tgt(t), Cnd(t), Src(t)) \mid t \in \Delta_A\} \cup \{(q, \epsilon, p) \mid p \in F_A\})$$

Given a word s let s^r denote the word 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 property $\overline{L(A)} = L(\overline{A^r})^r$.

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

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

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

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

Add the move $(\langle p, \mathbf{q}\rangle, z, \gamma, \langle p_i, \mathbf{q}'\rangle, \boldsymbol{z}_i)$ to T. For each $z' \in \boldsymbol{z}_i$ define $v = \langle\langle p_i, \mathbf{q}'\rangle, z'\rangle$, if $v \notin V$ then add v to V and if there exists $\rho \in \Delta_A$ such that $\mathit{Src}(\rho) = p'$ and $\mathit{Pop}(\rho) = z'$ then push v to S.

(iii) Go to (i).

(iv) Compute the set of final states $F = \{\langle p, \mathbf{q}\rangle \mid \langle p, \mathbf{q}\rangle \in \pi_1(V), p \in F_A, \mathbf{q} \cap F_B = \emptyset\}$. 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

$$\rho_1 = (\mathbf{q}, \bigwedge_{M_1 \models b_j} \psi_j \wedge \bigwedge_{M_1 \models \neg b_j} \neg\psi_j, \{q_j \mid M_1 \models b_j\})$$

$$\rho_2 = (\mathbf{q}, \bigwedge_{M_2 \models b_j} \psi_j \wedge \bigwedge_{M_2 \models \neg b_j} \neg\psi_j, \{q_j \mid M_2 \models b_j\})$$

that are composed with moves of A and added to T in Step (ii), where $M_1, M_2 \in \mathbf{M}$. We need to show that if $\mathit{Tgt}(\rho_1) \neq \mathit{Tgt}(\rho_2)$ (i.e., for some b_j, $M_1 \models b_j$ and

$M_2 \models \neg b_j$), then $Cnd(\rho_1) \wedge Cnd(\rho_2)$ is unsatisfiable, which holds because there is at least one ψ_j such that ψ_j is a conjunct of $Cnd(\rho_1)$ and $\neg\psi_j$ is a conjunct of $Cnd(\rho_2)$.

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 2 and the added constraint that $\bigvee \boldsymbol{a}$ is true in the computation of $\mathbf{M}$.

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

5.1 Difference Checking

The difference algorithm has a more efficient version in the case when A above also represents an SFA and the purpose is to find a *single* witness in $L(A)\setminus L(B)$. In this case the explicit construction of $L(A)\setminus L(B)$ is not needed since $Th(L(A)\setminus L(B))$ 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. This algorithm has the same complexity as the full construction in the general case, but may finish sooner when $L(A) \setminus L(B)$ is nonempty.

6 Implementation

The algorithms and the axiom generation discussed above, have been implemented in a prototype tool for analyzing regular expressions and context free grammars. The SMT solver Z3 [6] 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 4. The definitions of the axioms match very closely with the 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 below

```
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
```

Table 1. Sample regexes

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

Table 2. Experiments. For $1 \leq i \leq 10$, A_i is a ϵSFA for the regex in row i in Table 1. *Time* is $\sum_{1 \leq i,j \leq 10, i \neq j} t_{i,j}$, where $t_{i,j}$ is the time to generate a member $x \in L(A_i) \backslash L(A_j)$ where $i \neq j$.

Experiment	*Time*	*Formulas checked by Z3*
direct encoding	15s	$Th(A_i) \wedge Th(A_j) \wedge Acc^{A_i}(x,k) \wedge \neg Acc^{A_j}(x,k)$
difference algorithm	60s	$Th(A_i \setminus A_j) \wedge Acc^{A_i \setminus A_j}(x,k)$
difference checking	10s	No axioms for the automata are asserted, but Z3 is used for solving cube formulas.

and shows a simplified code snippet from the tool responsible for computing $Solutions(Cube(\boldsymbol{\varphi}, \boldsymbol{b}))$ in the difference construction algorithm in Section 5, where the solutions are generated incrementally using a context, and the *model generation* feature is used to extract solutions from Z3.

7 Evaluation

We conducted several experiments where we evaluated the performance of the difference algorithm and the axiomatization approach.[2] The following experiments were run on a laptop with an Intel dual core T7500 2.2GHz processor. We used a collection of 10 complex regexes r_i extracted from a case study in [13] that are representative for various practical usages. Table 1 shows the samples. For several r_i the corresponding automaton A_i has thousands of states. In all cases, $A_i \setminus A_j$ for $i \neq j$ is nonempty. There are a total of 90 such combinations.

Experiments are shown in Table 2. The table does not reflect experiments where the set difference is empty (i.e. for the case $i = j$ but assuming that A_j is made slighltly different from A_i in $A_i \setminus A_j$ so that the theories are not identical but accept the same words): in this case the *direct* encoding diverges when A_i encodes an infinite language. In contrast, the *difference* constructions remain

[2] More details are available in [17].

robust, i.e., the construction of $A \setminus B$ terminates with the empty automaton when $A \setminus B$ is empty.

To our knowledge, a system that comes closest to the scope of ours is the open source string constraint solver Hampi [11]. We conducted a similar experiment using Hampi. Given the regexes r_i in Table 1, Hampi's 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 [7] 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.

8 Related Work

The work presented here is a nontrivial extension of the work started in [18] where different ϵSFA algorithms and their effect on language acceptors for ϵSFAs (including minimization and determinization) are studied. The experiments in [18] failed in determinization, which needed the idea of solving *cube* formulas. Moreover, the approach of language acceptors presented in [18] does not support precise length constraints, and the axioms were not studied for ϵSPDAs. Theorem 1 generalizes a similar statement for ϵSFAs in [18].

Although, an extension of FAs with predicates has been suggested earlier [21], 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.

The Hampi [11] tool, that is a string constraint solver, supports encoding of difference constraints $L(R_1) \setminus L(R_2)$ between regular expressions R_1 and R_2, where R_1 can be obtained as a finitization of a context free grammar. Unlike in our case, Hampi turns string constraints over fixed-size string variables into a query to STP [7]. STP is a solver for bit-vectors and arrays. The input size needs to be fixed, since STP does not support axioms or algebraic data types, and potential combination with other theories, e.g. linear arithmetic, is not in the scope of STP.

A decision procedure for subset constraints over regular language variables is introduced in [9] by reasoning over dependency graphs. In contrast, we showed how finite push-down automata can be generalized by making transitions symbolic, and how a decision procedure can be embedded into a background theory of an SMT solver.

Several decision problems related to CFGs are studied in [2] 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 in a given grammar) of CFGs, that provides an advantage over an algorithm in [15], by providing a *witness* in case of ambiguity. Using SMT and the approach presented here, an interesting direction for future work is to study extensions of symbolic acceptors based on *grammars* that capture *parse trees*, which is also related to symbolic acceptors for tree-automata. A parse tree can be represented with an algebraic data-type based on the productions of the grammar. Potentially, this approach can be used for ambiguity checking and in addition to providing a witness, avoids the need to provide a priori depth-bounds.

Several program analysis techniques for programs with strings [22,5,16,20] build on automata libraries [12,1] that efficiently handle transitions over sets of characters as either BDDs [4] or interval constraints. Most of those approaches suffer from the separation of the decision procedures that are not tightly coupled. Constraints over strings are decided by one solver, while constraints over other domains are decided by other solvers, but the solvers usually cannot be combined in an efficient, sound or complete fashion. SMT solvers directly address this problem by combining decision procedures for a variety of theories. Particular examples from applications involving strings are: symbolic analysis of SQL queries [19] and analysis of .NET programs [13].

9 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 in combination with automata theoretic techniques. 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.

Acknowledgement. The Hampi comparison in Section 7 would not have been possible without the help of *Pieter Hooimeijer* who set up the whole environment for the experiment and provided scripts for converting the regexes in Table 1 to Hampi format.

References

1. BRICS finite state automata utilities, `http://www.brics.dk/automaton/`
2. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)

3. Blum, N., Koch, R.: Greibach Normal Form Transformation Revisited. Inf. Comput. 150(1), 112–118 (1999)
4. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: DAC 1990, pp. 40–45. ACM, New York (1990)
5. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. TACAS 2008, pp. 337–340. Springer, Heidelberg (2008)
7. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
8. Hodges, W.: Model theory. Cambridge Univ. Press, Cambridge (1995)
9. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: PLDI, pp. 188–198 (2009)
10. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
11. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: ISSTA 2009, pp. 105–116. ACM, New York (2009)
12. Klarlund, N.: Mona & Fido: The Logic-Automaton Connection in Practice. In: Nielsen, M. (ed.) CSL 1997. LNCS, vol. 1414, pp. 311–326. Springer, Heidelberg (1998)
13. Li, N., Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: Reggae: Automated test generation for programs using complex regular expressions. In: ASE 2009 (2009)
14. MSDN. .NET Framework Regular Expressions (2009), `http://msdn.microsoft.com/en-us/library/hs600312.aspx`
15. Schmitz, S.: Conservative ambiguity detection in context-free grammars. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 692–703. Springer, Heidelberg (2007)
16. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting Symbolic Execution with String Analysis. In: MUTATION 2007, pp. 13–22. IEEE, Los Alamitos (2007)
17. Veanes, M., Bjørner, N., de Moura, L.: Solving extended regular constraints symbolically. Technical Report MSR-TR-2009-177, Microsoft Research (2009)
18. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic Regular Expression Explorer. In: ICST 2010, IEEE, Los Alamitos (2010)
19. Veanes, M., Tillmann, N., de Halleux, J.: Qex: Symbolic SQL query explorer. In: LPAR-16. LNCS (LNAI). Springer, Heidelberg (2010)
20. Wassermann, G., Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. ACM TSEM 16(4), 14 (2007)
21. Watson, B.W.: chapter Implementing and using finite automata toolkits, pp. 19–36. Cambridge U. Press, Cambridge (1999)
22. Yu, F., Bultan, T., Cova, M., Ibarra, O.H.: Symbolic String Verification: An Automata-Based Approach. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 306–324. Springer, Heidelberg (2008)
23. Yu, F., Bultan, T., Ibarra, O.H.: Symbolic String Verification: Combining String Analysis and Size Analysis. In: Kowlaewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 322–336. Springer, Heidelberg (2009)