An Evaluation of Automata Algorithms for String Analysis*

Pieter Hooimeijer[1] and Margus Veanes[2]

[1] University of Virginia**
pieter@cs.virginia.edu

[2] Microsoft Research
margus@microsoft.com

Abstract. There has been significant recent interest in automated reasoning techniques, in particular constraint solvers, for string variables. These techniques support a wide range of clients, ranging from static analysis to automated testing. The majority of string constraint solvers rely on finite automata to support regular expression constraints. For these approaches, performance depends critically on fast automata operations such as intersection, complementation, and determinization. Existing work in this area has not yet provided conclusive results as to which core algorithms and data structures work best in practice.

In this paper, we study a comprehensive set of algorithms and data structures for performing fast automata operations. Our goal is to provide an apples-to-apples comparison between techniques that are used in current tools. To achieve this, we re-implemented a number of existing techniques. We use an established set of regular expressions benchmarks as an indicative workload. We also include several techniques that, to the best of our knowledge, have not yet been used for string constraint solving. Our results show that there is a substantial performance difference across techniques, which has implications for future tool design.

1 Introduction

There has been significant recent interest in decision procedures for string constraints. Traditionally, many analyses and automated testing approaches relied on their own built-in model to reason about string values [5,21,8,9,29,23]. Recent work on string analysis has focused on providing external decision procedures that reason about strings [11,15,24,2,28,27,12]. The separation of string decision procedures from their client analyses is desirable because it allows for the independent development of more scalable algorithms.

Existing string decision procedures vary significantly in terms of feature sets. Although most tools support regular expression constraints in one form

* Microsoft Research Technical Report MSR-TR-2010-90, July 2010, Updated August 2010.

** This work was done during an internship at Microsoft Research.

or another, there is no strong consensus on which features are necessary. Many programming languages provide a variety of string manipulating functions. For example, the user manual for PHP lists 111 distinct string manipulation functions [20], and the `System.Text` namespace in the .NET class library contains more than a dozen classes that deal with Unicode encodings [19]. Constraint solving tools must strike a balance between expressive utility and other concerns like scalability and generality.

In this paper, we focus on a subclass of string decision procedures that support regular expression constraints and use finite automata as their underlying representation. This includes the DRPLE [11], JSA [5], and Rex [28,27] tools, as well as a prototype from our recent work [12]. These tools are implemented in different languages, they parse different subsets of common regular expression idioms, they differ in how they use automata internally, and the automata data structures themselves are different. However, each approach relies crucially on the efficiency of basic automaton operations like intersection and determinization/complementation. Existing work provides reasonable information on the relative performance of the tools as a whole, but does not give any insight into the relative efficiency of the individual data structures.

Our goal is to provide an apples-to-apples comparison between the core algorithms that underlie these solvers. To achieve this, we have performed a faithful re-implementation in C# of several often-used automaton data structures. We also include several data structures that, to the best of our knowledge, have not yet featured in existing work on string decision procedures. We conduct performance experiments on an established set of string constraint benchmarks. We use both ASCII and UTF-16 encodings where possible, to investigate the impact of alphabet size on the various approaches. Our testing harness includes a relatively full-featured regular expression parser that is based on the .NET framework's built-in parser. By fixing factors like the implementation language and the front-end parser, and by including a relatively large set of regular expression features, we aim to provide practical insight into which data structures are advisable for future string decision procedure work.

This paper makes the following contributions:

- To the best of our knowledge, we provide the first performance comparison of automata data structures (and associated algorithms) designed for string decision procedures. We pay special attention to isolating the core operations of interest.
- We present several approaches that have not yet been used for string decision procedures. We demonstrate that these approaches frequently out-perform existing approaches.

The rest of this paper is structured as follows. Section 2 provides a brief example that highlights the utility of string decision procedures in the context of the Pex [26] unit testing framework. Section 3 gives formal definitions of the automata constructs of interest. Section 4 presents the design and implementation of the data structures that we implemented. In Section 5, we provide experi-

mental results using those data structures. We discuss closely related work in Section 6, and we conclude in Section 7.

2 Motivating Example

Before launching into the details of the automata operations of interest, we briefly illustrate the motivation that underlies automaton-based string decision procedures. To illustrate one concrete application scenario, consider the following C# method:

```
bool IsValidEmail(string s)
{
  string r1 = @"^[A-Za-z0-9]+@(([A-Za-z0-9\-])+\.)+([A-Za-z\-])+$";
  string r2 = @"^\d.*$";
  if (System.Text.RegularExpressions.Regex.IsMatch(s, r1))
    if (System.Text.RegularExpressions.Regex.IsMatch(s, r2))
      return false; //branch 1
    else
      return true;  //branch 2
  else
    return false;   //branch 3
}
```

In the context of parameterized unit testing with Pex [26] one objective is to generate values for the parameter s that provide branch coverage of the method. There are three return branches. Pex uses symbolic execution to generate path constraints and uses a constraint solver to solve those conditions for concrete parameter values. If any of those constraints include string operations (such as the use of `IsMatch` above), then we need a string decision procedure to solve those constraints. Some of the techniques discussed in this paper have been recently integrated into Pex. Note that characters in .NET use UTF-16 character encoding by default; it is challenging to make string decision procedures scale to such a large alphabet.

For this example, the path condition for branch 1 is $s \in L(r_1) \wedge s \in L(r_2)$ (e.g. $s =$ `"0@a.b"`), the path condition for branch 2 is $s \in L(r_1) \wedge s \notin L(r_2)$ (e.g. $s =$ `"a@b.c"`), and the path condition for branch 3 is $s \notin L(r_1)$ (e.g. $s =$ `"a@..b"`). Let A_1 be the SFA for r_1 and let A_2 be the SFA for r_2. Thus, solving the path conditions correspond to generating members in $L(A_1) \cap L(A_2)$, $L(A_1) \setminus L(A_2)$, and $\complement(L(A_1))$ (equivalently $\Sigma^* \setminus L(A_1)$), respectively. Note that if some path condition does not have a solution, this usually implies presence of dead-code, the techniques can thus also be used to enhance code reachability analysis.

3 Preliminaries

We assume familiarity with classical automata theory [13]. We work with a representation of automata where several transitions from a source state to a target

state may be combined into a single *symbolic move*. Symbolic moves have labels that denote *sets of characters* rather than individual characters. This representation naturally separates the structure of automata graph from the representation used for the character sets that annotate the edges. In this context, classical finite automata can be viewed as a special case in which each label is a singleton set. The following definition builds directly on the standard definition of finite automata.

Definition 1. A *Symbolic Finite Automaton* or *SFA* A is a tuple $(Q, \Sigma, \Delta, q^0, F)$, where Q is a finite set of *states*, Σ is the *input alphabet*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states* and $\Delta : Q \times 2^\Sigma \times Q$ is the *move relation*.

We indicate a component of an SFA A by using A as a subscript. The word *symbolic* stems from the intension that a move (q, ℓ, p) denotes the *set* of transitions $[\![(q, \ell, p)]\!] \stackrel{\text{def}}{=} \{(q, a, p) \mid a \in \ell\}$. We let $[\![\Delta]\!] \stackrel{\text{def}}{=} \cup\{[\![\rho]\!] \mid \rho \in \Delta\}$. An SFA A denotes the finite automaton $[\![A]\!] \stackrel{\text{def}}{=} (Q_A, \Sigma_A, [\![\Delta_A]\!], q^0_A, F_A)$. The *language* $L(A)$ *accepted by* A is the language $L([\![A]\!])$ accepted by $[\![A]\!]$.

An ϵSFA A may in addition have moves where the label is ϵ, denoting the corresponding epsilon move in $[\![A]\!]$. Given a move $\rho = (p, \ell, q)$, let $Src(\rho) \stackrel{\text{def}}{=} p$, $Tgt(\rho) \stackrel{\text{def}}{=} q$, $Lbl(\rho) \stackrel{\text{def}}{=} \ell$. We use the following notation for the set of moves starting from a given state q in A: $\Delta_A(q) \stackrel{\text{def}}{=} \{\rho \mid \rho \in \Delta_A, Src(\rho) = q\}$ 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 ϵSFA A is *clean* if for all $\rho \in \Delta^{\not\epsilon}_A$, $Lbl(\rho) \neq \emptyset$; A is *normalized* if for all $q \in Q_A$, if $\rho_1, \rho_2 \in \Delta^{\not\epsilon}_A(q)$ and $Tgt(\rho_1) = Tgt(\rho_2)$ then $\rho_1 = \rho_2$; A is *total* if for all states $q \in Q_A$, $\Sigma_A = \bigcup_{\rho \in \Delta_A(q)} Lbl(\rho)$. For making an SFA A total, do the following: for all $q \in Q_A$ such that $\ell = \complement(\bigcup_{\rho \in \Delta_A(q)} Lbl(\rho)) \neq \emptyset$, add (q, ℓ, sink) to Δ_A, and add the state sink to Q_A and the move $(\mathit{sink}, \Sigma, \mathit{sink})$ to Δ_A. Epsilon elimination from an ϵSFA is straightforward and makes use of union [28]. Also, any SFA can easily be made clean and normalized. When we say SFA we assume that epsilon moves are not present.

We assume a translation from regex (extended regular expression) patterns to ϵSFAs that follows closely the standard algorithm, see e.g., [13, Section 2.5]. A sample regex and corresponding ϵSFA are illustrated in Figure 1.

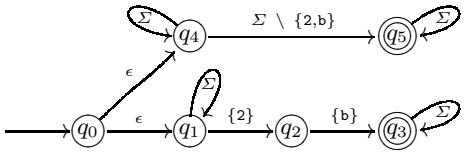

Fig. 1. ϵSFA generated from regex `2b|[^2b]`.

4 Automata Data structures and Algorithms

In this section, we describe the automaton data structures and algorithms of interest. We assume a graph-based data structure for the automata; each transition edge is annotated with a data structure that represents the label of that

transition. Section 4.1 describes a the data structures we use for those annotations. In Section 4.2, we discuss lazy and eager algorithms for two key operations on automata: language intersection (using the cross product construction) and language difference (using the subset construction). Later, in Section 5, we will evaluate experimentally how well each data structure/algorithm combination performs.

4.1 Representing character sets

We start by defining an interface for character set operations. This interface represents all the operations that the higher-level automata algorithms use to perform intersection and complementation. We then discuss several representations that can be used to implement this interface. A given representation affects both the performance of the algorithms discussed below; it also affects how effectively the algorithms can be combined with existing solvers.

Definition 2. A *minterm* of a finite nonempty sequence $(\ell_i)_{i\in I}$ of nonempty subsets of Σ is a sequence $(\ell'_i)_{i\in I}$ where each ℓ'_i is ℓ_i or $\complement(\ell_i)$ and $\bigcap_{i\in I} \ell'_i \neq \emptyset$, where $\bigcap_{i\in I} \ell'_i$ is the set *represented* by the minterm.

Intuitively, a minterm (ℓ'_1, ℓ'_2) of a sequence (ℓ_1, ℓ_2) of two nonempty subsets $\ell_1, \ell_2 \subseteq \Sigma$ represents a minimal nonempty region $\ell'_1 \cap \ell'_2$ of a *Venn diagram* for ℓ_1 and ℓ_2, see Figure 2. Note that the *collection of all minterms* of a given sequence of nonempty sets over Σ represents a *partition* of Σ.[3]

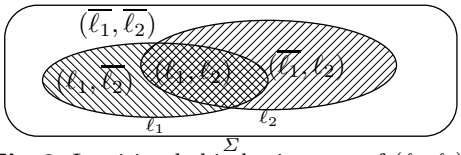

Fig. 2. Intuition behind minterms of (ℓ_1, ℓ_2).

Character set solver interface:

Boolean operations: union ($\cup$), intersection ($\cap$), and complement ($\complement$).

Nonemptiness check: decide if a given set is nonempty.

Minterm generation: compute all minterms of a sequence of sets.

For any character set representation, we can use an algorithm similar to that of Figure 3 to perform the minterm computation in terms of repeated complementation and intersections. This algorithm is similar in spirit to the standard *Quine-McCluskey algorithm* for minimizing Boolean functions. We compute intersections combinatorially in rank order; once a given combination reaches the empty set at rank i, we know that derived combinations of rank $i' > i$ are necessarily empty (and thus uninteresting). In practice, we use this algorithm for every character set representation.

For each character-set representation we discuss how the above operations are supported, and indicate how the minterm generation is implemented.

[3] A *partition of* Σ is a collection of nonempty subsets of Σ such that every element of Σ is in exactly one of these subsets.

Input: Finite nonempty sequence of nonempty sets $(\ell_i)_{i\in I}$.
Rank propagation: Iterate over *ranks* $r \leq |I|$ and construct S_r:

$$\begin{aligned} S_0 &= \{\emptyset\} \\ S_{r+1} &= \{J \cup \{i\} \mid J \in S_r, i \in I \setminus J, \ell_i \cap \textstyle\bigcap_{j\in J} \ell_j \neq \emptyset\} \end{aligned}$$

Output: Set of all $J \in \bigcup_r S_r$ such that $(\bigcap_{i\in J} \ell_i) \cap (\bigcap_{i\in I\setminus J} \mathtt{C}(\ell_i)) \neq \emptyset$.

Fig. 3. Minterm generation algorithm.

Character sets as BDDs In the general case, regexes, and string operations involving regexes, assume Unicode UTF-16 character encoding. This means that characters correspond to 16-bit bit-vectors. For the special case of ASCII range (resp. extended ASCII range) the number of bits is 7 (resp. 8). Independent of the number of bits, the general idea behind the BDD encoding is that each bit of a character corresponds to a variable of the BDD. The crucial point is to determine a good variable order for the BDD encoding. In order to do so, we considered typical uses of regexes, and in a separate evaluation determined an order which yielded BDDs of small size.

Based on the evaluation, a good choice of order turned out to be one where the highest bit has the lowest ordinal, in particular, for 16 bits, the MSB has ordinal 0 and the LSB has ordinal 15. This decision was in part based on the observation that typical regexes make extensive use of predefined character patterns called *character classes* where several bits are clustered and yield compact BDDs, such as `\w` (matches any word character), `\W` (matches any non-word character), `\d` (matches any decimal digit), `p{Lu}` (matches any single character in the Unicode general category `Lu` that is an uppercase letter), etc. There are a total of 30 general Unicode categories and the character class `\w` is a union of 7 of those categories (namely categories 0 (`Lu`), 1 (`Ll`), 2 (`Lt`), 3 (`Lm`), 4 (`Lo`), 8 (`Nd`) and 18 (`Pc`)). In total, the class `\w` *alone* denotes a set of characters consisting of 323 nonoverlapping ranges, totaling 47057 characters. Let $\beta_{\mathtt{p}}$ denote the BDD of a character pattern p. The BDD $\beta_{\backslash\mathtt{w}}$ has 1656 nodes. Note that Boolean operations with the above set interpretation correspond directly to Boolean operations over BDDs.

For example the pattern `[\w-[\da-d]]` denotes the set of all word characters that are not decimal digits or characters `a` through `d`, i.e., $\beta_{[\backslash\mathtt{w}-[\backslash\mathtt{da}-\mathtt{d}]]} = \beta_{\backslash\mathtt{w}} \cap \mathtt{C}(\beta_{\backslash\mathtt{d}} \cup \beta_{\mathtt{a}-\mathtt{d}})$. We write $[\![\beta]\!]$ for the set of characters represented by β, thus for example $[\![\beta_1 \cap \beta_2]\!] = [\![\beta_1]\!] \cap [\![\beta_2]\!]$. It is easy to write infeasible character patterns, for example `[\W-[\D]]` denotes an empty set since $[\![\beta_{\backslash\mathtt{d}}]\!] \subset [\![\beta_{\backslash\mathtt{w}}]\!]$ and

$$[\![\beta_{\backslash\mathtt{W}} \cap \mathtt{C}(\beta_{\backslash\mathtt{D}})]\!] = [\![\mathtt{C}(\beta_{\backslash\mathtt{w}}) \cap \mathtt{C}(\mathtt{C}(\beta_{\backslash\mathtt{d}}))]\!] = \mathtt{C}([\![\beta_{\backslash\mathtt{w}}]\!]) \cap [\![\beta_{\backslash\mathtt{d}}]\!].$$

Nonemptiness check of BDDs is trivial since the empty BDD $\beta^{\bot}$ is unique. Similarly, the BDD of all characters $\beta^{\top}$ is unique. Except for $\beta^{\bot}$ and $\beta^{\top}$, two BDDs are not guaranteed to be identical by construction even though they are isomorphic when representing the same sets. However, checking isomorphism (equivalence) of BDDs is linear. Equivalence checking is used prior to calling the minterm algorithm in order to eliminate duplicate sets from the input sequence.

Character sets as bitvector predicates A common alternative representation for character sets is to use interval arithmetic over bitvectors or integers. Here we assume that we are working in the context of a constraint solver that provides built-in support for bit vectors. We write BV^n for the sort of characters used by the solver, which is assumed to be a sort of n-bit vectors for a given fixed $n \in \{7, 8, 16\}$. Standard logic operations as well as standard arithmetic operations over BV^n, such as '$\leq$', are assumed to be built-in and can be used to form predicates for expressing character ranges. Let $\varphi_{\texttt{p}}(\chi)$ denote a predicate (with a single *fixed* free variable χ:BV^n) corresponding to the regex character pattern `p` and let $[\![\varphi_{\texttt{p}}]\!]$ denote the set of all characters a for which $\varphi_{\texttt{p}}(a)$ is true modulo the built-in theories. For example, consider BV^7 and the character pattern `\w`, the predicate $\varphi_{\texttt{\w}}$ is as follows where each disjunct corresponds to a Unicode category (the Unicode categories 2 , 3 and 4 are empty for the ASCII character range):

$$(\texttt{`A'} \leq \chi \wedge \chi \leq \texttt{`Z'}) \vee (\texttt{`a'} \leq \chi \wedge \chi \leq \texttt{`z'}) \vee (\texttt{`0'} \leq \chi \wedge \chi \leq \texttt{`9'}) \vee \chi = \texttt{`\_'}$$

where `'..'` is the bitvector representation of a character. The Boolean operations are directly supported by the corresponding built-in logical operators. For example $[\![\varphi_{[\texttt{\w}-[\texttt{\da-d}]]}]\!] = [\![\varphi_{\texttt{\w}} \wedge \neg(\varphi_{\texttt{\d}} \vee \varphi_{\texttt{a-d}})]\!] = [\![\varphi_{\texttt{\w}}]\!] \cap \complement([\![\varphi_{\texttt{\d}}]\!] \cup [\![\varphi_{\texttt{a-d}}]\!])$

For the ASCII range (or extended ASCII range), the direct range representation has several advantages by being succinct and taking advantage of the built-in optimizations of the underlying solver. Based on our experiments though, for full Unicode the representation produces predicates that do not scale. Instead, we use *if-then-else* (*Ite*) terms, supported in all state-of-the-art constraint solvers, to encode *Shannon expansions*, thus mimicking BDDs. An *Ite*-term is a term $\mathit{Ite}(\psi, t_1, t_2)$ that equals t_1, if ψ is true; equals t_2, otherwise. We explain the main idea. Given a BDD β the corresponding predicate $\mathit{Ite}[\beta]$ is constructed as follows where all shared subterms are constructed only once (and cached) and are thus maximally shared in the resulting term of the solver. Given a BDD β (other than $\beta^\bot$ or $\beta^\top$) we write $\mathit{BitIs0}(\beta)$ for the predicate over χ:BV^n, that is true if and only if the i'th bit of χ is 0, where $i = n - \mathit{ordinal}(\beta) - 1$ (recall that $\mathit{ordinal}(\beta)$ is the reverse bit position).

$$\mathit{Ite}[\beta] \stackrel{\text{def}}{=} \begin{cases} \mathit{true}, & \text{if } \beta = \beta^\top; \\ \mathit{false}, & \text{if } \beta = \beta^\bot; \\ \mathit{Ite}(\mathit{BitIs0}(\beta), \mathit{Ite}[\mathit{Left}(\beta)], \mathit{Ite}[\mathit{Right}(\beta)]), & \text{otherwise.} \end{cases}$$

It follows from the definition that $[\![\beta]\!] = [\![\mathit{Ite}[\beta]]\!]$.

Nonemptiness check of $[\![\varphi]\!]$ for a character predicate φ is given by checking if the formula φ is *satisfiable* by using the constraint solver. For minterm generation we use an incremental constraint solving technique that is sometimes called *cube-formula* solving [27]. The core idea behind the technique is as follows:

Given a finite sequence of character formulas $\boldsymbol{\varphi} = (\varphi_i)_{i<m}$ and distinct Boolean variables $\boldsymbol{b} = (b_i)_{i<m}$ define $\mathit{Cube}(\boldsymbol{\varphi}, \boldsymbol{b}) \stackrel{\text{def}}{=} \bigwedge_{i<m} \varphi_i \Leftrightarrow b_i$. A *solution* of $\mathit{Cube}(\boldsymbol{\varphi}, \boldsymbol{b})$ is an assignment M of truth values to $\boldsymbol{b}$ such that $M \models \exists\chi\, \mathit{Cube}(\boldsymbol{\varphi}, \boldsymbol{b})$. Given a solution M of $\mathit{Cube}(\boldsymbol{\varphi}, \boldsymbol{b})$, let $\varphi_M \stackrel{\text{def}}{=} \bigwedge_{i<m, M \models b_i} b_i \wedge \bigwedge_{i<m, M \models \neg b_i} \neg b_i$.

Input: ϵSFAs A and B over Σ.
Initialize: Eliminate epsilons from A and B.
Let S be a stack, initially $S = (\langle q_A^0, q_B^0 \rangle)$.
Let V be a hashtable, initially $V = \{\langle q_A^0, q_B^0 \rangle\}$.
Let Δ be a hashtable, initially $\Delta = \emptyset$.
Search: While S is nonempty, repeat the following.
Pop $\langle q_1, q_2 \rangle$ from S.
For each $\rho_1 \in \Delta_A(q_1)$ and $\rho_2 \in \Delta_B(q_2)$, let $\ell = Lbl(\rho_1) \cap Lbl(\rho_2)$, if $\ell \neq \emptyset$:
Let $p_1 = Tgt(\rho_1)$, $p_2 = Tgt(\rho_2)$ and add $(\langle q_1, q_2 \rangle, \ell, \langle p_1, p_2 \rangle)$ to Δ.
If $\langle p_1, p_2 \rangle \notin V$ then add $\langle p_1, p_2 \rangle$ to V and push $\langle p_1, p_2 \rangle$ to S.
Output: $A \times B = (\langle q_A^0, q_B^0 \rangle, \Sigma, V, \{q \in V \mid q \in F_A \times F_B\}, \Delta)$.

Fig. 4. Product algorithm. Constructs $A \times B$ such that $L(A \times B) = L(A) \cap L(B)$.

We use the following iterative model generation procedure to generate the set S of all solutions of $Cube(\boldsymbol{\varphi}, \boldsymbol{b})$. Although the procedure is exponential in (n in) the *worst case* due to the inherent complexity of the problem, it seems to work well in practice [27]. The algorithm assumes that the solver is capable of model generation, rather than just checking existence of a model, which is the case for most state-of-the-art constraint solvers and in particular SMT solvers.

Minterm generation with cubes: Input is a nonempty sequence of satisfiable character predicates $\boldsymbol{\varphi}$. Output is the set $\mathbf{M}$ of solutions of $Cube(\boldsymbol{\varphi}, \boldsymbol{b})$.
Initialize: Let $\mathbf{M}_0 = \emptyset$ and $\psi_0 = Cube(\boldsymbol{\varphi}, \boldsymbol{b})$.
Iterate: If ψ_i has a model M let $\mathbf{M}_{i+1} = \mathbf{M}_i \cup \{M\}$ and $\psi_{i+1} = \psi_i \wedge \neg\varphi_M$, otherwise let $\mathbf{M} = \mathbf{M}_i$ and stop.

4.2 Primitive automata algorithms

The algorithms on automata that we are most interested in, and that we will focus on here, are *product* $A \times B$ and *difference* $A - B$, where difference provides a way of checking *subset* constraints between regular expressions as well as doing *complementation* $\bar{B}$ such that $L(\bar{B}) = \Sigma^* \setminus L(B)$ and *determinization* of B, $Det(B)$, as special cases. The algorithms assume a representation of SFAs where move labels are symbolic and use a *character set solver* that provides the functionality discussed in Section 4.1. Both algorithms also have *lazy* versions that do not construct the full automaton: lazy difference is *subset checking* $L(A) \subseteq L(B)$ (with witness in $L(A) \setminus L(B)$ if $L(A) \subsetneq L(B)$), and lazy product is *disjointness checking* $L(A) \cap L(B) = \emptyset$ (with witness in $L(A) \cap L(B)$ if $L(A) \cap L(B) \neq \emptyset$). We do not describe the lazy versions explicitly, since, apart from not constructing the resulting automaton, they are similar to the eager versions.

The product algorithm is shown in Figure 4. Note that the character set solver is used for performing intersection and nonemptiness check on labels. The difference algorithm, shown in Figure 5, makes, in addition, essential use of minterm generation during an implicit determinization of the subtracted automaton B in $A - B$. Intuitively, the algorithm is a combined product and complementation algorithm. The main advantage of the combination is *early pruning of search stack* S by keeping the resulting automaton *clean*. In our implementation, the difference algorithm uses a standard set implementation to represent state sets,

Input: ϵSFAs A and B over Σ.
Initialize: Eliminate epsilons from A and B and make B *total*.
Let $q^0 = \langle q^0_A, State\{q^0_B\}\rangle$.
Let S be a stack, initially $S = (q^0)$.
Let V, F, and Δ be hashtables. Initially $V = \{q^0\}$ and $\Delta = \emptyset$.
Initially $F = \{q^0\}$, if $q^0_A \in F_A$ and $q^0_B \notin F_B$; $F = \emptyset$, otherwise.
Search: While S is nonempty repeat the following.
Pop $\langle p, Q\rangle$ from S. Let $\Delta_B(Q) = \{\rho_i\}_{i\in I}$ and let $\ell_i = Lbl(\rho_i)$ for $i \in I$.
Compute minterms: Compute $\mathbf{M}$ as the set of all minterms of the sequence $(\ell_i)_{i\in I}$. Here minterms are given as subsets of I. For $J \in \mathbf{M}$ let $\ell_J = \bigcap_{i\in J} \ell_i \cap \bigcap_{i\in I\setminus J} \complement(\ell_i)$.
For each minterm $J \in \mathbf{M}$:
For each A **move** $\rho \in \Delta_A(p)$ such that $Lbl(\rho) \cap \ell_J \neq \emptyset$:
Let $P = State\{\, Tgt(\rho_i) \mid i \in J\}$.
Let $q = \langle Tgt(\rho), P\rangle$.
Add $(\langle p, Q\rangle, Lbl(\rho) \cap \ell_J, q)$ to Δ.
If $q \notin V$ then add q to V and push q to S.
If $Tgt(\rho) \in F_A$ and $P \cap F_B = \emptyset$ then add q to F.
Output: $A - B = (q^0, \Sigma, V, F, \Delta)$.

Fig. 5. Difference algorithm. Constructs $A - B$ such that $L(A - B) = L(A) \setminus L(B)$.

denoted by $State\{\ldots\}$ in Figure 5, to represent subsets of automata states. The difference algorithm uses *three* different kinds of set data structures, each with different strengths: character sets, state sets, and hashtables for algorithm variables, where the character set representation depends on the character set solver.

Note that the difference algorithm reduces to complementation of B when $L(A) = \Sigma^*$, e.g., when $A = (q^0_A, \Sigma, \{q^0_A\}, \{q^0_A\}, \{(q^0_A, \Sigma, q^0_A)\})$. Then the condition $Lbl(\rho) \cap \ell_J \neq \emptyset$ above is trivially true, since $Lbl(\rho) \cap \ell_J = \ell_J$, for $\rho \in \Delta_A$. Consequently, there is no pruning of the search stack S then with respect to A, and the full complement $\bar{B}$ is constructed. Moreover, $\bar{B}$ is *deterministic*, since for any two distinct moves $(\langle p, Q\rangle, \ell_J, q)$ and $(\langle p, Q\rangle, \ell_{J'}, q')$ that are added to Δ above, $\ell_J \cap \ell_{J'} = \emptyset$ by definition of minterms. It follows that the difference algorithm also provides a *determinization* algorithm for B: construct $\bar{B}$ as above and let $Det(B) = (q^0_{\bar{B}}, \Sigma_{\bar{B}}, Q_{\bar{B}}, Q_{\bar{B}} \setminus F_{\bar{B}}, \Delta_{\bar{B}})$.

5 Experiments

In this section we first compare the performance of the product and difference algorithms with respect to different character set representations and eager vs. lazy versions of the algorithms. For predicate representation of character sets we use Z3 as the constraint solver. Integration with Z3 uses the .NET API that is publically available [32]. All experiments were run on a laptop with an Intel dual core T7500 2.2GHz processor. We then compare the performance of our implementation with JSA [5] that is based on a well-established open-source automata library (`dk.brics.automaton`).

Our experiment uses a set of ten benchmark regexes that have previously been used to evaluate string constraint solving tools [12,28]. The regexes are representative for various practical usages and originate from a case study in [18]. The sizes of the automata constructed from the regexes are shown in Figure 6.

For each pair $(A_i, A_j)_{i,j<10}$ we conducted the following experiments to compare different character set representations, algorithmic choices, and the effect

	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
$\|Q\|$:	36	30	31	17	11	18	4573	104	2228	42
$\|\Delta^\epsilon\|$:	25	15	10	11	4	14	8852	99	3570	32
$\|\Delta^\ell\|$:	36	24	28	15	11	13	148	96	524	40

Fig. 6. Sizes of ϵSFAs A_{i-1} constructed for regexes $\#i$ for $1 \leq i \leq 10$ in [28, Table I], where each regex is assumed to have an implicit start-anchor `^` and end-anchor `$`.

Difference

	Eager		Lazy	
	Empty	**Nonempty**	**Empty**	**Nonempty**
BDD-ASCII	(8).33/.007	(78)36/.008	(8).33/.006	(87)41/.001
BDD-UTF16	(8).56/.02	(78)38/.02	(8).52/.012	(87)41/.01
Pred-ASCII	(8)1.6/.06	(78)72/.08	(8)1.6/.06	(87)58/.02
Pred-UTF16	(8)5.5/.12	(78)179/.24	(8)5.3/.12	(87)66/.11
Range-ASCII	(8).9/.03	(78)67/.03	(8)1/.03	(87)44/.003
Hash-ASCII	(8).9/.03	(78)67/.03	(8)1/.03	(87)45/.003
brics-ASCII	(9)32/.015	(72)273/.016		
brics-UTF16	(9)39/.11	(72)341/.44		

Product

	Eager		Lazy	
	Empty	**Nonempty**	**Empty**	**Nonempty**
BDD-ASCII	(29)9.7/.001	(26)90/.002	(29)9.7/.001	(26)19/.001
BDD-UTF16	(29)9.7/.001	(26)92/.003	(29)9.7/.001	(26)19/.001
Pred-ASCII	(29)10/.003	(26)142/.003	(29)10/.004	(26)25/.007
Pred-UTF16	(29)10/.01	(26)150/.05	(29)10/.01	(26)25/.03
Range-ASCII	(29)10/.002	(23)16/.005	(29)10/.002	(26)69/.001
Hash-ASCII	(29)10/.002	(23)16/.005	(29)10/.002	(26)70/.001
brics-ASCII	(25)66/.015	(19)65/.016		
brics-UTF16	(25)66/.03	(19)70/.09		

Fig. 7. Experimental evaluation of the SFA algorithms. Each entry in the tables has the form $(n)t/m$ where n is the *number of combinations solved*, t is the *total time* it took to solve the n instances, m is the *median*. Time is in seconds. For product, the eager experiment constructs $A_i \times A_j$ for $0 \leq i \leq j < 10$; the lazy experiment tests emptiness of $L(A_i) \cap L(A_j)$ for $0 \leq i \leq j < 10$. For difference, the eager experiment constructs $A_i - A_j$ for $0 \leq i, j < 10$; the lazy experiment tests if $L(A_i) \subseteq L(A_j)$ ($L(A_i - A_j) = \emptyset$) for $0 \leq i, j < 10$. Timeout for each instance (i, j) was 20 min.

of the size of the alphabet. For product we ignored the order of the arguments due to commutativity, thus there are 55 pairs in total, and for difference there are 100 pairs in total. Figure 7 shows the evaluation results for the difference algorithm and the product algorithm, respectively. The times exclude the construction time of ϵSFAs from the regexes but *include* the epsilon elimination time to convert ϵSFAs to SFAs that is a preprocessing step in the algorithms. The total time to construct the ϵSFAs for the 10 regexes (including parsing) was 0.33 seconds (for both UTF16 as well as ASCII). For parsing the regexes we use the built-in regex parser in .NET.

The top columns correspond to the eager vs. lazy versions of the algorithms and the secondary columns correspond to whether the result is an empty automaton or a nonempty automaton. The rows correspond to the different algorithmic choices: *BDD-X* denotes the use of the BDD based solver where X is ASCII

or UTF16; *Pred-ASCII* denotes the use of predicate encoding of character sets using Z3 predicates over BV^7; *Pred-UTF16* denotes the use of predicate encoding of character sets using Z3 *Ite*-term encodings of BDDs over BV^{16}; *Range-ASCII* denotes the use of standard range representation for ASCII character sets as *hashsets of character pairs*; *Hash-ASCII* denotes the use of *hashsets of individual characters* as character sets. We excluded the evaluation results for most of the *Range-UTF16* and *Hash-UTF16* cases, since too many instances either timed out or caused an out-of-memory exception. The only case that completed for the same number of instances was lazy difference with *Range-UTF16* character set representation, but is e.g. 70 times slower in total experiment time compared to *BDD-UTF16*. In the difference experiment, 5 instances either timed out or caused an out-of-memory exception, involving the automata A_6 and A_8 in all cases. One of the hardest instances that terminated was checking $L(A_8) \subseteq L(A_9)$ that took 14 minutes with *Range-UTF16* and 1.3 seconds with *BDD-UTF16*, where the minterm generation algorithm is used heavily.

With the BDD representation, the size of the alphabet turns out not to play a major role in the algorithms, although the BDD sizes are typically considerably larger for UTF16 (in hundreds or even thousands of nodes) compared to ASCII (in tens on nodes). This is a useful indication that BDDs work surprisingly well as character sets and may even work for UTF32 encoding (which we have not tried). In several non-BDD cases, many individual experiments timed out. The BDD character set based algorithms are an *order of magnitude faster in the median for UTF16*. As expected, all the lazy versions of the algorithms are faster in the nonempty case. The eager constructions are however needed when converting the resulting SFAs to *symbolic language acceptors* for combination with other theories [28]. Symbolic language acceptors for SFAs are made use of for example in the parameterized unit testing framework Pex [23], where regexes are handled in branch conditions of .NET programs using SFAs with Pred-UTF16 representation of character sets, integration of the algorithms in this paper into Pex is currently ongoing work.[4]

Comparison with `brics`. For this comparison we first serialized the automata A_i, $i < 10$, in textual format. This is for two reasons: 1) to provide a fair comparison *at the level of automata algorithms*, i.e., to exclude differences in the automata constructed from the regexes; 2) to avoid semantic differences regarding the meaning of the notations used in the regexes. In the measurements we bootstrapped the automata by excluding the time to deserialize and to reconstruct the automata in *brics*, but included the time to add epsilon transitions (that add extra character interval transitions), as this is effectively equivalent to epsilon elimination using the *brics* library.

The Java code responsible for the lazy difference experiment is

```
... construct Ai, Aj, epsAi, epsAj ...
long t = System.currentTimeMillis();
boolean empty = (Ai.addEpsilons(epsAi)).subsetOf(Aj.addEpsilons(epsAj));
t = System.currentTimeMillis() - t;
```

[4] `http://www.pexforfun.com/`

	Product		Difference	
	Empty	**Nonempty**	**Empty**	**Nonempty**
BDD-ASCII	.022/.001	.024/.001	.32/.005	.33/.001
BDD-UTF16	.022/.001	.024/.001	.54/.015	.78/.002
Pred-ASCII	.07/.003	.07/.004	1.6/.06	2/.01
Pred-UTF16	.2/.01	.2/.01	5.2/.12	8.2/.05
brics-ASCII	.14/.001	.2/.015	.1/.016	.8/.016
brics-UTF16	.6/.03	1.4/.05	3.6/.05	24.8/.08

Fig. 8. Lazy difference and product experiment times with A_6 and A_8 excluded. Each entry is *total*/*median* in seconds. For product, 21 instances are empty and 15 instances are nonempty. For difference, 8 instances are empty and 56 instances are nonempty.

using the `bircs.Automaton` method `subsetOf`. The eager experiment used the `bircs.Automaton` method `minus`. The code for the product experiment is similar (using `intersection`). For running each instance we assigned 1.5GB to the java runtime (which was the maximum possible). For all cases involving A_6, the product and difference experiments timed out or caused an out-of-memory exception. For the product experiment all cases involving A_8 also timed out. The instance $L(A_8) \subseteq L(A_8)$ did not time out using brics, while caused an out-of-memory exception using our tool during minterm generation.

For a better comparison of the easy cases we conducted a separate experiment where both A_6 and A_8 are excluded from the experiments and considered the lazy versions only. Thus, there are total of 36 product instances and 64 difference instances. The outcome of this experiment is shown in Figure 8. Let t_X^{op} denote the total time for experiment row X and operation op in Figure 8. Then:

$$t^{\text{prod}}_{\text{brics}-\text{ASCII}}/t^{\text{prod}}_{\text{BDD}-\text{ASCII}} \approx 7, \quad t^{\text{prod}}_{\text{brics}-\text{UTF16}}/t^{\text{prod}}_{\text{BDD}-\text{UTF16}} \approx 43,$$
$$t^{\text{diff}}_{\text{brics}-\text{ASCII}}/t^{\text{diff}}_{\text{BDD}-\text{ASCII}} \approx 1.4, \; t^{\text{diff}}_{\text{brics}-\text{UTF16}}/t^{\text{diff}}_{\text{BDD}-\text{UTF16}} \approx 21.$$

Technique Name	Data structure	Corresponds to
Christensen, Moller, et al. (JSA) [5]	Char. ranges (Java, Unicode)	Eager Range
Hooimeijer and Weimer (DPRLE) [11]	Single-char. hashset (OCaml; ASCII)	Eager Hashset
Hooimeijer and Weimer [12]	Char. ranges (C++; ASCII)	Lazy Range
Minamide [21,29]	Single-char. functional set (OCaml; ASCII)	Eager Hashset
Veanes, Halleux, and Tillmann [28]	Unary pred. (C# and Z3[32]; Unicode)	Lazy predicate
Henriksen, Jensen, et al. [10,30]	BDDs C	Eager BDD

Fig. 9. Existing automata-based string analyses and, the data structures they use, and the closest-matching experimental prototype tested in this paper.

6 Related Work

In this section we discuss related work, focusing on automated reasoning techniques; Figure 9 provides a brief overview. DPRLE [11] has two associated imple-

mentations: a fully verified core algorithm written in Gallina [6], and the OCaml implementation used for the experiments. The Gallina specification is designed primarily to help discharge proof obligations (using Coq's built-in inversion lemmas for cons lists, for example). This specification is of limited interest, since it is not actually designed to be executed. The OCaml implementation relies heavily on OCaml's built-in hashtable data structure. The transition function is modeled using two separate mappings: one for character-consuming transitions (`state -> state -> character set`) and one for ϵ–transitions (`state -> state set`). This representation reflects the fact that the main DPRLE algorithms rely heavily on tracking states across automaton operations [11].

The Rex tool provides a SFA representation that is similar to the formal definition given in Section 3. The core idea, based on work by van Noord and Gerdeman [22], is to represent automaton transitions using logical predicates. Rex works in the context of *symbolic language acceptors*, which are first-order encodings of symbolic automata into the theory of algebraic datatypes. The Rex implementation uses the Z3 satisfiability modulo theories(SMT) solver [7] to solve the produced constraints. The encoding process and solving process are completely disjoint. This means that many operations, like automaton intersection, can be offloaded to the underlying solver. For example, to find a string w that matches two regular expressions, r_1 and r_2, Rex can simply assert the existence of w, generate symbolic automaton encodings for r_1 and r_2, and assert that s is accepted by both those automata. We refer to this as a *Boolean encoding* of the string constraints.

The initial Rex work [28] explores various optimizations, such as minimizing the symbolic automata prior to encoding them. These optimizations make use of the underlying SMT solver to find combinations of edges that have internally-consistent move conditions. Subsequent work [27] explored the trade-off between the Boolean encoding and the use of automata-specific algorithms for language intersection and language difference. In this case, the automata-specific algorithms make repeated calls to Z3 to solve *cube formulae* to enumerate edges with co-satisfiable constraints. In practice, this approach is not consistently faster than the Boolean encoding.

The Java String Analyzer (JSA) by Christensen *et al.* [5] uses finite automata internally to represent strings. The underlying `dk.brics.automaton` library is not presented as a constraint solving tool, but it is used in that way. The library represents automata using a pointer-based graph representation of node and edge objects; edges represent contiguous character ranges. The library, which is written in Java, includes a deterministic automaton representation that is specialized for matching given strings efficiently. This is not a common use case for constraint solving, since the goal there is to efficiently *find* string assignments rather than verify them. Several other approaches include a built-in model of string operations; Minamide [21] and Wassermann and Su [29] rely on an ad-hoc OCaml implementation that uses that language's built-in applicative data structures.

In previous work, we present a lazy backtracking search algorithm for solving regular inclusion constraints [12]. The underlying automaton representation, written in C++, is based on the Boost Graph Library [25], which allows for a variety of adjacency list representations. We annotate transitions with integer ranges, similar to JSA. The implementation pays special attention to memory management, using fast pool allocation for small objects such as the abstract syntax tree for regular expressions. This implementation uses lazy algorithms intersection and determinization algorithms, allowing for significant performance benefits relative to DPRLE [11] and the original Rex [28] implementation.

Other Approaches. Bjørner *et al.* describe a set of string constraints based on common string library functions [2]. The approach is based on a direct encoding to a combination of theories provided by the Z3 SMT solver [7]. Supported operations include substring extraction (and, in general, the combination of integer constraints with string constraints), but the work does not provide an encoding for regular sets.

The Hampi tool [15] uses an eager bitvector encoding from regular expressions to bitvector logic. The encoding does not use quantification, and requires the enumeration of all positional shifts for every subexpression. The Kaluza tool extends this approach to systems of constraints with multiple variables and concatenation [24].

The MONA implementation [10] provides decision procedures for several varieties of monadic second–order logic (M2L). MONA relies on a highly-optimized BDD-based representation for automata. The implementation has seen extensive engineering effort; many of the optimizations are described in a separate paper [16]. We are not aware of any work that investigates the use of multi-terminal BDDs for nondeterministic finite automata directly. We believe this may be a promising approach, although the use of BDDs complicates common algorithms like automaton minimization. In this paper, we restrict our attention to a class of graph-based automata representations.

There is a wide range of application domains that, at some level, rely on implicit or explicit automata representations. Fang *et al.* [31] use an automaton-based method to model string constraints and length bounds for abstract interpretation. There is significant work on using automata for arithmetic constraint solving; Boigelot and Wopler provide an overview [4]. A subset of work on explicit state space model checking (e.g., [3]) has focused on general algorithms to perform fast state space reduction. Similarly, there is theoretical work on word equations that involves automata [1,17], as well as algorithms work on the efficient construction of automata from regular expressions (e.g., [14]).

7 Conclusions

In this paper we presented a study of automata representations for efficient intersection and difference constructions. We conducted this study to evaluate which combination of data structures and algorithms is most effective in the context

of string constraint solving. Existing work in this area has consistently included performance comparisons at the tool level, but has been largely inconclusive regarding which automata representations work best in general. To answer this question, we re-implemented a number of data structures in the same language (C#) using a front-end parser that correctly handles a large subset of .NET's regular expression language, and using a UTF-16 alphabet. Our experiments showed that, over the sample inputs under consideration, the BDD-based representation provides the best performance when paired with the lazy versions of the intersection and difference algorithms. This suggests that, future string decision procedure work can achieve significant direct benefits by using this combination of data structure (BDD) and algorithms (lazy) as a starting point. To the best of our knowledge, no existing tools currently use this combination.

Acknowledgement We thank Nikolaj Bjørner for discussions that lead to the BDD character set representation idea and for continuous support with Z3.

References

1. Sebastian Bala. Regular language matching and other decidable cases of the satisfiability problem for constraints between regular open terms. In *STACS*, pages 596–607, 2004.
2. Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS'09*, volume 5505 of *LNCS*, pages 307–321. Springer, 2009.
3. Stefan Blom and Simona Orzan. Distributed state space minimization. *J. Software Tools for Technology Transfer*, 7(3):280–291, 2005.
4. Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: An overview. In *ICLP 2002*, pages 1–19.
5. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, pages 1–18, 2003.
6. Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.
7. Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS. Springer, 2008.
8. Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI '08*, Tucson, AZ, USA, June 9–11, 2008.
9. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05*, pages 213–223, 2005.
10. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95*, volume 1019 of *LNCS*. Springer, 1995.
11. Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *PLDI*, pages 188–198, 2009.
12. Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *ASE 2010: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010.
13. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

14. Lucian Ilie and Sheng Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.
15. Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: a solver for string constraints. In *ISSTA '09*, pages 105–116. ACM, 2009.
16. Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.
17. Michal Kunc. What do we know about language equations? In *Developments in Language Theory*, pages 23–27, 2007.
18. Nuo Li, Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *ASE'09*, 2009.
19. MSDN Library. System.text namespace. In *http://msdn.microsoft.com/en-us/library/system.text.aspx*, June 2010.
20. PHP Manual. Pcre; posix regex; strings. In *http://php.net/manual/en/book.strings.php*, December 2009.
21. Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *WWW '05*, pages 432–441, 2005.
22. Gertjan Van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:2001, 2001.
23. Pex. http://research.microsoft.com/projects/pex.
24. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, 2010.
25. Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, December 2001.
26. Nikolai Tillmann and Jonathan de Halleux. Pex - white box test generation for .NET. In *TAP'08*, volume 4966 of *LNCS*, pages 134–153, Prato, Italy, April 2008. Springer.
27. Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. Symbolic automata constraint solving. In C. Fermüller and A. Voronkov, editors, *LPAR-17*, volume 6397 of *LNCS/ARCoSS*, pages 640–654. Springer, 2010.
28. Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST'10*. IEEE, 2010.
29. Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI'07*, pages 32–41. ACM, 2007.
30. Fang Yu, Muath Alkhalaf, and Tevfik Bultan. An automata-based string analysis tool for php. In *TACAS'10*, LNCS. Springer, 2010.
31. Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic String Verification: Combining String Analysis and Size Analysis. In *TACAS*, pages 322–336, 2009.
32. Z3. http://research.microsoft.com/projects/z3.