Modeling Regular Replacement for String Constraint Solving

Xiang Fu
Hofstra University
Xiang.Fu@hofstra.edu

Chung-Chih Li
Illinois State University
cli2@ilstu.edu

Abstract

Bugs in user input sanitation of software systems often lead to vulnerabilities. Among them many are caused by improper use of regular replacement. This paper presents a precise modeling of various semantics of regular substitution, such as the declarative, finite, greedy, and reluctant, using finite state transducers (FST). By projecting an FST to its input/output tapes, we are able to solve atomic string constraints, which can be applied to both the forward and backward image computation in model checking and symbolic execution of text processing programs. We report several interesting discoveries, e.g., certain fragments of the general problem can be handled using less expressive deterministic FST. A compact representation of FST is implemented in SUSHI, a string constraint solver. It is applied to detecting vulnerabilities in web applications.

1 Introduction

User input sanitation has been widely used by programmers to assure robustness and security of software. *Regular replacement* is one of the most frequently used approaches by programmers. For example, at both client and server sides of a web application, it is often used to perform format checking and filtering of command injection attack strings. As software bugs in user input sanitation can easily lead to vulnerabilities, it is desirable to employ automated analysis techniques for revealing such security holes. This paper presents the finite state transducer models of a variety of regular replacement operations, geared towards automated analysis of text processing programs.

One application of the proposed technique is symbolic execution [10]. In [3] we outlined a unified symbolic execution framework for discovering command injection vulnerabilities. The target system under test is executed as usual except that program inputs are treated as symbolic literals. A path condition is used to record the conditions to be met by the initial input, so that the program will execute to a location. At critical points, e.g., where a SQL query is submitted, path conditions are paired with attack patterns. Solving these constraints leads to attack signatures.

```
1 <?php
2   $msg = $_POST["msg"];
3   $sanitized =  preg_replace("/\<script.*?\>.*?\<\/script.*?\>/i","", $msg);
4   save_to_db($sanitized)
5 ?>
```

Listing 1: Vulnerable Sanitation against XSS Attack

In the following, we use an example to demonstrate the idea of the above research and motivate the modeling of regular replacement in this paper. Consider a PHP snippet in Listing 1, which takes a message as input and posts it to a bulletin. To prevent the Cross-Site Scripting (XSS) attack, the programmer calls `preg_replace()` to remove any pair of `<script>` and `</script>` tags. Unfortunately, the protection is insufficient. Readers can verify that `<<script></script>script>alert('a')</script>` is an attack string. After `preg_replace()`, it yields `<script>alert('a')</script>`.

We now show how the attack signature is generated, assuming the availability of symbolic execution. By symbolically executing the program, variable `$msg` is initialized with a symbolic literal and let it be x. Assume α is the regular expression `<script.*?>.*?</script.*?>` and ε is the empty string. After

line 3, variable `$sanitized` has a symbolic value represented by string expression $x^{-}_{\alpha\to\varepsilon}$, and it is a replacement operator that denotes the effects of `preg_replace`, using the *reluctant* semantics (see "$*?$" in formula). Then at line 4 where the SQL query is submitted, a string constraint can be constructed as below, using an existing attack pattern. The equation asks: can a JavaScript snippet be generated after the `preg_replace` protection?

$$x^{-}_{\alpha\to\varepsilon} \equiv \texttt{<script.*?>alert('a')</script.*?>}$$

To solve the above equation, we first model the reluctant regular replacement $x^{-}_{\alpha\to\varepsilon}$ as a finite state transducer (FST) and let it be $\mathscr{A}_1$. The right hand side (RHS) of the equation is a regular expression, and let it be r. It is well known that the identity relation $Id(r) = \{(w,w) \mid w \in L(r)\}$ is a regular relation that can be recognized by an FST (let it be $\mathscr{A}_2$). Now let $\mathscr{A}$ be the composition of $\mathscr{A}_1$ and $\mathscr{A}_2$ (by piping the output tape of $\mathscr{A}_1$ to input tape of $\mathscr{A}_2$). Projecting $\mathscr{A}$ to its input tape results in a finite state automaton (FSA) that represents the solution of x.

Notice that a precise modeling that distinguishes the various regular replacement semantics is necessary. For example, a natural question following the above analysis is: *If we approximate the reluctant semantics using the greedy semantics, could the static analysis be still effective?* The answer is negative: When the `*?` operators in Listing 1 are treated as `*`, the analysis reports no solution for the equation, i.e., a false negative report on the actually vulnerable program.

In this paper, we present the modeling of regular replacement operations. §2 covers preliminaries. §3 and §4 present the modeling of various regular replacement semantics. §5 introduces tool support. §6 discusses related work. §7 concludes.

2 Preliminaries

This section formalizes several typical semantics of regular substitution, and then introduces a variation of the standard finite state transducer model. We introduce some notations first. Let Σ represent the alphabet and R the set of regular expressions over Σ. If $\omega \in \Sigma^*$, ω is called a word. Given a regular expression $r \in R$, its language is denoted as $L(r)$. When $\omega \in L(r)$ we say ω is an instance of r. We sometimes abuse the notation as $\omega \in r$ when the context is clear that r is a regular expression. A regular expression r is said to be *finite* if $L(r)$ is finite. Clearly, $r \in R$ is finite if and only if there exists a constant length bound $n \in N$ s.t. for any $\omega \in L(r)$, $|w| \leq n$. We assume $\# \notin \Sigma$ is the *begin marker* and $\$ \notin \Sigma$ is the *end marker*. They will be used in modeling procedural regular replacement in §4. Let $\Sigma_2 = \Sigma \cup \{\#, \$\}$. Assume Ψ is a second alphabet which is disjoint with Σ. Given $\omega \in (\Sigma \cup \Psi)^*$, $\pi(\omega)$ denotes the projection of ω to Σ s.t. all the symbols in Ψ are removed from ω. Let $0 \leq i < j \leq |\omega|$, $\omega[i, j]$ represents a substring of ω starting from index i and ending at $j-1$ (index counts from 0). Similarly, $\omega[i]$ refers to the element at index i. We use NFST, DFST to denote the nondeterministic and deterministic FST, respectively. Similar are NFSA and DFSA for finite state automata.

There are three popular semantics of regular replacement, namely *greedy*, *reluctant*, and *possessive*, provided by many programming languages, e.g., in `java.utils.regex` of J2SE. We concentrate on two of them: the greedy and the reluctant. The greedy semantics tries to match a given regular expression pattern with the longest substring of the input while the reluctant semantics works in the opposite way. From the theoretical point of view, it is also interesting to define a *declarative* semantics for string replacement. A declarative replacement $\gamma_{r\to\omega}$ replaces every occurrence of a regular pattern r with ω.

Definition 2.1. Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\varepsilon \notin r$). The *declarative replacement*, denoted as $\gamma_{r\to\omega}$, is defined as:

$$\gamma_{r\to\omega} = \begin{cases} \{\gamma\} & \text{if } \gamma \notin \Sigma^* r \Sigma^* \\ \{\nu_{r\to\omega}\,\omega\,\mu_{r\to\omega} \mid \gamma = \nu\beta\mu \text{ and } \beta \in r\} & \text{otherwise} \end{cases}$$

∎

The greedy and reluctant semantics are also called *procedural*, because both of them enforce a *left-most* matching. The replacement procedure is essentially a loop which examines each index of a word, from left to right. Once there is a match of the regular pattern r, the greedy replacement performs the longest match, and the reluctant replaces the shortest.

Definition 2.2. Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\varepsilon \notin r$). The *reluctant replacement* of r with ω in γ, denoted as $\gamma^-_{r\to\omega}$, is defined recursively as $\gamma^-_{r\to\omega} = \{\nu\omega\mu^-_{r\to\omega}\}$ where $\gamma = \nu\beta\mu$, $\nu \notin \Sigma^* r \Sigma^*$, $\beta \in r$, and for every $\nu_1, \nu_2, \beta_1, \beta_2, \mu_1, \mu_2 \in \Sigma^*$ with $\nu = \nu_1\nu_2$, $\beta = \beta_1\beta_2$, $\mu = \mu_1\mu_2$: if $\nu_2 \neq \varepsilon$ then $\nu_2\beta_1 \notin r$ and $\nu_2\beta\mu_1 \notin r$; and, if $\beta_2 \neq \varepsilon$ then $\beta_1 \notin r$. ∎

Note that in the above definition, "if $\nu_2 \neq \varepsilon$ then $\nu_2\beta_1 \notin r$ and $\nu_2\beta\mu_1 \notin r$" enforces left-most matching", i.e., there does not exist an earlier match of r than β; similarly, "if $\beta_2 \neq \varepsilon$ then $\beta_1 \notin r$" enforces shortest matching, i.e., there does not exist a shorter match of r than β.

Definition 2.3. Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\varepsilon \notin r$). The *greedy replacement*, denoted as $\gamma^+_{r\to\omega}$, is defined recursively as $\gamma^+_{r\to\omega} = \{\nu\omega\mu^+_{r\to\omega}\}$ where $\gamma = \nu\beta\mu$, $\nu \notin \Sigma^* r \Sigma^*$, $\beta \in r$, and for every $\nu_1, \nu_2, \beta_1, \beta_2, \mu_1, \mu_2 \in \Sigma^*$ with $\nu = \nu_1\nu_2$, $\beta = \beta_1\beta_2$, $\mu = \mu_1\mu_2$: if $\nu_2 \neq \varepsilon$ then $\nu_2\beta_1 \notin r$ and if $\mu_1 \neq \varepsilon$ then $\nu_2\beta\mu_1 \notin r$. ∎

Example 2.4. Let $\gamma = aaa$ with $a \in \Sigma$, (i) $\gamma_{aa\to b} = \{ba, ab\}$, $\gamma^+_{aa\to b} = \{ba\}$, $\gamma^-_{aa\to b} = \{ba\}$. (ii) $\gamma_{a^+\to b} = \{b, bb, bbb\}$, $\gamma^+_{a^+\to b} = \{b\}$, and $\gamma^-_{a^+\to b} = \{bbb\}$. ∎

Notice that in the above definitions, $\varepsilon \notin r$ is required for simplicity. In practice, precise Perl/Java regex semantics is followed for handling $\varepsilon \in r$. For example, in SUSHI, given $\gamma = a$, $r = a^*$, and $\omega = b$, $\gamma^-_{r\to\omega} = \{bab\}$ and $\gamma^+_{r\to\omega} = \{bb\}$. When $\beta \in \gamma^-_{r\to\omega}$, we often abuse the notation and write it as $\beta = \gamma^-_{r\to\omega}$, given the following lemma. Similar applies to $\gamma^+_{r\to\omega}$.

Lemma 2.5. For any $\gamma, \omega \in \Sigma^*$ and $r \in R$: $|\gamma^+_{r\to\omega}| = |\gamma^-_{r\to\omega}| = 1$.

In the following, we briefly introduce the notion of FST and its variation, using the terminology in [7]. We demonstrate its application to modeling the declarative replacement.

Definition 2.6. Let Σ^ε denote $\Sigma \cup \{\varepsilon\}$. A finite state transducer (FST) is an enhanced two-taped nondeterministic finite state machine described by a quintuple $(\Sigma, Q, q_0, F, \delta)$, where Σ is the alphabet, Q the set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ the set of final states, and δ is the transition function, which is a total function of type $Q \times \Sigma^\varepsilon \times \Sigma^\varepsilon \to 2^Q$. ∎

It is well known that each FST accepts a regular relation which is a subset of $\Sigma^* \times \Sigma^*$. Given $\omega_1, \omega_2 \in \Sigma^*$ and an FST $\mathscr{M}$, we say $(\omega_1, \omega_2) \in L(\mathscr{M})$ if the word pair is accepted by $\mathscr{M}$. Let $\mathscr{M}_3$ be the composition of two FSTs $\mathscr{M}_1$ and $\mathscr{M}_2$, denoted as $\mathscr{M}_3 = \mathscr{M}_1 || \mathscr{M}_2$. Then $L(\mathscr{M}_3) = \{(\mu, \nu) \mid (\mu, \eta) \in L(\mathscr{M}_1)$ and $(\eta, \nu) \in L(\mathscr{M}_2)$ for some $\eta \in \Sigma^*\}$. We introduce an equivalent definition of FST below.

Definition 2.7. An augmented finite state transducer (AFST) is an FST $(\Sigma, Q, q_0, F, \delta)$ with the transition function augmented to type $Q \times \mathscr{R} \to 2^Q$, where $\mathscr{R}$ is the set of regular relations over Σ. ∎

In practice, we would often restrict the transition function of an AFST to the following two types: (1) $Q \times R \times \Sigma^* \to 2^Q$. In a transition diagram, we label the arc from q_i to q_j for transition $q_j \in \delta(q_i, r : \omega)$

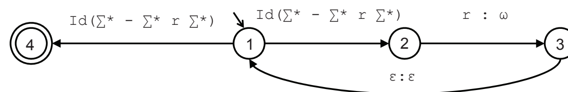


Figure 1: An FST for $s_{r\to\omega}$

by $r : \omega$; and (2) $Q \times \{Id(r) \mid r \in R\} \to 2^Q$, where $Id(r) = \{(\omega, \omega) \mid \omega \in L(r)\}$. In a transition diagram, an arc of type (2) is labeled as $Id(r)$.

Now, we can use an AFST to model the declarative string replacement $s_{r \to \omega}$ for any $\omega \in \Sigma^*$ and $r \in R$ (with $\varepsilon \notin r$). Figure 1 shows the construction, which presents an AFST that accepts $\{(s, \eta) \mid s \in \Sigma^*$ and $\eta \in s_{r \to \omega}\}$. In other words, given any two $s, \eta \in \Sigma^*$, we can use the AFST to check if η is a string obtained from s by replacing every occurrence of patterns in r with ω. We alternatively use FST and AFST for the time being without loss of generality.

3 DFST and Finite Replacement

This section shows that regular replacement with finite language pattern can be modeled using DFST, under certain restrictions. We fix the notation of DFST first. Intuitively, for a DFST, at any state $q \in Q$ the input symbol uniquely determines the destination state and the symbol on output tape. If there is a transition labeled with ε on the input, then this is the only transition from q.

Definition 3.1. An FST $\mathscr{A} = (\Sigma, Q, s_0, F, \delta)$ is *deterministic* if for any $q \in Q$ and any $a \in \Sigma$ the following is true. Let $t_1, t_2 \in \{a, \varepsilon\}$, $b_1, b_2 \in \Sigma^\varepsilon$, and $q_1, q_2 \in Q$. $q_1 = q_2$, $b_1 = b_2$, and $t_1 = t_2$ if $q_1 \in \delta(q, t_1 : b_1)$ and $q_2 \in \delta(q, t_2 : b_2)$. ∎

Lemma 3.2. Let $\$ \notin \Sigma$ be an end marker. Given a *finite* regular expression $r \in R$ with $\varepsilon \notin r$ and $\omega_2 \in \Sigma^*$, there exist DFST $\mathscr{A}^-$ and $\mathscr{A}^+$ s.t. for any $\omega, \omega_1 \in \Sigma^*$: $\omega_1 = \omega^-_{r \to \omega_2}$ iff $(\omega\$, \omega_1\$) \in L(\mathscr{A}^-)$; and, $\omega_1 = \omega^+_{r \to \omega_2}$ iff $(\omega\$, \omega_1\$) \in L(\mathscr{A}^+)$.

We briefly describe how $\mathscr{A}^+$ is constructed for $\omega^+_{r \to \omega_2}$, similar is $\mathscr{A}^-$. Given a finite regular expression r, and assume its length bound is n. Let $\Sigma^{\leq n} = \bigcup_{0 \leq i \leq n} \Sigma^i$. Then $\mathscr{A}^+$ is defined as a quintuple $(\Sigma \cup \{\$\}, Q, q_0, F, \delta)$. The set of states $Q = \{q_1, \ldots, q_{|\Sigma^{\leq n}|}\}$ has $|\Sigma^{\leq n}|$ elements, and let $\mathscr{B} : \Sigma^{\leq n} \to Q$ be a bijection. Let $q_0 = \mathscr{B}(\varepsilon)$ be the initial state and the only final state. A transition $(q, q', a : b)$ is defined as follows for any $q \in Q$ and $a \in \Sigma \cup \{\$\}$, letting $\beta = \mathscr{B}^{-1}(q)$: (case 1) if $a \neq \$$ and $|\beta| < n$, then $b = \varepsilon$ and $q' = \mathscr{B}(\beta a)$; or (case 2) if $a \neq \$$ and $|\beta| = n$: if $\beta \notin r\Sigma^*$, then $b = \beta[0]$ and $q' = \mathscr{B}(\beta[1 : |\beta|]a)$; otherwise, let $\beta = \mu\nu$ where μ is the longest match of r, then $b = \omega_2$ and $q' = \mathscr{B}(\nu a)$; or (case 3) if $a = \$$, then $b = \beta^+_{r \to \omega_2}\$$ and $q' = q_0$.

Intuitively, the above algorithm simulates the left-most matching. It buffers the current string processed so far, and the buffer size is the length bound of r. Once the buffer is full (case 2), it examines the buffer and checks if there is a match. If not, it emits the first character and produces it as output; otherwise, it produces ω_2 on the output tape. Clearly, $\mathscr{B}$ is feasible because of the bounded length of r.

4 Procedural Replacement

The modeling of procedural replacement is much more complex than that of the declarative semantics. The general idea is to compose a number of finite state transducers for generating and filtering begin and end markers for the regular pattern in the input word. We start with the reluctant semantics. Given reluctant replacement $S^-_{r \to \omega}$, the modeling consists of four steps.

4.1 Modeling Left-Most Reluctant Replacement

Step 1 (DFST Marker for End of Regular Pattern): The objective of this step is to construct a DFST (called $\mathscr{A}_1$) that marks the end of regular pattern r, given $S^-_{r \to \omega}$. We first construct a deterministic FSA $\mathscr{A}$ that accepts r s.t. $\mathscr{A}$ does not have any ε transition. We use (q, a, q') to denote a transition from

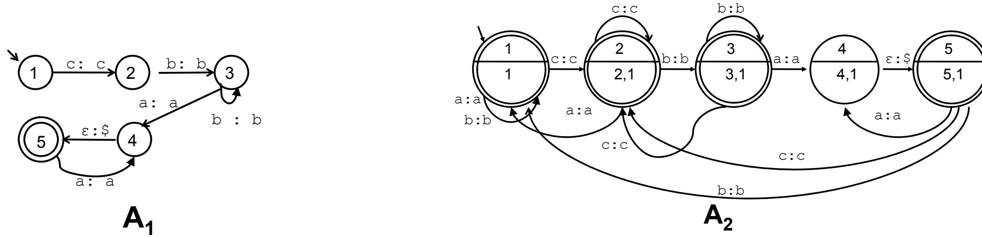


Figure 2: DFST End Marker

state q to q' that is labeled with $a \in \Sigma$. Then we modify each final state f of FSA as below: (1) make f a non-final state, (2) create a new final state f' and establish a transition ε from f to f', (3) for any outgoing transition (f,a,s) create a new transition (f',a,s) and remove that outgoing transition from f (keeping the ε transition). Thus the ε transition is the only outgoing transition of f. Then convert the FSA into a DFST $\mathscr{A}_1$ as below: for an ε transition, its output is \$ (end marker); for every other transition, its output is the same as the input symbol.

Example 4.1. A_1 in Figure 2 is the DFST generated for regular expression cb^+a^+. ∎

Step 2 (Generic End Marker): Note that on the input tape, $\mathscr{A}_1$ only accepts r. We would like to generalize $\mathscr{A}_1$ so that the new FST (called $\mathscr{A}_2$) will accept any word on its input tape. For example, A_2 in Figure 2 is a generalization of A_1, and $(ccbbaa, ccbba\$a\$) \in L(A_2)$.

Step 2 is formally defined as follows. Given $\mathscr{A}_1 = (\Sigma \cup \{\$\}, Q_1, q_0^1, F_1, \delta_1)$ as described in Step 1, $\mathscr{A}_2$ is a quintuple $(\Sigma \cup \{\$\}, Q_2, q_0^2, F_2, \delta_2)$. A *labeling function* $\mathscr{B} : Q_2 \to 2^{Q_1}$ is a bijection s.t. $\mathscr{B}(q_0^2) = \{q_0^1\}$. For any $t \in Q_2$ and $a \in \Sigma$: $t' \in \delta_2(t, a:a)$ iff $\mathscr{B}(t') = \{s' \mid \exists s \in \mathscr{B}(t) \ s.t.\ s' \in \delta_1(s, a:a)\} \cup \{q_0^1\}$. Clearly, $\mathscr{B}$ models a collection of states in $\mathscr{A}_1$ that can be reached by the substrings consumed so far on $\mathscr{A}_2$. Note that there is at most one state reached by a substring, because $\mathscr{A}_1$ is deterministic. Hence, the collection of states is always finite. The handling of the only ε transition in $\mathscr{A}_2$ is similar.

Example 4.2. A_2 in Figure 2 is the result of applying the above algorithm on A_1. Clearly, for A_2, $\mathscr{B}(1) = \{1\}$, $\mathscr{B}(2) = \{2,1\}$, and $\mathscr{B}(3) = \{3,1\}$. Running $(ccbb, ccbb)$ on A_2 results a partial run to state 3. For $(ccbb, ccbb)$, there are five substring pairs to be observed: $(ccbb, ccbb)$, (cbb, cbb), (bb, bb), (b, b), and $(\varepsilon, \varepsilon)$. Among them, only (cbb, cbb) and $(\varepsilon, \varepsilon)$ can be extended to match r (i.e., cb^+a^+). Clearly, if run them on A_1, they would result in partial runs that end at states 3 (by (cbb, cbb)) and 1 (by $(\varepsilon, \varepsilon)$). This is the intuition of having $\mathscr{B}(3) = \{3,1\}$ in A_2. The labeling function $\mathscr{B}$ keeps track of the potential substrings of match by recording those states of A_1 that could be reached by the substrings. ∎

The following lemma states that $\mathscr{A}_2$ inserts an end marker \$ after each occurrence of regular pattern r, and there are no duplicate end markers inserted (even when empty string $\varepsilon \in r$).

Lemma 4.3. For any $r \in R$ there exists a DFST $\mathscr{A}_2$ s.t. for any $\omega \in \Sigma^*$, there is one and only one $\omega_2 \in (\Sigma \cup \{\$\})^*$ with $(\omega, \omega_2) \in L(\mathscr{A}_2)$ and $\omega = \pi(\omega_2)$ such that ω_2 satisfies the following: for any $0 \le x < |\omega_2|$, $\omega_2[x] = \$$ iff $\pi(\omega_2[0,x]) \in \Sigma^* r$; and for any $1 \le x < |\omega_2|$, if $\omega_2[x] = \$$, then $\omega_2[x-1] \neq \$$.

Step 3 (Begin Marker of Regular Pattern): From $\mathscr{A}_2$ we can construct a *reverse* transducer $\mathscr{A}_3$ by reversing all transitions in $\mathscr{A}_2$ and replacing the end marker \$ with the begin marker #. Then create a new initial state s_0, add ε transitions from s_0 to each final state in $\mathscr{A}_2$, and make the original initial state of $\mathscr{A}_2$ the final state in $\mathscr{A}_3$. For example, the A_3 shown in Figure 3 is a reverse of A_2 in Figure 2. Clearly, $(aabbcc, \#a\#abbcc) \in L(A_3)$, and A_3 marks the beginning for pattern $r = a^+b^+c$.

Lemma 4.4. For any $r \in R$ there exists an FST $\mathscr{A}_3$ s.t. for any $\mu \in \Sigma^*$, there exists one and only one $\nu \in (\Sigma \cup \{\#\})^*$ with $(\mu, \nu) \in L(\mathscr{A}_3)$. ν satisfies the following: (i) $\mu = \pi(\nu)$, and, (ii) for $0 \le i < |\nu|$: $\nu[i] = \#$ iff $\pi(\nu[i, |\nu|]) \in r\Sigma^*$, and (iii) for $1 \le i < |\nu|$: if $\nu[i] = \#$ then $\nu[i-1] \neq \#$,

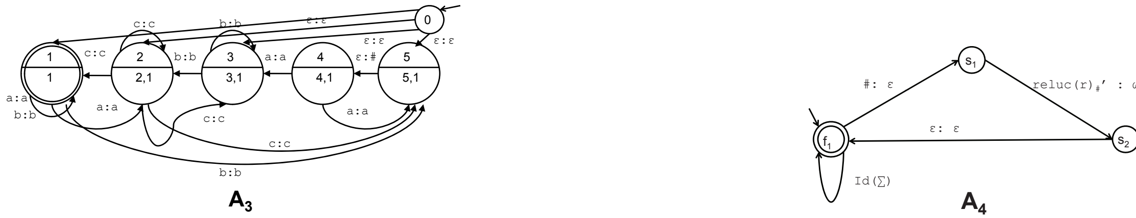


Figure 3: Begin Marker and Reluctant Replacement Transducers

The beauty of the nondeterminism is that $\mathscr{A}_3$ can always make the "smart" decision to enforce there is one and only one run which "correctly" inserts the label #. Any incorrect insertion will never reach a final state. The nondeterminism gives $\mathscr{A}_3$ the "look ahead" ability.

Step 4 (Reluctant Replacement): Next we define an automaton for implementing the reluctant replacement semantics. Given a DFSA $\mathscr{M}$, let $\mathscr{M}_1$ be the new automaton generated from $\mathscr{M}$ by removing all the outgoing transitions from each final state of $\mathscr{M}$. We have the following result: $L(\mathscr{M}_1) = \{s \mid s \in L(\mathscr{M}) \wedge \forall s' \prec s : s' \notin L(\mathscr{M})\}$. Clearly $\mathscr{M}_1$ implements the "shortest match" semantics. Given $r \in R$, let $\texttt{reluc}(r)$ represent the result of applying the above "reluctant" transformation on r.

We still need to filter the extra begin markers during the replacement process. Given a regular language $\mathscr{L} = \texttt{reluc}(r)$, let $\mathscr{L}_\#$ represent the language generated from $\mathscr{L}$ by nondeterministically inserting #, i.e., $\mathscr{L}_\# = \{\mu \mid \mu \in (\Sigma \cup \{\#\})^* \wedge \pi(\mu) \in \mathscr{L}\}$. Clearly, to recognize $\mathscr{L}_\#$, an automaton can be constructed from $\mathscr{A}$ (which accepts $\mathscr{L}$) by attaching a self loop transition (labeled with #) to each state of $\mathscr{A}$. Let $\mathscr{L}_\#' = \mathscr{L}_\# \cap \overline{\Sigma_2^* \#}$ (this is to avoid removing the begin marker for the next match). Now given regular language $\mathscr{L}_\#'$ and $\omega \in \Sigma^*$, it is straightforward to construct an FST $\mathscr{A}_{\mathscr{L}_\#' \times \omega}$ s.t. $(\mu, \nu) \in L(\mathscr{A}_{\mathscr{L}_\#' \times \omega})$ iff $\mu \in \mathscr{L}_\#'$ and $\nu = \omega$. Intuitively, given any μ (interspersed with #) that matches r, the FST replaces it with ω.

An automaton $\mathscr{A}_4$ (as shown in Figure 3) can be defined. Intuitively, $\mathscr{A}_4$ consumes a symbol on both the input tape and output tape unless encountering a begin marker #. Once a # is consumed, $\mathscr{A}_4$ enters the replacement mode, which replaces the shortest match of r with ω (and also removes extra # in the match). Thus, piping it with $\mathscr{A}_3$ directly leads to the precise modeling of reluctant replacement.

Lemma 4.5. Given any $r \in R$ and $\omega \in \Sigma^*$, and let $\mathscr{A}_r$ be $\mathscr{A}_3 || \mathscr{A}_4$, then for any $\omega_1, \omega_2 \in \Sigma^*$: $(\omega_1, \omega_2) \in L(\mathscr{A}_r)$ iff $\omega_2 = \omega_{1}{}_{r \to \omega}^{-}$.

4.2 Modeling Left-Most Greedy Semantics

Handling the greedy semantics is more complex. We have to insert both begin and end markers for the regular pattern and then apply a number of filters to ensure the longest match. The first action is to insert begin markers using $\mathscr{A}_3$ as described in the previous section. Then the second action is to insert an end marker \$ *nondeterministically* after each substring matching r. Later, additional markers will be filtered, and improper marking will be rejected. We call this FST $\mathscr{A}_2'$. Given $r \in R$ and $\omega \in \Sigma^*$, $\mathscr{A}_2'$ can be constructed so that for any $\omega_1 \in \Sigma_2^*$ and $\omega_2 \in \Sigma_2^*$: $(\omega_1, \omega_2) \in L(\mathscr{A}_2')$ iff (i) $\pi(\omega_1) = \pi(\omega_2)$, and (ii) for any $0 \leq i < |\omega_2|$, $\pi(\omega_2[0,i]) \in \Sigma^* r$ *if* $\omega_2[i] = \$$, and (iii) for any $1 \leq i < |\omega_2|$, if $\omega_2[i] = \$$ then $\omega_2[i-1] \neq \$$. Notice that $\mathscr{A}_2'$ is different from $\mathscr{A}_2$ in that the \$ after a match of r is *optional*. Clearly, $\mathscr{A}_2'$ can be modified from $\mathscr{A}_2$ by simply adding an $\varepsilon : \varepsilon$ transition from f (old final state) to f' (new final state) in $\mathscr{A}_2$, e.g., to add an $\varepsilon : \varepsilon$ transition from state 4 to 5 in A_2 in Figure 2. Also #:# transitions are needed for each state to keep the # introduced by $\mathscr{A}_3$.

Then we need a filter to remove extra markers so that every \$ is paired with a #. Note we do not yet make sure that between the pair of # and \$, the substring is a match of r. We construct the AFST

as follows. Let $\mathscr{A}_f = (\Sigma_2, Q, q_0, F, \delta)$. Q has two states q_0 and q_1. $F = \{q_0\}$. The transition function δ is defined as below: (i) $\delta(q_0, Id(\Sigma)) = \{q_0\}$, (ii) $\delta(q_0, \$: \varepsilon) = \{q_0\}$, (iii) $\delta(q_0, \# : \#) = \{q_1\}$, (iv) $\delta(q_1, \# : \varepsilon) = \{q_1\}$, (v)$\delta(q_1, Id(\Sigma)) = \{q_1\}$, (vi) $\delta(q_1, \$: \$) = \{q_0\}$.

Now we will apply three FST filters (represented by three identity relations $Id(L_1)$, $Id(L_2)$, and $Id(L_3)$), for filtering the nondeterministic end marking. L_1, L_2, and L_3 are defined as below:

$$\begin{aligned} L_1 &= \overline{\Sigma_2^* \#(\bar{r} \cap \Sigma^*)\$\Sigma_2^*} && (1) \\ L_2 &= \overline{\Sigma_2^*[\hat{}\#](\$ \cap r_{\#,\$})} \cap \overline{\Sigma_2^*[\hat{}\#](\$\Sigma\Sigma_2^* \cap r_{\#,\$}\Sigma_2^*)} && (2) \\ L_3 &= \overline{\Sigma_2^* \#(r_{\#,\$} \cap (\Sigma^*\$(\Sigma^+)_{\#,\$}))\Sigma_2^*} && (3) \end{aligned}$$

The intuition of L_1 is to make sure that the substring between each pair of # and \$ is a match of r. The motivation of L_2 is for preventing removing too many # symbols by $\mathscr{A}_f$ (due to improper insertion of end markers by $\mathscr{A}_2'$). $Id(L_2)$ handles two cases: (1) to avoid removing the begin markers at the end of input word if the pattern r includes ε; and (2) to avoid removing begin markers for the next instance of r. Consider the following example for case (2): given $S^+_{a^* \to c}$ and the input word *bab*, the correct marking of begin and end markers should be #\$*b*#*a*\$#\$*b*#\$ (which leads to *cbccbc* as output). However the following incorrect marking could pass $Id(L_1)$ and $Id(L_3)$, if not enforcing the $Id(L_2)$ filter: #\$*b*#*a*\$*b*#\$. The cause is that an ending marker \$ (e.g., the one before the last *b*) may trigger $\mathscr{A}_f$ to remove a good begin marker # that precedes an instance of r (i.e., ε). Filter $Id(L_2)$ is thus defined for preventing such cases.

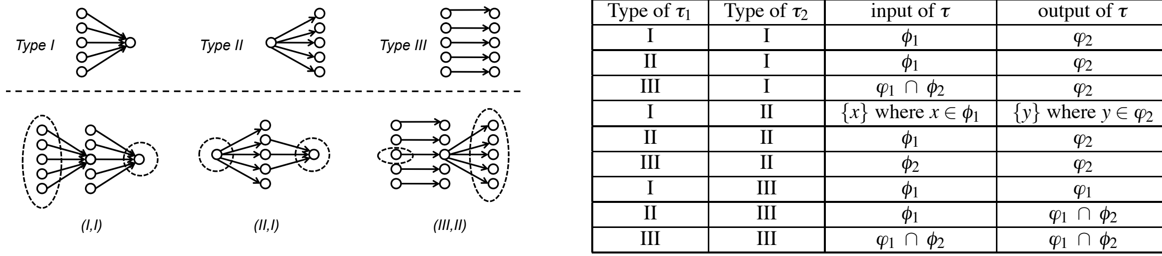
Finally, L_3 is defined for ensuring longest match. Note that filter $Id(L_3)$ will be applied after $Id(L_1)$ and $Id(L_2)$ which have guaranteed the pairing of markers and the proper contents between each pair of markers. L_3 eliminates cases where starting from # there is a substring (when projected to Σ) matches r and the string contains at least one \$ inside (implying that there is a longer match than the substring between the # and its matching \$). Note that $(\Sigma^+)_{\#,\$}$ refers to a word in Σ^+ interspersed with begin/end markers, i.e., for any $\omega \in (\Sigma^+)_{\#,\$}$, $|\pi(\omega)| > 0$. We also need an FST $\mathscr{A}_4'$, which is very similar to $\mathscr{A}_4$. $\mathscr{A}_4'$ enters (and leaves) the replacement mode, once it sees the begin (and the end) marker. Then we have the following:

Lemma 4.6. Given any $r \in R$ and $\omega \in \Sigma^*$, let $\mathscr{A}_g$ be $\mathscr{A}_3 || \mathscr{A}_2' || \mathscr{A}_f || \mathscr{A}_{Id(L_1)} || \mathscr{A}_{Id(L_2)} || \mathscr{A}_{Id(L_3)} || \mathscr{A}_4'$, then for any $\omega_1, \omega_2 \in \Sigma^*$: $\omega_2 = {\omega_1}^+_{r \to \omega}$ iff $(\omega_1, \omega_2) \in L(\mathscr{A}_g)$.

5 SISE Constraint and SUSHI Solver

This work is implemented as part of a constraint solver called SUSHI [4], which solves SISE (Simple Linear String Equation) constraints. Intuitively, a SISE equation can be regarded as a variation of word equation [13]. It is composed of word literals, string variables, and various frequently seen string operators such as substring, concatenation, and regular replacement. To solve SISE, an automata based approach is taken, where a SISE is broken down into a number of atomic string operations. Then the solution process consists of a number of backward image computation steps. We now briefly describe the part related to regular replacement.

It is well known that projecting an FST to its input tape (by removing the output symbol from each transition) results in a standard finite state machine. Similar applies to the projection to output tape. We use `input`$(\mathscr{A})$ and `output`$(\mathscr{A})$ to denote the input and output projection of an FST $\mathscr{A}$. Given an atomic SISE constraint $x_{r \to \omega} \equiv r_2$, the solution pool of x (backward image of the constraint) is defined as $\{\mu \mid \mu_{r \to \omega} \in L(r_2)\}$. Given a regular expression ν, the forward image of $\nu_{r \to \omega}$ is defined as $\{\mu \mid \mu \in \alpha_{r \to \omega} \text{ and } \alpha \in \nu\}$. Clearly, let $\mathscr{A}$ be the corresponding FST of $x_{r \to \omega}$, the backward image can be computed using `input`$(\mathscr{A} || Id(r_2))$. Similarly, given $\mu_{r \to \omega}$, the forward image is `output`$(Id(\mu) || \mathscr{A})$.



Type of τ_1	Type of τ_2	input of τ	output of τ
I	I	ϕ_1	φ_2
II	I	ϕ_1	φ_2
III	I	$\varphi_1 \cap \phi_2$	φ_2
I	II	$\{x\}$ where $x \in \phi_1$	$\{y\}$ where $y \in \varphi_2$
II	II	ϕ_1	φ_2
III	II	ϕ_2	φ_2
I	III	ϕ_1	φ_1
II	III	ϕ_1	$\varphi_1 \cap \phi_2$
III	III	$\varphi_1 \cap \phi_2$	$\varphi_1 \cap \phi_2$

Figure 4: SUSHI FST Transition Set

5.1 Compact Representation of FST

SUSHI relies on `dk.brics.automaton` [15] for FSA operations. We use a self-made Java package for supporting FST operations [16]. Note that there are existing tools related to FST, e.g., the FSA toolbox [16]. In practice, to perform inspection on user input, FST has to handle a large alphabet represented using 16-bit Unicode. In the following, we introduce a compact representation of FST. A collection of FST transitions can be encoded as a *SUSHI FST Transition Set* (SFTS) in the following form:

$$\mathscr{T} = (q, q', \phi : \varphi)$$

where q, q' are the source and destination states, the *input charset* $\phi = [n_1, n_2]$ with $0 \le n_1 \le n_2$ represents a range of input characters, and the *output charset* $\varphi = [m_1, m_2]$ with $0 \le m_1 \le m_2$ represents a range of output characters. $\mathscr{T}$ includes a set of transitions with the same source and destination states: $\mathscr{T} = \{(q, q', a : b) \mid a \in \phi \text{ and } b \in \varphi\}$. For $\mathscr{T} = (q, q', \phi : \varphi)$, however, it is required that if $|\phi| > 1$ and $|\varphi| > 1$, then $\phi = \varphi$. For ϕ and φ, ε is represented using $[-1, -1]$. Thus, there are three types of SFTS (excluding the ε cases), as shown in the following. *Type I:* $|\phi| > 1$ and $|\varphi| = 1$, thus $\mathscr{T} = \{(q, q', a : b) \mid a \in \phi \text{ and } \varphi = \{b\}\}$. *Type II:* $|\phi| = 1$ and $|\varphi| > 1$, thus $\mathscr{T} = \{(q, q', a : b) \mid b \in \varphi \text{ and } \phi = \{a\}\}$. *Type III:* $|\varphi| = |\phi| > 1$, thus $\mathscr{T} = \{(q, q', a : a) \mid a \in \phi\}$. The top-left of Figure 4 gives an intuitive illustration of these SFTS types (which relates the input and output chars).

The algorithms for supporting FST operations (such as union, Kleen star) should be customized correspondingly. In the following, we take FST composition as one example. Let $\mathscr{A} = (\Sigma, Q, q, F, \delta)$ be the composition of $\mathscr{A}_1 = (\Sigma, Q_1, q_0^1, F_1, \delta_1)$ and $\mathscr{A}_2 = (\Sigma, Q_2, s_0^2, F_2, \delta_2)$. Given $\tau_1 = (t_1, t_1', \phi_1 : \varphi_1)$ in $\mathscr{A}_1$ and $\tau_2 = (t_2, t_2', \phi_2 : \varphi_2)$ in $\mathscr{A}_2$, where $\varphi_1 \cap \phi_2 \neq \emptyset$, an SFTS $\tau = (s_1, s_2, \phi : \varphi)$ is defined for $\mathscr{A}$ s.t. $s_1 = (t_1, t_2)$, $s_2 = (t_1', t_2')$, and the input/output charset of τ is defined as the table in Figure 4 (note all entries except for (I,II) produce one SFTS only). For example, when both τ_1 and τ_2 are type I, we have $\phi = \phi_1$ and $\varphi = \varphi_2$. The bottom left of Figure 4 shows the intuition of the algorithm. The dashed circles represent the corresponding input/output charset.

5.2 Evaluation

We are interested in whether the proposed technique is efficient and effective in practice. We list here four SISE equations for stress-testing the SUSHI package. Note that each equation is parametrized by an integer n. **eq1**: $x^{+}_{\mathtt{a}^{+} \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$; **eq2**: $x^{-}_{\mathtt{a}^{+} \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$; **eq3**: $x^{+}_{\mathtt{a}^{*} \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$; **eq4**: $x^{-}_{\mathtt{a}^{*} \to \mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n, 2n\}$. The following table displays the running results when n is 41. (more data in [4]). It displays the max size of FST and FSA used in the solution process.

Equation	FST States	FST Transitions	FSA States	FSA Transitions	Time (Seconds)
eq1(41)	5751	16002	125	207	155.281
eq2(41)	5416	5748	83	124	162.469
eq3(41)	631	1565	2	2	492.281
eq4(41)	126	177	0	0	14.016

The technique scales well in practice. We applied SUSHI in discovering SQL injection vulnerabilities and XSS attacks in FLEX SDK (see technical report [4]). The running cost ranges from 1.4 to 74 seconds on a 2.1Ghz PC with 2GB RAM (with SISE equation size ranging from 17 to 565). [1]

6 Related Work

Recently, string analysis has received much attention in security and compatibility analysis of programs (see e.g., [5, 12]). In general, there are two interesting directions of string analysis: (1) *forward analysis*, which computes the image (or its approximation) of the program states as constraints on strings; and, (2) *backward analysis*, which usually starts from the negation of a property and computes backward. Most of the related work (e.g., [2, 11, 18]) falls into the category of forward analysis. This work can be used for both forward and backward image computation. Compared with forward analysis, it is able to generate attack signatures as evidence of vulnerabilities.

Modeling regular replacement distinguishes our work from several related work in the area. For example, one close work to ours is the HAMPI string constraint solver [9]. HAMPI supports solving string constraints with context-free components, which are unfolded to regular language. HAMPI, however, supports neither constant string replacement nor regular replacement, which limits its ability to reason about sanitation procedures. Similarly, Hooimeijer and Weimer's work [6] in the decision procedure for regular constraints does not support regular replacement. A closer work to ours is Yu's automata based forward/backward string analysis [18]. Yu uses a language based replacement [17], which introduces imprecision in its over-approximation. Conversely, our analysis considers the delicate differences among the typical regular replacement semantics and provides more accurate analysis. In [1], Bjørner *et al.* uses first order formula on bit-vector to model string operations except replacement. We conjecture that it can be extended by using recursion in their first order framework for defining replaceAll semantics.

FST is the major modeling tool in this paper. It is mainly inspired by [7, 14, 8] in computational linguistics, for processing phonological and morphological rules. In [8], an informal discussion was given for the semantics of left-most longest matching of string replacement. This paper has given the formal definition of replacement semantics and has considered the case where ε is included in the search pattern. Compared with [7] where FST is used for processing phonological rules, our approach is lighter given that we do not need to consider the left and right context of re-writing rules in [7]. Thus more DFST can be used, which certainly has advantages over NFST, because DFST is less expressive. For example, in modeling the reluctant semantics, compared with [7], our algorithm does not have to non-deterministically insert begin markers and it does not need extra filters, thus more efficient. It is interesting to compare the two algorithms and measure the gain in performance by using more DFST in modeling, which remains one of our future work.

Limitation of the Model: It is shown in [4] that solving SISE constraint is decidable (with worst complexity 2-EXPTIME). This may seem contradictory with the conclusion in [1]. The decidability is achieved by restricting SISE as described below. SISE requires that each variable appears at most once and all variables must be appear in LHS. This permits a simple recursive algorithm that reduces the solution process into a number of backward image computation steps. However, it may limit the expressiveness of SISE in certain application scenario. SISE supports regular replacement, substring, concatenation operators, however, it does not support operators related to string length, e.g., indexOf and length operators. It is interesting to extend the framework to support mixed numerical and string operators, e.g., encoding numeric constraints using automata as described by Yu *et al.* in [18], or translating string constraints to first order formula on bit-vectors as shown by Bjørner *et al.* [1].

[1] SISE equation size is measured by the combined length of constant words, variables, and operators included in the equation.

7 Conclusion

This paper presents the finite state transducer models of various regular substitutions, including the declarative, finite, reluctant, and greedy replacement. A compact FST representation is implemented in a constraint solver SUSHI. The presented technique can be used for analyzing programs that process text and communicate with users using strings. Future directions include modeling mixture of greedy and reluctant semantics, handling hybrid numeric/string constraints, and context free components.
Acknowledgment: This paper is inspired by the discussion with Fang Yu, Tevfik Bultan, and Oscar Ibarra. We thank the anonymous reviewers for very constructive comments that help improve the paper.

References

[1] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools AND Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pages 322–336. Springer, 2009.

[2] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[3] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A Static Analysis Framework for Detecting SQL Injection Vulnerabilities. In *Proceedings of 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, pages 87 – 96, 2007.

[4] Xiang Fu, Chung chih Li, and Kai Qian. On simple linear string equations. `http://people.hofstra.edu/Xiang_Fu/XiangFu/publications/techrep09b.pdf`, 2009.

[5] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 697–698, 2004.

[6] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languagees. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198, 2009.

[7] Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistic*, 20(3):331–378, 1994.

[8] L. Karttunen, J p. Chanod, G. Grefenstette, and A. Schiller. Regular expressions for language engineering. *Natural Language Engineering*, 2:305–328, 1996.

[9] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proc. of 2009 International Symposium on Testing and Analysis (ISSTA'09)*, 2009.

[10] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[11] Christian Kirkegaard and Anders Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*, August 2006.

[12] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium*. USENIX Association, 2005.

[13] M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.

[14] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.

[15] A. Møller. The dk.brics.automaton package. *available at* `http://www.brics.dk/automaton/`.

[16] Gertjan van Noord. Fsa utilities: A toolbox to manipulate finite-state automata on simple linear string equations. `http://www.let.rug.nl/~vannoord/papers/fsa/fsa.html`, 1998.

[17] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In *Proc. of the 15th SPIN Workshop on Model Checking Software*, pages 306–324, 2008.

[18] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Proceedings of the 15th International Conference on Tools AND Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 322–336. Springer, 2009.