On Precise Modeling of Regular Replacement

Xiang Fu¹ and Chung-Chih Li²

¹ Hofstra University, Xiang.Fu@hofstra.edu
² Illinois State University, cli2@ilstu.edu

Abstract. This paper studies the precise modeling of various semantics of regular substitution, such as the declarative, finite, greedy, and reluctant replacement, using finite state transducers (FST) as filters. By projecting an FST of regular replacement 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. We report several interesting discoveries, e.g., certain fragments of the general problem can be handled using less expressive deterministic finite state transducers. A compact representation of FST and the customized algorithms for FST operations are developed. It is shown that the precise modeling of regular replacement can help discover delicate security vulnerabilities of software systems with very few false positives.

1 Introduction

Regular replacement is a frequently used tool for user input sanitation, e.g., escaping control characters, removing javascript tags in web 2.0 applications, and filtering improper language in blog posts. With regular replacement, however, programmers can still introduce bugs into sanitation code. It is desirable to automatically discover those defects that lead to vulnerabilities. This paper discusses the formal modeling of various semantics of regular replacements for program analysis. We show that such modeling is the key for solving string constraints, which can be used in both model checking and symbolic execution.

Consider one example called BadLogin, originally introduced in [5]. The java application has a log-in page, which applies a number of sanitation procedures for each user input: e.g., to replace the single quote character with its escaping form (a sequence of two single quotes), and to limit the size of each user string to 16 characters. The sanitation operation can be represented by the following notation: $x_{\cdot \cdot}^{'\cdot'}[0,16]$, where $x_{\cdot \cdot}^{'\cdot'}$ represents replacement of ' with $''$, and $[0][16]$ is the substring operator. String replacement is an essential component of Simple Linear String Equation (SLSE) [8], which is used for capturing attack patterns. The following is one SLSE for attacking the BadLogin example.

\[ \text{uname}=\cdot \circ x_{\cdot \cdot}^{'\cdot'}[0,16] \circ ' \text{ AND pwd}=\cdot \circ y_{\cdot \cdot}^{'\cdot'}[0,16] \circ ' \equiv \text{uname}=((\text{'''})\text{''}\text{''} \text{ OR } \text{ 'uname>}' \text{'} \text{'} \]

The above SLSE is a concatenation of five string terms, with $\circ$ denoting concatenation. It targets at the WHERE clause of a SQL query in BadLogin which
verifies user identity. The equation asks: After applying the string sanitation on the username (represented by \(x\)) and password (represented by \(y\)) which are entered by a user, is it possible to bypass the password checking and make the WHERE clause essentially a tautology (by OR \(uname<>''\)). After solving the SLSE equation (using SUSHI package [7]), we are able to generate the shortest attack strings, e.g., \(x = a'\ldots'\) and \(y = '' \text{ OR } uname<>''\).

The key to solving SLSE equations is how regular replacement can be precisely modeled. Once the replacement operators can be defined using a finite state transducer, the solution of an atomic SLSE equation on replacement can be easily achieved by projecting finite state transducer (FST) to its input tape, which results in a finite state machine. This paper presents the precise modeling of various regular string replacement semantics and a compact representation (and customized algorithms) for FST. The algorithm is implemented in a Java package called SUSHI, which is applied for analyzing programs that manipulate strings. It is shown that precise modeling of string replacement can help reduce false positives during security analysis.

The rest of the paper is organized as follows. §C2 provides the formal definition of string replacement semantics. §C3 introduces several variations of finite state transducer (FST) and its customized compact representation in SUSHI. §C4 and §C5 present the modeling of various regular replacement semantics, including finite, greedy, and reluctant semantics, using finite state transducer. §C6 introduces tool support and presents experimental data. §C7 concludes and discusses related work.

2 Regular Replacement Semantics

Three different semantics of regular replacement, namely greedy, reluctant, and possessive, are provided in java.util.regex [16], with the greedy semantics being default. 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. The possessive semantics is similar to the reluctant semantics and does not allow matching algorithm to trace back. In the following, we consider two simplified semantics which are very close to the real Java regex semantics: the completely greedy and the completely reluctant semantics. They can be enforced syntactically by requiring that only greedy or reluctant operators are used. From the theoretical point of view, it is also interesting to define a declarative semantics for string replacement.

In the following, we introduce some notations first. Let \(\Sigma\) represent the alphabet and \(R\) the set of regular expressions over \(\Sigma\). If \(\omega \in \Sigma^*\), \(\omega\) 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 \(\omega\) 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. Regular expressions are built using quantifiers such as *, +, and ?.
Definition 1. Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\epsilon \notin r$). The declarative replacement, denoted as $\gamma_{r \rightarrow \omega}$, is defined as:

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

Definition 2. Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\epsilon \notin r$). The reluctant replacement, denoted as $\gamma_{r \rightarrow \omega}$, is defined recursively as $\gamma_{r \rightarrow \omega} = \{ \nu r \omega \nu'_{r \rightarrow \omega} \}$ 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 $\nu_2 \notin \epsilon$ then $\nu_2 \beta_1 \notin r$ and $\nu_2 \beta_1 \mu_2 \notin r$; and, if $\beta_2 \notin \epsilon$ then $\beta_1 \notin r$.

Definition 3. Let $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\epsilon \notin r$). The greedy replacement, denoted as $\gamma_{r \rightarrow \omega}^+$, is defined recursively as $\gamma_{r \rightarrow \omega}^+ = \{ \nu r \omega \mu_{r \rightarrow \omega}^+ \}$ 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 $\nu_2 \notin \epsilon$ then $\nu_2 \beta_1 \notin r$ and if $\mu_1 \notin \epsilon$ then $\nu_2 \beta_1 \mu_2 \notin r$.

Intuitively, both the completely reluctant and greedy semantics enforce leftmost 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 replaces the longest match, and the reluctant replacement replaces the shortest. The declarative semantics does not enforce the left-most matching and could produce multiple words as the result of replacement.

The following two examples illustrate the difference of the above semantics:
let $\gamma = aaa$ with $a \in \Sigma$; (i) $\gamma_{aa \rightarrow b} = \{ ba \}$, $\gamma_{aa \rightarrow b}^+ = \{ ba \}$, $\gamma_{a \rightarrow b} = \{ ba \}$, $\gamma_{a \rightarrow b}^+ = \{ ba \}$.

(ii) $\gamma_{a \rightarrow b}^+ = \{ b, bb, bbb \}$, $\gamma_{a \rightarrow b}^+ = \{ b \}$, and $\gamma_{a \rightarrow b} = \{ bb \}$.

Notice that in the above definitions, $\epsilon \notin r$ is required for simplicity. In practice, precise Java semantics is followed in SUSHI [7] for handling $\epsilon \in r$. For example, in SUSHI, given $\gamma = a$, $r = a^*$, and $\omega = b$, $\gamma_{r \rightarrow \omega} = \{ bab \}$ and $\gamma_{r \rightarrow \omega}^+ = \{ bb \}$. When $\omega \in \gamma_{r \rightarrow \omega}$, we often abuse the notation and write it as $\omega = \gamma_{r \rightarrow \omega}$. Similar applies to $\gamma_{r \rightarrow \omega}^+$, given the following lemma.

Lemma 1. For any $\gamma, \omega \in \Sigma^*$ and $r \in R$ (with $\epsilon \notin r$): $\gamma_{r \rightarrow \omega}^+$ and $\gamma_{r \rightarrow \omega}^+$ are both singleton sets.

3 Finite State Transducer and Declarative Replacement

This section briefly introduces finite state transducer (FST) [10], the major modeling tool for various semantics of regular replacement. We demonstrate the application of FST first using declarative replacement. Then we introduce a compact representation of FST in practice.

Definition 4. Let $\Sigma^*$ denote $\Sigma \cup \{ \epsilon \}$. 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 $\Sigma$ is the alphabet, $Q$ the set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ the set of final states, and $\delta$ is the transition function, which is a total function of type $Q \times \Sigma^* \times \Sigma^* \rightarrow 2^Q$. 
A transition from a state \( q \) to \( q' \) (labeled with input char \( a \) and output char \( b \)) is written as \( q' \in \delta(q, a : b) \). But sometimes we also write it as a tuple \( (q, q', a : b) \) for convenience. It is well known that each finite state transducer accepts a regular relation which is a subset of \( \Sigma^* \times \Sigma^* \). Given \( \omega_1, \omega_2 \in \Sigma^* \) and an FST \( M \), we say \( (\omega_1, \omega_2) \in L(M) \) if the word pair is accepted by \( M \). Clearly, regular relation is closed under union, concatenation, and Kleen star, but not closed under complementation and intersection. FST is also closed under composition. Let \( M_3 \) be the composition of two FSTs \( M_1 \) and \( M_2 \), denoted as \( M_3 = M_1 \circ M_2 \). Then \( L(M_3) = \{(\mu, \nu) \mid (\mu, \eta) \in L(M_1) \text{ and } (\eta, \nu) \in L(M_2) \text{ for some } \eta \in \Sigma^*\} \). In the following, we introduce an equivalent definition of FST called augmented finite state transducer.

**Definition 5.** An augmented finite state transducer (AFST) is an FST \((\Sigma, Q, q_0, F, \delta)\) with the transition function augmented to type \( Q \times R \rightarrow 2^Q \), where \( R \) is the set of regular relations over \( \Sigma \).

![Fig. 1. An FST for \( s_{r \rightarrow \omega} \)](image)

In practice, we would often restrict the transition function of an AFST to the following two types: (1) \( Q \times R \times \Sigma^* \rightarrow 2^Q \). In a transition diagram, we label the arc from \( q_i \) to \( q_j \) for transition \( q_j \in \delta(q_i, r : \omega) \) by \( r : \omega \); and (2) \( Q \times \{\text{Id}(r) \mid r \in R\} \rightarrow 2^Q \), where \( \text{Id}(r) = \{(\omega, \omega) \mid \omega \in L(r)\} \). In a transition diagram, an arc of type (2) is labeled as \( \text{Id}(r) \).

Now, we can use an AFST to model the declarative string replacement \( s_{r \rightarrow \omega} \) for any \( \omega \in \Sigma^* \) and \( r \in R \) (with \( \epsilon \notin r \)). Figure 1 presents an AFST that accepts \( \{(s, \eta) \mid s \in \Sigma^* \text{ and } \eta \in s_{r \rightarrow \omega}\} \).

### 3.1 Compact Representation of FST

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, which is inspired by the `dk.bricks.automaton` [15] package. The customized algorithms are implemented in SUSHI [7].

A collection of FST transitions can be encoded as a *SUSHI FST Transition Set* (SFTS) in the following form:

\[
T = (q, q', \phi : \varphi)
\]

where \( q, q' \) are the source and destination states, the *input charset* \( \phi = [n_1, n_2] \) with \( 0 \leq n_1 \leq n_2 \) represents a range of input characters, and the *output charset*...
\( \varphi = [m_1, m_2] \) with \( 0 \leq m_1 \leq m_2 \) represents a range of output characters. \( \mathcal{T} \) includes a set of transitions with the same source and destination states:
\[
\mathcal{T} = \{(q, q', a : b) \mid a \in \phi \text{ and } b \in \varphi\}. 
\]
For \( \mathcal{T} = (q, q', \phi : \varphi) \), however, it is required that if \( |\phi| > 1 \) and \( |\varphi| > 1 \), then \( \phi = \varphi \). For \( \phi \) and \( \varphi \), \( \epsilon \) is represented using \([-1, -1]\). Thus, there are three types of SFTS (excluding the \( \epsilon \) cases), as shown in the following. The top half of Figure 2 gives an intuitive illustration of these SFTS types (which relates the input and output chars).

- **Type I**: \( |\phi| > 1 \) and \( |\varphi| = 1 \), thus \( \mathcal{T} = \{(q, q', a : b) \mid a \in \phi \text{ and } \varphi = \{b\}\} \).

- **Type II**: \( |\phi| = 1 \) and \( |\varphi| > 1 \), thus \( \mathcal{T} = \{(q, q', a : b) \mid b \in \varphi \text{ and } \phi = \{a\}\} \).

- **Type III**: \( |\varphi| = |\phi| > 1 \), thus \( \mathcal{T} = \{(q, q', a : a) \mid a \in \phi\} \).

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 \( \mathcal{A} = (\Sigma, Q, \delta) \) be the composition of \( \mathcal{A}_1 = (\Sigma, Q_1, q_0, F_1, \delta_1) \) and \( \mathcal{A}_2 = (\Sigma, Q_2, s_0, F_2, \delta_2) \). Clearly, \( Q = \{(t, t') \mid t \in Q_1 \wedge t' \in Q_2\}, \) \( q_0 = (q_0, s_0^1) \), \( F = \{(t, t') \mid t \in F_1 \wedge t' \in F_2\} \). Given \( \tau_1 = (t_1, t'_1, \phi_1 : \varphi_1) \) in \( \mathcal{A}_1 \) and \( \tau_2 = (t_2, t'_2, \phi_2 : \varphi_2) \) in \( \mathcal{A}_2 \), where \( \varphi_1 \cap \phi_2 \neq \emptyset \), an SFTS \( \tau = (s_1, s_2, \phi : \varphi) \) is defined for \( \mathcal{A} \) s.t. \( s_1 = (t_1, t_2), s_2 = (t'_1, t'_2) \), and the input/output charset of \( \tau \) is defined as below:

<table>
<thead>
<tr>
<th>Type of ( \tau_1 )</th>
<th>Type of ( \tau_2 )</th>
<th>input of ( \tau )</th>
<th>output of ( \tau )</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>I</td>
<td>( \phi_1 )</td>
<td>( \varphi_2 )</td>
</tr>
<tr>
<td>II</td>
<td>I</td>
<td>( \phi_1 )</td>
<td>( \varphi_2 )</td>
</tr>
<tr>
<td>III</td>
<td>I</td>
<td>( \phi_1 \cap \phi_2 )</td>
<td>( \varphi_2 )</td>
</tr>
<tr>
<td>I</td>
<td>II</td>
<td>( \phi_1 )</td>
<td>( \varphi_2 )</td>
</tr>
<tr>
<td>II</td>
<td>II</td>
<td>( \phi_2 )</td>
<td>( \varphi_2 )</td>
</tr>
<tr>
<td>III</td>
<td>III</td>
<td>( \phi_1 \cap \phi_2 )</td>
<td>( \varphi_2 \cap \phi_2 )</td>
</tr>
</tbody>
</table>

Notice that except when \( \tau_1 \) is type I and \( \tau_2 \) is type II, all cases produce one SFTS only. The \((I, II)\) case produces \(|\phi_1| \times |\varphi_2|\) SFTSes (with each SFTS modeling one standard FST transition). The bottom of Figure 2 shows the intuition of the algorithm. The dashed circles represent the corresponding input/output charset.

Fig. 2. SUSHI FST Transition Set
\( \epsilon \) transitions are handled separately. For example, given \( \tau_1 = (t_1, t'_1, \phi_1 : \epsilon) \) in \( A_1 \) and \( \tau_2 = (t_2, t'_2, \phi_2, \varphi_2) \), one SFTS \( \tau \) for \( A \) can be defined as \( (s_1, s_2, \phi : \epsilon) \) where \( s_1 = (t_1, t_2) \) and \( s_2 = (t'_1, t'_2) \), i.e., only \( A_1 \) moves in the composition.

4 Regular Substitution Restricted to Finite Language

The FST used for modeling the declarative semantics is a nondeterministic finite state transducer (NFST). This section shows that regular replacement with finite language pattern can be modeled using deterministic finite state transducer (DFST), under certain restrictions. We fix some notations first. 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 \( \omega \in L(r) \), \( |\omega| \leq n \). We call a regular expression \( r \) fixed length, if for any \( \omega_1, \omega_2 \in r: |\omega_1| = |\omega_2| \).

**Definition 6.** An FST \( A = (\Sigma, Q, s_0, F, \delta) \) is deterministic if for any \( q \in Q \) and any \( a \in \Sigma \) the following is true: let \( t_1, t_2 \in \{ a, \epsilon \} \), \( b_1, b_2 \in \Sigma^* \), and \( q_1, q_2 \in Q \), then \( q_1 \in \delta(q, t_1 : b_1) \) and \( q_2 \in \delta(q, t_2 : b_2) \) \( \Rightarrow q_1 = q_2 \) and \( b_1 = b_2 \).

Intuitively, for a DFST, at any state \( q \in Q \) the input symbol uniquely determines the destination state and the symbol to consume on output tape. If there is a transition labeled with \( \epsilon \) on the input, then this is the only transition from \( s \). Clearly, given any DFST \( A \) and any \( \omega_1, \omega'_1, \omega_2, \omega'_2 \in \Sigma^* \), if both \( (\omega_1, \omega_2) \) and \( (\omega'_1, \omega'_2) \) have partial runs on \( A \) and \( \omega_1 \) is a strict prefix of \( \omega'_1 \) (written as \( \omega_1 \prec \omega'_1 \)), then \( \omega_2 \prec \omega'_2 \) because the input word uniquely determines the partial run.

**Lemma 2.** Let \( \$ \not\in \Sigma \) be an end marker. Given a finite regular expression \( r \in R \) with \( \epsilon \not\in r \) and \( \omega_2 \in \Sigma^* \), there exist DFST \( A^- \) and \( A^+ \) s.t. for any \( \omega, \omega_1 \in \Sigma^* \):

\[
\omega_1 = \omega_{r-\omega_2}^r \text{ iff } (\omega, \omega_1 \$) \in L(A^-); \text{ and, } \omega_1 = \omega_{r^+}^r_\omega \text{ iff } (\omega \$, \omega_1 \$) \in L(A^+).
\]

We briefly describe how \( A^+ \) is constructed for \( \omega_{r^+}^r_{\omega_2} \), similar is \( A^- \). The DFST \( A^+ \) does not have any transition with \( \epsilon \) on input. 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 \( A^+ \) is defined as a quintuple \( (\Sigma \cup \{ \$ \}, Q, q_0, F, \delta) \). The set of states \( Q = \{ q_1, \ldots, q_{|\Sigma^{|n}|} \} \) has \( |\Sigma^{\leq n}| \) elements, and let \( B : \Sigma^{\leq n} \to Q \) be a bijection. Let \( q_0 = B(\epsilon) \) 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 \( \beta = B^{-1}(q) \) (case 1) if \( a \neq \$ \) and \( |\beta| < n - 1 \), then \( b = \epsilon \) and \( q' = B(3a) \); or (case 2) if \( a \neq \$ \) and \( |\beta| = n - 1 \); if \( \beta \not\in r \Sigma^* \), then \( b = \beta[0] \) and \( q' = B(\beta[1 : |\beta|][a]) \); otherwise, let \( \beta = \mu \nu \) where \( \mu \) is the longest match of \( r \), then \( b = \omega_2 \) and \( q' = B(\nu a) \); or (case 3) if \( a = \$ \), then \( b = \beta_{r=\omega_2}^r \$ \) and \( q' = q_0 \).

Shown in Figure 3 is a part of the DFST for \( s_{(ab)(abb)}^r \). Clearly, the length bound of regular pattern is 3. For input word \( abab\$ \), the only matching output word is \( acc\$ \), as recognized by the DFST (visiting states 1, 2, 3, 4, 2, 1).

**Corollary 1.** If \( r \in R \) is fixed length, then for any \( \omega, \omega_2 \in \Sigma^* : \omega_{r^+\omega_2}^r = \omega_{r-\omega_2}^r \).
It is then interesting to know if the general problem of regular substitution can be represented using DFST. The conclusion is false. Assume there exists a DFST \( A \) s.t. \( L(A) = \{ (\omega, \omega_1) \mid \omega_1 = \omega_a^+ b - \epsilon c \} \). Clearly, \( (a^n, a^n) \in L(A) \) and \( (a^n b, c) \in L(A) \). Since \( A \) is deterministic, and \( a^n \) is a prefix of \( a^n b \), it easily leads to the contradiction that \( a^n \) is a prefix of \( c \). Thus the assumption cannot be true. Using the same technique, one can show that \( \{ (\omega, \omega \$) \mid \omega \in \Sigma^* \} \) is not accepted by any DFST.

5 Reluctant and Greedy Replacement

This section introduces the modeling of the completely reluctant and greedy replacement. The general idea is to compose a number of NFSTs and DFSTs for generating and filtering begin and end markers for the regular pattern in the input word. The modeling of the reluctant semantics consists of four steps, and similar is the greedy semantics.

5.1 Modeling Left-most Reluctant Replacement

We fix some notations first. A finite state machine (FSA) is denoted using a quintuple \( (\Sigma, Q, q_0, F, \delta) \). Clearly, \( \Sigma, Q, q_0, F \) are the alphabet, set of states, initial state, and set of final states, respectively. \( \delta : Q \times \Sigma \rightarrow 2^Q \) is the transition function. We sometimes abuse the notation and use \( (q, a, q') \in \delta \) to denote a transition from state \( q \) to \( q' \) that is labeled with \( a \in \Sigma \).

**Step 1 (DFST for Marking End of Regular Pattern):** Given reluctant replacement \( S_{\rightarrow \omega} \), the objective of this step is to construct a DFST that marks the end of regular pattern \( r \). We first construct a deterministic FSA \( A \) that accepts \( r \) s.t. \( A \) does not have any \( \epsilon \) transition and has only one final state \( f \). Then we modify the 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 \( \epsilon \) 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 \( \epsilon \) transition). Thus the \( \epsilon \) transition is the only outgoing transition of \( f \). Then convert the resulting automaton into a DFST \( A_1 \) as below: for the \( \epsilon \) transition, the output symbol is \( \$ \) (end marker); for all other transitions, output is the identical symbol.
Example 1. $A_1$ in Figure 4 is the DFST generated for regular expression $cb^+a^+$.

Step 2 (Generic End Marker): Note that the input tape of $A_1$ only accepts $r$. We would like to generalize $A_1$ so that the new FST would accept any word on its input tape. For example, $A_2$ in Figure 4 is a generalization of $A_1$, and $(ccbb$a, ccbb$\$) $\in L(A_2)$.

Step 2 is formally defined as follows. Given $A_1 = (\Sigma, Q_1, q_0^1, F_1, \delta_1)$ as described in Step 1, $A_2$ is a quintuple $(\Sigma, Q_2, q_0^2, F_2, \delta_2)$. A labeling function $B : Q_2 \rightarrow 2^{Q_1}$ is a bijection s.t. $B(q_0^2) = \{q_0^1\}$. For any $t \in Q_2$ and $a \in \Sigma$: $(t, t', a : a) \in \delta_2$ iff $B(t') = \{s' \mid \exists s \in B(t) \land (s, s', a : a) \in \delta_1\} \cup \{q_0^1\}$. Clearly, $B$ models a collection of states in $A_1$ that can be reached by the substrings consumed so far on $A_2$. Note that there is at most one state reached by a substring, because $A_1$ is deterministic. The collection of states is always finite. The handling of the only $\$ transition in $A_2$ is similar. A state $t \in F_2$ iff there is no outgoing transition from $t$ that is labeled with $\$ on the output part.

Example 2. $A_2$ in Figure 4 is the result of applying the above algorithm on $A_1$. Clearly, for $A_2$, $B(1) = \{1\}$, $B(2) = \{2, 1\}$, and $B(3) = \{3, 1\}$. Running $(ccbb, ccbb)$ on $A_2$ results a partial run to state 3. There are five substrings to be traced for potential matching of $r$: $(ccbb, ccbb)$, $(cbb, cbb)$, $(bb, bb)$, $(b, b)$, and $(\epsilon, \epsilon)$. Clearly, if run on $A_1$, they would result in partial runs that end at states 3 (by $(cbb, cbb)$) and 1 (by $(\epsilon, \epsilon)$). This is the intuition of having $B(3) = \{3, 1\}$.

Given $\omega \in (\Sigma \cup \Psi)^*$, let $\pi(\omega)$ be the projection of $\omega$ to $\Sigma$ s.t. all the symbols in $\Psi$ are removed from $\omega$. Let $0 \leq i \leq j \leq |\omega|$, $\omega[i, j]$ represents a substring of $\omega$ starting from index $i$ and ending at $j - 1$. Similarly, $\omega[i]$ refers to the element at index $i$. Notice that for any word, index starts from 0. We have the following lemma.

Lemma 3. For any $r \in R$ there exists an FST $A_2$ s.t. for any $\omega \in \Sigma^*$, there is one and only one $\omega_2 \in (\Sigma \cup \{\$\})^*$ with $(\omega, \omega_2) \in L(A_2)$. In addition, $\omega_2$ satisfies
the following: for any $0 \leq x < |\omega_2|$, $\omega_2[x] = \$$ iff $\pi(\omega_2[0,x]) \in \Sigma^* r$; and for any $1 \leq x < |\omega_2|$, if $\omega_2[x] = \$$, then $\omega_2[x-1] \neq \$.

Corollary 2. $\mathcal{A}_2$ is a DFST.

**Step 3 (Marking Beginning of Regex Pattern):** From $\mathcal{A}_2$ we can construct a reverse transducer $\mathcal{A}_3$ by reversing all transitions in $\mathcal{A}_2$ and replacing the end marker $\$$ with the begin marker $\#$ $\notin \Sigma$. Then create a new initial state $s_0$ and add $\epsilon$ transitions to each final state in $\mathcal{A}_2$ and make the original initial state of $\mathcal{A}_2$ the final state. This results in $\mathcal{A}_3$. For example, the $\mathcal{A}_3$ shown in Figure 5 is a reverse of $\mathcal{A}_2$ in Figure 4. Clearly, $(aabbcc, \#a\#abbcc) \in L(\mathcal{A}_3)$, with $\mathcal{A}_3$ marks the beginning for pattern $r = a^+ b^+ c$.

**Lemma 4.** For any $r \in R$ there exists an FST $\mathcal{A}_3$ s.t. for any $\mu \in \Sigma^*$, there exists one and only one $\nu \in (\Sigma \cup \{\#\})^*$ with $(\mu, \nu) \in L(\mathcal{A}_3)$. $\nu$ satisfies the following: (i) $\mu = \pi(\nu)$, and, (ii) for $0 \leq i < |\nu|$: $\nu[i] = \#$ iff $\pi(\nu[i], |\nu|) \in r\Sigma^*$, and (iii) for $1 \leq i < |\nu|$: if $\nu[i] = \#$ then $\nu[i-1] \neq \#$.

The beauty of the nondeterminism is that $\mathcal{A}_3$ 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 the automata the “look ahead” ability.

**Step 4 (Inserting Begin Marker):** From $\mathcal{A}_3$ we can construct a reverse transducer $\mathcal{A}_4$ by reversing all transitions in $\mathcal{A}_3$, replacing the begin marker $\#$ with the end marker $\$$, then create a new initial state $s_0$ and add $\epsilon$ transitions to each final state in $\mathcal{A}_3$ and make the original initial state of $\mathcal{A}_3$ the final state. This results in $\mathcal{A}_4$. Clearly, $\mathcal{A}_4$ is a reverse of $\mathcal{A}_3$ in Figure 5.
Step 4 (Reluctant Replacement): Next we define an automaton for implementing the reluctant replacement semantics. We need to define an FST for filtering the extra begin markers during the replacement process. Given a regular language \( L \subseteq \Sigma^* \), let \( L_\# \) represent the language generated from \( L \) by nondeterministically inserting \#, i.e., \( L_\# = \{ \mu \mid \mu \in (\Sigma \cup \{\#\})^* \land \pi(\mu) \in L \} \).

Clearly, given \( \mathcal{A} \) that recognizes \( L \), an FSA \( \mathcal{A}' \) can be constructed from \( \mathcal{A} \) for accepting \( L_\# \), by inserting self loop (labeled with \#) to each state of \( \mathcal{A} \). Given any \( L_\# \in (\Sigma \cup \{\#\})^* \) and \( \omega \in \Sigma^* \), it is straightforward to construct an FST \( \mathcal{A}_{\# \times \#} \) s.t. \((\mu, \nu) \in L(\mathcal{A}_{\# \times \#}) \iff \mu \in L_\# \) and \( \nu = \omega \). Intuitively, given any \( \mu \) (interspersed with \#) that matches \( \mathcal{A} \), the FST replaces it with \( \omega \).

Up to now, we have solved the problem of filtering the extra \# in a substring that matches \( \mathcal{A} \). We still need to implement the reluctant semantics. This can be achieved easily as below. Given a DFSA \( \mathcal{A} \), let \( \mathcal{A}' \) be the new automaton generated from \( \mathcal{A} \) by removing all the outgoing transitions from each final state of \( \mathcal{A} \). We have the following result: \( L(\mathcal{A}') = \{ s \mid s \in L(\mathcal{A}) \land \forall s' \text{s.t. } s' < s : s' \notin L(\mathcal{A}) \} \). Clearly \( \mathcal{A}' \) implements the “shortest match” semantics.

Given regular expression \( r \), let \( \text{reluc}(r) \) represent the result of applying the above “reluctant” transformation on \( r \). An automaton \( \mathcal{A}_4 \) (as shown in Figure 6) can be defined. Intuitively, \( \mathcal{A}_4 \) consumes a symbol on both the input tape and output tape unless encountering a \#. Once a begin marker \# is consumed, \( \mathcal{A}_4 \) enters the replacement mode, which replaces the shortest match of \( r \) with the target string \( \omega \).

**Lemma 5.** Given any \( r \in R \) and \( \omega \in \Sigma^* \), there is an FST \( \mathcal{A} \) s.t. for any \( \omega_1, \omega_2 \in \Sigma^* : \omega_1, \omega_2 \in L(\mathcal{A}) \iff \omega_2 = \omega_1 \mathbf{r}_{\approx \omega} \).

The FST in the above lemma can be simply constructed as \( \mathcal{A}_3 \circ \mathcal{A}_4 \), with \( \mathcal{A}_3 \) generating the begin markers first, and \( \mathcal{A}_4 \) for accomplishing the reluctant replacement.

### 5.2 Modeling Left-Most Greedy Semantics

The modeling of greedy semantics is more complex than that of reluctant semantics. The general idea is to insert markers and then apply a number of filters for ensuring the longest match. Let \#, \$ \notin \Sigma \) be the start and end marker, respectively, and let \( \Sigma_2 = \Sigma \cup \{\#, \$\} \). In the following, given \( r \in R \) and \( \omega \in \Sigma^* \), we build an AFST for the \( \mathbf{r}_{\approx \omega} \) operator.

The first action is to insert begin markers using \( \mathcal{A}_3 \) 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 \( \mathcal{A}'_2 \). Given \( r \in R \) and \( \omega \in \Sigma^* \), \( \mathcal{A}'_2 \) can be constructed so that for any \( \omega_1 \in (\Sigma \cup \{\#\})^* \) and \( \omega_2 \in \Sigma_2^* : (\omega_1, \omega_2) \in L(\mathcal{A}'_2) \iff (i) \pi(\omega_1) = \pi(\omega_2), \) and (ii) for any \( 0 \leq i < |\omega_2|, \pi(\omega_2[i]) \in \Sigma^* r \text{ if } \omega_2[i] = \$, and (iii) for any \( 1 \leq i < |\omega_2|, \text{ if } \omega_2[i] = \$ \text{ then } \omega_2[i-1] \neq \$. Notice that \( \mathcal{A}'_2 \) is different from \( \mathcal{A}_2 \) in that the $ after a match of \( r \) is optional. Clearly, \( \mathcal{A}'_2 \) can be modified from \( \mathcal{A}_2 \) by simply adding an \( \epsilon \) from
f (old final state) to \( f' \) (new final state) in \( A_2 \), e.g., to add an \( \epsilon : \epsilon \) transition from state 4 to 5 in Figure 4. Also \# : # transitions are needed for each state to keep the \# introduced by \( A_3 \).

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 \( A_f = (\Sigma_2, Q, q_0, F, \delta) \). \( Q \) has two states \( q_0 \) and \( q_1 \). \( F = \{ q_0 \} \). The transition function \( \delta \) is defined as below: (i) \( \delta(q_0, Id(\Sigma)) = \{ q_0 \} \), (ii) \( \delta(q_0, $ : \epsilon) = \{ q_0 \} \), (iii) \( \delta(q_0, # : #) = \{ q_1 \} \), (iv) \( \delta(q_1, # : \epsilon) = \{ q_1 \} \), (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. \( L_1 \) is defined using the following regular expression:

\[
\Sigma_2^* #(r \cap \Sigma^+)S \Sigma_2^*
\]

The intuition of \( L_1 \) is to make sure that the substring between each pair of # and $ is a match of \( r \). \( L_2 \) is defined as below:

\[
\Sigma_2^* [(\# \cap r \cap $ \cap #) \cup \Sigma_2^* \cap (\Sigma \Sigma_2^* \cap r \cap # \cap $)] \Sigma_2^*
\]

The motivation of \( L_2 \) is for preventing removing too many # symbols. \( Id(L_2) \) handles two cases: (1) to avoid removing the pair of markers at the end of input word if the pattern \( r \) includes \( \epsilon \); and (2) to avoid the removing of begin markers for the next instance of \( r \). Consider the following example for case (2): given \( S_{\Sigma, r, c}^* \) and the input word \( bab \), the correct marking of begin and end markers should be \#b#a$#$b# (which leads to cbecbc as output). However the following incorrect marking could pass, without the \( Id(L_2) \) filter: \#b#a$#$b$.

The trick is that an ending marker $ (e.g., the one before the last b) may trigger \( A_f \) to remove a good begin marker # that precedes an instance of \( r \). Filter \( Id(L_2) \) is thus defined for preventing such cases. Finally, \( L_3 \) is defined for ensuring longest match:

\[
\Sigma_2^* #(r \cap # \cap (\Sigma^+ S(\Sigma^+)) \cap # \cap $) \Sigma_2^*
\]

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 \( \Sigma \)) 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^+ \cap #) \cap $ refers to a word in \( \Sigma^+ \) interspersed with begin/end markers, i.e., for any \( \omega \in (\Sigma^+ \cap #) \cap $, \( |\pi(\omega)| > 0 \).

**Lemma 6.** Given any \( r \in R \) and \( \omega \in \Sigma^* \), there is an FST \( A^+ \) s.t. for any \( \omega_1, \omega_2 \in \Sigma^* : \omega_2 = \omega_1^+ \omega_2 \) iff \( (\omega_1, \omega_2) \in L(A^+) \).

Let \( A_{Id(L_1)}, A_{Id(L_2)}, \) and \( A_{Id(L_3)} \) be the finite transducers that accept \( Id(L_1) \), \( Id(L_2) \), and \( Id(L_3) \), respectively. Clearly, \( A^+ \) can be constructed as the composition of the following FST filters:

\[
A_3 \circ A_2' \circ A_f \circ A_{Id(L_1)} \circ A_{Id(L_2)} \circ A_{Id(L_3)}
\]
The precise modeling of regular substitution semantics can be applied to both forward and backward image computation for symbolic model checking and static analysis of programs that manipulate strings (e.g., web applications). In the following, we present the applications and the tool support.

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 \( \text{input}(A) \) and \( \text{output}(A) \) to denote the input and output projection of an FST \( A \).

Let \( x \) be a variable, \( r, r_2 \in R, \omega \in \Sigma^* \), and \( x_{r \rightarrow \omega} \) be a regular substitution (using one of the declarative, greedy, and reluctant semantics discussed in this paper). An atomic string equation on regular replacement can be written as below: \( x_{r \rightarrow \omega} \equiv r_2 \).

The solution pool of \( x \) (backward image of the constraint) is defined as \( \{ \mu \mid \mu_{r \rightarrow \omega} \in L(r_2) \} \). Given a regular expression \( \nu \), the forward image of \( \nu_{r \rightarrow \omega} \) is defined as \( \{ \mu \mid \mu \in \alpha_{r \rightarrow \omega} \text{ and } \alpha \in \nu \} \). Clearly, given \( x_{r \rightarrow \omega} \equiv r_2 \), let \( A \) be the corresponding FST of \( x_{r \rightarrow \omega} \), the solution pool of variable \( x \) can be computed using \( \text{input}(A \circ \text{Id}(r_2)) \). Similarly, given \( \mu \in R \), the forward image of \( \mu_{r \rightarrow \omega} \) can be obtained by \( \text{output}(\text{Id}(\mu) \circ A) \).

The above is a part of SLSE, a general framework on solving constraints of strings. Single Linear String Equation (SLSE) is a variation of the word equation problem. User inputs are treated as unknown variables in an equation, and they can be combined with constant words using concatenation, substring, substitution, and other popular string operations provided by prevalent programming languages. We take an automata based approach for solving SLSE. The basic idea is to break down an SLSE into a number of atomic string operations. Then the solution process consists of a number of backward image computation steps. Solving atomic cases of regular replacement is an essential component of the algorithm for solving SLSE equations. The details of handling other string operators can be found in [8].

```java
StrExp x = StrExp.fromVar(new Variable("x"));
StrExp lhs = StrExp.replaceAll(x, "." , " . " );
RegExpConst rhs = new RegExpConst("uname = '(\[ ' ' ]| ' ) * ' ");
SLSE slse = new SLSE(lhs, rhs);
Solution sol = se.solveMax();
SolutionItem si = sol.findVarByName("x");
Automaton au = ((RegExpConst) si.value).getAutomaton();
if (!au.run("aaaa ")){
    fail("error!");
}
```

Listing 6.1. Solve SLSE Using SUSHI

### 6.1 Tool Support

The precise modeling of regular substitution is implemented as part of a Java package named SUSHI for solving string constraints. SUSHI consists of two
Java packages: automaton utilities and constraint solver. The automaton utility package provides the support of finite state transducer. It supports frequently used operations, e.g., concatenation, Kleen star, projecting input/output tape, constructing identity relation, composition etc. The finite state automaton is directly modeled using the `dk.bricks.automaton` package. For example, when projecting the input tape of a finite state transducer, the result is an instance of `dk.bricks.automaton`. SUSHI provides an easy way to use API for constructing and solving SLSE equations. Figure 6.1 shows a program snippet for solving an atomic SLSE equation: \( x^+_\_\_ \equiv \text{uname}='[^'|''']['^*'] \).

6.2 Experiments

In theory, the complexity of solving SLSE is EXPTIME. We conducted a number of experiments which verify the applicability of the tool in practice. In the following, we listed four SLSE equations for stress-testing the SUSHI package. Note that each equation is parametrized by an integer \( n \). eq1: \( x^+_a\_b{n,n} \equiv b\{2n,2n\} \); eq2: \( x^+_a\_b{n,n} \equiv b\{2n,2n\} \); eq3: \( x^+_a\_b{n,n} \equiv b\{2n,2n\} \); eq4: \( x^+_a\_b{n,n} \equiv b\{2n,2n\} \).

Figure 7 displays the data of running results. The first column displays the parameterized equation. For example, \( \text{eq1}(16) \) is the equation \( \text{eq1} \) parameterized by 16. The next two columns display the number of states and transitions of the FST used for modeling the regular substitution on the left hand side of each equation. Then columns 4 and 5 present the number of states and transitions of the FSA representation of the solution. The last column is the time spent on solving the solution. All experiments are conducted on a PC with 2.1Ghz Intel Core2 CPU and 2GB RAM.

As shown in Figure 7, Equations \( \text{eq1} \) and \( \text{eq3} \) take more time to verify because the construction of greedy replacement operators need more FST filters. It is interesting to note that the solution of \( \text{eq1} \) contains a collection of strings that are generated by interspersing \( b^n \) with one \( a^+ \) component (except solution \( b\{2n,2n\}_2 \)). While the solution of \( \text{eq3} \) if always a singleton set \( \{a^+\} \). Equation \( \text{eq4} \) does not have any solution.

7 Related Work and Conclusion

Analyzing text processing programs has received much attention recently due to its important application in security vulnerability scanning (e.g., [5, 6, 18, 1, 4, 13, 12]), and compatibility checking (e.g., [2, 3, 9]). There are two interesting directions of string analysis: (1) forward analysis, e.g., [3, 18], which computes the image (or its approximation) of the program states as constraints on strings and other primitive data types; and (2) backward analysis, e.g., [5, 6, 17, 12], which usually starts from the negation of a property and computes backward. This paper presented a technique for modeling regular substitutions that can be used for both the forward and backward analysis of string manipulating programs.

Finite state transducer is the major modeling tool used in this paper. FST and its variations (such as weighted FST) have been widely used in computational linguistics for processing phonological and morphological rules (see e.g. [10,
In [11], 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 $\epsilon$ is included in the search pattern. Compared with [10] 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 [10]. Thus more DFST can be used, which certainly has advantages over NFST, as it is less expressive. For example, in modeling the reluctant semantics, compared with [10], our algorithm does not need to non-deterministically insert markers and it does not need extra filters.

This paper has presented our investigation of formal modeling of various semantics of regular substitution, including declarative, finite, pure reluctant, and pure greedy replacement. We made several interesting discoveries that several fragments of the general problem can be modeled using deterministic finite state transducer, which is weaker than nondeterministic machines. A compact representation of FST is designed, and customized algorithm are developed for speeding up FST operations. The algorithms have been implemented as a part of the SUSHI package, which solves Simple Linear String Equations (SLSE) [8]. It is a continuation of our earlier efforts [5, 6] of building a general symbolic execution framework for automated vulnerability scanning. Future directions include modeling the possessive semantics and the SLSE with hybrid use of greedy and reluctant replacement operators.

**Acknowledgment:** We would like to thank Fang Yu, Tevfik Bultan, and Oscar Ibarra for the constructive discussion of this research.

**References**


