Regular Paper

Solving String Constraints with Streaming String Transducers

Qizhen Zhu[1,a)] Hitoshi Akama[1,b)] Yasuhiko Minamide[1,c)]

Received: May 7, 2019, Accepted: August 22, 2019

Abstract: We present a procedure to solve the satisfiability problem of string constraints consisting of (i) string concatenation and rational transductions of string variables restricted to be in the "straight-line" fragment, (ii) regular constraints to string variables, and (iii) integer constraints involving the length of string variables. We represent each atomic string constraint by a streaming string transducer. By the sequential composition of streaming string transducers, we obtain a single streaming string transducer. The input straight-line constraint is satisfiable if and only if the domain of the composed streaming string transducer is not empty. In addition, by calculating the Parikh image of the composed streaming string transducers, we can represent the constraints among the length of string variables as a semi-linear set. Then the integer constraints together with the Parikh image can be solved by existing SMT solvers. We have implemented this procedure and performed experiments on several string constraints. Our implementation is slower than other solvers for general cases but performs better for some special cases.

Keywords: string constraints, streaming string transducers, transducers

1. Introduction

Recently, there have been a great amount of works on solving string constraints. String constraint solvers are useful in numerous areas such as security, web programming, and model checking. For example, the vulnerability of cross-site scripting is typically caused by improper manipulating of strings and can be checked by string constraint solvers. However, the combination of concatenation and rational transductions of string variables leads to a problem which is undecidable by a simple reduction to PCP [19].

Lin and Barceló considered a fragment of string constraints called *straight-line* which accommodates concatenation and rational transductions [18]. They proved that the satisisfiability of straight-line string constraints is decidable and still decidable when they are extended with integer constraints. Their decision procedure transforms straight-line constraints to another fragment called AC (acyclic) by removing concatenation from constraints and then applies the decision procedure for AC. Holík et al. implemented the solver SLOTH based on their work and succinct representation of string constraints using alternating finite automata, and demonstrated that their solver handles constraints derived from PHP and JavaScript programs [16]. Later Chen et al. implemented the solver OSTRICH that handles string constraints with specific conditions by computing pre-image of regular languages under transductions [12]. OSTRICH is faster and can support a wider class of transductions than SLOTH. However, the support of integer constraints is limited since in general integer constraints do not follow their restrictions.

As a model of string transformations that accommodates concatenation, Alur and Černý introduced *streaming string transducers* (SSTs) [9]. An SST is an automaton with a finite set of string variables and updates variables using the current contents of its variables at each transition. For updates, it can concatenate the contents of variables. Finally, it uses the contents of the string variables to compute the output string at the end of the input string. It was shown that SSTs are equi-expressive to MSO string transductions and two-way finite-state transducers [9], [14]. Using this result, it was shown that SSTs are closed under sequential composition [9].

In this paper, we apply a class of SSTs called *bounded-copy* SSTs to study string constraints including transducers and concatenation. The second author of this paper gave a concrete construction for the composition of bounded-copy SSTs [4]. That is key to our development.

The first advantage of utilizing SSTs lies in that we can support a wider class of transductions such as string reverse and string replacement using regular expressions as shown in Ref. [17]. Furthermore, we can design a simple procedure checking satisfiability of straight-line constraints based on the standard construction for sequential composition and Parikh image.

We develop a decision procedure to solve the satisfiability problem of straight-line string constraints combined with integer constraints by using bounded-copy SSTs. We represent each atomic string constraint with a bounded-copy SST. By the sequential composition of SSTs, we obtain a single bounded-copy SST. The input straight-line string constraint is satisfiable if and only if the domain of the composed SST is not empty. In addition, by calculating the Parikh image of the composed SST, we

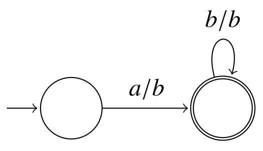
[1] Department of Mathematical and Computing Science, Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
[a)] zhu.q.ac@m.titech.ac.jp
[b)] akamahitoshi@gmail.com
[c)] minamide@is.titech.ac.jp

can represent the constraints among the length of string variables as a semi-linear set. Then the integer constraints together with the Parikh image can be solved by existing SMT solvers (e.g., the Z3 SMT solver). We have implemented our decision procedure in Scala and conducted experiments on several string constraints. The program is available on https://github.com/minamide-group/sst.

Organization Firstly, we recall the concepts of transductions, finite-state transducers, and and semi-linear sets in Section 2. In Section 3, we review the definition of SST and briefly explain the sequential composition of SSTs. In Section 4, we define straight-line string constraints. We transform constraints into SSTs in Section 5. In Section 6, we show how to combine straight-line constraints with integer constraints. Our implementation and experimental results are presented in Section 7. In Section 8, we review related research on string constraint solvers and composition of SSTs. Finally, we conclude with possible future works in Section 9.

2. Preliminaries

Let Σ and Γ be two alphabets. We call a binary relation between Σ^* and Γ^* a *transduction*. A *nondeterministic finite-state transducer* is a tuple $\mathcal{T} = (\Sigma, \Gamma, Q, q_0, \delta, F)$, where Σ is the input alphabet, Γ is the output alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the transition relation. For example, the following transducer recognizes the transduction $\{(ab^i, b^{i+1}) \mid i \in \mathbb{N}\}$.



A transduction is said to be *rational* if it is recognized by some finite-state transducer. We use $[\![\mathcal{T}]\!]$ to denote the transduction recognized by $\mathcal{T}$.

Let M be a commutative monoid with a binary operation $+$ and an identity element 1_M. A *linear set* over M constructed by $\{v_0, v_1, v_2, \ldots, v_k\} \subseteq M$ is a set of the form $\{v_0 + \lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_k v_k \mid \lambda_1, \lambda_2, \ldots, \lambda_k \in \mathbb{N}\}$. A *semi-linear set* is a finite union of linear sets.

Let Σ be an alphabet $\{a_1, a_2, \ldots, a_l\}$ where the order of $a_1, a_2, \ldots, a_l$ is arbitrary but fixed. The *Parikh image* of a string $w \in \Sigma^*$ is $\Psi(w) = (c_1, c_2, \ldots, c_l)$ where c_i is the number of occurrences of a_i in w. The Parikh image of a language $\mathcal{L} \subseteq \Sigma^*$ is $\Psi(\mathcal{L}) = \{\Psi(w) \mid w \in \mathcal{L}\}$.

3. Streaming String Transducers

Alur and Černý introduced (deterministic) streaming string transducers (SSTs) for the verification of single-pass list-processing programs [9]. An SST is an automaton with a finite set of string variables and updates variables using the current contents of its variables at each transition. Finally, it uses the contents of the string variables to compute the output string at the end of the input string. Although SSTs were extended to nondeterministic ones [7], we mainly use deterministic SSTs in this paper and just say SSTs for deterministic SSTs.

3.1 Definition of SSTs

For the definition of SSTs, we introduce the monoid of variable updates. Let X and Γ be a finite set of variables and an alphabet. Then, we call a function from X to $(X \cup \Gamma)^*$ a *variable update* and write $M_{X,\Gamma}$ for the set of variable updates over X and Γ. We use the following notation for $f \in M_{X,\Gamma}$ such that $f(x) = a\,x$, $f(y) = x\,y$, and $f(z) = z$.

$$f = [x := a\,x;\ y := x\,y;\ z := z]$$

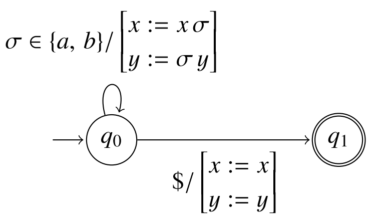
We sometimes omit assignments which assign a variable to itself. For example, we may write $[x := a\,x;\ y := x\,y]$ for f.

The set of variable updates $M_{X,\Gamma}$ constitutes a monoid with the composition of variable updates and the unit element $1_{M_{X,\Gamma}}(x) = x$ for all $x \in X$. For $f \in M_{X,\Gamma}$ and $x \in X$, we write $|f|_x$ for the number of occurrences of the variable x on the right-hand sides of f. For example, $|f|_x = 2$ and $|f|_y = 1$ for the f above. We say $f \in M_{X,\Gamma}$ is *K-copy* if $\max_{x \in X} |f|_x \leq K$.

Formally, an SST $\mathcal{S}$ is an 8-tuple $(\Sigma, \Gamma, Q, X, q_0, \delta, \eta, F)$, where Q is a finite set of states, q_0 is an initial state, Σ and Γ are input and output alphabets, respectively, X is a finite set of variables over Γ^*, $\delta : Q \times \Sigma \to Q$ is a state-transition function, $\eta : Q \times \Sigma \to M_{X,\Gamma}$ is a variable-update function, and $F : Q \hookrightarrow (X \cup \Gamma)^*$ is a partial output function.

We first explain the semantics of SSTs informally using an example.

Example 3.1 Let us consider the following SST whose output function is given by $F(q_1) = xy$. It outputs ww^R for an input $w\$$ where $w \in \{a, b\}^*$.


$\sigma \in \{a,\ b\}/ \begin{bmatrix} x := x\,\sigma \\ y := \sigma\,y \end{bmatrix}$
q_0
q_1
$\$/ \begin{bmatrix} x := x \\ y := y \end{bmatrix}$

For an input string $ab\$$, it takes the following transition and finally it outputs $abba$ by applying $[x := ab; y := ba]$ to $xy = F(q_1)$.

$$\left(q_0, \begin{matrix} x := \epsilon \\ y := \epsilon \end{matrix}\right) \overset{a}{\Rightarrow} \left(q_0, \begin{matrix} x := a \\ y := a \end{matrix}\right) \overset{b}{\Rightarrow} \left(q_0, \begin{matrix} x := ab \\ y := ba \end{matrix}\right) \overset{\$}{\Rightarrow} \left(q_1, \begin{matrix} x := ab \\ y := ba \end{matrix}\right) \quad \square$$

To define the semantics of SSTs formally, we extend the state-transition and variable-update functions for strings as follows.

$$\begin{aligned} \hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, \sigma w) &= \hat{\delta}(\delta(q, \sigma), w) && (\sigma \in \Sigma, w \in \Sigma^*) \\ \hat{\eta}(q, \varepsilon) &= 1_{M_{X,\Gamma}} \\ \hat{\eta}(q, \sigma w) &= \eta(q, \sigma) \circ \hat{\eta}(\delta(q, \sigma), w) && (\sigma \in \Sigma, w \in \Sigma^*) \end{aligned}$$

Then, the output string of SST $[\![\mathcal{S}]\!](w)$ for an input string $w \in \Sigma^*$ is defined as follows:

$$[\![\mathcal{S}]\!](w) = \begin{cases} \hat{\varepsilon}(\hat{\eta}(q_0, w)(F(\hat{\delta}(q_0, w)))) & \hat{\delta}(q_0, w) \in \mathrm{dom}(F) \\ \perp & \hat{\delta}(q_0, w) \notin \mathrm{dom}(F) \end{cases}$$

where $\hat{\varepsilon} : (X \cup \Gamma)^* \to \Gamma^*$ is the function that removes variables in a given string.

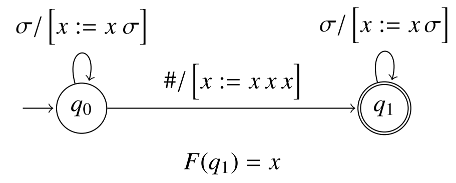
Example 3.2 This definition is equivalent to the informal semantics described above. For example, we have the following composition for the SST in Example 3.1 and the input string $ab\$$.

$$\begin{bmatrix} x := xa \\ y := ay \end{bmatrix} \circ \begin{bmatrix} x := xb \\ y := by \end{bmatrix} \circ \begin{bmatrix} x := x \\ y := y \end{bmatrix} = \begin{bmatrix} x := xab \\ y := bay \end{bmatrix}$$

Then, we have $\hat{\varepsilon}([x := xab; y := bay](xy)) = abba$. □

We say an SST is *K-bounded copy* if for every $q \in Q$ and input string $w \in \Sigma^*$, the variable update $\hat{\eta}(q, w)$ is K-copy. An SST is *bounded copy* if it is K-bounded copy for some K[*1]. Especially, a 1-bounded copy SST is called a *copyless SST*.

Example 3.3 The following is an example of 3-bounded-copy SST. It accepts string $w\#w'$ ($w, w' \in \{a, b\}^*$) and outputs $w^3 w'$. In the transition diagram, the transition $\sigma/\left[x := x\,\sigma\right]$ represents the two transitions $a/\left[x := x\,a\right]$ and $b/\left[x := x\,b\right]$.



The same transduction can be given as a copyless SST by introducing auxiliary variables. □

3.2 Sequential Composition of SSTs

Our work significantly depends on the construction for sequential composition of two SSTs: for given two SSTs $\mathcal{S}_1$ and $\mathcal{S}_2$, we need to construct SST $\mathcal{S}$ such that $[\![\mathcal{S}]\!] = [\![\mathcal{S}_2]\!] \circ [\![\mathcal{S}_1]\!]$. It was first shown that SSTs are closed under composition by using the result that the expressiveness of SSTs exactly coincide with that of MSO string transductions [7]. Our work is based on the following construction of the second author for bounded-copy SSTs [4].

Theorem 3.4 Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be K-bounded-copy and L-bounded-copy SSTs. Then, there exists a $K{\cdot}L{\cdot}|Q_2|$-bounded-copy SST $\mathcal{S}$ such that $[\![\mathcal{S}]\!] = [\![\mathcal{S}_2]\!] \circ [\![\mathcal{S}_1]\!]$. Furthermore,

- the number of the states of $\mathcal{S}$ is bounded by $|Q_1| \cdot |Q_2|^{|Q_2||X|} \cdot C^{|Q_2||X|}$.
- the number of the variables of $\mathcal{S}$ is bounded by $|Q_2| \cdot |X| \cdot |Y| \cdot (L + 1)$.

where Q_1 and Q_2 are the state sets of $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, X and Y are the variable sets of $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, and C is a constant defined as follows.

$$C = |\{f \in M_{Y,\emptyset} \mid \text{f is L-copy }\}| \qquad \square$$

The formalization in Isabelle is available on https://github.com/akamah/sst-isabelle. This theorem itself is not a contribution of this paper.

It should be noted that the for the composition of more than two SSTs, the order of composition has a significant impact on the size of the composed SST. Let us consider the composition of three copyless SSTs $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ whose variables sets are X, Y, and Z. If we compose them by $[\![\mathcal{S}_3]\!] \circ ([\![\mathcal{S}_2]\!] \circ [\![\mathcal{S}_1]\!])$, the composed SST has the following number of variables.

$$4 \cdot |Q_3| \cdot (|Q_2| \cdot |X| \cdot |Y|) \cdot |Z|$$

On the other hand, if we compose them by $([\![\mathcal{S}_3]\!] \circ [\![\mathcal{S}_2]\!]) \circ [\![\mathcal{S}_1]\!]$, the composed SST has the following number of variables:

$$4 \cdot |Q_{2,3}| \cdot |X| \cdot (|Q_3| \cdot |Y| \cdot |Z|)$$

where $|Q_{2,3}|$ is the state set of the composition of $\mathcal{S}_2$ and $\mathcal{S}_3$. Since $|Q_{2,3}|$ is exponential in $|Q_3|$, the former composition is much better at least on the number of variables. Hence, we have chosen the order of the former composition in our implementation when we compose more than two SSTs.

4. Straight-Line String Constraints

Let $X = \{x_0, x_1, \ldots, x_{n-1}\}$ be a set of string variables indexed by one integer. Let $i, j, k \in \mathbb{N}$ denote the indices of string variables. Then we define *straight-line constraints* over X [18].

Definition 4.1 (Basic Straight-Line Constraint). Let w be a constant string, $j, k < i$, and T be a transduction. We call $x_i = e_i$ an *atomic constraint*, where e_i is an expression defined as follows.

$$e_i ::= w \mid x_j \mid x_j \cdot x_k \mid T(x_j)$$

Let m be a natural number less than n. A *basic straight-line constraint* φ_{sl} is a conjunction of atomic constraints defined as follows.

$$\varphi_{sl} ::= (x_m = e_m) \wedge (x_{m+1} = e_{m+1}) \wedge \cdots \wedge (x_{n-1} = e_{n-1}) \qquad \square$$

For example, the following constraint is in straight-line format.

$$x_2 = x_0 \cdot x_1$$
$$x_3 = x_2.\mathit{replaceAll}(a, bb)$$

The next constraint is not in straight-line format because x_2 is restricted by x_3.

$$x_2 = x_0 \cdot x_3$$
$$x_3 = x_2.\mathit{replaceAll}(a, bb)$$

We extends the basic straight-line constraints with regular language membership on string variables and integer constraints involving the length of string variables.

Definition 4.2 (Straight-Line Constraint). Let R_i ($0 \le i < n$) be regular languages. A *regular constraint* φ_{reg} is a conjunction of regular language membership on string variables.

$$\varphi_{reg} ::= \bigwedge (x_i \in R_i)$$

Let $c \in \mathbb{N}$ be an integer constant, u be an variable over $\mathbb{Z}$, and $|x|$ be the length of content of string variable x. We define an integer expression t as follows.

$$t ::= c \mid u \mid |x| \mid t + t \mid t - t$$

[*1] This definition of bounded copy SSTs does not directly corresponds to that of Ref. [8]. It is basically an SST with a finite transition monoid in Ref. [15].

Then an *integer constraint* φ_{int} is defined as follows.

$$\varphi_{int} ::= t = t \mid t < t \mid \varphi_{int} \land \varphi_{int} \mid \neg\varphi_{int}$$

A *straight-line constraint* φ is defined as follows.

$$\varphi ::= \varphi_{sl} \land \varphi_{reg} \land \varphi_{int}$$ □

Let θ_{str} be an interpretation from a string variable to a string in Σ^*. We extend θ_{str} for e as follows.

$$\theta_{str}(w) = w \text{ for } w \in \Sigma^* \qquad \theta_{str}(T(x)) = T(\theta_{str}(x))$$
$$\theta_{str}(x_1 \cdot x_2) = \theta_{str}(x_1) \cdot \theta_{str}(x_2)$$

Let θ_{int} be an interpretation from an integer variable to a number in $\mathbb{Z}$ and $\theta = \theta_{str} \cup \theta_{int}$. Then we extends θ for t as follows.

$$\theta(c) = c \text{ for } c \in \mathbb{Z} \qquad \theta(|x|) = |\theta_{str}(x)|$$
$$\theta(t_1 + t_2) = \theta(t_1) + \theta(t_2) \qquad \theta(t_1 - t_2) = \theta(t_1) - \theta(t_2)$$

Given an interpretation θ, we use val_θ to evaluate constraints to a value in $\{tt, ff\}$ as follows.

$$val_\theta(\varphi_1 \land \varphi_2) = tt \quad \text{iff} \quad val_\theta(\varphi_1) = tt \text{ and } val_\theta(\varphi_2) = tt$$
$$val_\theta(\neg\varphi) = tt \quad \text{iff} \quad val_\theta(\varphi) = ff$$
$$val_\theta(x = e) = tt \quad \text{iff} \quad \theta_{str}(x) = \theta_{str}(e)$$
$$val_\theta(x \in R) = tt \quad \text{iff} \quad \theta_{str}(x) \in R$$
$$val_\theta(t_1 = t_2) = tt \quad \text{iff} \quad \theta_{int}(t_1) = \theta_{int}(t_2)$$
$$val_\theta(t_1 < t_2) = tt \quad \text{iff} \quad \theta_{int}(t_1) < \theta_{int}(t_2)$$

A constraint φ is *satisfiable* if there is an interpretation θ such that $val_\theta(\varphi) = tt$, and we call θ a *witness* for φ. Otherwise φ is *unsatisfiable*. We use $\theta \models \varphi$ to denote $val_\theta(\varphi) = tt$.

Example 4.3 The following constraint replaces all the occurrences of < and >, and then checks whether the result contains < or >. Finally it checks whether $x_0 \cdot x_1$ contains the same number of < and >. This check is achieved because $|x_3| - |x_2| = 3|x_2|_<$ and $|x_4| - |x_3| = 3|x_3|_>$, where $|x|_\sigma$ denotes the number of occurrences of σ in string variable x.

$$x_2 = x_0 \cdot x_1$$
$$x_3 = x_2.replaceAll(<, \&lt;)$$
$$x_4 = x_3.replaceAll(>, \&gt;)$$
$$|x_4| - |x_3| = |x_3| - |x_2|$$

This constraint is satisfiable because it is evaluated to be tt under the following witness.

$$[x_0 \mapsto <,\ x_1 \mapsto >,\ x_2 \mapsto <>,\ x_3 \mapsto \&lt;>,\ x_4 \mapsto \&lt;\&gt;]$$ □

The previous research of Lin et al. [18] solved a straight-line constraint φ by transforming it into φ' which preserves the satisfiability of φ and there is no concatenation in φ'. Then φ' is solved by applying a result in the theory of rational relations [10]. The size of φ' is at most exponential to that of φ.

The transformation is based on the following properties of regular languages and rational transductions.

• Let R be a regular language. Then there exist a natural number n, R'_i and $R''_i (1 \le i \le n)$ such that the following statement holds.

$$w'w'' \in R \iff \bigvee_{i=1}^{n} (w' \in R'_i \land w'' \in R''_i)$$

• Let T be a rational transduction. Then there exist a natural number n, T'_i and $T''_i (1 \le i \le n)$ such that the following statement holds.

$$w = T(w'_1 w''_1) \iff$$
$$\exists w'w''.(w = w'w'') \land \bigvee_{i=1}^{n} (w' = T'_i(w'_1) \land w'' = T''_i(w''_1))$$

We use a simple example to explain their approach.

Example 4.4 Consider the following constraint.
(1) $y = x_0 \cdot x_1$
(2) $y \in R$
(3) $z = T(y)$

In order to remove the concatenation $x_0 \cdot x_1$, we need to introduce two fresh variables y_0 and y_1 as follows.
(1′) $y_0 = x_0$
(2′) $y_1 = x_1$

The regular constraint $y \in R$ is then split into two parts: $\bigvee_i (y_0 \in R'_i \land y_1 \in R''_i)$ and we can nondeterministicly choose one pair of R'_i and R''_i.
(3′) $y_0 \in R'_i$
(4′) $y_1 \in R''_i$

We also need to split T and z as $\bigvee_j (z_0 = T'_j(y_0) \land z_1 = T''_j(y_1))$. Similarly we nondeterministicly choose one pair of T'_j and T''_j.
(5′) $z_0 = T'_j(y_0)$
(6′) $z_1 = T''_j(y_1)$

The new constraint consists of (1′), (2′), (3′), (4′), (5′), and (6′). SLOTH uses this approach to handle the combination of concatenation and transduction. □

5. String Constraints to SSTs

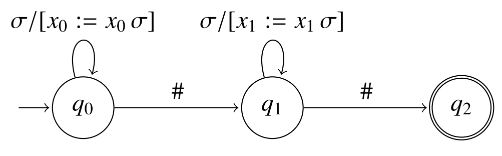
In this section, we show all the atomic constraints and regular constraints can be transformed into bounded-copy SSTs which preserve the satisfiability of the original constraints. The original constraint is satisfiable if and only if the obtained SST accepts a non-empty language. Then we sequentially compose all the SSTs together to generate a single SST. The straight-line constraint is satisfiable if and only if the composed SST accepts a non-empty language.

5.1 Atomic Constraints

For an atomic constraint $x_i = e_i$, we transform this constraint into an SST $\mathcal{S}_i$, such that $\mathcal{S}_i$ accepts strings of the form $(\Sigma^*\#)^i$ representing $x_0\#x_1\#\ldots\#x_{i-1}\#$, where # is a symbol indicating the end of the content of the previous variable. $\mathcal{S}_i$ computes the value of x_i and outputs $(\Sigma^*\#)^{i+1}$ representing $x_0\#x_1\#\ldots\#x_{i-1}\#x_i\#$.

Consider $x_i = e_i$ where e_i is a concatenation of string variables. Note that a single variable or a string can also be transformed in a similar manner. The SST $\mathcal{S}_i$ for this constraint has i variables and $(i + 1)$ states. It records the values of variables from the input string, outputs them and appends the value of e_i at the end.

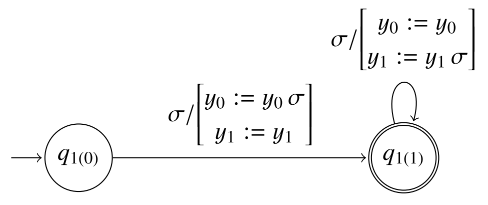
Example 5.1 $x_2 = x_1 \cdot x_0$ is transformed into the following SST. Each x_i ($i \in \{0, 1\}$) is recorded at state q_i. The final state q_2 outputs values of all variables with the value of x_2, which is $x_1 x_0$. The values in the output are also split by #.



$\mathcal{S}_2$. $F(q_2) = x_0\#x_1\#x_1x_0\#$ □

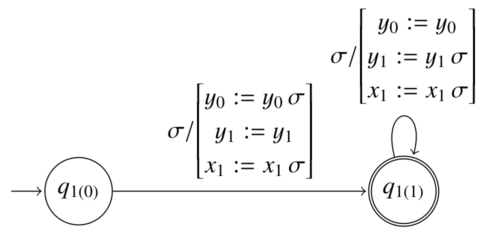
Let T be a transduction realized by an SST $\mathcal{S}_r$ with the state set Q_r and the variable set X_r. Then, a constraint $x_i = T(x_j)$ is transformed into an SST $\mathcal{S}_i$ with $(i + 1 + |X_r|)$ variables and $(i + |Q_r|)$ states. Firstly, we modify $\mathcal{S}_r$ to $\mathcal{S}'_r$ such that $\mathcal{S}'_r$ works in the same way as $\mathcal{S}_r$ but updates the value of x_j in each transition. Then we create an SST $\mathcal{S}'_i$ that accepts $(\Sigma^*\#)^i$ and outputs $(\Sigma^*\#)^{i+1}$. At last we replace q_j in $\mathcal{S}'_i$ with $\mathcal{S}'_r$ and get the result $\mathcal{S}_i$.

Example 5.2 Consider the operation of string insertion $x_3 = x_1.\mathit{insert}(1, aba)$, where $x_1.\mathit{insert}(1, aba)$ is the string obtained by inserting aba at the index 1 of x_1. This function can be represented by the following $\mathcal{S}_r$.


$\sigma/\begin{bmatrix} y_0 := y_0 \\ y_1 := y_1\,\sigma \end{bmatrix}$
$\sigma/\begin{bmatrix} y_0 := y_0\,\sigma \\ y_1 := y_1 \end{bmatrix}$
$q_{1(0)}$
$q_{1(1)}$

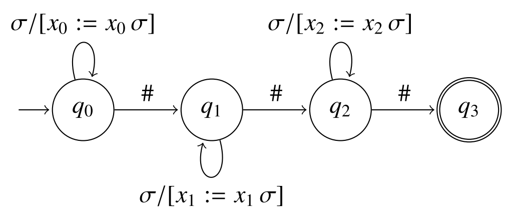
$\mathcal{S}_r$. $F(q_1) = y_0\,aba\,y_1$

Then we create $\mathcal{S}'_r$, which represents the same function as $\mathcal{S}_r$ but updates the value of x_1 at the same time.


$\sigma/\begin{bmatrix} y_0 := y_0 \\ y_1 := y_1\,\sigma \\ x_1 := x_1\,\sigma \end{bmatrix}$
$\sigma/\begin{bmatrix} y_0 := y_0\,\sigma \\ y_1 := y_1 \\ x_1 := x_1\,\sigma \end{bmatrix}$
$q_{1(0)}$
$q_{1(1)}$

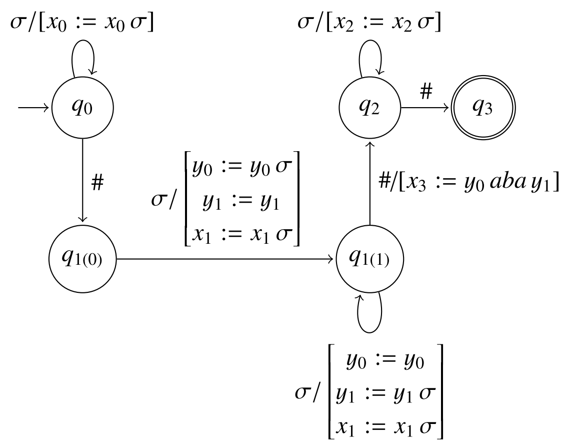
$\mathcal{S}'_r$. $F(q_1) = y_0\,aba\,y_1$

The following $\mathcal{S}'_3$ accepts $(\Sigma^*\#)^3$.


$\sigma/[x_0 := x_0\,\sigma]$
$\sigma/[x_2 := x_2\,\sigma]$
q_0
#
q_1
#
q_2
#
q_3
$\sigma/[x_1 := x_1\,\sigma]$

$\mathcal{S}'_3$. $F(q_3) = x_0\#x_1\#x_2\#x_3\#$

After replacing q_1 with $\mathcal{S}'_r$ and modifying transitions, we obtain the following $\mathcal{S}_3$.


$\sigma/[x_0 := x_0\,\sigma]$
$\sigma/[x_2 := x_2\,\sigma]$
q_0
q_2
#
q_3
#
$\sigma/\begin{bmatrix} y_0 := y_0\,\sigma \\ y_1 := y_1 \\ x_1 := x_1\,\sigma \end{bmatrix}$
$\#/[x_3 := y_0\,aba\,y_1]$
$q_{1(0)}$
$q_{1(1)}$
$\sigma/\begin{bmatrix} y_0 := y_0 \\ y_1 := y_1\,\sigma \\ x_1 := x_1\,\sigma \end{bmatrix}$

$\mathcal{S}_3$. $F(q_3) = x_0\#x_1\#x_2\#x_3\#$ □

The following theorem is clear from the construction.

Theorem 5.3 Let $\mathcal{S}_i$ be the SST obtained from $x_i = e_i$. Then $[\![\mathcal{S}_i]\!]$ is a partial function from $(\Sigma^*\#)^i$ to $(\Sigma^*\#)^{i+1}$ and the following two statements are equivalent.

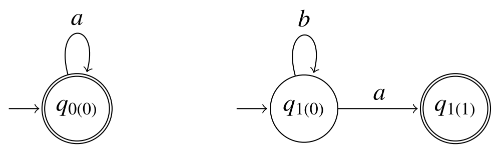
(1) $[\![\mathcal{S}_i]\!](w_0\# \ldots w_{i-1}\#) = w_0\# \ldots w_{i-1}\#w_i\#$

(2) $[x_0 \mapsto w_0, \ldots, x_{i-1} \mapsto w_{i-1}, x_i \mapsto w_i] \models x_i = e_i$ □

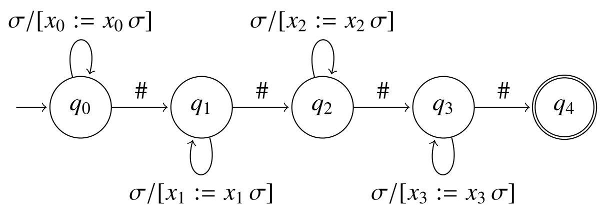
5.2 Regular Constraints

We transform a regular constraint $\bigwedge_{i=0}^{n-1}(x_i \in R_i)$ into $\mathcal{S}_n$ that accepts $(\Sigma^*\#)^n$. We replace each q_i in $\mathcal{S}_n$ with the DFA representing R_i, then $\mathcal{S}_n$ checks whether the regular constraint is satisfied by the input. We assume there is exactly one regular language R_i for each x_i. If there are more than one R for a single variable, we compute the intersection of them. If a variable x is not restricted by any R, then x is in Σ^*.

Example 5.4 Consider $(x_0 \in a^*) \wedge (x_1 \in b^*a)$ for $\{x_0, x_1, x_2, x_3\}$. The regular languages a^* and b^*a are represented by following DFAs.

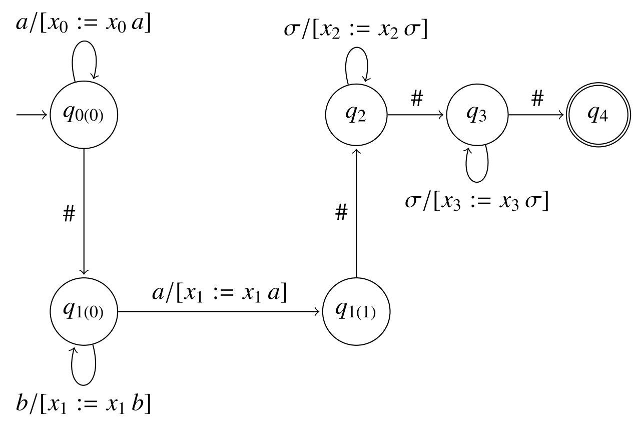


Then we create $\mathcal{S}_4$ that accepts $(\Sigma^*\#)^4$.


$\sigma/[x_0 := x_0\,\sigma]$
$\sigma/[x_2 := x_2\,\sigma]$
q_0
#
q_1
#
q_2
#
q_3
#
q_4
$\sigma/[x_1 := x_1\,\sigma]$
$\sigma/[x_3 := x_3\,\sigma]$

$\mathcal{S}_4$

Then q_0 and q_1 are replaced with the above DFAs. Because x_2 and x_3 are not restricted by any regular constraint, and q_2 and q_3 express Σ^*, we do not need to do anything with them. Then $\mathcal{S}_4$ becomes:



$\mathcal{S}_4$ □

Theorem 5.5 Let $\mathcal{S}_n$ be the SST obtained from $\bigwedge(x_i \in R_i)$. Then $[\![\mathcal{S}_n]\!]$ is a partial function from $(\Sigma^*\#)^n$ to $(\Sigma^*\#)^n$ and the following two statements are equivalent.

(1) $[\![\mathcal{S}_n]\!](w_0\# \dots w_{n-1}\#) = w_0\# \dots w_{n-1}\#$

(2) $[x_0 \mapsto w_0, \dots, x_{n-1} \mapsto w_{n-1}] \models \bigwedge(x_i \in R_i)$ □

The theorem is also clear from the construction of the SST.

By Theorem 3.4, Theorem 5.3, and Theorem 5.5, we have the following theorem which shows that we can solve the satisfiability of φ_{sl} and φ_{reg} by checking whether the domain of the composed SST is empty.

Theorem 5.6 Given $\varphi_{sl} = \bigwedge_{i=m}^{n-1}(x_i = e_i)$ and $\varphi_{reg} = \bigwedge(x_i \in R_i)$, let $\mathcal{S}_i$ be the SST obtained from $x_i = e_i$, $\mathcal{S}_n$ be the SST obtained from φ_{reg}, and $\mathcal{S}$ be the sequential composition of $\mathcal{S}_m, \dots, \mathcal{S}_n$. Then $[\![\mathcal{S}]\!]$ is a partial function from $(\Sigma^*\#)^m$ to $(\Sigma^*\#)^n$ and the following two statements are equivalent.

(1) $[\![\mathcal{S}]\!](w_0\# \dots w_{m-1}\#) = w_0\# \dots w_{n-1}\#$

(2) $[x_0 \mapsto w_0, \dots, x_{n-1} \mapsto w_{n-1}] \models \varphi_{sl} \wedge \varphi_{reg}$ □

This theorem is proved by induction on the number of SSTs.

6. Integer Constraints

In this section we firstly introduce Parikh image of SSTs. Then we explain how to solve length constraints by using Parikh image.

6.1 Parikh Image of SSTs

We define the Parikh image of an SST $\mathcal{S}$ as $\Psi(\mathcal{S}) = \{\Psi(w') \mid \exists w. [\![\mathcal{S}]\!](w) = w'\}$. Since SSTs are expressively equivalent to MSO (monadic second-order logic) definable transductions [9] and the Parikh image of an MSO definable transduction is a semi-linear set [13], the Parikh image of an SST can also be represented by a semi-linear set.

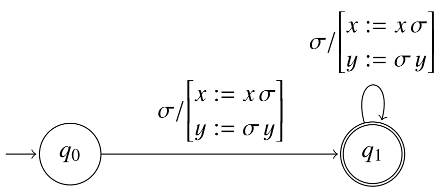
Theorem 6.1 Given a bounded-copy SST $\mathcal{S}$, there exists a nondeterministic transducer $\mathcal{T}_{\mathbb{N}}$ that reads a string and outputs a vector of non-negative integers such that the following two statements hold:

(1) if $[\![\mathcal{S}]\!](w) = w'$, then $(w, \Psi(w')) \in [\![\mathcal{T}_{\mathbb{N}}]\!]$,

(2) if $(w, v) \in [\![\mathcal{T}_{\mathbb{N}}]\!]$, then there exists w' such that $\Psi(w') = v$ and $[\![\mathcal{S}]\!](w) = w'$. □

The construction of $\mathcal{T}_{\mathbb{N}}$ and the outline of the proof are given in Appendix A.2. The construction is inspired by [5], [17]. By applying the state elimination on $\mathcal{T}_{\mathbb{N}}$, we can compute a semi-linear set representing the Parikh image of $\mathcal{S}$.

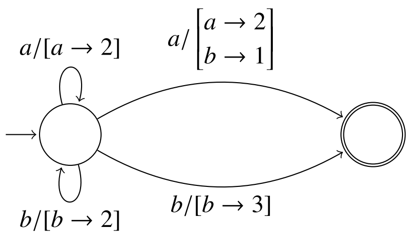
Example 6.2 Let σ be a symbol in $\{a, b\}$. Consider the following SST.



$\sigma/\begin{bmatrix} x := x\,\sigma \\ y := \sigma\,y \end{bmatrix}$
$\sigma/\begin{bmatrix} x := x\,\sigma \\ y := \sigma\,y \end{bmatrix}$
q_0
q_1

$F(q_1) = x\,b\,y$

We generate the following transducer from the SST. In order to be comprehensible, we use a map from output symbols to integers to represent the output in each transition. For instance, the transition labeled with $a/[a \to 2]$ means that the number of a increases by 2 in this transition.



The Parikh image of this SST is $\{v_0 + \lambda_1 v_1 + \lambda_2 v_2 \mid \lambda_1, \lambda_2 \in \mathbb{N}\}$, where $v_0 = (0, 1)$, $v_1 = (2, 0)$ and $v_2 = (0, 2)$. The number of a that this SST outputs is $(0 + 2\lambda_1 + 0\lambda_2)$, and the number of b is $(1 + 0\lambda_1 + 2\lambda_2)$. □

6.2 Solving Integer Constraints by using Parikh Image

In order to combine string constraints with integer constraints which involve the length of contents of string variables, firstly we modify the last SST $\mathcal{S}_n$ such that every x_i only contains one kind of symbols a_i. The length of x_i is equivalent to the number of occurrences of a_i in the output strings of the composed SST. Then we compute the Parikh image to calculate the letter counts of all symbols in the output strings.

Let $\mathcal{S}_n$ be an SST obtained from a regular constraint that outputs strings in the form $w_0\#w_1\#...\#w_{n-1}\#$, where w_i is the content of the variable x_i $(0 \leq i < n)$. We modify $\mathcal{S}_n$ to $\mathcal{S}'_n$ such that $\mathcal{S}'_n$ outputs:

$$\underbrace{a_0...a_0}_{|w_0|}\underbrace{a_1...a_1}_{|w_1|}\dots\underbrace{a_{n-1}...a_{n-1}}_{|w_{n-1}|}$$

where the number of a_i is the length of w_i. In other words, we have $\Psi([\![\mathcal{S}'_n]\!](w_0\#w_1\# \dots \#w_{n-1}\#)) = (|w_0|, |w_1|, \dots, |w_{n-1}|)$.

$\mathcal{S}'_n$ can be obtained by modifying the variable update function of $\mathcal{S}_n$, such that each variable contains only a single kind of symbol which is distinct from any other variable.

Then the composed SST also outputs strings in the form $a_0^* a_1^* \dots a_{n-1}^*$. We compute the Parikh image of the composed SST which represents the constraints on variable length.

Theorem 6.3 Given $\varphi_{sl} = \bigwedge_{i=m}^{n-1}(x_i = e_i)$ and $\varphi_{reg} = \bigwedge(x_i \in R_i)$, let $\mathcal{S}_i$ be the SST obtained from $x_i = e_i$, $\mathcal{S}_n$ be the SST obtained from φ_{reg}, $\mathcal{S}'_n$ be the SST obtained by modifying $\mathcal{S}_n$, $\mathcal{S}'$ be the sequential composition of $\mathcal{S}_m, \dots, \mathcal{S}_{n-1}, \mathcal{S}'_n$, and $\mathcal{T}_{\mathbb{N}}$ be the transducer obtained from $\mathcal{S}'$ by the Theorem 6.1. Then $[\![\mathcal{T}_{\mathbb{N}}]\!]$ is a transduction between $(\Sigma^*\#)^m$ and $\mathbb{N}^n$ and the following two statements are equivalent.

(1) $(w_0\#\ldots w_{m-1}\#,(c_0,\ldots,c_{n-1}))\in[\![\mathcal{T}_{\mathbb{N}}]\!]$

(2) $[x_0\mapsto w_0,\ldots,x_{n-1}\mapsto w_{n-1}]\models\varphi_{sl}\wedge\varphi_{reg}$ and

$\bigwedge_{i=0}^{n-1}|w_i| = c_i forsomew_m,\ldots,w_{n-1}$ □

This theorem follows directly from Theorem 5.6 and Theorem 6.1.

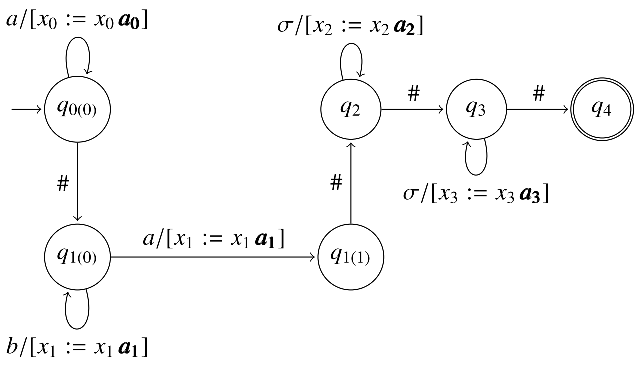
Example 6.4 Reconsider examples in Section 5. We have SSTs transformed from the following constraints.

$\mathcal{S}_2 : x_2 = x_1x_0$

$\mathcal{S}_3 : x_3 = x_1.insert(1,aba)$

$\mathcal{S}_4 : (x_0\in a^*)\wedge(x_1\in b^*a)$

We add an integer constraint $|x_3| > |x_2|$ and modify $\mathcal{S}_4$ into $\mathcal{S}_4'$ as follows.


$a/[x_0 := x_0\,\boldsymbol{a_0}]$
$\sigma/[x_2 := x_2\,\boldsymbol{a_2}]$
$q_{0(0)}$
q_2
q_3
q_4
#
#
#
#
$\sigma/[x_3 := x_3\,\boldsymbol{a_3}]$
$q_{1(0)}$
$a/[x_1 := x_1\,\boldsymbol{a_1}]$
$q_{1(1)}$
$b/[x_1 := x_1\,\boldsymbol{a_1}]$

$\mathcal{S}_4'$. $F(q_4) = x_0x_1x_2x_3$

Then we compose $\mathcal{S}_2$, $\mathcal{S}_3$ and $\mathcal{S}_4'$ together, and compute its Parikh image, which is $\{v_0+\lambda_1v_1+\lambda_2v_2 \mid \lambda_1,\lambda_2\in\mathbb{N}\}$ where $v_0 = (0,1,1,4)$, $v_1 = (1,0,1,0)$ and $v_2 = (0,1,1,1)$. Intuitively, v_1 means that when $|x_0|$ increases by 1, $|x_2|$ will increase by 1. And v_2 means that when $|x_1|$ increases by 1, both of $|x_2|$ and $|x_3|$ will increase by 1.

We convert this Parikh image into following integer constraints.

(1) $\lambda_1\geq 0$
(2) $\lambda_2\geq 0$
(3) $|x_0| = 0+\lambda_1+0\lambda_2$
(4) $|x_1| = 1+0\lambda_1+\lambda_2$
(5) $|x_2| = 1+\lambda_1+\lambda_2$
(6) $|x_3| = 4+0\lambda_1+\lambda_2$

Finally we submit the conjunction of the integer constraints from Parikh image together with the input $|x_3| > |x_2|$ to an SMT solver and get a *sat* answer. □

7. Implementation and Experiments

7.1 Implementation

We have implemented our procedure and optimization in Scala. Currently, our solver supports a part of SMT-libv2 format [11] including `declare-fun`, `assert`, `check-sat`, and `get-model` commands with string operations such as `str.substr`, `str.++`, `str.len` and so on.

The output includes the satisfiability of the input constraint. If the input constraint is satisfiable, our solver can also find a witness by following method. If there is no length constraint on any variable, we search the composed SST for an accepted input and let the composed SST process the input to get a witness that contains the values of all variables. If there are length constraints on variables, firstly we obtain a witness wit_0 that specifies the length of each variable by an SMT solver. We search the transducer generated by the method in Section 6.1 for an accepted input which produces an output that is equivalent to wit_0. Then the composed SST processes the input and outputs a witness.

The program structure is in **Fig. 1**. Firstly we convert the atomic constraints and regular constraints into SSTs. Then we compose the SSTs together. If the composed SST accepts an empty language, the result is *"unsat"*, which means the input constraint is unsatisfiable. Otherwise, we calculate the Parikh image of the composed SST, submit the integer constraints together with the Parikh image to the Z3 SMT solver, and get the result.

Example 7.1

$x_1 = ba\,x_0\,ab$

$x_2 = x_1.replaceAll(abb,bb)$

$x_2\in b^*aab$

$|x_2| = |x_1|$

The above constraint is represented as the following text.

```
(declare-fun x0 () String)
(declare-fun x1 () String)
(declare-fun x2 () String)
(assert (= x1 (str.++ "ba" x0 "ab")))
(assert (= x2 (str.replaceall x1 "abb" "bb")))
(assert (str.in.re x2
        (re.++ (re.* (str.to.re "b"))
               (str.to.re "aab"))))
(assert (= (str.len x2) (str.len x1)))
(check-sat)
(get-model)
```

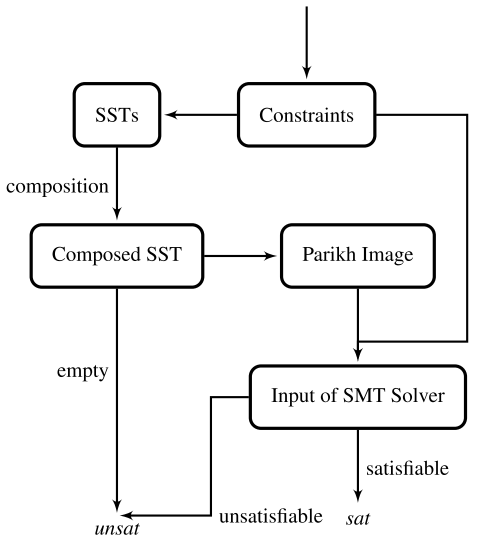


Fig. 1 Overview of our solver.

Firstly, we declare variables `x0`, `x1`, `x2` with the `declare-fun` command. Then we use the `assert` command to express the constraint above. Here the `str.++`, `str.replaceall`, `str.in.re`, and `str.len` represent the string concatenation, string replacing, regular language membership and string length operations respectively. `str.to.re` converts a string into a regular expression. `re.*`, `re.++`, and `re.union` represent the star, concatenation, and union operations on regular expressions. The `check-sat` command checks the satisfiability of the set of all assertions. The `get-model` command returns a possible assignment of the variables if the assertions are satisfiable.

The output is as follows. `sat` means this constraint is satisfiable. `model` defines the values of the variables.

```
sat
(model
        (define-fun x0 () String
                '')
        (define-fun x1 () String
                'baab')
        (define-fun x2 () String
                'baab')
)
```

□

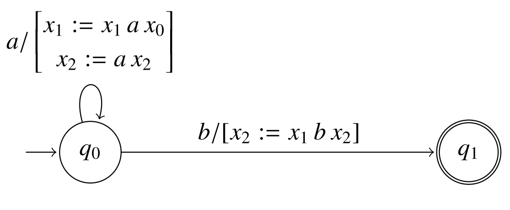
7.2 Optimization: Removing Redundant Variables

The SST composition algorithm generates lots of states and variables as we described in Theorem 1. However, only a few of them are necessary. We give a method to remove redundant variables, which are never used by the output function or always contain the empty string.

Firstly, we search for all unreachable states and remove them. Then, find all the variables which may be used by the output function. We will calculate the *used variable set* of the initial state. At first, the used variable set of each final state q_f is all the variables in $F(q_f)$, and the used variable set of each non-final state is $\emptyset$. Then we find all the variables that are assigned to the used variables in a transition and add them to the used variable set of the origin state of the transition. Repeat this step until the used variable set of each state does not change.

We also need to find all the variables which may contain a nonempty string. We will calculate the union of the *nonempty variable set* of all the final states. At first, the nonempty variable set of the initial state is $\emptyset$. Then we find all the variables assigned by alphabets or by the nonempty variables in a transition, and add them to the nonempty variable set of the destination of the transition.

Example 7.2 Consider the following SST.


$a/\begin{bmatrix} x_1 := x_1\,a\,x_0 \\ x_2 := a\,x_2 \end{bmatrix}$
q_0
$b/[x_2 := x_1\,b\,x_2]$
q_1

$F(q_1) = x_2$

At the beginning, the used variable set of q_1 is $\{x_2\}$. Then by the transition (q_0, b, q_1), we can update the used variable set of q_0 to $\{x_1, x_2\}$ because x_1 and x_2 are assigned to x_2 in this transition. Similarly, by the transition (q_0, a, q_0), the used variable set of q_0 is updated to $\{x_0, x_1, x_2\}$.

The nonempty variable sets of q_0 is initially $\emptyset$. By the transition (q_0, a, q_0), the nonempty variable sets of q_0 is updated to $\{x_1, x_2\}$, because they are assigned by non-empty strings in this transition. By the transition (q_0, b, q_1), the nonempty variable sets of q_1 is updated to $\{x_1, x_2\}$.

At last, we find the intersection of the used variables and the nonempty variables. In this example, $\{x_0, x_1, x_2\} \cap \{x_1, x_2\} = \{x_1, x_2\}$ is the set of the non-redundant variables. □

7.3 Experiments

Firstly we use one test case to show details of composition of SSTs and removal of redundant variables. We solve Case 1 with and without the optimization of removing redundant states and variables. Here we use only $\{a, b\}$ as the alphabet.

Case 1.

(1) $x_1 = x_0.replaceAll(a, b)$

(2) $x_2 = a \cdot x_1 \cdot a$

(3) $x_2 \in ab^*a$

This constraint is transformed into the following SSTs.

	$\|Q\|$	$\|X\|$	$\|\delta\|$
(1)	3	2	3
(2)	4	2	6
(3)	7	3	10

The results of the composition of SSTs are in **Table 1**. Without optimization, the size of the variable set grows extremely fast. When composing 3 SSTs together, $|X|$ becomes 1344. This case is satisfiable because the domain of the composed SST is not empty. With the optimization, $|X|$ of the second composed SST is reduced from 1344 to 3. □

We also conducted experiments to compare our procedure with SLOTH [16] and OSTRICH [12]. All the test cases use the 8-bit extended ASCII alphabet. Experiments were executed on a computer with Intel Core i5-5257U CPU @ 2.70 GHz and 8 GiB RAM.

Case 2, Case 3 and Case 4 contain constraints with *replaceAll* functions. Case 5 uses a series of concatenation. Case 6, Case 7, Case 8 and Case 9 contain integer constraints, while Case 8 and Case 9 also use *reverse* functions. We explain some of the test cases while others are shown in Appendix A.1. The results are listed in **Table 2**. We use - to denote the solver is not able to handle this test case (whether the constraint is not supported or the constraint causes an out-of-memory error), # to denote the solver gives an incorrect answer, and * to denote the solver gives a correct answer but cannot generate a witness.

Case 2 (ReplaceAll I).

$$x_1 = x_0.replaceAll(\langle sc \rangle, \epsilon)$$
$$x_1 \in \langle sc \rangle$$

This constraint replaces all the occurrences of $\langle sc \rangle$ with the

Table 1 Results of optimization.

	unoptimized			optimized		
composed SST	$\lvert Q\rvert$	$\lvert X\rvert$	$\lvert\delta\rvert$	$\lvert Q\rvert$	$\lvert X\rvert$	$\lvert\delta\rvert$
(1)–(2)	5	32	15	4	2	6
(1)–(3)	15	1344	45	14	3	21

Table 2 Results of comparing with other solvers.

	Run Time (sec)							
	Case 2	Case 3	Case 4	Case 5 (k = 9)	Case 6	Case 7	Case 8	Case 9
Our Solver	1.257	7.819	2.270	2.995	1.492	1.357	1.384	1.275
SLOTH	1.737	2.058	8.321	-	1.508/*	1.776/*	-	-
OSTRICH	1.107/#	1.285	1.264	1.199	-	-	1.471	-

Table 3 Results of Case 5 with different k.

	SLOTH			OSTRICH	Our Solver
k	AFAs	Final AFA states	Run Time (sec)	Run Time (sec)	Run Time (sec)
1	4	12	1.744	1.017	1.010
2	6	18	1.361	1.052	1.197
3	10	30	1.367	1.051	1.281
4	18	54	1.454	1.084	1.422
5	34	102	1.623	1.065	1.556
6	66	198	2.122	1.082	1.707
7	130	390	3.446	1.099	1.951
8	258	774	7.323	1.096	2.291
9	514	1542	- (OOM)	1.199	2.995

empty string, and then checks whether the result may still be $\langle sc\rangle$. It is satisfiable because we have the following witness.

$$x_0 = \langle sc\langle sc\rangle\rangle \qquad x_1 = \langle sc\rangle$$

SLOTH and our solver solve this constraint correctly, while OSTRICH gives an *unsat* answer. □

Case 2 shows that we cannot remove all the occurrences of $\langle sc\rangle$ by simply replacing them with empty strings. If we use a nonempty string to replace $\langle sc\rangle$, then Case 2 will be unsatisfiable. However, the concatenation of strings without $\langle sc\rangle$ may still lead to a result containing $\langle sc\rangle$ as shown in the next test case.

Case 3 (ReplaceAll II).

$$x_2 = x_0.replaceAll(\langle sc\rangle, a)$$
$$x_3 = x_1.replaceAll(\langle sc\rangle, a)$$
$$x_4 = x_2 \cdot x_3$$
$$x_4 \in a\langle sc\rangle a$$

This constraint replaces all the occurrences of $\langle sc\rangle$ in x_0 and x_1 with a, then checks whether the concatenation of the results may contain $\langle sc\rangle$. It is satisfiable because of the following witness.

$$x_0 = a\langle s \qquad x_1 = c\rangle a \qquad x_2 = a\langle s$$
$$x_3 = c\rangle a \qquad x_4 = a\langle sc\rangle a$$

This constraint is handled by SLOTH, OSTRICH and our solver. As shown in Table 2, our solver is much slower than other solvers. The main reason is due to the fast growth of the size of the composed SST. The final composed SST has 4797 states and 100 variables. □

Case 5 (Concatenation)

$$x_1 = x_0 \cdot x_0$$
$$x_2 = x_1 \cdot x_1$$
$$\cdots$$
$$x_k = x_{k-1} \cdot x_{k-1}$$
$$x_1 \in (ab)^+$$
$$x_k \in (aa)^+$$

This constraint is obviously unsatisfiable. We run this test case for k from 1 to 9. The results are shown in **Table 3**. OSTRICH solves this constraint efficiently, while our solver is slower than OSTRICH. SLOTH uses alternating finite automata (AFA) to represent constraints. As shown in the results, the number of states of AFAs grows exponentially, which is consistent with what we discussed in **Example 4.4**. When $k = 9$, SLOTH encounters an out-of-memory (OOM) error. □

Above all, our solver is slower than SLOTH and OSTRICH when the constraint is supported by them. However, our solver supports the general straight-line constraints including concatenation, transductions realized by SSTs, and integer constraints.

8. Related Works

There have been numerous works on solving string constraints [1], [2], [3], [16], [18], [20]. Among them, we review the works that accommodate transducers and concatenation. We also review the previous research on the sequential composition of SSTs.

Lin and Barceló [18] showed that string constraints in straight-line format are decidable. They gave a method to solve straight-line string constraints including concatenation, regular language membership, and finite-state transducers. Their fragment supports constraints of the form $x = T(y)$, where T is a transduction represented by a possibly nondeterministic finite-state transducer. On the other hand, we represent T by a deterministic SST. The classes of transductions represented by nondeterministic finite-state transducers and deterministic SST are incomparable. However, the class of transductions represented by deterministic SSTs is larger than that of functional transductions represented by finite-state transducers since deterministic SSTs are equi-expressive to deterministic two-way finite-state transducers [9], [14]. Lin and Barceló also showed that their constraints are still decidable when extended with linear arithmetic constraints involving the length of string variables and *IndexOf* constraints.

Holík et al. implemented the string constraint solver SLOTH based on the research of Lin [18] with alternating finite automata as succinct representations of languages [16]. Their solver performs better than ours for most cases where regular expressions are complex.

Chen et al. developed a framework that solves straight-line string constraints by computing pre-image of regular languages under transductions [12]. It supports constraints that satisfy two specific restrictions *RegMonDec* and *RegInvRel*. Based on the framework, they implemented OSTRICH that can handle string constraints involving concatenation, one-way and two-way finite-state transducers(possibly non-deterministic), regular-expression matching, reverse functions, and replaceAll functions. The replacement of their replaceAll functions could contain string variables. The support of integer constraints is limited since in general integer constraints do not follow their restrictions.

Abdulla et al. developed a CEGAR framework to solve string constraints based on flat automata [1]. Their method can handle general string constraints which are not restricted to the straight-line fragment and include string concatenation, equations, transducers, context-free grammars and length constraints.

The sequential composition of SSTs plays a crucial role in this work. A sketch of the construction was first given for copyless SSTs in Refs. [7], [9]. However, it was pointed out later that it does not preserve the restriction of copyless SSTs. Our work is based on the construction of the second author for bounded-copy SSTs [4]. He formalized the construction in Isabelle/HOL and verified its correctness. As far as we know, the only work other than his construction that concretely gives a construction is that of Alur and D'antoni [6], which describes a construction of the composition of streaming tree transducers. However, their construction is not direct in the sense that it depends on the factorization of variable updates into atomic ones.

9. Conclusion

We have implemented a string constraint solver for straight-line constraints involving string concatenation, functions represented by streaming string transducers, regular language membership, and length constraints. Our solver transforms atomic constraints and regular constraints into streaming string transducers, applies the sequential composition to them, and computes the Parikh image of the composed streaming string transducer. Our current implementation is slow in general but fast for some special string constraints.

Future Work Currently, our solver only handles functions from strings to strings represented by deterministic SSTs. Since nondeterministic SSTs are also closed under composition, we believe that our approach can be extended for nondeterministic SSTs. However, we have to give a concrete construction of composition before the extension.

Acknowledgments This work was supported by JSPS KAKENHI Grant Numbers 15K00087 and 19K11899.

References

[1] Abdulla, P.A., Atig, M.F., Chen, Y.-F., Diep, B.P., Holík, L., Rezine, A. and Rümmer, P.: Flatten and Conquer: A Framework for Efficient Analysis of String Constraints, *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.602–617 (2017).

[2] Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P. and Stenman, J.: String Constraints for Verification, *Computer Aided Verification*, pp.150–166 (2014).

[3] Abdulla, P.A., Atig, M.F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P. and Stenman, J.: Norn: An SMT Solver for String Constraints, *Computer Aided Verification*, pp.462–469 (2015).

[4] Akama, H.: Composition of bounded-copy streaming string transducers and its formal verification, Master's thesis, Tokyo Institute of Technology (2019).

[5] Alur, R. and Cerný, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs, *Proc. 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.599–610 (2011).

[6] Alur, R. and D'antoni, L.: Streaming tree transducers, *Journal of the ACM (JACM)*, Vol.64, No.5, pp.31:1–31:55 (2017).

[7] Alur, R. and Deshmukh, J.V.: Nondeterministic Streaming String Transducers, *Automata, Languages and Programming*, pp.1–20 (2011).

[8] Alur, R., Filiot, E. and Trivedi, A.: Regular Transformations of Infinite Strings, *2012 27th Annual IEEE Symposium on Logic in Computer Science*, pp.65–74 (2012).

[9] Alur, R. and Černý, P.: Expressiveness of streaming string transducers, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pp.1–12 (2010).

[10] Barceló, P., Figueira, D. and Libkin, L.: Graph Logics with Rational Relations, *Logical Methods in Computer Science*, Vol.9, No.3 (2013).

[11] Barrett, C., Fontaine, P. and Tinelli, C.: The SMT-LIB Standard Version 2.0, *Proc. 8th International Workshop on Satisfiability Modulo Theories* (2010).

[12] Chen, T., Hague, M., Lin, A.W., Rümmer, P. and Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations, *Proc. ACM on Programming Languages*, Vol.3, pp.49:1–49:30 (2019).

[13] Courcelle, B.: Monadic Second-Order Definable Graph Transductions: A Survey, *Theoretical Computer Science*, Vol.126, pp.53–75 (1994).

[14] Engelfriet, J. and Hoogeboom, H.J.: MSO Definable String Transductions and Two-way Finite-state Transducers, *ACM Trans. Computational Logic*, Vol.2, No.2, pp.216–254 (2001).

[15] Filiot, E., Krishna, S.N. and Trivedi, A.: First-order Definable String Transformations, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science*, Vol.29, pp.147–159 (2014).

[16] Holík, L., Janků, P., Lin, A.W., Rümmer, P. and Vojnar, T.: String Constraints with Concatenation and Transducers Solved Efficiently, *Proc. ACM on Programming Languages*, Vol.2, pp.4:1–4:32 (2017).

[17] Kagae, M. and Minamide, Y.: Equivalence Checking of Streaming String Transducers and Its Application to Regular Expression Replacement, *PRO*, Vol.8, No.3, pp.1–10 (2015).

[18] Lin, A.W. and Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS, *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.123–136 (2016).

[19] Morvan, C.: On Rational Graphs, *Foundations of Software Science and Computation Structures*, pp.252–266 (2000).

[20] Zheng, Y., Zhang, X. and Ganesh, V.: Z3-str: A z3-based string solver for web application analysis, *Proc. 2013 9th Joint Meeting on Foundations of Software Engineering*, pp.114–124 (2013).

Appendix

A.1 Test Cases in Table 2

Case 4 (ReplaceAll III).

$$x_1 = x_0.replaceAll(b, c)$$
$$x_2 = x_1.replaceAll(c, d)$$

...

$x_9 = x_8.replaceAll(j, k)$

$x_0 \in aaba$

This constraint is satisfiable because we have the following witness.

$x_0 = aaba \quad x_1 = aaca \quad x_2 = aada$

$x_3 = aaea \quad x_4 = aafa \quad x_5 = aaga$

$x_6 = aaha \quad x_7 = aaia \quad x_8 = aaja$

$x_9 = aaka$

SLOTH, OSTRICH and our solver can solve this constraint. □

Case 6 (Integer I).

$x_1 = x_0.replaceAll(ab, c)$

$|x_0| >= |x_1| + 5$

This constraint is satisfiable because we have the following witness.

$x_0 = abacacbababbccbaaabbcacccabccacacabbbbbbaccb$

$x_1 = cacacbccbccbaacbcaccccccacaccbbbbbaccb$

$|x_0| = 44 \quad |x_1| = 38$

OSTRICH cannot handle this constraint. SLOTH handles this constraint but fails to generate a witness. □

Case 7 (Integer II).

$x_2 = x_0.replaceAll(a, bb)$

$x_3 = x_1.replaceAll(a, bbb)$

$x_0 \in a^+$

$x_1 \in a^+$

$|x_2| = |x_3|$

This constraint is satisfiable because we have the following witness.

$x_0 = aaa \quad x_1 = aa$

$x_2 = bbbbbb \quad x_3 = bbbbbb$

OSTRICH cannot handle this constraint. SLOTH handles this constraint but fails to generate a witness. □

Case 8 (Reverse I).

$x_1 = x_0.reverse$

$x_0 \in (abc)^+$

$x_1 \in (cba)^+$

$|x_0| = |x_1|$

This constraint is satisfiable because we have the following witness.

$x_0 = abcx_1 = cba$

SLOTH cannot handle this constraint. □

Case 9 (Reverse II).

$x_1 = x_0.reverse$

$x_0 \in (abc)^+$

$x_1 \in (cba)^+$

$|x_0| < |x_1|$

This constraint is unsatisfiable. OSTRICH and SLOTH cannot handle this constraint. □

A.2 Construction of $\mathcal{T}_\mathbb{N}$

For finite sets X and Y, a function $f : Y \to X \to \mathbb{N}$ can be considered as a $|X| \times |Y|$ matrix on $\mathbb{N}$.

For $f : Y \to X \to \mathbb{N}$ and $g : Z \to Y \to \mathbb{N}$, we define $f \cdot g : Z \to X \to \mathbb{N}$ as follows.

$$(f \cdot g)(z)(x) = \sum_{y \in Y} g(z)(y) f(y)(x)$$

For $f_1 : X \to Y \to \mathbb{N}$ and $f_2 : X \to Y \to \mathbb{N}$, we define $f_1 + f_2 : X \to Y \to \mathbb{N}$ as follows.

$$(f_1 + f_2)(x)(y) = f_1(x)(y) + f_2(x)(y)$$

For $f : X \to Y \to \mathbb{N}$ and $v : X \to \mathbb{N}$, we define $f \cdot v : Y \to \mathbb{N}$ as follows.

$$f \cdot v(y) = \sum_{x \in X} v(x) f(x)(y)$$

We define functions $\Psi_X : (X \cup \Gamma)^* \to X \to \mathbb{N}$ and $\Psi_\Gamma : (X \cup \Gamma)^* \to \Gamma \to \mathbb{N}$ as follows.

$$\Psi_X(w)(x) = |w|_x$$
$$\Psi_\Gamma(w)(a) = |w|_a$$

We extend Ψ_X to $M_{X,\Gamma} \to X \to \mathbb{N}$ and Ψ_Γ to $M_{X,\Gamma} \to \Gamma \to \mathbb{N}$ as follows.

$$\Psi_X(\alpha)(x) = \Psi_X(\alpha(x))$$
$$\Psi_\Gamma(\alpha)(x) = \Psi_\Gamma(\alpha(x))$$

Lemma 1. For any $\alpha_1, \alpha_2 \in M_{X,\Gamma}$, we have the following equations.

$$\Psi_X(\alpha_1 \circ \alpha_2) = \Psi_X(\alpha_1) \cdot \Psi_X(\alpha_2)$$
$$\Psi_\Gamma(\alpha_1 \circ \alpha_2) = \Psi_\Gamma(\alpha_1) \cdot \Psi_X(\alpha_2) + \Psi_\Gamma(\alpha_2)$$

Lemma 2. For any $\alpha \in M_{X,\Gamma}$ and $w_f \in (X \cup \Gamma)^*$, we have the following equation.

$$\Psi(\hat{\epsilon}(\alpha(w_f))) = \Psi_\Gamma(\alpha) \cdot \Psi_X(w_f) + \Psi_\Gamma(w_f)$$

Then we construct a nondeterministic transducer $\mathcal{T}_\mathbb{N} = (\Sigma, \Gamma \to \mathbb{N}, Q^A, Q_0^A, \Delta^A, F^A)$ from $\mathcal{S} = (\Sigma, \Gamma, Q, X, q_0, \delta, \eta, F)$, with each component defined as follows.

$$Q^A = \{q_\perp\} \cup \{(q, \Psi_X(\alpha) \cdot \Psi_X(w_f)) \mid \exists w,\ q_f.\hat{\delta}(q, w) = q_f \wedge \\ \hat{\eta}(q, w) = \alpha \wedge F(q_f) = w_f\}$$
$$Q_0^A = \{(q_0, B) \mid (q_0, B) \in Q^A\}$$

$F^A = \{q_\perp\}$
$\Delta^A \subseteq Q^A \times \Sigma \times (\Gamma \to \mathbb{N}) \times Q^A$

Then Q_A is finite since $\mathcal{S}$ is bounded-copy. Δ^A contains the following transitions.
For any $q_f \in dom(F)$ and $F(q_f) = w_f$,

$$(q_f, \Psi_X(w_f)) \xrightarrow{\epsilon/\Psi_\Gamma(w_f)} q_\perp \in \Delta^A.$$

For $(q, B) \in Q^A$, $\delta(q', \sigma) = q$, and $\eta(q', \sigma) = \alpha$,

$$(q', \Psi_X(\alpha) \cdot B) \xrightarrow{\sigma/\Psi_\Gamma(\alpha) \cdot B} (q, B) \in \Delta^A.$$

Lemma 3. If $\hat{\delta}(q, w) = q_f$, $F(q_f) = w_f$, and $\hat{\eta}(q, w) = \alpha$, then we have the following transition in Δ^A.

$$(q, \Psi_X(\alpha) \cdot \Psi_X(w_f)) \xrightarrow{w/\Psi_\Gamma(\alpha) \cdot \Psi_X(w_f) + \Psi_\Gamma(w_f)} q_\perp$$

Lemma 4. If we have a transition $(q, B) \xrightarrow{w/v} q_\perp \in \Delta^A$, then there exist a finial state q_f and a variable update α such that $\hat{\delta}(q, w) = q_f$, $\hat{\eta}(q, w) = \alpha$, $B = \Psi_X(\alpha) \cdot \Psi_X(w_f)$, and $v = \Psi_\Gamma(\alpha) \cdot \Psi_X(w_f) + \Psi_\Gamma(w_f)$ where $w_f = F(q_f)$.
It is easy to prove Lemma 3 and Lemma 4 by induction on $|w|$.
Outline of the Proof of Theorem 1:
(1) If $[\![\mathcal{S}]\!](w) = w'$, then there exists q_f such that $F(q_f) = w_f$, $\hat{\delta}(q_0, w) = q_f$, $\hat{\eta}(q_0, w) = \alpha$, and $\hat{\epsilon}(\alpha(w_f)) = w'$.
By Lemma 3, we have the following transition in $\mathcal{T}_\mathbb{N}$.

$$(q_0, \Psi_X(\alpha) \cdot \Psi_X(w_f)) \xrightarrow{w/\Psi_\Gamma(\alpha) \cdot \Psi_X(w_f) + \Psi_\Gamma(w_f)} q_\perp$$

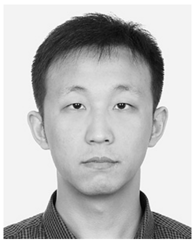
Then by Lemma 2, we have $\Psi_\Gamma(\alpha) \cdot \Psi_X(w_f) + \Psi_\Gamma(w_f) = \Psi_\Gamma(\hat{\epsilon}(\alpha(w_f)))$. Therefore we have the following transition.

$$(q_0, \Psi_X(\alpha) \cdot \Psi_X(w_f)) \xrightarrow{w/\Psi_\Gamma(w')} q_\perp$$

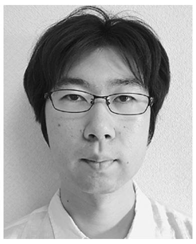
(2) If $(w, v) \in [\![\mathcal{T}_\mathbb{N}]\!]$, then we have the following transition for some B.

$$(q_0, B) \xrightarrow{w/v} q_\perp$$

By Lemma 4, there exist q_f and α such that $\hat{\delta}(q_0, w) = q_f$, $\hat{\eta}(q_0, w) = \alpha$, and $v = \Psi_\Gamma(\alpha) \cdot \Psi_X(w_f) + \Psi_\Gamma(w_f)$ where $w_f = F(q_f)$.
Let $w' = \hat{\epsilon}(\alpha(w_f))$. By Lemma 2, we have $\Psi(w') = v$. □



Qizhen Zhu received his B.Sc. degree in computer science and technology from Zhejiang University, China in 2016. He is currently undertaking a master course at Tokyo Institute of Technology. His research interests include automata theory and software verification.



Hitoshi Akama received his M.Sc. degree and graduated from Tokyo Institute of Technology in 2019. He is interested in formal language theory and its application to software verification.

Yasuhiko Minamide received his M.Sc. and Ph.D. degrees from Kyoto University in 1993 and 1997, respectively. Since 2015, he has been a professor at the Department of Mathematical and Computing Science, Tokyo Institute of Technology. His research interests focus on software verification and programming languages. He is also interested in the theory and applications of automata and formal languages. He is a member of ACM, IPSJ, and JSSST.