Internship report - Streaming String Transducers

Jérémy Ledent
supervised by Anca Muscholl
LaBRI, University of Bordeaux

June 3 to July 12, 2013

1 Introduction

In formal language theory, two very different models sometimes turn out to describe the same class of languages. This usually shows that there is a fundamental concept described by those models. A well-known example is the class of regular languages, which can be characterized by logic (monadic second order (MSO) logic), algebra (syntactic monoids), and many computational models (automata). In particular, it was shown by Rabin and Scott [11] that two-way finite state automata are equivalent to finite state automata, even in presence of non-determinism.

However, these results do not hold when, instead of languages, we consider *transductions*, that is, relations from strings to strings. Indeed, two-way finite state transducers are strictly more powerful than their one-way counterparts. The non-deterministic versions of these models are also strictly more powerful than the deterministic ones. Nonetheless, some notable equalities have been proven between those classes of transductions: in [8], Engelfriet and Hoogeboom have shown that two-way deterministic generalized sequential machines (2DGSM) define the same class of transductions as deterministic MSO string transducers (DMSOS).

Recently, Alur has described a new model for defining string transductions [1], deterministic streaming string transducers (DSST), which is equally expressive as 2DGSM and MSO logic [4]. An interesting feature of this model compared to 2DGSM is that it only does a one-way pass through its input, using a finite number of string variables in order to compute the output. Moreover, when we consider the non-deterministic models, NSST are shown to be as expressive as non-deterministic MSO logic [3], whereas they are incomparable to 2NGSM. Alur has also extended his model to string-to-tree and tree-to-tree transductions, with interesting results in terms of expressiveness and decidability [2].

During this internship, I focused on the deterministic versions of those models. More precisely, I looked at the equivalence between DSST and 2DGSM in terms of resource usage, and tried to exhibit a relation between the number of variables of a DSST and the number of back-and-forth moves of an equivalent 2DGSM. Section 2 introduces the definitions of the different models. Section 3 proves some expressiveness results between DSST, DMSOS and 2DGSM. Section 4 deals with relations between the resources of DSST and 2DGSM.

2 Definitions

2.1 DSST

Description *Deterministic streaming string transducers* (DSSTs), defined in [4], in addition to all transductions implemented by deterministic one-way finite transducers, can implement transductions such as reversing a string and swapping substrings.

A DSST reads the input in a single left-to-right pass. In addition to a finite set of states, it has a finite set of string variables that it uses to produce the output.

In each step, a DSST reads an input symbol, changes its state, and concurrently updates all its string variables using a copyless assignment.

Copyless assignments The right-hand sides in a copyless assignment consist of a concatenation of string variables and output symbols, with the restriction that in a parallel assignment, a variable can appear at most once across all right-hand sides.

For instance, let $X = \{x, y\}$ be the set of string variables, and let $\alpha, \beta, \gamma \in \Gamma^*$ be strings of output symbols. Then, the update $(x, y) = (\alpha.x.\beta.y.\gamma, \varepsilon)$ is a copyless assignment, as it contains only one occurrence of x and y each.

On the other hand, the assignment $(x, y) = (x.y, y.\alpha)$ is not copyless as the variable y appears in the right-hand sides twice.

Formal definition A DSST W is a tuple $(\Sigma, \Gamma, Q, q_0, X, F, \delta_1, \delta_2)$, where:

- Σ is the input alphabet
- Γ is the output alphabet
- $Q = \{q_0, \ldots, q_n\}$ is the set of states
- q_0 is the initial state
- $X = \{x_1, \ldots, x_m\}$ is the set of string variables
- $F : Q \to (\Gamma \cup X)^*$ is the partial output function such that for each $q \in Q$ and $x \in X$, there is at most one occurrence of x in $F(q)$
- $\delta_1 : Q \times \Sigma \to Q$ is the state-transition function
- $\delta_2 : Q \times \Sigma \times X \to (\Gamma \cup X)^*$ is the variable-update function such that for each $q \in Q$ and $a \in \Sigma$ and $x \in X$, there is at most one occurrence of x in the set of strings $\{\delta_2(q, a, y) \mid y \in X\}$

At the beginning of the computation, the DSST is in state q_0 and all variables are mapped to the empty string.
From now on, an assignment such that $\delta_2(q, a, x) = \alpha.x.\beta.y$ and $\delta_2(q, a, y) = \gamma$ will be noted either $(\alpha.x.\beta.y, \gamma)$ or $\left| \begin{array}{l} x := \alpha.x.\beta.y \\ y := \gamma \end{array} \right.$.

2.2 2DGSM

DGSM A *deterministic generalized sequential machine* (DGSM) is similar to a deterministic finite automaton (DFA) except that on each transition, it can output symbols from an output alphabet Γ. The output is written from left to right on a write-only output tape. Formally, a DGSM is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where (Q, Σ, q_0, F) are defined as for DFA, but the transition function is $\delta : Q \times \Sigma \to Q \times \Gamma^*$.

2DGSM A *two-way deterministic generalized sequential machine* (2DGSM), introduced in [9], is a finite-state device with a two-way read-only input tape and a one-way write-only output tape.

In each step, the machine reads an input symbol, changes its state, writes a finite string on its output tape, and moves its reading head according to the finite-state control. The head either moves to the left ($-$) or to the right ($+$).

The string on the input tape is assumed to be $\vdash w \dashv$, where $\vdash$ and $\dashv$ ($\notin \Sigma$) are special symbols known as the left and right end-markers. If it moves right from $\dashv$ or left from $\vdash$, the 2DGSM halts and rejects its input

It is possible that the computation of the machine may not terminate; however, only halting runs contribute to the output. The output of the machine is the string on the output tape if the machine terminates in a designated final state.

Formally, a 2DGSM is specified as the tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q,Σ, Γ, q_0 and F are defined as for DGSMs, while the transition function is $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to Q \times \{-, +\} \times \Gamma^*$.

Visit number A 2DGSM is said to have a *visit number* of k if k is the maximum number of visits done to a position of an input word on which it halts and accepts.

It is useful to note that the visit number of a 2DGSM is bounded by its number of states (otherwise, it would loop because of the determinism). We can also note that for every 2DGSM which may reject an input by looping (and therefore doesn't halt), there is a equivalent 2DGSM which always halts. This result was proven by Sipser, in [12].

2.3 DMSOS

As in [8], to define *deterministic MSO-definable string transductions* (DMSOS), we are going to use the same definition as for MSO-definable graph transductions.

String graph A string $w = w_1 w_2 \cdots w_k$ is viewed as a *string graph* G_w with $k + 1$ vertices $v_0, v_1, \cdots v_k$, with an edge from each v_i to v_{i+1} labeled with the symbol w_i.

Then, a MSO string transduction will be a graph transduction between two string graphs.

Formal definition An *MSO formula* over an alphabet Σ, to be interpreted over a string graph G_w, consists of Boolean connectives, quantifiers, first-order variables that range over vertices of G_w , monadic second-order variables that range over sets of vertices of G_w, and atomic formulas of the form $\text{edge}_a(x, y)$, for $a \in \Sigma$, meaning that the vertex x has an a-labeled edge to the vertex y.

A DMSOS T from input alphabet Σ to output alphabet Γ is defined by:

- A finite copy set C
- Vertex formulas $\varphi^c(x)$, for each $c \in C$, each of which is an MSO formula over Σ with one free first-order variable x.
- Edge formulas $\varphi_\gamma^{c_1,c_2}(x, y)$, for each $\gamma \in \Gamma$ and $c_1, c_2 \in C$, each of which is an MSO formula over Σ with two free first-order variables x and y.

Given an input string w, consider the following output graph: for each vertex x in G_w and $c \in C$, there is a vertex x^c in the output iff the formula $\varphi^c(x)$ holds, and for all vertices x^{c_1} and y^{c_2}, there is a γ-labeled edge from x^{c_1} to y^{c_2} iff the formula $\varphi_\gamma^{c_1,c_2}(x,y)$ holds. If this graph is the string graph corresponding to the string u over Γ then $[\![T]\!](w) = u$, and if this graph is not a string graph, then $[\![T]\!](w)$ is undefined.

3 Expressiveness of DSST

In [8], Engelfriet and Hoogeboom proved that DMSOS = 2DGSM. In a recent paper, Alur and Cerny [4] showed that DSST are equally expressive to these two other models.

3.1 Proof of $DSST \subseteq DMSOS$

First, let us formalize Alur's proof of the reduction from DSST to DMSOS.

Theorem 1. *For every deterministic streaming string transducer W there exists a deterministic MSO transducer T such that $[\![W]\!] = [\![T]\!]$.*

Proof. The idea is to find an MSO-definable graph transformation that is able to simulate the behavior of any streaming string transducer. Let's see the general idea on an example: the transduction f_3 from Alur's article [4], which replaces each symbol b by as many bs as there are as between this occurrence of b and the previous occurrence of b:

$$f_3(a^{i_1}ba^{i_2}b\cdots a^{i_k}ba^{i_{k+1}}) = a^{i_1}b^{i_1}a^{i_2}b^{i_2}\cdots a^{i_k}b^{i_k}a^{i_{k+1}}$$

This transduction is implemented by a DSST with one state q_0 and two variables x and y. When it reads the letter a, it stores a in x and b in y. When it reads the letter b, it concatenates x and y in x, and resets y to ε. At the end, the output is in the variable x. For example, $f_3(aaba) = aabba$.

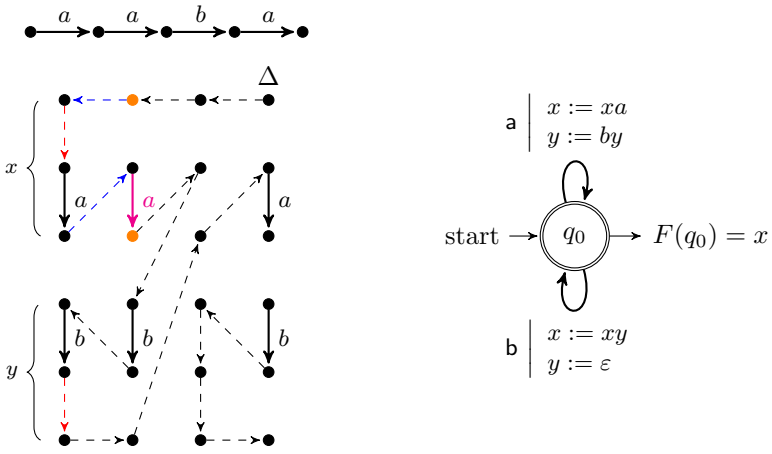


Figure 1: Graph transformation associated with the DSST implementing f_3

On top of figure 1 is the graph representation of the input string $aaba$. Under each letter of the input string, 6 nodes represent the new assignments to the variables when the

DSST reads this letter. The top three nodes represent the assignment to x, and the bottom three nodes represent the assignment to y. For example, under the second letter a of the input string, the assignment $x := xa$ is represented in the top three nodes. The symbol a is represented by a labeled edge (pink), and the variable x is written by connecting the former content of the variable x from the previous column, using dashed (ε-labeled) edges (blue). After this assignment, the new content of the variable x can be read between the two orange nodes. In the first column, variables are represented by a dashed edge (red) since they are empty in the beginning of the computation. In the last column, the symbol Δ denotes the beginning of the output, and some variables might be linked depending on the output function (it it not the case here since the output is just x).

Let us formalize the MSO formulas defining this graph transformation. Let $W = (\Sigma, \Gamma, Q, q_0, V, F, \delta_1, \delta_2)$ be a deterministic streaming string transducer, where $Q = \{q_0, \dots, q_n\}$ and $V = \{v_1, \dots, v_m\}$ is the set of string variables (noted v in order to avoid confusion with the x's and y's in the logic formulas).

Firstly, in order to represent an assignment of the form $v := w_1 \cdots w_k$ with k symbols $w_i \in (\Gamma \cup V)$ in the right-hand side, we need $k+1$ nodes. Thus, the copy-set C must be such that $|C| = m \times p$, where m is the number of variables and $p = \max_{q,a,v}(|\delta_2(q,a,v)| + 1)$.

For all $c \in C$, let $\varphi^c(x) = true$, i.e., every node has $|C|$ copies.

Now, we have to define the formulas $\varphi_\gamma^{c_1,c_2}(x,y)$, which denote that there is an edge from the c_1^{th} copy of x to the c_2^{th} copy of y, labeled by γ. We need a formula $\text{States}(X_0, \cdots, X_n)$ which means that the second-order variables X_i describe a run of the DSST.

We first define $\text{out}_a(x) = \exists y.\text{edge}_a(x,y)$ which means that we read a in position x.

$$
\begin{aligned}
&\text{States}(X_0, \cdots, X_n) = \\
&\quad \forall x \Bigg(\quad \left(\bigwedge_{i \neq j} \neg(x \in X_i \wedge x \in Xj) \wedge \bigvee_i x \in X_i \right) \\
&\qquad \wedge \quad (\text{first}(x) \Rightarrow x \in X_0) \\
&\qquad \wedge \quad \left(\bigwedge_{\substack{a \in \Sigma \\ i,j \in [\![0,n]\!] \\ \delta_1(q_i,a)=q_j}} (x \in X_i \wedge \neg\text{last}(x) \wedge \text{out}_a(x) \Rightarrow \exists y (y \in X_j \wedge \text{edge}_a(x,y))) \right) \\
&\qquad \wedge \quad \left(\text{last}(x) \Rightarrow \bigvee_{\substack{i \in [\![0,n]\!] \\ F(q_i) \text{ is defined}}} x \in X_i \right) \\
&\quad \Bigg)
\end{aligned}
$$

For node x of the input string and each variable v_i, we need p copies of x. We name the elements of the copy-set: $C = \{v_i^\alpha \mid i \in [\![1,m]\!], \alpha \in [\![1,p]\!]\}$. Then we write the formulas $\varphi_\gamma^{c_1,c_2}(x,y)$ depending on c_1 and c_2 as follows:

There is a vertical edge labeled by γ between two consecutive copies of a node iff they correspond to the same variable v_i and the current state and letter read correspond to

a transition where the letter γ appears in the right place in the right-hand side of the assignment. We get the formula:

$$\varphi_{\gamma}^{v_i^{\alpha},v_i^{\alpha+1}}(x,y) = \exists X_0 \cdots \exists X_n (\text{States}(X_0,\cdots,X_n) \wedge \left(x = y \wedge \bigvee_{\substack{k,a \\ \delta_2(q_k,a,v_i)[\alpha]=\gamma}} (x \in X_k \wedge \text{out}_a(x)) \right))$$

The case of the letter ε is particular: we also have to draw an ε-labeled edge in the first column for the empty variables at the beginning.

$$\varphi_{\varepsilon}^{v_i^{\alpha},v_i^{\alpha+1}}(x,y) = \exists X_0 \cdots \exists X_n (\text{States}(X_0,\cdots,X_n) \wedge (\left(x = y \wedge \text{first}(x) \wedge \bigvee_{\substack{k,a,j \\ \delta_2(q_k,a,v_i)[\alpha]=v_j}} \text{out}_a(x) \right) \vee \left(x = y \wedge \bigvee_{\substack{k,a \\ |\delta_2(q_k,a,v_i)|<\alpha}} (x \in X_k \wedge \text{out}_a(x)) \right)))$$

In order to draw the diagonal ε-labeled edges from one column to the next one, we have to check if there is a symbol corresponding to the right variable at the right position:

$$\varphi_{\varepsilon}^{v_i^{p},v_j^{\alpha+1}}(x,y) = \exists X_0 \cdots \exists X_n (\text{States}(X_0,\cdots,X_n) \wedge \left((y = x+1) \wedge \bigvee_{\substack{k,a \\ \delta_2(q_k,a,v_j)[\alpha]=v_i}} (x \in X_k \wedge out_a(x)) \right))$$

For the edges from a column to the previous one, we get a similar formula $\varphi_{\varepsilon}^{v_i^{\alpha},v_j^{1}}(x,y)$ where $y = x+1$ is replaced by $x = y+1$. We also need to have formulas for the edges which link the variables together in the last column, and which write the Δ label on the first node. The formulas $\varphi_{\gamma}^{c_1,c_2}(x,y)$ which don't correspond to any of those cases are set to *false*. If it corresponds to several different cases, we have to merge the formulas.

In the end, we have successfully defined all the formulas using monadic second-order logic. Thus, this graph transformation is MSO-definable. In order to get a string graph, we have to delete the ε labeled edges and the nodes which are not accessible from the first one. This transformation is also MSO-definable and was formalized by Engelfriet and Hoogeboom [8]. Since MSO-definable transductions are closed under sequential composition [6], we have finally proven the theorem.

□

3.2 Direct reduction from DSST to 2DGSM

Alur has shown that DSST = DMSOS = 2DGSM [4]. The disadvantage of his proof is that there is no direct reduction between DSST and 2DGSM. We will now try to write a direct reduction from DSST to 2DGSM, and see if it helps us find a relation between the number of variables of a DSST and the number of visits of the associated 2DGSM.

3.2.1 Use of Hopcroft's property

We are going to use the following property [5]:

Theorem 2 (Hopcroft)**.** DGSM $\circ$ 2DGSM $\subseteq$ 2DGSM

Proof. The idea of the proof can be given by the beginning of the article of Chytil and Jakl [6] and relies on the construction of a "dgsm inverse" explained in Hopcroft and Ullman's article [9].

Let A_1 be a DGSM and A_2 a 2DGSM. We want to construct a 2DGSM A_3 such that $A_3 = A_1 \circ A_2$. On an input w, A_1 produces an output w', which is then used as an input by A_2, which finally produces the output word w''. We want A_3 to simulate this composition by outputting w'' on the input w.

In fact, A_3 does the same computation as A_2, but it also needs to keep track of the state in which A_1 would be at the current position. This way, when reading a letter of w, it can determine what letters of w' are output by A_1, and thus simulate the behavior of A_2. This is easily done when A_2 moves right by following the transitions in A_1's automaton. But when A_2 goes left, A_3 needs to do one step to the left in the run of A_1. This amounts to "guessing" in which state A_1 was before reading the last letter. This is done by the following algorithm:

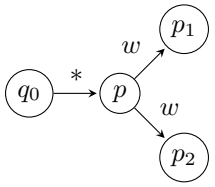
Algorithm for reverse run (based on [9]) Assume that $A = (Q, \cdots)$ is a DGSM, then a 2DGSM $B = (Q', \cdots)$ with $Q \subseteq Q'$ can be constructed with the following behavior:

Let $w = a_1 \cdots a_n$ be an input to A and a_i a position in w such that A reaches state r at position a_{i-1} and state q at position a_i. Then, starting on w at position a_i in state q, the 2DGSM B ends in position a_{i-1} in state r. Let us use two functions: δ^*, the next state function defined as a mapping from $Q \times \Sigma^*$ to Q, and γ, the preceding states function, defined as follows: $\gamma(q, w) = \{p \mid \delta^*(p, w) = q\}$.

B needs to find the preceding state r of A in the current run. In order to do that, B looks at the value of $\gamma(q, a_{i-1})$. If it contains only one state, then this is the desired state r and the algorithm is finished.

Let $\gamma(q, a_{i-1}) = \{q_1, \cdots, q_s\}$, $s \geq 2$.

We remark that, A being deterministic, the following configuration cannot occur (with $w \in \Sigma^*$, and $p_1 \neq p_2$):



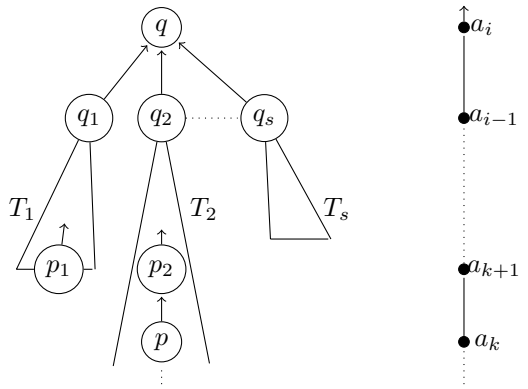
So we can define s disjoint subtrees $T_1, \cdots T_s$ as follows:

$$T_j = \{(p, j) \mid 1 \leq k < i-1,\ \delta^*(p, a_k \cdots a_{i-2}) = q_j\}$$

B goes down in the subtrees, computing successively $\gamma(q_j, a_k \cdots a_{i-2})$ for $1 \leq j \leq s$ and $k = i-2, i-3, \cdots$. Two configurations may appear:

1. B reaches the following configuration for some k:

   - $\gamma(q_j, a_{k+1} \cdots a_{i-2})$ is non empty for at least two subtrees;
   - $\gamma(q_j, a_k \cdots a_{i-2})$ is empty except for exactly one subtree (which is one among the non empty subtrees one step before). If this subtree is T_l, then q_l is the state r we are looking for.



   In this example, $\gamma(q_j, a_{k+1} \cdots a_{i-2})$ is non empty for T_1 and T_2, and at the next step the only non-empty one is T_2. The desired state is thus $r = q_2$.

   Let $m \neq m'$ such that $\gamma(q_m, a_{k+1} \cdots a_{i-2})$ and $\gamma(q_{m'}, a_{k+1} \cdots a_{i-2})$ are non empty. B then selects two states p_m and $p_{m'}$ such that p_m is in $\gamma(q_m, a_{k+1} \cdots a_{i-2})$ and $p_{m'}$ is in $\gamma(q_{m'}, a_{k+1} \cdots a_{i-2})$.

2. B reaches the left endmarker ; only one tree T_l contains the initial state q_0 (otherwise, A would not be deterministic), the desired state is $r = q_l$. It can be obtained by computing $\delta^*(q_0, a_1 \cdots a_{i-2})$. Then, B can find, for distinct m and m', two states p_m and $p_{m'}$, in $\gamma(q_m, a_2 \cdots a_{i-2})$ and $\gamma(q_{m'}, a_2 \cdots a_{i-2})$ respectively.

When having found p_m and $p_{m'}$, B uses them to come back to state q. It thus computes $\delta^*(p_m, a_{k+1} \cdots a_j)$ and $\delta^*(p_{m'}, a_{k+1} \cdots a_j)$, for successive values of j, starting with $j = k + 1$. As T_m and $T_{m'}$ are disjoint, $\delta^*(p_m, a_{k+1} \cdots a_j) = \delta^*(p_{m'}, a_{k+1} \cdots a_j) = q \Leftrightarrow j = i - 1$. Therefore, B is able to go back to the position a_{i-1}, in the state r, and continue its computation.

□

3.2.2 General idea

Given a DSST W, we want to build a 2DGSM which simulates W. Using Hopcroft's property, we decompose it into a DGSM and a 2DGSM. First, the DGSM labels each letter of the input string with the operation done on the variables of the DSST at this position. Then, the 2DGSM produces the same output as W, using this labeled string as input.

Example 1 We show the idea on an example: $(A^nB^p)^k \to (a^nc^pb^nd^p)^k$, with two string variables x and y. We can implement this translation via a DGSM and a 2DGSM, this composition being equivalent to a 2DGSM.

We define the following DSST implementing this transduction:

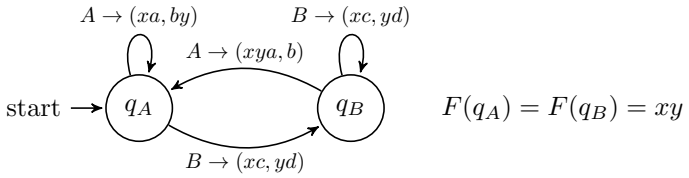


Figure 2: DSST implementing the transduction $(A^nB^p)^k \to (a^nc^pb^nd^p)^k$

Let the input be $ABAB$. The idea is first to label the letters with the operations applied to them, done by a DGSM.

Then we use a 2DGSM to form the output, by "following" the formation of the string variables (see figure 3).

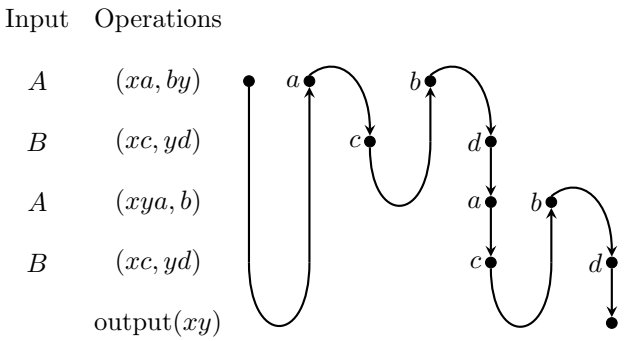


Figure 3: Here, the output of 2DGSM is *acbdacbd*

The 2DGSM first goes through the whole input to see which string variable will be used first in the output (here it is x). It then browses the input in reverse to know what are the output symbols used in this variable, adding to its output all symbols which are written before the string variable in the operation. When reaching the beginning of the output or completing the value of the string variable, the 2DGSM goes on in the valuation of the following string variables.

3.2.3 Formal definition

Let $W = (\Sigma, \Gamma, Q, q_0, X, F, \delta_1, \delta_2)$, with n states and m variables.

Construction of the DGSM The DGSM labeling the states of the input word has input alphabet Σ and output alphabet $\widehat{\Sigma} = \{(\delta_2(q, a, x))_{x \in X} \mid q \in Q, a \in \Sigma\} \cup \{F(q) \mid q \in Q\}$. Its set of states is Q and for each transition $\delta_1(q, a) = q'$ of W, there is a transition

$\delta(q,a) = (q', (\delta_2(q,a,x))_{x\in X})$. In the final state q_f, it also adds a symbol indicating the output $F(q_f)$. We get as output of this DGSM the sequence of operations that W does on its variables while reading the input.

Construction of the 2DGSM Now, we need to build a 2DGSM T with input alphabet $\widehat{\Sigma}$ and output alphabet Γ. Its set of states is $\widehat{Q} = \{q_0, q_{acc}\} \cup \{x_i^L, x_i^R \mid x_i \in X\}$. We have $|\widehat{Q}| = 2m + 2$.

- In the initial state q_0, T goes right until it finds the end of the input string. The last symbol indicates what the output of the DSST is: if the first variable used in the output is x_i, T goes in state x_i^L to start writing the content of the variable x_i.
- In state x_i^L, T writes down the symbols that were added at the left of the variable x_i, while going from right to left on the input string.
- In state x_i^R, T has just finished writing the current content of the variable x_i. At this position, thanks to the copyless restriction of DSST, there is a unique assignment of the form $x_j := u.x_i.v$, where $u, v \in (\Gamma \cup X)^*$. If v doesn't contain any variable, then T outputs v, goes to state x_j^R and does one step to the right. If the first variable appearing in v is x_k, then T outputs all the symbols in v before x_k, goes to state x_k^L and does one step to the left, to start writing down the contents of x_k.
- When the output is fully written, T goes to the accepting state q_{acc}.

For readability, suppose we have only three variables x, y and z, and one of the symbols in $\widehat{\Sigma}$ is $\varphi = \left| \begin{array}{l} x := a.x.b \\ y := c \\ z := d.z.e.y.f \end{array} \right.$ where $a, b, c, d, e, f \in \Sigma$.

Then we have the following transitions:

1. $(x^L, \varphi) \to (x^L, a, -)$
2. $(x^R, \varphi) \to (x^R, b, +)$
3. $(y^L, \varphi) \to (y^R, c, +)$
4. $(y^R, \varphi) \to (z^R, f, +)$
5. $(z^L, \varphi) \to (z^L, d, -)$
6. $(z^R, \varphi) \to (y^L, e, -)$

The transition 4 is particularly tricky: the state y^R means that the current content of variable y has just been written. So we continue writing what comes after the symbol y in the right-hand side of the assignment $z := d.z.e.y.f$, i.e., the letter f. But now, we need to go to state z^R since the variable which has just been written is z, not y.

In the general case, that kind of construction is always possible thanks to the copyless restriction of the DSST. With m variables, we have to add exactly $2m$ transitions for each symbol in $\widehat{\Sigma}$. We also need to have transitions for the symbol indicating the beginning of the input: $(x_i^L, \vdash) \to (x_i^R, \varepsilon, +)$. The transitions for the output function F are similar to those for the operations on variables.

In Figure 4 we show the 2DGSM of example 1, working on the alphabet of operations $\widehat{\Sigma} = \{(xa, by), (xc, yd), (xya, b), output(xy)\}$.

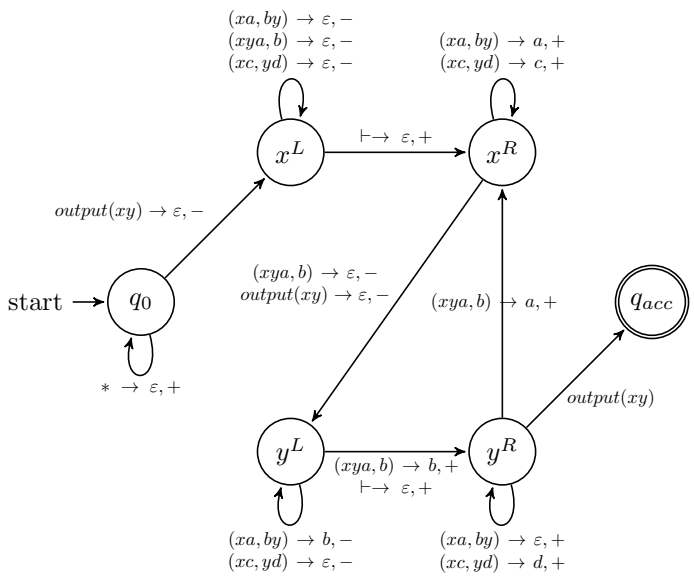


Figure 4: 2DGSM of example 1

Proof of correctness. Let us represent the computation of the DSST by an ordered tree. In order to explain how such a tree is constructed, we will focus on example 1 (see Figure 5), but this proof works in the general case. The internal nodes of the tree are labeled by variables, and the leaves are labeled by symbols in the output alphabet $\Gamma \cup \{\varepsilon\}$. The root is a special node with no label.

- The input $ABAB$ is written on the left, from bottom to top. The corresponding operations on variables are written next to it.
- At the top of the tree is the root (level 0), which links to each symbol x and y of the output (level 1).
- At the next levels, the children of the node x depend of the assignment to x. For example, at level 2, the operation is $x := xc$ so we add two nodes x and c whose father is the x node from the previous level. This construction is always possible thanks to the copyless restriction of the DSST.
- At the last level, each variable x and y is linked to the symbol ε since they are empty in the beginning.

The output of the DSST is obtained by performing a depth-first search (DFS) of the tree and writing the letters in the leaves of the trees in the order we visit them. In figure 5, the output is *acbdacbd*. Let us follow the run of the 2DGSM from Figure 4 in order to see that it does exactly the same DFS and thus outputs the same word as the DSST from Figure 2.

- First, in state q_0, it goes to the end of the input word until it reaches the symbol $output(xy)$ indicating the output of the DSST. This corresponds to the level 1 of the tree. It then goes to state x^L since x is the first variable that appears in the output (i.e., the leftmost child of the root).

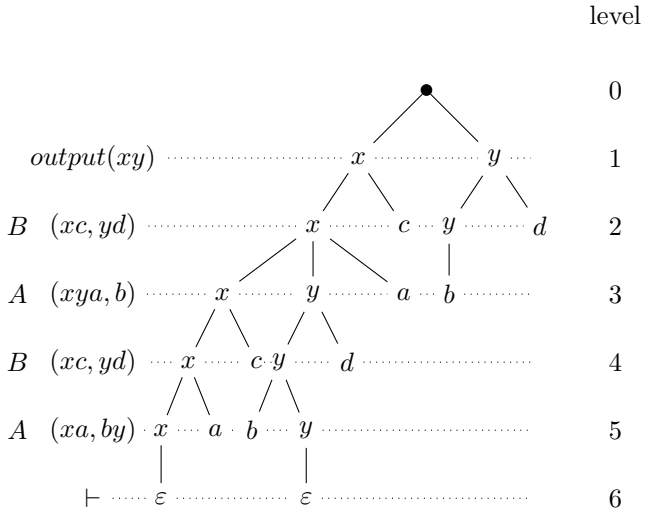


Figure 5: Tree associated with the computation of ABAB in example 1

- In state x^L, it goes down the x-labeled branch, and writes down ε symbols since the x node never has left siblings. When it reaches the symbol $\vdash$, it interprets x as ε and goes to state x^R to start going up.
- In state x^R, it goes up the x-labeled branch and writes down the right siblings of x: a and c. Then at level 3 it encounters the variable y and goes to state y^L.
- In state y^L, it goes down the y branch and writes a b since it is a left sibling of y. Then it goes back up and writes the letter d.
- When it reaches level 3 in state y^R, it outputs the right sibling of y (letter a), and then goes to state x^R since the father of y is x.
- It then keeps going up and outputs c. When it reaches level 1 in the stats x^R, the part of the output contained in variable x has been written. It goes to state y^L to start writing the second part of the output.
- It goes down until level 3 where the operation $y := b$ is done. This corresponds to a leaf of the tree so it writes a b and goes back up in state y^R to write the last d.
- When it reaches the first level in state y^R, the output is finished, so it goes to the accepting state. The output is *acbdacbd*.

□

Number of visits If the DSST W has m variables, after labeling the input with a DGSM, we can build a 2DGSM with $2m + 1$ non-accepting states which gives the same output as W. Hence, the number of visits on each letter of the input is bounded by $2m + 1$.

Theorem 3. *Given a* DSST *with n states and m string variables, we construct an equivalent* 2DGSM *of size $\mathcal{O}\big(2^{n \log n} \times m\big)$.*

Proof. This is due to Hopcroft's construction (see page 7) : we compose a DGSM A of size $\mathcal{O}(n)$ with a 2DGSM B of size $\mathcal{O}(m)$. In order to compute the reverse run of A, we need a sub-automaton which memorizes at most n disjoint sets of states. Its thus needs $\mathcal{O}(n^n) = \mathcal{O}\big(2^{n \log n}\big)$ states. Moreover, we also have to remember the current state of B, hence the total size is $\mathcal{O}\big(2^{n \log n} \times m\big)$. □

In the next section, we will introduce machines with *regular look-ahead (RLA)* : given a $\text{DSST}_{\text{RLA}}$with m string variables, we construct a $\text{2DGSM}_{\text{RLA}}$of size $\mathcal{O}(m)$ (see Theorem 4).

4 Relation between number of variables of a DSST and the visit number of the associated 2DGSM

In the previous section, we showed that a DSST with m variables can be simulated by a $(2m+1)$-visiting 2DGSM *after* labeling the input with a DGSM. Unfortunately, Hopcroft's construction doesn't preserve the number of visits of the 2DGSM. In fact, it is not possible to find a relation between the number of variables and the number of visits, without taking into account the number of states.

4.1 Counter-examples

Counter-example 1 Consider the transduction $R^n_{copy} \subseteq \Sigma^* \times \Sigma^*$ (where $\Sigma = (\bigcup^n \Sigma_i) \cup \{\#\}$ and the Σ_i are pairwise disjoint), defined as the following set:
$\{(w_1\#w_2\#\dots\#w_n\#\alpha, w_\alpha) \mid n \geq 0, w_i \in \Sigma_i^*, 0 \leq \alpha \leq n\}$.
The goal is here to copy the α^{th} word.

This tranduction can be implemented by a DSST using n string variables $x_1, \dots, x_n$, recording the values of the n words w_i. After reading the end of the input, the value of x_α is given as the output.

Because of the use of n different alphabets Σ_i, a single string variable cannot be used to record the value of two different words. Thus, the transducer needs at last n string variables to record the values of the n different words.

The transduction R_{copy} can also be implemented by a 2DGSM: the 2DGSM reads the input till the end, recording the value of α. Then, the 2DGSM moves backwards, until the beginning of the α^{th} word, and copies the value of w_α. Thus, this transducer visits three times the positions contained in w_α, one time the positions before, and two times the positions after these. So the visit number of this 2DGSM is 3.

Counter-example 2 Consider the transduction $R_{ex} \subseteq \Sigma^* \times \Sigma^*$ defined as the following set: $\{(w_1a_1w_2a_2\dots w_ka_kw_{k+1}, a_{k+1}^{n_{k+1}}a_k^{n_k}\dots a_2^{n_2}a_1^{n_1}) \mid w_i \in (\Sigma \setminus \{a_i, \dots, a_k\})^*, n_i = |w_i|\}$, where $a_1, \dots, a_{k+1}$ are fixed and pairwise different. Informally, the input is divided into k input blocks w_i, where w_i begins at the first occurrence of a_{i-1}, and ends at the first occurrence of a_i. The output depends of the size of each block, from the last to the first.

This transduction can be implemented by the DSST from figure 6, using a single string variable x. The state indicates in which block the reading head is: when reading a_i in state q_i, it goes to state q_{i+1}. The letters are written to the left of variable x in order to reverse the order of the blocks in the output.

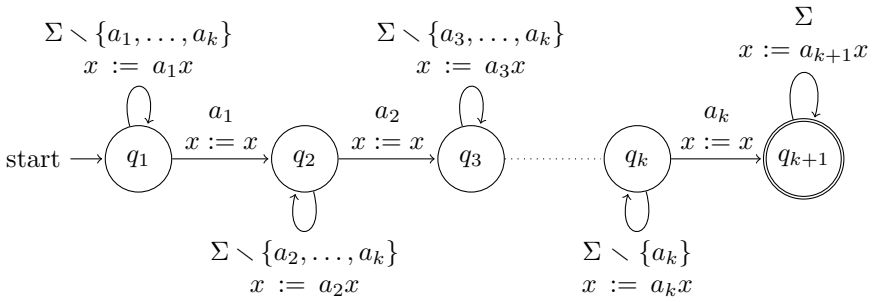


Figure 6: DSST implementing the transduction R_{ex}

The transduction R_{ex} can also be implemented by a 2DGSM. Since it must write its output from left to right, it must first find the last block w_{k+1} and output its length. Then, being at the end on the input, it must determine where the block w_k begins, i.e., find the first occurrence of a_{k-1}. Since the 2DGSM has a finite memory and the input word can be as long as we want, this cannot be done without going back to the beginning of the input. In fact, every time it writes down the length of a block w_i, it must go back to the beginning of the input in order to find the previous block w_{i-1}. Thus, this 2DGSM is necessarily $(2k+1)$-visiting.

We have shown with these two counter-examples that there is no relation between the number of variables used in a DSST and the visit number of an equivalent 2DGSM ; they can get as big as we want, without influencing one another.

4.2 Adding regular look-around / look-ahead

We are now going to consider slightly different versions of DSST and 2DGSM, by allowing them to test a condition on their input on each transition. The transition can be done iff the condition is verified.

4.2.1 Definitions

- A DSST with *regular look-ahead* (DSST_{RLA}) is an extension of the DSST model in which the machine can test at each transition whether the suffix of the input word situated to the right of the reading head belongs to some regular language.

- A 2DGSM with *regular look-around* ($\text{2DGSM}_{\text{RLA}}$) is an extension of the 2DGSM model in which the machine can test at each transition whether the prefix to the left and the suffix to the right of the reading head belong to some regular languages.

Remark In order to preserve determinism, for a given state q and input letter a, all tests must be done on regular languages that are disjoint.

4.2.2 From $\mathbf{DSST_{RLA}}$ to $\mathbf{2DGSM_{RLA}}$

Theorem 4. *For every* DSST_{RLA} *with* k *string variables, there exists an equivalent* $(2k+1)$*-visiting* $2\text{DGSM}_{\text{RLA}}$*. The number of states of the* $2\text{DGSM}_{\text{RLA}}$ *is equal to* $2k+2$*.*

Proof. Let W be a $\text{DSST}_{\text{RLA}}$with k variables. We want to construct a $2\text{DGSM}_{\text{RLA}}T$ which simulates W, with a number of visits bounded by $2k+1$.

The idea is the same as in section 2.2.3, except that the work of the DGSM is now done using regular look-around tests. In fact, the only thing T needs to know at a given position is the operation done by W on its variables. This is determined by the state of W on that position.

Let Q be the set of states of W. For every $q \in Q$, let A_q be a deterministic finite automaton (DFA) defined as follows:

- Its set of states is Q, its initial state is the same as W's
- Its transition function δ is the same as the state-transition function of W
- Its set of accepting states is $F = \{q\}$

Let R_q be the regular language recognized by the DFA A_q. A prefix $w_1 \cdots w_i$ of the input word is in R_q iff the DSST W is in state q at position $i+1$. Therefore, T can guess the state of W using a regular look-around test on the prefix to the left of the reading head.The tests to the right are used to simulate the regular look-ahead behavior of W. Moreover, since the DFAs A_q are deterministic and have disjoint sets of accepting states, the languages R_q are disjoint. Hence, the determinism is preserved. Since the set of states of T is still $\widehat{Q} = \{q_0, q_{acc}\} \cup \{x_i^L, x_i^R \mid x_i \in X\}$ as in section 2.2.3, its number of visits is bounded by $2k+1$.

□

4.2.3 From $\mathbf{2DGSM_{RLA}}$ to $\mathbf{DSST_{RLA}}$

Theorem 5. *For every* k*-visiting* $2\text{DGSM}_{\text{RLA}}$*, there exists an equivalent* DSST_{RLA} *with* $\lceil \frac{k}{2} \rceil$ *string variables. The size of the* DSST_{RLA} *is exponential in the number of states of the* $2\text{DGSM}_{\text{RLA}}$*.*

Proof. Let T be a k-visiting $2\text{DGSM}_{\text{RLA}}$. We want to simulate it with a $\text{DSST}_{\text{RLA}}W$. First, let us explain the idea of the construction.

General idea Let us consider a run of a 3-visiting $2\text{DGSM}_{\text{RLA}}T$. Such a run is shown on figure 7. T does some back-and-forth moves on the input word w and outputs a word $w_1.w_2 \cdots w_9$. There are 4 special positions on the input word which correspond to the turns of T and are labeled (a), (b), (c) and (d). Since T is 3-visiting, we must show that it can be simulated by a $\text{DSST}_{\text{RLA}}W$ with 2 variables x and y. Let us assume that, thanks to regular look-ahead tests, W can determine the positions of those turns. We will explain later how this is done.

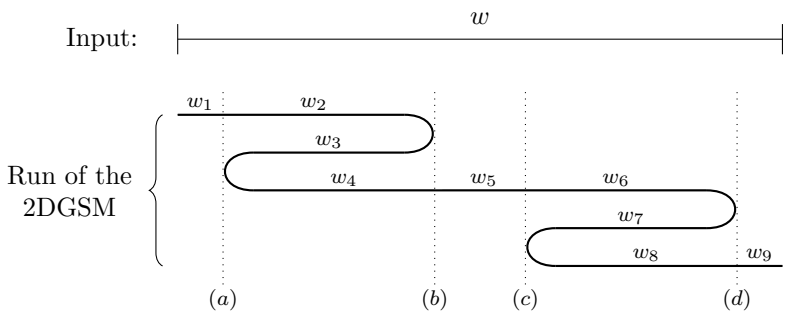


Figure 7: Run of a 2DGSM on input w

The behavior of W is the following:

- It starts writing w_1, the beginning of the output, in the variable x, until it finds a turn.
- When it reaches (a), $x = w_1$ and $y = \varepsilon$. It starts writing w_2 in x, and $w_3.w_4$ in y. This can be done by adding the symbols of w_3 at the left of y, and the symbols of w_4 at the right.
- When it reaches (b), $x = w_1.w_2$ and $y = w_3.w_4$. W concatenates the content of the two variables in x ($x := xy$) and frees the variable y ($y := \varepsilon$). It starts writing w_5 at the right of x.
- When it reaches (c), it does the same thing as at turn (a): it writes the next part w_6 of the input in x, and starts writing $w_7.w_8$ in y.
- When it reaches (d), it concatenates x and y as at turn (b), so $x = w_1 \cdots w_8$ and $y = \varepsilon$.
- When it reads the last letter of the input, the output $w_1 \cdots w_9$ is stored in variable x.

We notice that there are two types of turns: left turns (such as (a) and (c)) and right turns (such as (b) and (d)). At each left turn, one new variable is used. At each right turn, one used variable is freed and can be re-used later. A $2k+1$-visiting $2\text{DGSM}_{\text{RLA}}$can do at most k consecutive left turns, the associated DSST_{RLA}thus needs $k+1$ variables (it also needs one variable for the first left-to-right pass of the $2\text{DGSM}_{\text{RLA}}$).

Crossing sequences In order to guess the run of the $2\text{DGSM}_{\text{RLA}}$, we are going to use the same construction as for the reduction from 2DFA to NFA in [10].

Let us represent a run of a 2DFA on its input tape by labeling each boundary between two tape squares with the sequence of states in which it is crossed. For example, figure 8 shows a 2DFA recognizing the language of words in $\{0,1\}^*$ with no two successive occurrences of 1, along with a labeled run on input 101001.

- The ordered list of states under the boundary between two input squares is called a *crossing sequence*. Note that if the input is accepted, a state cannot be repeated twice in the same crossing sequence with the head going in the same direction (otherwise, the 2DFA would loop). Moreover, since the head starts at the left of

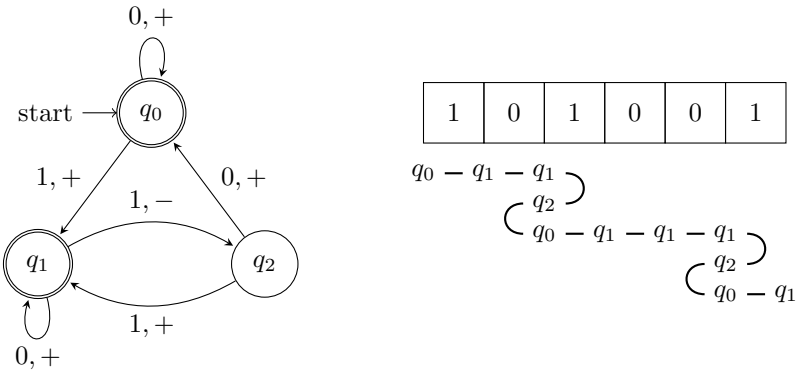


Figure 8: Run of a 2DFA on input 101001

the input, the head must be moving right the first time a boundary is crossed. Subsequent crossings must be in alternate directions.

- A crossing sequence is *valid* if it is of length at most $2s$ with s the number of states of the 2DFA, and no two odd-numbered and no two even-numbered states are identical. The number of valid crossing sequences is finite.
- A valid crossing sequence $q_1, \cdots, q_k$ *matches* $p_1, \cdots, p_l$ on a symbol a (where $q_1, \cdots, q_k$ is the sequence under the left boundary of a and $p_1, \cdots, p_l$ is under the right boundary of a) if these two sequences are consistent assuming that:
  1. a is initially reached in state q_1
  2. whenever the head moves left in state q_i, it comes back in state q_{i+1}
  3. whenever the head moves right in state p_i, it comes back in state p_{i+1}

We can now construct a NFA which recognizes the same language as the 2DFA: its set of states is the set of crossing sequences, and its transitions are such that $\delta(\pi, a) = \{\rho \mid \rho \text{ is a valid crossing sequence that is matched by } \pi \text{ on input } a\}$. On an input w, this NFA guesses the crossing sequences which correspond to the run of the 2DFA on w. The proof of this result is in [10].

One important property of this NFA is *non-ambiguity*, i.e., for every input word w, there is only one run that accepts w. Indeed, if there were two accepting runs of the NFA on an input, we could construct two distinct runs of the original 2DFA on input w, which is absurd because it is deterministic.

Let us first consider a 2DGSM T (without regular look-ahead). If we ignore the output tape and consider it as a 2DFA, we can construct the corresponding NFA M which has as states the crossing sequences of T, as explained previously. The $\text{DSST}_{\text{RLA}}W$ simulating T also has as states the crossing sequences of T, but it must behave in a deterministic manner:

Assume the input is $w = w_1 \cdots w_k$, and the input head is on w_i. After reading the prefix $w_1 \cdots w_{i-1}$, W is in state π. In the NFA M, when reading w_i in state π, several transitions are possible. Let $P = \delta(\pi, w_i)$ be the set of possible new states. For each $\rho \in P$, let R_ρ be the regular language recognized by the NFA M_ρ which is similar to M

except ρ is the initial state. The suffix $w_{i+1} \cdots w_k$ of the input is in R_ρ iff ρ is the next state in the unique accepting run of M. Therefore, the transitions of W are $\delta_1(\pi, w_i) = \rho$ with the regular test $w_{i+1} \cdots w_k \in R_\rho$. Thanks to the non-ambiguity of M, the languages R_ρ are disjoint, hence W can determine the crossing sequences of T using deterministic regular look-ahead tests.

If we now consider that T is a $2\text{DGSM}_{\text{RLA}}$, the same construction still holds. The regular look-around tests T does on the left of its input head can be done by simulating the associated automata along the run of W. Those done on the right of the input head can be tested along with the regular look-ahead tests of W, since regular languages are closed under intersection. W is still able to determine the crossing sequences of T.

Formal construction Let $T = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a k-visiting $2\text{DGSM}_{\text{RLA}}$. We simulate it with a DSST_{RLA} $W = (Q', \Sigma, \Gamma, X, \delta_1, \delta_2, q_0', F')$ where:

- $Q' = \Sigma \times CS$ where CS is the set of valid crossing sequences of T.
- δ_1 behaves as explained in the previous part in order to guess the crossing sequences of T using regular look-ahead tests. It also remembers the last letter read in the first component of the state: instead of $\delta_1(\pi, a_i, R_\rho) = \rho$, the transition becomes $\delta_1((a_{i-1}, \pi), a_i, R_\rho) = (a_i, \rho)$.
- $X = \{x_1, \ldots, x_s\}$ is of size $s = \lceil \frac{k}{2} \rceil$.
- q_0' and F' are defined as for the NFA which guesses crossing sequences. In the final states, the function F' outputs x_1.

The operations on variables are made such that at all times, if the current crossing sequence is $\pi = q_1 \cdots q_k$, the first variable contains the string output by T before it reaches this position in state q_1 ; the second one contains the string output by T between the moment it leaves this position in state q_2 and the moment it returns in state q_3, etc. We might have to swap the contents of some variables in order to keep them in the right order, but it allows us to know which variables are used and which variables are free by looking at the length of the crossing sequence. Let us illustrate this by an example:

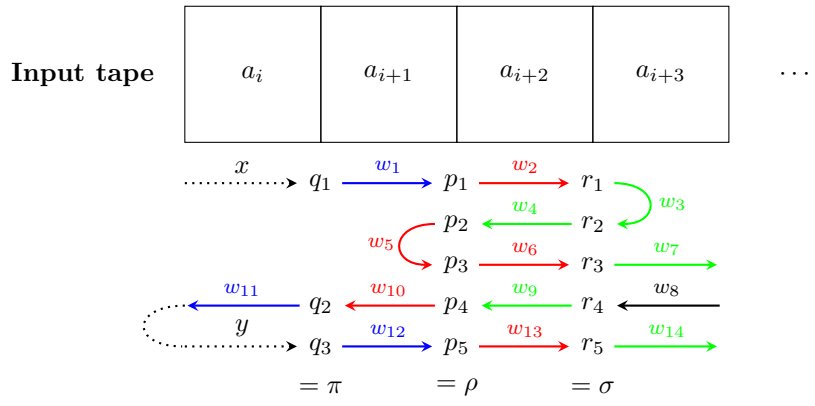


Figure 9: Three successive steps of the DSST

In figure 9 is shown a local part of the run of a $2\text{DGSM}_{\text{RLA}}$ T we want to simulate. Let x, y and z be three variables of the DSST_{RLA} W working on the crossing sequences

$\pi = q_1, q_2, q_3$, $\rho = p_1, p_2, p_3, p_4, p_5$ and $\sigma = r_1, r_2, r_3, r_4, r_5$. The words from w_1 to w_{14} are output by T on those transitions. Let us look at the operations W does on its variables. The goal is to handle correctly the variables x, y, z and to keep them in the right order.

- First, W is in state (a_i, π). Since π is of length 3, we know that x and y are used, and z is free. The contents of x and y are shown in figure 9 by dotted arrows: x contains the output before q_1, and y contains the output between q_2 and q_3.
- W reads the letter a_{i+1}. Knowing a_i, π and a_{i+1}, we can treat all transitions leaving a state of π (the blue ones on the figure). Then, W goes to state (a_{i+1}, ρ). The assignment is:
$$\left|\begin{array}{l} x := x.w_1 \\ y := w_{11}.y.w_{12} \\ z := \varepsilon \end{array}\right.$$
- W reads the letter a_{i+2}. It can now treat all the transitions leaving ρ (red). Since ρ is of length 5, a new variable must be used. Since the new turn is between p_2 and p_3, it is stored in y, and the content of y is put in z. W goes to state (a_{i+2}, σ) and does the assignment:
$$\left|\begin{array}{l} x := x.w_2 \\ y := w_5.w_6 \\ z := w_{10}.y.w_{13} \end{array}\right.$$
- W reads the letter a_{i+3} and treats the transitions leaving σ (green). There is a right-turn between r_1 and r_2, so the contents of x and y must be concatenated in x. In order to preserve the order, the content of z is stored in y, and z is freed:
$$\left|\begin{array}{l} x := x.w_3.w_4.y.w_7 \\ y := w_9.z.w_{14} \\ z := \varepsilon \end{array}\right.$$

Remark It is possible to do a similar construction with fewer states, without remembering the previous letter read. When doing a transition $\delta_1(\pi, a) = \rho$, we treat the output of T in the odd-numbered states of π, and in the even-numbered states of ρ, when reading letter a. For example, in the previous example, the assignment between π and ρ when reading a_{i+1} would be
$$\left|\begin{array}{l} x := x.w_1 \\ y := w_5 \\ z := w_{10}.y.w_{12} \end{array}\right.$$
However, with this construction, we must use k string variables instead of $\lceil \frac{k}{2} \rceil$ (because of the case where the 2DGSM$_{\text{RLA}}$zigzags around a crossing sequence).

□

5 Conclusion

The robustness of the streaming string transducer model and its derivatives is supported by their equivalence to other models such as two-way transducers and monadic second-order logic. In this paper, we have taken a closer look at the equivalence between DSST and 2DGSM, and shown a relation between the resources of these two models.

There are still many open problems on this model, among them the equivalence problem for k-valued NSST. We have tried reusing the result on 2NGSM by Culik and Karhumaki [7], either by adapting the proof (without succes), or by proving the equivalence of k-valued NSST with k-valued 2NGSM. The inclusion k-val. 2NGSM $\subseteq$ k-val. NSST can

be proven using a construction similar to the one in section 4.2.3, but we have found no proof or counter-example for the other inclusion.

References

[1] R. Alur and P. Cerny. Streaming transducers for algorithmic verification of single-pass list processing programs. In *Proceedings of the 38th Annual ACM Symposium on Principles of Programming Languages*, 2011.

[2] R. Alur and L. D'Antoni. Streaming tree transducers. In *39th International Colloquium on Automata, Languages, and Programming*, 2012.

[3] R. Alur and J.V.Deshmukh. Nondeterministic streaming string transducers. In *ICALP 2011*, volume Part II of *LNCS 6756*, pages 1–20, 2011.

[4] R. Alur and P.Cerny. Expressiveness of streaming string transducers. In *Proc. of Fondations of Software Technology and Theoretical Computer Science*, pages 1–12, 2010.

[5] J.E. Hopcroft A.V. Aho and J.D. Ullman. A general theory of translation. In *Mathematical Systems Theory 3*, pages 195–221, 1969.

[6] M. Chytil and V. Jakl. Serial composition of two-way finite-state transducers and simple programs on strings. In *ICALP*, pages 135–147, 1977.

[7] K. Culik and J. Karhumaki. The equivalence of finite valued transducers (on hdt0l languages) is decidable. *Theoretical Computer Science*, 47:71–84, 1989.

[8] J. Engelfriet and H.J. Hoogeboom. MSO-definable string transductions and two-way finite-state transducers. In *ACM Transactions on Computational Logic 2*, pages 216–254, 2001.

[9] J.E. Hopcroft and J.D. Ullman. An approach to a unified theory of automata. *The Bell System Technical Journal 46*, pages 1793–1829, 1967.

[10] J.E.Hopcroft and J.D.Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[11] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[12] M. Sipser. Halting space-bounded computations. *19th Annual Symposium on Foundations of Computer Science*, pages 73–74, 1978.