CHARLES UNIVERSITY IN PRAGUE

FACULTY OF MATHEMATICS AND PHYSICS

MASTER THESIS

JAKUB BARTÁK

Recognition of picture languages

Institution: Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc.

Study branch: Theoretical Computer Science

Acknowledgments

I would like to thank all who participated on the creation of this thesis. My supervisor RNDr. František Mráz CSc. who provided me with thesis idea and patiently guided the work. His colleague Martin Plátek CSc. who helped with the initial forming of the two-dimensional restarting automaton concept and all my friends who helped with emendation and readibility of the text – most notably Lenka Bartůňková and Jana Přesličková. Last but not least I would like to thank the God for his guidance and blessing.

Contents

Abstract

Název práce: Rozpoznávání obrázkových jazyků
Autor: Jakub Barták
Obor: Teoretická informatika
Vedoucí diplomové práce: RNDr. František Mráz, CSc.
e-mail vedoucího: mraz@ksvi.mff.cuni.cz
Abstrakt: Předkládáme zde transformaci jednodimenzionálního zkracujícího restartovacího automatu do dvou dimenzí. Výsledný automat (zvaný dvou-dimenzionální restartovací automat – 2RA) má jak zajímavé uzávěrové vlastnosti, tak blízkou vazbu na třídu jazyků REC (Recognizable languages), čímž se třída jazyků které je schopen rozpoznat ukazuje být zajímavým přínosem pro hierarchii dvou-dimenzionálních jazyků.
Klíčová slova: Dvou-dimenzionální jazyky, restartovací automaty, třída jazyků REC

Title: Recognition of picture languages
Author: Jakub Barták
Branch of study: Theoretical Computer Science
Supervisor: RNDr. František Mráz, CSc.
Supervisor's email address: mraz@ksvi.mff.cuni.cz
Abstract: We present a transformation of one-dimensional shrinking restarting automaton concept into two-dimensions. The resulting automaton (called Two-dimensional Restarting Automaton – 2RA) proved to have an interesting closure properties as well as close relation to the class or recognizable languages (REC) which qualify it as a notable member of two-dimensional language hierarchy.
Keywords: Two-dimensional languages, Restarting automata, REC class

Introduction

Theoretical automata models and grammars are nowadays commonly accepted terms helping in wide range of problems and challenges from guessing the upper bounds of some computation problems to grammar checking of complicated languages like Czech. Most of the current automata and grammars are working with the basic one-dimensional object – string. We can see many reasons for that. The historical one, as first computers were solving problems like cryptography where string is the basic object. The reason of complexity, as handling one-dimensional objects is far easier than two (or more)-dimensional. And the "necessity" one, as the need for using the two-dimensional objects – pictures has risen a few decades after the need for handling strings.

Today the one-dimensional language hierarchy is quite rich with many known relations among the contained classes. We have Chomsky hierarchy for the one-dimensional grammars and all grammar classes there have their respective automata. There are still some open questions remaining, but it is safe to say that the space of one-dimensional languages is mapped.

When we switch into the two dimensions the situation changes dramatically. Despite the fact that first two-dimensional automaton was defined by M. Blum and C. Hewitt back in 1967 there is still no consensus about ground class (like regular languages in one-dimensional case), there is no known hierarchy and we can find only several results that are being slowly linked together. This is especially unfortunate as the computer processed images can be seen virtually everywhere these days.

Lately there have been some attempts to establish the ground class for two-dimensional languages by A. Restivo and D. Giammarresi and although it is probably too soon to decide whether their proposal will be accepted or not, they deserve much credit for advancing the research concerning two-dimensional languages and their hierarchy.

To help in those efforts we decided to try to bring the successful model of restarting automaton into two-dimensions and define an automaton that will be a new interesting member of the two-dimensional language hierarchy or automaton counterpart to some already known concept like tiling systems.

The thesis itself is divided into four chapters. The first chapter contains short introduction into two-dimensional languages theory as well as presentation of some not well-known models and concepts. In second chapter is the definition of restarting automaton with short discussion of how the definition emerged. In the third chapter are the theoretical results concerning the class of languages restarting automaton accepts, like closure properties and position in two-dimensional language hierarchy. The last (fourth) chapter contains the "practical" results of how the restarting automaton handles some basic two-dimensional objects like lines, crosses and trees.

Chapter 1

Introduction to Two-dimensional languages

The goal of this chapter is to give an insight into a part of the theory of two-dimensional languages needed to understand this thesis. The chapter is divided into two sections. In the first section you can find fundamental definitions of pictures, picture languages and basic operations on them. The second section covers some advanced or not that well-known topics of the given theory.

1.1 Basic definitions

You can find most of this chapter in almost every book about two-dimensional language theory (for example [4]).

1.1.1 Pictures and picture languages

First we define a basic two-dimensional element called picture. We can see pictures in the same way as computers do in case of bitmap graphics. This creates close resemblance to one-dimensional (string) languages and even equality in case of pictures of size $1 \times k$. On the other hand this approach causes problems with graphical objects which are "un-squarable" by their nature (such as circles).

Definition 1.1 *A* picture *over an alphabet Σ is a two-dimensional rectangular array (matrix) of elements of Σ. The set of pictures of size (m, n) is denoted by $\Sigma^{m,n}$. All pictures over Σ are denoted by $\Sigma^{*,*}$.* Picture language *L is subset of $\Sigma^{*,*}$.*

Let $P \in \Sigma^{*,*}$ be a picture. Then, $rows(P)$, resp. $cols(P)$ denotes the number of rows, resp. columns of P. Usually the measures are called height and width respectively of the picture. The pair $rows(P) \times cols(P)$ is called the size of P. We say that P is a *square* picture of size n if $rows(P) = cols(P) = n$.

For theoretical purposes it is sometimes useful to define an empty picture. We denote empty picture by λ symbol (same as in one-dimensional case). It is the only picture of size 0×0. Note that there are no pictures of size $0 \times k$ or $k \times 0$ for any $k > 0$.

Furthermore sometimes we need to refer to a specific symbol in the picture. For that purpose we define the following abbreviation. Let i, j be integers such that $1 \leq i \leq rows(P), 1 \leq j \leq cols(P), P(i, j)$ denotes the symbol in P at coordinates (i, j). Picture of size 1×1 has exactly one symbol at coordinates $(1, 1)$.

Example 1.1 *To give an example of a picture language, let* $\Sigma = \{0, 1\}$, *L be the set consisting exactly of all square pictures $P \in \Sigma^{*,*}$, where*

$$P(i,j) = \begin{cases} 0 \text{ if } i + j \text{ is an odd number} \\ 1 \text{ otherwise} \end{cases}$$

Our language is the language of all "chessboard" like pictures. Pictures in L of sizes 1, 2 and 3 follow.

$$\begin{array}{|c|}\hline 1 \\ \hline \end{array} \qquad \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} \qquad \begin{array}{|c|c|c|}\hline 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline 1 & 0 & 1 \\ \hline \end{array}$$

To prevent automata to "fall off the picture" during computation and to relieve ourselves from constantly speaking about borders of pictures we define *boundary picture*. The original picture is surrounded by a special symbol, so automaton working on the tape is informed when it is about to step out of the boundaries. Usually it is required that after automaton steps on the boundary symbol the next action must be the reverse step "back".

Definition 1.2 *For every picture $P \in \Sigma^{m,n}$ we define* boundary picture *$\hat{P}$ of size* $(m + 2, n + 2) \in \Sigma \cup \{\#\}$ *where $\# \notin \Sigma$:*

$$\hat{P}(i,j) = \begin{cases} P(i-1, j-1) \text{ for } 2 \leq i \leq rows(P), 2 \leq j \leq cols(P) \\ \# \quad \text{otherwise} \end{cases}$$

Example 1.2 *Boundary pictures of the pictures from Example 1.1 then look like:*

#	#	#
#	*1*	#
#	#	#

#	#	#	#
#	*1*	*0*	#
#	*0*	*1*	#
#	#	#	#

#	#	#	#	#
#	*1*	*0*	*1*	#
#	*0*	*1*	*0*	#
#	*1*	*0*	*1*	#
#	#	#	#	#

Unless specified otherwise, we always use boundary pictures for computations instead of "normal" ones.

1.1.2 Operations on pictures

We define a set of basic operations allowing us to work effectively with pictures and picture languages in the same manner as in the case of strings and string languages. Some operations are almost the same as in the one-dimensional case, others split to two cases (horizontal and vertical). Concatenation operation is an example of the latter.

Definition 1.3 *The* column concatenation *of two pictures P and Q (denoted by $P \obar Q$) is a partial operation, defined only if $rows(P) = rows(Q)$, and it is given by*

$$P = \boxed{\begin{matrix} P_{1,1} & \dots & P_{1,n} \\ \vdots & \ddots & \vdots \\ P_{m,1} & \dots & P_{m,n} \end{matrix}} \quad Q = \boxed{\begin{matrix} Q_{1,1} & \dots & Q_{1,n'} \\ \vdots & \ddots & \vdots \\ Q_{m,1} & \dots & Q_{m,n'} \end{matrix}}$$

$$P \obar Q = \boxed{\begin{matrix} P_{1,1} & \dots & P_{1,n} & Q_{1,1} & \dots & Q_{1,n'} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ P_{m,1} & \dots & P_{m,n} & Q_{m,1} & \dots & Q_{m,n'} \end{matrix}}$$

Similarly, the row concatenation *of two pictures P and Q (denoted by $P \ominus Q$) is a partial operation, defined only if $cols(P) = cols(Q)$, and it is given by*

$$P = \boxed{\begin{matrix} P_{1,1} & \dots & P_{1,n} \\ \vdots & \ddots & \vdots \\ P_{m,1} & \dots & P_{m,n} \end{matrix}} \quad Q = \boxed{\begin{matrix} Q_{1,1} & \dots & Q_{1,n} \\ \vdots & \ddots & \vdots \\ Q_{m',1} & \dots & Q_{m',n} \end{matrix}}$$

$$P \ominus Q = \boxed{\begin{matrix} P_{1,1} & \dots & P_{1,n} \\ \vdots & \ddots & \vdots \\ P_{m,1} & \dots & P_{m,n} \\ Q_{1,1} & \dots & Q_{1,n} \\ \vdots & \ddots & \vdots \\ Q_{m',1} & \dots & Q_{m',n} \end{matrix}}$$

Moreover, the column and the row concatenation of P and the empty picture λ is always defined and λ is the neutral element for both these operations.

Besides the concatenations, we define an unary operation called rotation.

Definition 1.4 *Let P be a picture. The* (clockwise) rotation *of P, indicated as P^R, is defined by*

$$P = \boxed{\begin{matrix} P_{1,1} & \dots & P_{1,n} \\ \vdots & \ddots & \vdots \\ P_{m,1} & \dots & P_{m,n} \end{matrix}} \qquad P^R = \boxed{\begin{matrix} P_{m,1} & \dots & P_{1,1} \\ \vdots & \ddots & \vdots \\ P_{m,n} & \dots & P_{1,n} \end{matrix}}$$

Please note that although we denote rotation in the same way as "reverse" in case of the one-dimensional languages, the resulting picture significantly differs from the "mirror image" we would expect to get. Only in case of pictures of size $(1, k)$ the P^{R^R} produces the "mirror image" as we are accustomed.

To fill the missing option of creating the "mirror image" of some pictures we define the following operation:

Definition 1.5 *Let P be a picture. The* vertical mirroring *of P, indicated as $P^{(v\_mirr)}$, is defined by*

$$P = \boxed{\begin{matrix} P_{1,1} & \dots & P_{1,n} \\ \vdots & \ddots & \vdots \\ P_{m,1} & \dots & P_{m,n} \end{matrix}} \qquad P^{(v\_mirr)} = \boxed{\begin{matrix} P_{1,n} & \dots & P_{1,1} \\ \vdots & \ddots & \vdots \\ P_{m,n} & \dots & P_{m,1} \end{matrix}}$$

the horizontal mirroring *indicated as $P^{(h\_mirr)}$, is defined by*

$$P = \boxed{\begin{matrix} P_{1,1} & \dots & P_{1,n} \\ \vdots & \ddots & \vdots \\ P_{m,1} & \dots & P_{m,n} \end{matrix}} \qquad P^{(h\_mirr)} = \boxed{\begin{matrix} P_{m,1} & \dots & P_{m,n} \\ \vdots & \ddots & \vdots \\ P_{1,1} & \dots & P_{1,n} \end{matrix}}$$

Remark 1.1 *Notice that vertical mirroring can be acquired as combination of horizontal mirroring and rotation (and vice-versa).* $P^{(v\_mirr)} = ((((P^R)^{(h\_mirr)})^R)^R)^R$

We now define a projection of a picture. Aside from the fact that such transformations are often seen in "real" world of pictures, projection is a key element of a later defined tiling systems.

Definition 1.6 *Let $P \in \Gamma^{*,*}$ be a picture and π be a function $\pi : \Gamma \to \Sigma$. The* projection by mapping π *of picture P is the picture $P' \in \Sigma^{*,*}$ such that $P'(i, j) = \pi(P(i, j))$, for all $1 \le i \le rows(P), 1 \le j \le cols(P)$.*

Where there is no danger of ambiguity we will use $\pi(P)$ to indicate the projection of picture P by mapping π. The definition of projection of a picture can be extended in a natural way to sets of pictures.

Definition 1.7 *Let $L \subseteq \Gamma^{*,*}$ be a picture language and π be a function $\pi : \Gamma \to \Sigma$. The* projection by mapping π *of L is the language $L' = \{\, P' \mid \forall P \in L : P' = \pi(P) \,\} \subseteq \Sigma^{*,*}$.*

We have defined all basic operations that will be used in this thesis. The following section deals with main computation models working with the picture languages.

1.2 Advanced topics

In the second section of this introduction we cover some more advanced and specific topics used further in the thesis. Mainly several automata and other classification concepts which form together the core of the two-dimensional language hierarchy.

Remark 1.2 *Because two-dimensional theory is a fast growing field nowadays, this part of the thesis is probably most likely to evolve. We strongly recommend checking the latest articles and books for the newest results.*

1.2.1 Basic families of picture languages

In case the of one-dimensional languages we are in a situation of already mapped hierarchy of languages and known relations among them. Above that, most of the ground languages have "nice" properties like closures on almost all operations. When we move to two-dimensions the situation changes dramatically. So far, there is no consensus about the ground class (which are the regular languages in one-dimension), most of the proposed two-dimensional automata have different strength in their deterministic and non-deterministic versions and although there are many separate results on several automata there is no "unified" theory equivalent to the Chomsky hierarchy in one-dimension. And this is despite the fact that the first definition of the two-dimensional finite automaton was published by M. Blum and C. Hewitt back in 1967 [1].

To give some insight into the situation on the field we present several definitions of automata models and mention their properties.

One of the most straightforward generalization of the string automata is the four-way finite automaton defined by M. Blum and C. Hewitt in 1967 [1].

It can be imagined as a simple finite state automaton capable of moving on the tape in four directions: *Left, Right, Up, Down*.

4-way finite automaton

Definition 1.8 *A non-deterministic (deterministic)* four-way finite automaton, *referred to as 4NFA (4DFA), is a 7-touple* $A = (\Sigma, Q, \Delta, q_0, q_a, q_r, \delta)$ *where:*

- Σ *is the input alphabet;*
- Q *is a finite set of states;*
- $\Delta = \{R, L, U, D\}$ *is the set of "directions";*
- $q_0 \in Q$ *is the "initial" state;*
- $q_a, q_r \in Q$ *are the "accepting" and the "rejecting" states, respectively;*
- $\delta : Q \backslash \{q_a, q_r\} \times \Sigma \to 2^{Q \times \Delta}$ *(or* $\delta : Q \backslash \{q_a, q_r\} \times \Sigma \to Q \times \Delta$ *in the deterministic case, respectively) is the transition function.*

When there is no need to specify whether we are speaking about deterministic or non-deterministic version of the automaton, we refer to a four-way automaton as 4FA.

4FA recognizes a picture $P \in \Sigma^{*,*}$ if, starting from the position $(1, 1)$ in the initial state, it possibly moves around and eventually halts in an accepting state q_a. Notice that during the recognition process the 4FA is not required to read all the positions of the input picture; moreover the finite control can come back to a given position as many times as needed. Let us give an example of a picture language recognized by a 4FA:

Example 1.3 *Let* $\Sigma = \{0, 1\}$ *be an alphabet and let* $L \subseteq \Sigma^{*,*}$ *be the language of pictures which first column is equal to the last one. Then L is recognized by a 4DFA that operates as follows. It scans a picture* $P \in L$ *row by row from left to right, proceeding from top to bottom, and by checking at the same time that all positions contain letters in* Σ *and that the leftmost letter in a row is equal to the rightmost one.*

Although 4FA is a quite simple model it has some inconvenient properties which discredit it from the position of ground class automaton in two-dimensions. More specifically.

Theorem 1.1 ([12]) $\mathcal{L}$*(4DFA) and* $\mathcal{L}$*(4NFA) are NOT closed under row and column concatenation.*

Regarding the boolean operations between languages, the following theorem holds.

Theorem 1.2 ([7]) *$\mathcal{L}$(4DFA) and $\mathcal{L}$(4NFA) are closed under boolean union and intersection operations. Moreover $\mathcal{L}$(4DFA) is closed under complement.*

Note that the question of whether the family $\mathcal{L}$(4NFA) is closed under complement is still open.

The probably most inconvenient property is that family $\mathcal{L}$(4DFA) is strictly included in $\mathcal{L}$(4NFA) (both theorem and proof regarding that can be found in [4]).

As we will see the property that deterministic version of some automaton has lesser strength than its non-deterministic version is so common in two-dimensions that one may even ask whether it is a necessary property in recognition of two-dimensional languages in general or we just have been so far unsuccessful in finding the "right" concept.

On-line tessellation automaton

Completely different approach to the two-dimensional automaton was introduced by K. Inoue and A. Nakamura in [6]. They defined two-dimensional on-line tessellation automaton (2OTA), which is an example of a cellular automaton (i.e. automaton that operates on the entire tape simultaneously). Informally we can imagine 2OTA as an array of simple computation units that cover the whole input picture. Every unit is capable of changing its internal state depending on the state of its neighbors and the symbol on the tape. Whole computation of 2OTA then resembles a "wave" which starts in the upper left corner and ends in the lower right corner.

Formal definition given here is taken from [4] and is slightly different from the original one given by Inoue et al.

Definition 1.9 *A non-deterministic (deterministic)* two-dimensional on-line tessellation automaton, *referred as to 2OTA (2DOTA), is completely defined by $A = (\Sigma, Q, I, F, \delta)$ where:*

- *Σ is the input alphabet;*
- *Q is a finite set of states;*
- *$I \subseteq Q$ (or $I = \{i\} \subseteq Q$ in the deterministic case, respectively) is the set of "initial" states;*
- *$F \subseteq Q$ is the set of "final" (or "accepting") states;*

- $\delta : Q \times Q \times \Sigma \rightarrow 2^Q$ (*or* $\delta : Q \times Q \times \Sigma \rightarrow Q$ *in the deterministic case, respectively*) *is the transition function.*

A run of A on a picture $P \in \Sigma^{*,*}$ consists of associating a state (from the set Q) to each position (i, j) of P. Such state is given by transition function δ and depends on the states already associated to positions $(i - 1, j)$, $(i, j - 1)$ and on the symbol $P(i, j)$. All positions are initialized with one of initial states and when the coordinates of the position are not valid (like in case of border positions) we assume the invalid position is in one of the initial states as well.

A 2OTA A recognizes a picture P if there exists a run of A on P such that the state associated to position $(rows(P), cols(P))$ is a final state.

We give an example of a language recognized by 2OTA.

Example 1.4 *Let $\Sigma = \{a\}$ and let $L \subseteq \Sigma^{*,*}$ be the language of all squares over Σ. A 2OTA recognizes pictures of L by associating state "1" to positions in the main diagonal and states "2" and "3" to positions above and below such diagonal, respectively. A picture will be accepted if the position of the bottom-right corner contains state "1". Formally, L is recognized by the following 2OTA $A = (\Sigma, Q, I, F, \delta)$ defined as follows:*

- $Q = \{0, 1, 2, 3\}$;
- $I = \{0\}$;
- $F = \{1\}$;
- $\delta(0, 0, a) = \delta(2, 3, a) = 1$;
  $\delta(0, 1, a) = \delta(0, 2, a) = \delta(2, 1, a) = \delta(2, 2, a) = 2$;
  $\delta(1, 0, a) = \delta(3, 0, a) = \delta(1, 3, a) = \delta(3, 3, a) = 3$.

The families of two-dimensional languages recognized by a 2OTA and a 2DOTA are denoted by $\mathcal{L}$(2OTA) and $\mathcal{L}$(2DOTA), respectively. We now present some results on those language families.

Theorem 1.3 ([6]) *$\mathcal{L}$(2OTA) is closed under row and column concatenation.*

Theorem 1.4 ([6]) *$\mathcal{L}$(2OTA) is closed under projection.*

Theorem 1.5 ([6]) *The family $\mathcal{L}$(2DOTA) is strictly included in $\mathcal{L}$(2OTA).*

Theorem 1.6 ([6]) *$\mathcal{L}$(4NFA) $\subset$ $\mathcal{L}$(2OTA).*

$\mathcal{L}$(2DOTA) and $\mathcal{L}$(4DFA) are not related by any inclusion relations. There are examples of picture languages recognized by 4DFA and not recognized by any 2DOTA and vice versa.

Notice that a 2OTA reduces to a standard finite automaton on words when restricted to operate on one-row pictures only. This becomes even more interesting when we realize that 2OTA is stronger than 2NFA – which, when restricted to operate on one-row pictures, resembles standard automaton far more.

This is another example of the fact that the transition to two dimensions is neither easy nor intuitive.

Tiling Systems

We now define language family of tiling systems which has a close connection to on-line tessellation automata although this is not clear at first glimpse. This family of languages was first introduced by D. Giammarresi and A. Restivo in [5] and proposed as a ground class for two-dimensional languages. Since then, several new results regarding properties of this class have been published. They follow after the main definition and listing of the properties. Tiling systems will also have close relation to two-dimensional restarting automaton.

Tiling systems take as starting point a characterization of recognizable string languages in terms of local languages and projections. Namely, any recognizable (by means of finite automata) string language can be obtained as projection of a local string language over a larger alphabet (cf. Theorem 6.1 in [3]). This notion is then extended to the two-dimensional case: precisely, we define local picture languages by means of a set of square arrays of side-length two, here called "tiles", which represent the only allowed blocks of that size in the pictures of the language. Then we say that a two-dimensional language is "tiling recognizable" if it could be obtained as a projection of a local picture language.

Local two-dimensional languages

Definition 1.10 *Given a picture P of size (m, n), let $h \leq m, k \leq n$: we denote by $B_{h,k}(P)$ the set of all blocks (or sub-pictures) of P of size (h, k). We call* tile *a square picture of size* $(2, 2)$.

We now define local and domino-local languages which are core elements of tiling systems.

Definition 1.11 *Let Γ be a finite alphabet. A two-dimensional language $L \subseteq \Gamma^{*,*}$ is* local *if there exists a finite set Θ of tiles over the alphabet $\Gamma \cup \{\#\}$ such that $L = \{P \in \Gamma^{*,*} | B_{2,2}(\hat{P}) \subseteq \Theta\}$.*

Θ represents the set of allowed blocks for pictures belonging to the local language L. Given a language L, we can consider the set Θ as the set of all possible blocks of size (2,2) of pictures that belong to L (note that we are considering boundary pictures). The language L is local if, given such a set Θ, we can exactly retrieve the language L. We will assume implicitly that the empty picture λ belongs to L if and only if Θ contains the tile with four # symbols. The family of local picture languages will be denoted by LOC.

Example 1.5 *Let $\Gamma = \{0, 1\}$ be an alphabet and Θ be the following set of tiles over Γ.*

$$\Theta = \left\{ \begin{matrix}
\begin{array}{|c|c|}\hline 1&0\\\hline 0&1\\\hline\end{array} &
\begin{array}{|c|c|}\hline 0&0\\\hline 1&0\\\hline\end{array} &
\begin{array}{|c|c|}\hline 0&1\\\hline 0&0\\\hline\end{array} &
\begin{array}{|c|c|}\hline 0&0\\\hline 0&0\\\hline\end{array} &
\begin{array}{|c|c|}\hline \#&1\\\hline \#&0\\\hline\end{array} &
\begin{array}{|c|c|}\hline \#&0\\\hline \#&0\\\hline\end{array} &
\begin{array}{|c|c|}\hline 0&0\\\hline \#&\#\\\hline\end{array} \\
\begin{array}{|c|c|}\hline 0&1\\\hline \#&\#\\\hline\end{array} &
\begin{array}{|c|c|}\hline 0&\#\\\hline 1&\#\\\hline\end{array} &
\begin{array}{|c|c|}\hline 0&\#\\\hline 0&\#\\\hline\end{array} &
\begin{array}{|c|c|}\hline \#&\#\\\hline 0&0\\\hline\end{array} &
\begin{array}{|c|c|}\hline \#&\#\\\hline 1&0\\\hline\end{array} &
\begin{array}{|c|c|}\hline \#&\#\\\hline \#&1\\\hline\end{array} &
\begin{array}{|c|c|}\hline \#&\#\\\hline 0&\#\\\hline\end{array} \\
\begin{array}{|c|c|}\hline \#&0\\\hline \#&\#\\\hline\end{array} &
\begin{array}{|c|c|}\hline 1&\#\\\hline \#&\#\\\hline\end{array} & & & & &
\end{matrix} \right\}$$

The language $L = L(\Theta)$ is the language of square pictures in which all main diagonal positions carry symbol "1", whereas the remaining positions carry symbol "0".

Notice that the language of squares over a one-letter alphabet is not a local language because there is no "local strategy" to compare the number of rows and columns using only one symbol.

In [4] (page 242) *hv-local* picture languages are defined, where the square tiles of side 2 are replaced by "dominoes" that correspond to two kinds of tiles: horizontal dominoes of size (1,2) and vertical dominoes of size (2,1). As those "domino-languages" are used for example in family of languages called SDREC we give formal definition (calling them "domino-local" instead of "hv-local").

Definition 1.12 *Let Γ be a finite alphabet. Then two-dimensional language $L \subseteq \Gamma^{*,*}$ is* domino-local *if there exists a finite set Δ of dominoes over the alphabet $\Gamma \cup \{\#\}$ such that language $L = \{P \in \Gamma^{*,*} | (B_{1,2}(\hat{P}) \cup B_{2,1}(\hat{P})) \subseteq \Delta\}$.*

It is easy to understand the following remark so we include it without the proof.

Remark 1.3 *If $L \subseteq \Gamma^{*,*}$ is a domino-local two-dimensional language then L will be a local language. Opposite is not true, that is there are languages that are local but not domino-local. This can be easily seen on the language of squares over $\Sigma = \{0, 1\}$ filled with 0's with exception of main diagonal which is filled with 1's.*

With the definition of two-dimensional local languages, we have all the tools necessary to define the tiling systems.

Definition 1.13 *A* tiling system *(TS) is 4-tuple $T = (\Sigma, \Gamma, \Theta, \pi)$ where Σ and Γ are two finite alphabets, Θ is finite set of tiles over the alphabet $\Gamma \cup \{\#\}$ and $\pi : \Gamma \to \Sigma$ is a projection.*

The tiling system T recognizes a language L over the alphabet Σ as follows: $L = \pi(L')$ where $L' = L(\Theta)$ is the local language over Γ corresponding to the set of tiles Θ. We write $L = L(T)$ and we say that L is the language recognized by T. We say that a language $L \subseteq \Sigma^{*,*}$ is *recognizable by tiling systems* (or *tiling recognizable*) if there exists a tiling system $T = (\Sigma, \Gamma, \Theta, \pi)$ such that $L = L(T)$. We denote by $\mathcal{L}(TS)$ the family of all two-dimensional languages recognizable by tiling systems. In other words $L \in \mathcal{L}(TS)$ if it is a projection of some local language.

Example 1.6 *Let $\Sigma = \{a\}$ be a one-letter alphabet and let L be the language of squares over Σ, that is $L = \{P | cols(P) = rows(P)\} \subseteq \Sigma^{*,*}$. Language L is recognizable by tiling systems. We can take as underlying local language L', the one from Example 1.5, i.e. the language of squares over the alphabet $\Gamma = \{0, 1\}$ with 1's in the main diagonal and 0's in the other positions. The applied projection $\pi : \Gamma \to \Sigma$ is then defined in a way $\pi(0) = \pi(1) = a$. It is easy to see that $L = \pi(L')$.*

The projection defined in tiling systems can be seen as a way to "hide" some auxiliary symbols used in generation of the result picture. In the example mentioned above we used auxiliary diagonal to assure that the picture is square. This diagonal was later "overwritten" by the projection.

Remark 1.4 *It is interesting that despite the fact that $\mathcal{L}$(domino-local) $\subset$ LOC, when we create domino-systems in the same manner, as tiling systems are defined (with domino-local languages instead of local), those two family classes are equivalent. The proof (found in [4] page 243-245) is based on the idea that we can express property of "being an allowed sub-picture of size (2,2) of picture in L" by means of dominoes over a larger alphabet Γ. Projection then takes care of the rest.*

We mention following properties of tiling systems.

Theorem 1.7 ([4]) *The family $\mathcal{L}(TS)$ is closed under projection.*
The family $\mathcal{L}(TS)$ is closed under row and column concatenation.
The family $\mathcal{L}(TS)$ is closed under union and intersection.
The family $\mathcal{L}(TS)$ is NOT closed under complement.

D. Giammarresi and A. Restivo further proved the equivalence between on-line tessellation automata and tiling systems. This connection (aside from some other properties of tiling systems) gives them a strong ground to define class REC (recognizable languages). REC class contains 4 equivalent language families, namely languages recognized by on-line tessellation automata $\mathcal{L}(2OTA)$, languages recognized by finite tiling systems $\mathcal{L}(TS)$, languages defined by existential monadic second order formulas $\mathcal{L}(EMSO)$ and languages defined by a complementation-free regular expressions with projection $\mathcal{L}(PCFRE)$. As mentioned earlier, REC class was proposed by D. Giammarresi and A. Restivo as ground class for two-dimensional languages. Regardless whether we agree with such proposal or not, REC represents a robust class and therefore a good choice for comparison with our new model of two-dimensional restarting automaton.

We can see the connections between all mentioned classes in following figure.

Deterministic REC

Because of the use of projection in tiling systems (and because of their equivalence with 2OTA) there is an implanted non-determinism in class REC. Therefore there have been several attempts to define a class of languages which could be called "deterministically recognizable". The first such attempt was made by authors of original REC class D.Giammarresi et al. in article [13, From Determinism to Non-determinism in Recognizable Two-Dimensional Languages]. Several classes were defined there, but we mention only two of them. The first one is called UREC which stands for "unambiguous REC". Informally, it contains languages where every picture has a unique counter-image in its corresponding local language (in "orientation free notion"). Exact definition can be found in [5].

For a better idea we can imagine that every successful finding of some pre-image of picture in UREC must end up with the same picture. Please note that this does not rule out possible backtracking during the computation.

In the case we wanted to remove such backtracking possibility, our goal would be to have each tile of the final picture corresponding to a single tile

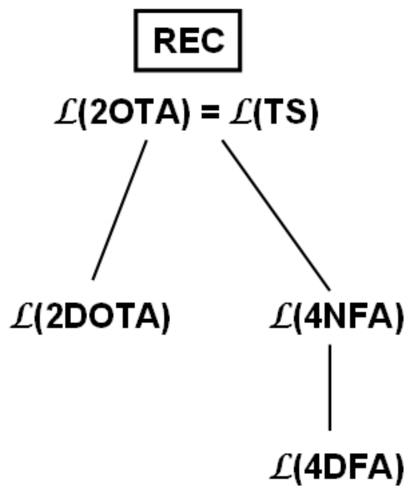

Figure 1.1: Inclusions among the classes mentioned above. Line between two classes means that the lower class is included in the higher. This inclusion need not to be strict.

in the underlying local language. This is indeed how DREC class is defined. Definition can be found in [13] but because it contains some ideas used later in the thesis we present it here as well.

At first, imagine that when we would like to determine the original pre-image of some picture P, the result could depend on where we start the transformation. That is because after we have determined the first symbol, the surrounding tiles are forced to bind to this choice and therefore have reduced number of possible pre-image tiles. In ideal situation this number is reduced to a single tile and we can continue in the process. As in two-dimensional case we have four corners where to begin (each time we end in the opposite corner) there are four processing directions (called corner-to-corner directions) defined: *tl2br, tr2bl, bl2tr* and *br2tl*; *tl2br* stands for top left to bottom right etc. . DREC class is then defined in following way.

Definition 1.14 *A tiling system* $(\Sigma, \Gamma, \Theta, \pi)$ *is* tl2br-deterministic *if for any* γ_1, $\gamma_2, \gamma_3 \in \Gamma \cup \{\#\}$ *and* $\sigma \in \Sigma$ *there exists at most one tile* $\begin{array}{|c|c|}\hline \gamma_1 & \gamma_2 \\ \hline \gamma_3 & \gamma_4 \\ \hline \end{array} \in \Theta$, *with* $\pi(\gamma_4) = \sigma$. *Similarly we define d-deterministic tiling systems for any corner-to-corner direction.*

Recognizable two-dimensional language L will be *deterministic*, if it admits a d-deterministic tiling system for some corner-to-corner direction d. Moreover, we denote by DREC the class of Deterministic Recognizable Two-dimensional Languages.

There is one interesting property considering the class DREC.

Theorem 1.8 ([13]) *The class DREC is equal to the closure by rotation of $\mathcal{L}(DOTA)$.*

Considering the inclusions among the classes mentioned above following theorem holds.

Theorem 1.9 ([13]) *UREC is properly included in REC. DREC is properly included in UREC.*

Therefore we get $DREC \subset UREC \subset REC$.

Sudoku-Deterministic REC

Last class we mention here is the class of Sudoku-deterministically recognizable picture languages (SDREC) defined by B. Borchert and K. Reinhardt in [2]. SDREC class takes slightly different approach to the notion of determinism in recognizable languages than the class DREC. Instead of guessing the original tile on position x, y in one shot, it keeps the set of possible pre-image possibilities and reduce them iteratively in the very same manner the Sudoku-puzzle is solved.

For easier definition of the class, domino-systems are used instead of tiling-system (but we know from Remark 1.4 that they are equivalent). Formally:

Let $T = (\Sigma, \Gamma, \Delta, \pi)$ be a domino tiling system. Given a picture P over the alphabet Σ we define a picture s_P of the same size in which we initialize every position (i, j) by the set $s_P(i, j) := \pi^{-1}(P(i, j)) \in 2^\Gamma$ of possible pre-image symbols. In one step of the sudoku-deterministic process we discard all the possibilities that does not conform with the set of allowed dominoes Δ (on the processed picture at the moment). Formally:

For $s, s' \in (2^\Gamma)^{*,*}$ (of the same size) we allow a step $\hat{s} \Rightarrow_{sd(T)} \hat{s}'$ if for all positions (i, j) in s we have that the set $\hat{s}'(i, j)$ consists of the elements in the set $\hat{s}(i, j)$ which have in each of the four neighboring sets a respective element such that the respective domino tile is in Δ. Even more formally:

$s'(i, j) = \{x \in s(i, j) \mid \exists y_1,\ y_2,\ y_3,\ y_4 \in \Gamma \cup \{\#\}$ such that $y_1 \in \hat{s}(i + 1, j)$, $y_2 \in \hat{s}(i - 1, j)$, $y_3 \in \hat{s}(i, j + 1)$, $y_4 \in \hat{s}(i, j - 1)$ and $\begin{array}{|c|c|}\hline x & y_1 \\ \hline\end{array}$, $\begin{array}{|c|c|}\hline y_2 & x \\ \hline\end{array}$, $\begin{array}{|c|}\hline x \\ \hline y_3 \\ \hline\end{array}$, $\begin{array}{|c|}\hline y_4 \\ \hline x \\ \hline\end{array}$ $\in \Delta\}$.

Note that only elements from $s'(i,j)$ are dropped which will never be the symbol of a valid pre-image at this position. In other words: one step excludes one or more possibilities for which one can be sure about that it will not be a solution, and after this step new opportunities for excluding more possibilities may show up – this is how usually a Sudoku puzzle is solved.

For a given domino tiling system $T = (\Sigma, \Gamma, \Delta, \pi)$ let the accepted language $\mathcal{L}_{sd}(T)$ be defined as the set of pictures P for which there exists a way to transform the initialized picture $\hat{s}_P$ in finitely many steps into a picture in which every position consists of exactly one element and which cannot be transformed further. Formally :

$$\mathcal{L}_{sd}(T) := \{P \in \Sigma^{*,*} \mid \exists P' \in \mathcal{L}(T) \text{ such that } \hat{s}_P \Rightarrow^*_{sd(T)} \{\hat{P}'\}\}.$$

The $\{\hat{P}'\}$ stands for the picture which is of the same size as $\hat{P}'$ and has instead of a letter $P'(i,j)$ the singleton set $\{P'(i,j)\}$ as a letter at position (i,j).

The class of the Sudoku-deterministically recognizable picture languages SDREC is defined as the languages L for which there is a domino tiling system T recognizing L in that way i.e.

Definition 1.15 SDREC $:= \{L \subseteq \Sigma^{*,*} \mid$ there is a $T = (\Sigma, \Gamma, \Delta, \pi)$ such that $L = \mathcal{L}_{sd}(T)\}$

Following theorems concerning SDREC class holds. 4AFA stands for 4-way alternating finite automaton which we mention here only because of the following theorem and its significance. More about 4AFA can be found in [8].

Theorem 1.10 ([2]) $\mathcal{L}(4AFA) \subseteq SDREC$

Theorem 1.11 ([2]) $DREC \subseteq SDREC$

The second theorem is not very surprising seeing how SDREC class works. What is interesting on the first theorem is its combination with the result from [8] where it was proven that 4AFA contains some non-recognizable picture languages and therefore SDREC does not belong to REC. We can see the update figure with newly added classes and connections between them on Figure 1.2.

Following questions concerning the subject are still open:

- is DREC or even SDREC contained in 4AFA ?
- is there some computational model capturing exactly REC ∩ co-REC ?

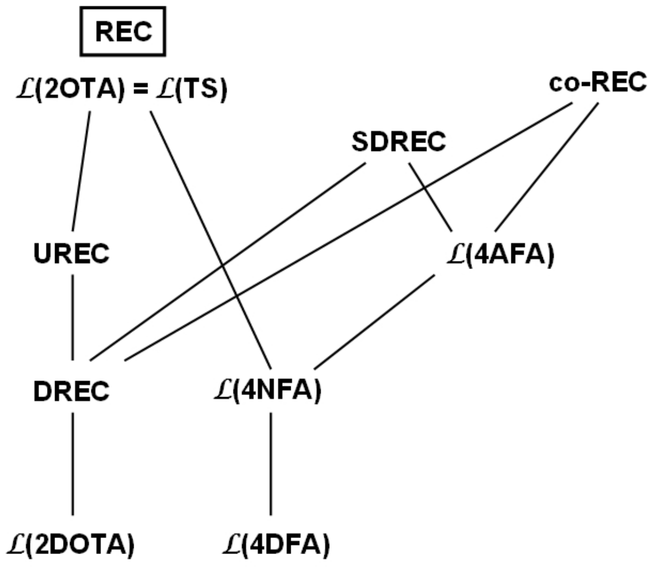

Figure 1.2: Inclusions among all mentioned classes

- author is also unaware of any known relation between UREC and SDREC classes.

Remark 1.5 *Just to imagine how complex the problems become when switching from one into two dimensions, note that all the classes mentioned above when taken into single dimension, collapse to the class of regular languages.*

We have defined several automata for recognition of two-dimensional languages and mentioned their properties. Special concern was given to class of REC and its subclasses, namely the deterministic ones. This approach has two justifications. Firstly we will use some of the ideas and results in the definition of two-dimensional restarting automaton and it is required that the reader is familiar with them. Secondly REC class is very robust and with attention given to it nowadays it is the best choice for feasible comparison with two-dimensional restarting automaton.

This chapter cannot be in any way taken as satisfactory introduction to theory of two-dimensional languages. For better understanding of the subject we strongly recommend checking on the latest articles, mainly from D. Giammarresi or K. Inoue and searching their respective bibliographies. Several key articles can be found in this thesis bibliography as well.

Chapter 2

Defining Two-dimensional Restarting automata

In the following chapter we go through several proposals of two-dimensional restarting automaton definitions and after taking all pros and cons into account we come out with two definitions for both deterministic and non-deterministic version).

2.1 Definition discussion

2.1.1 Idea behind restarting automata

The concept of restarting automata was first introduced by P. Jančar, F. Mráz, M. Plátek and J. Vogel in [9] but perhaps better (and more recent) summary on the subject can be found in [14]. This concept allowed authors elegant characterization of deterministic context-free languages and helped in attempts of solving the grammar checking problem for Czech language. Restarting automata are of several types, but the general idea can be described as follows.

Restarting automaton has a finite control unit, a head with a read/write window attached to a tape, and it works in certain cycles. In a cycle, it starts in the initial state and it moves the head from left to right along the word on the tape; according to its instructions, it can at some point rewrite the content of its read/write window by shorter string and "restart" – i.e. reset the control unit to the initial state and place the head on the left end of the (shorter) word. The computation halts in an accepting or a rejecting state and an input word is accepted if there exists a computation ending in an accepting state. As usual, both non-deterministic and deterministic versions of the automaton are defined. A natural property of monotonicity is also considered (during any computation, "the places of rewriting do not increase their distances from the right end") and shows that deterministic monotonic restarting automata nicely characterize the class of deterministic context-free languages (DCFL) [10].

The property of "restarting itself" is quite useful when used in proofs as an "error preserving property". Imagine the restarting automaton used for grammar checking. Every rewriting is constructed in a way leaving out some parts not affecting the (non)correctness of the sentence. On the input sentence

"The little boys I mentioned runs very quickly"

we get consequently

"The boys I mentioned runs very quickly"
"boys I mentioned runs very quickly"
"boys runs very quickly"
"boys runs quickly"

and finally we obtain the "error core"

"boys runs"

The second property which we will use is the fact that every rewriting instruction somehow "lowers the weight" of the input tape. In one dimension, this is done by length–reducing rewriting steps or weight–reducing rewritings used for shrinking restarting automaton [11]. Because in two dimensions it would be complicated to reduce the dimensions of the tape (what becomes of a picture when we remove some inner positions ?) our approach will be to assign unique weight to each symbol in the input alphabet Σ (and the optional working alphabet Γ) and force every rewriting instruction to lower the overall weight of the input picture.

Remark 2.1 *It can be easily seen, that for every finite input, the "weight reducing" property necessarily leads to finite computation sequence.*

2.1.2 Definition proposals for 2DRA

Our goal is to define an automaton which will be easy to handle, both in real computations and proofs. Moreover we require that the automaton is "weak" in terms of language class it accepts. We take from one-dimensional shrinking restarting automaton the property of "lowering the weight" of the tape and the property of "restart" after every rewriting instruction. In case of rewriting we also require the change of the tape is "local" in some sense.

We divide the following discussion of the definition into three sections:

- How does the automaton read the tape ?
- How the rewriting instructions are defined ?
- When does the automaton accept the input ?

We propose several solutions for each aspect of our automaton, but we choose the best combination at the end rather then in the process. This is because some reading strategies are better when combined with certain rewriting strategies etc.

Reading

There are two main approaches we can take when deciding how the automaton will read the input.

- Automaton does not read input in a common sense but rather finds first available rewriting position and performs the rewriting.
- Automaton is capable of reading input and storing information about it in its internal states. Then it performs rewriting on the position it decides.

In both cases we have to define in which direction is the input read. In the second case the only difference would be in time complexity of the algorithms as automaton can always read the whole input and then decide where to perform the rewriting. In the first case however, the decision could affect the automaton strength as the "first" rewriting position would change depending on the direction input is read.

Technically we can allow automaton to move freely in all four directions acting in a same way as 4-way finite automaton. Such approach can be applied though in the second case only.

Rewriting

Rewriting poses probably the greatest challenge as it can be performed in many different ways. In case of one-dimensional restarting automaton we find the strategy of "meta-instructions" which consists of rewritten core symbols and a surrounding defined by some regular expression. This approach allows single rewriting instruction to capture variety of real situations. In two dimensions we lack the elegance of the one-dimensional regular expressions. Therefore we try to use local languages and tiles instead. Especially local languages allow us to capture the variety of surroundings, yet are both easy to define and easy to handle. Tiles on the other hand allow defining fixed surrounding of some "pixel".

Following options are therefore proposed for rewriting:

- Automaton rewrites fixed tile of size $n \times n$ to another fixed tile of the same size. n should be in any case finite, but we consider only $n = 1$, $n = 2$ and $n = 3$ as larger "tiles" would be hard to handle.

    1. Size of 1×1 (single pixel is rewritten at a time).
    2. Size of 2×2 (classic "tile" – at most 4 pixels are rewritten).
    3. Size 3×3 (i.e. at most 9 pixels at a time).

- Automaton rewrites fixed tile as above, but takes into consideration the tile surroundings. This is more like the one-dimensional case where automaton rewrites fixed word to fixed word, but takes regular expression around rewritten word into consideration. In two dimensions we will use tiles of size 2×2 which are used in tiling systems. Size of surroundings taken into account in rewriting has to be also specified. We suggest surrounding of the size 4×4, making it the rewritten tile center of the area.

Last thing regarding the rewriting is whether automaton should use working alphabet or not. By working alphabet we mean set of symbols which cannot be present in input picture and but can be used during the computation.

Accepting

Again several approaches are possible.

- Automaton accepts when the whole image contains only tiles from given (acceptance) set. And (optionally) no rewriting instruction can be used.

- Automaton can "read" the input image in the same way as a 4-way Turing machine and accepts when it reaches acceptance state (with transitions defined as in Turing machine). In other words, automaton accepts when picture belongs to some pre-defined local language.

- Combination of both approaches mentioned above. Automaton accepts when there are no rewriting instructions available, all tiles of the whole image are in acceptance tile set AND automaton reaches acceptance state after reading the whole image. This approach allows us to recognize languages containing n dots (single black pixels) where n is fixed or generally count occurrences of lines/squares/anything within the picture.

Two viable definition drafts emerged from the proposals mentioned above. First would work as Turing machine with three major exceptions. It would be bounded by the size of input picture, it would have to "restart" after every rewriting performed on the input tape and all rewritings would have to be performed in a way to lower the overall "weight" of the picture. Our second option is an automaton working with tiles. It rewrites tile to another tile (still reducing the overall weight of the picture) and accepts when picture belongs to some local language. We dismissed the option to allow the automaton using tiles to rewrite the tile if its surrounding belongs to some local language as too complicated to handle.

After several discussions and first attempts to prove some of the automaton properties we decided to continue with the proposal of automaton using tiles and created following definition.

Definition 2.1 Two-dimensional restarting automaton*, referred to as 2RA, is a 5-touple* $A = (\Sigma, \Gamma, \Theta_f, \delta, \mu)$ *where* Σ *is finite input alphabet,* Γ *is finite working alphabet,* $\Theta_f \subseteq (\Sigma \cup \Gamma \cup \{\#\})^{2,2}$ *is set of accepting tiles,* $\mu : \Sigma \cup \Gamma \rightarrow \mathbb{N}$ *is a weight function and* $\delta : (\Sigma \cup \Gamma \cup \{\#\})^{2,2} \rightarrow (\Sigma \cup \Gamma \cup \{\#\})^{2,2}$ *is set of rewriting rules satisfying condition that in every rule only single pixel on the tile is changed and all rewritings* $a \rightarrow b$ *conform with* $\mu(a) > \mu(b)$.

Note that by working alphabet Γ we understand set of symbols disjoint with input alphabet Σ. Automaton then works in a following way. Starting in an upper left corner of an input picture $P \in \Sigma^{*,*}$ automaton reads the input from left to right and from up to down ending in a lower right corner (in the same manner as westerners do read books). When it founds a tile for which rewriting rule is defined it performs the rewriting and restarts (i.e. goes to upper left corner again). When no rewriting rule can be executed for the whole picture, automaton verifies whether the picture belongs to the local language defined by Θ_f and if so it accepts. Formally:

Let $A = (\Sigma, \Gamma, \Theta_f, \delta, \mu)$ be a two-dimensional restarting automaton and P_1, P_2 be two pictures over the alphabet $\Sigma \cup \Gamma$ of the same size. We say

that the picture P_1 can be directly reduced to the picture P_2, denoted by $P_1 \vdash_A P_2$, if there exists two integers i, j, $1 \leq i \leq rows(P_1)$, $1 \leq j \leq cols(P_2)$ such that $P_1(k,l) = P_2(k,l)$ for all pairs of the indices k, l where $1 \leq k \leq rows(P_1)$, $1 \leq l \leq cols(P_2)$ except the pairs (i, j), $(i, j+1)$, $(i+1, j)$,$(i+1, j+1)$ and there exists a rule $\begin{array}{|c|c|}\hline P_1(i,j) & P_1(i,j+1) \\ \hline P_1(i+1,j) & P_1(i+1,j+1) \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline P_2(i,j) & P_2(i,j+1) \\ \hline P_2(i+1,j) & P_2(i+1,j+1) \\ \hline \end{array}$.
Moreover there is no rule in δ that could be applied to any tile on the positions $\begin{array}{|c|c|}\hline (m,n) & (m,n+1) \\ \hline (m+1,n) & (m+1,n+1) \\ \hline \end{array}$, where $1 \leq m \lneq i, 1 \leq n \leq cols(P_1)$ or $m = i, 1 \leq n \lneq j$.

We say that P_1 *can be reduced* to P_2 (denoted by $P_1 \vdash_A^* P_2$) if there exist a sequence of reductions $Q_1 \vdash_A Q_2, Q_2 \vdash_A Q_3, ..., Q_{n-1} \vdash_A Q_n$ where $n \geq 1$, $Q_1 = P_1$ and $Q_n = P_2$. Obviously, the relation $\vdash_A^*$ is the reflexive and transitive closure of the relation $\vdash_A$.

Definition 2.2 *Let $A = (\Sigma, \Gamma, \Theta_f, \delta, \mu)$ be a 2RA. The language accepted by A is the set*

$$L(A) = \{P \in \Sigma^{*,*} | \exists Q \in (\Sigma \cup \Gamma)^{*,*} : P \vdash_A^* Q \text{ and } Q \in L(\Theta_f)\}$$

In preliminary discussions automaton with internal states that would be allowed to "count" the occurrences of some tiles was proposed but we abandoned that direction.

Careful reader probably noticed, that automaton is by definition non-deterministic as for one "working" tile several rewriting rules can be defined. We define deterministic version of two-dimensional restarting automata (2DRA) by restricting set of rewriting rules for single "working" tile to at most one rule. Formally:

Definition 2.3 Deterministic two-dimensional restarting automaton*, referred to as 2DRA, is a 5-touple $A = (\Sigma, \Gamma, \Theta_f, \delta, \mu)$ where Σ is finite input alphabet, Γ is finite working alphabet, $\Theta_f \subseteq (\Sigma \cup \Gamma \cup \{\#\})^{2,2}$ is set of accepting tiles, $\mu : \Sigma \cup \Gamma \rightarrow \mathbb{N}$ is a weight function and $\delta : (\Sigma \cup \Gamma \cup \{\#\})^{2,2} \rightarrow (\Sigma \cup \Gamma \cup \{\#\})^{2,2}$ is set of rewriting rules satisfying following conditions:*

- *In every rule only single pixel on the tile is changed*
- *Rule changing pixel $a \rightarrow b$ conform with $\mu(a) > \mu(b)$*
- *For every tile $T = (\Sigma \cup \Gamma \cup \{\#\})^{2,2}$ there exists at most one rule in δ.*

We can see two-dimensional restarting automaton as an integration of thoughts from two original concepts. From the one-dimensional restarting automaton (namely shrinking restarting automaton) we took the idea of performing single rewriting operations at a time and bounding the possible rewritings by forced lowering of the picture weight. From the tiling systems and local languages we took the idea of using tiles as a basic two-dimensional objects.

To give an example of language recognized by two-dimensional restarting automaton and to show how such automaton could be defined in "practice" we use the language of squares over one-letter alphabet from Example 1.4.

Example 2.1 *Let $\Sigma = \{a\}$ and let $L \subseteq \Sigma^{*,*}$ be the language of all squares over Σ. Deterministic two-dimensional restarting automaton A recognizing L is be defined as follows. $A = (\Sigma, \Gamma, \Theta_f, \delta, \mu)$ where*

- $\Sigma = \{a\}$
- $\Gamma = \{1\}$
- $\mu = (\text{'}a\text{'} \rightarrow 2; \text{'}1\text{'} \rightarrow 1)$
- $\delta = \left\{ \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & a \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & 1 \\ \hline \end{array}, \begin{array}{|c|c|}\hline 1 & a \\ \hline a & a \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline 1 & a \\ \hline a & 1 \\ \hline \end{array} \right\}$
- $\Theta_f = \left\{ \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & 1 \\ \hline \end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline 1 & a \\ \hline \end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline a & a \\ \hline \end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline a & \# \\ \hline \end{array}, \begin{array}{|c|c|}\hline a & \# \\ \hline a & \# \\ \hline \end{array}, \begin{array}{|c|c|}\hline a & \# \\ \hline 1 & \# \\ \hline \end{array}, \begin{array}{|c|c|}\hline 1 & \# \\ \hline \# & \# \\ \hline \end{array}, \begin{array}{|c|c|}\hline a & 1 \\ \hline \# & \# \\ \hline \end{array}, \begin{array}{|c|c|}\hline a & a \\ \hline \# & \# \\ \hline \end{array}, \begin{array}{|c|c|}\hline \# & a \\ \hline \# & \# \\ \hline \end{array}, \begin{array}{|c|c|}\hline \# & a \\ \hline \# & a \\ \hline \end{array}, \begin{array}{|c|c|}\hline \# & 1 \\ \hline \# & a \\ \hline \end{array}, \begin{array}{|c|c|}\hline 1 & a \\ \hline a & 1 \\ \hline \end{array}, \begin{array}{|c|c|}\hline a & 1 \\ \hline a & a \\ \hline \end{array}, \begin{array}{|c|c|}\hline a & a \\ \hline 1 & a \\ \hline \end{array}, \begin{array}{|c|c|}\hline a & a \\ \hline a & a \\ \hline \end{array} \right\}$

Automaton uses one-letter working alphabet from which it creates artificial diagonal to ensure the input is a square picture. The weight function μ is defined in a way that the only rewritings allowed are from 'a' to '1'. We can notice that the weight function is interesting only up to the point of its existence. The most complex part of this automaton is the definition of the accepted local language. This is mainly because the input picture was simpler then the accepted one. In case of the reverse (i.e. complicated input picture and simple accepting local language) we would probably get far more rewriting rules then the number of the tiles in the accepted local language.

With the definitions done we will now focus on the closure properties of 2(D)RA and make an attempt to set it up to the current hierarchy of two-dimensional languages.

Chapter 3

Two-dimensional Restarting automata properties

This chapter contains results regarding properties of two-dimensional restarting automata like closures on operations defined in Chapter 1 and attempts to position the class of languages accepted by 2RA into existing hierarchy of two-dimensional languages. Our first guess that the 2RA accepts the same class as the tiling systems (because what it does is something like "reverse projection") can run into trouble because both tiling systems (TS) and the on-line tessellation automata (2OTA) only READ the input picture. 2RA on the other hand rewrites the input and therefore it is likely to be "stronger". We show at the end of this chapter that we directly proved only the inclusion $TS \subseteq 2RA$ and failed to decide whether it is strict or not.

Before we start working on closure properties of two-dimensional restarting automaton we first set up a lemma that will ease our future work. We have isolated a technique that will allow us to create from two general restarting automata, two restarting automata which accepts same languages but have disjoint working alphabets and disjoint symbols in acceptance set Θ_f. The property that those sets are disjoint will be crucial in several proofs.

Lemma 3.1 *Let* $A_1 = (\Sigma_1, \Gamma_1, \Theta_{f1}, \delta_1, \mu_1)$ *and* $A_2 = (\Sigma_2, \Gamma_2, \Theta_{f2}, \delta_2, \mu_2)$ *be two two-dimensional restarting automata. Then there exist two-dimensional restarting automata* A_3 *and* A_4 *with disjoint working alphabets and disjoint symbols in acceptance sets such that* $\mathcal{L}(A_3) = \mathcal{L}(A_1)$ *and* $\mathcal{L}(A_4) = \mathcal{L}(A_2)$.

Proof: Note that if acceptance sets of both automata A_3, A_4 use only symbols from working alphabets they are disjoint (as their working alphabets are disjoint).

To create automata with disjoint working alphabets it is sufficient to simply add new index $_1$ to all working symbols from one of the automata and replace all occurrences of those symbols in rewriting rules and acceptance set in the respective automaton with their indexed counterparts.

More formally, let $A_3 = A_1$ and $A_4 = (\Sigma_4, \Gamma_4, \Theta_{f4}, \delta_4, \mu_4)$ where $\Sigma_4 = \Sigma_2$, $\Gamma_{4-work} = \{g_1 | g \in \Gamma_2\}$. $\Theta_{f4-work} = \Theta_{f2}$ and $\delta_{4-work} = \delta_2$ where in both $\Theta_{f4-work}$ and δ_{4-work} every occurrence of symbol from Γ_2 has been replaced with respective symbol from Γ_{4-work}. In case the input alphabets Σ_1 and Σ_2 were disjoint we could simply set $\Gamma_4 = \Gamma_{4-work}$, $\Theta_{f4} = \Theta_{f4-work}$ and $\delta_4 = delta_{4-work}$. μ_4 would be defined in a same way as μ_2 with all the symbols from Γ_2 replaced by their respective indexed counterparts from Γ_4.

In the case the input alphabets Σ_1 and Σ_2 are overlapping we add all symbols from input alphabet Σ_2 to working alphabet Γ_4 (again with the new index $_1$) and create new set of rewriting rules which at the beginning of the computation transform all the symbols from the input alphabet to their respective symbols in working alphabet. We then replace the symbols in acceptance set in the same manner.

Again more formally. $\Sigma_{4-work} = \{s_1 | s \in \Sigma_4\}$, $\Gamma_4 = \Gamma_{4-work} \cup \Sigma_{4-work}$. $\Theta_{f4} = \Theta_{f4-work}$ where every occurrence of symbol from Σ_4 has been replaced with respective symbol from Σ_{4-work}. At last $\delta_4 = \delta_{4-work} \cup \delta_{4-inputTOwork}$, where in δ_{4-work} every occurrence of symbol from Σ_4 has been replaced with respective symbol from Σ_{4-work} and $\delta_{4-inputTOwork}$ contains for every $s \in \Sigma_4$ the following 4 rules:

$$\begin{array}{|c|c|}\hline s & * \\ \hline * & * \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline s_1 & * \\ \hline * & * \\ \hline\end{array} \quad \begin{array}{|c|c|}\hline * & s \\ \hline * & * \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline * & s_1 \\ \hline * & * \\ \hline\end{array} \quad \begin{array}{|c|c|}\hline * & * \\ \hline s & * \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline * & * \\ \hline s_1 & * \\ \hline\end{array} \quad \begin{array}{|c|c|}\hline * & * \\ \hline * & s \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline * & * \\ \hline * & s_1 \\ \hline\end{array}$$

where '$*$' stands for any symbol. The computation of the automaton A_4 then looks like following. Before using any rewriting rule from the actual "computation" set it first uses the new rewriting rules that just change symbols on the tape from the input to the working alphabet (this is because there are no other rewriting rules handling the input symbols). Then it continues with the original computation. μ_4 is defined in a way allowing all the rewritings in δ_4 – for example in a same way as μ_2 with following exceptions: All the "indexed" input symbols from Γ_4 take place of the original input symbols from Σ_2 and the original symbols have assigned weight far greater then any other symbol from Γ_4. Then every input symbol can be rewritten to its "indexed" counterpart and all the rewrites from the original automaton are allowed as well. Thus $L(A_2) \subseteq L(A_4)$. The inverse inclusion comes from the fact that A_4 at first rewrites the input symbols to their respective counterparts in the working alphabet (and we have shown that those are the rewriting that need to be done first) and then it simulates the automaton A_2. Therefore any computation of A_4 stripped of the first rewriting of the input symbols can be performed by A_2 as well.

Note that if A_2 is deterministic then A_4 is deterministic as well (same holds for A_1 and A_3).

q.e.d.

3.0.3 Closure properties of 2RA

We now mention all closure properties of 2RA we were able to prove.

Lemma 3.2 *The family $\mathcal{L}(2RA)$ is closed under projection.*

Proof: Let Σ_1, Σ_2 be two finite alphabets and let $\varphi : \Sigma_1 \rightarrow \Sigma_2$ be a projection. We will prove that if picture language $L_1 \subseteq \Sigma_1^{*,*}$ is recognizable by 2RA then $L_2 = \varphi(L_1)$ is recognizable by 2RA, too.

Let $A_1 = (\Sigma_1, \Gamma_1, \Theta_{f1}, \delta_1, \mu_1)$ be a two-dimensional restarting automaton recognizing L_1. We create two-dimensional restarting automaton A_2 recognizing L_2 in the following manner.

$A_2 = (\Sigma_2, \Gamma_2, \Theta_{f1}, \delta_2, \mu_2)$ where $\Gamma_2 = \Gamma_1 \cup \Sigma_1$. $\delta_2 = \delta_1 \cup \delta_{2-work}$ where δ_{2-work} contains rewriting rules creating an inverse transformation to φ.

Formally, $\delta_{2-work} = \left\{ d | \forall s \in \Sigma_1 \; d = \begin{array}{|c|c|} \hline \varphi(s) & * \\ \hline * & * \\ \hline \end{array} \rightarrow \begin{array}{|c|c|} \hline s & * \\ \hline * & * \\ \hline \end{array} \right\}$ where '$*$' stands for any symbol from $\Sigma_2 \cup \Gamma_2$.

The computation is then very similar to the computation of automaton in the proof of Lemma 3.1. Before automaton does any "computational" rewritings it first "reverses" the projection applied to input alphabet and then continues with the computation of A_1. Therefore we can see that if A_2 accepts input picture P, then there exists P_1 accepted by A_1 such that $P = \varphi(P_1)$. Note that for single picture P there could be multiple possibilities of "pre-images" P_1. A_2 has to non-deterministically choose one of them. It is also clear that A_2 cannot accept any pictures outside the $\varphi(\mathcal{L}(A_1))$ as it actually "simulates" A_1 in its own computation.

q.e.d.

We now prove that the family of languages recognized by 2RA is closed under row and column concatenation. As proofs for the row and column cases would be almost the same, we formally prove only the column concatenation closure. We do so in the following way. First we non-deterministically mark the parts of the concatenated picture with separated indices. Then we use the rewriting rules of the left picture automaton on the left part and the rewriting rules of the right picture automaton on the right part.

Lemma 3.3 *The family $\mathcal{L}(2RA)$ is closed under row and column concatenation.*

Proof: As stated above we will prove only the column concatenation, as the proof for the row concatenation is similar.

Let L_1 and L_2 be two picture languages recognized by two-dimensional restarting automata $A_1 = (\Sigma, \Gamma_1, \Theta_{f1}, \delta_1, \mu_1)$ and $A_2 = (\Sigma, \Gamma_2, \Theta_{f2}, \delta_2, \mu_2)$ respectively (we assume w.l.o.g. that the input alphabets are the same). Our goal is to construct two-dimensional restarting automaton recognizing language $L = L_1 \oplus L_2$ containing all pictures P, such that $P = P_1 \oplus P_2$ where $P_1 \in L_1$ and $P_2 \in L_2$.

We can assume w.l.o.g. (as a result of the Lemma 3.1) that both working alphabets and symbols in acceptance sets are disjoint.

The proof is based on the idea that we are able to non-deterministically mark both original parts of the concatenated picture and then simulate the computation of the automaton A_1 on the left part while simulating the automaton A_2 on the right part. The resulting automaton will accept if both A_1 and A_2 accepts.

Imagine marking by adding new index to all symbols on the tape that belong to P_1 (P_2 respectively). In this way we do not destroy any information about the input picture.

More formally we will use following rewriting rules:

$$\begin{array}{|c|c|}\hline \# & \# \\ \hline \# & s \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & s_1 \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline \# & \# \\ \hline *_1 & s \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline *_1 & s_1 \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline \# & \# \\ \hline *_1 & s \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline *_1 & s_2 \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline \# & \# \\ \hline *_2 & s \\ \hline \end{array} \rightarrow$$

$$\begin{array}{|c|c|}\hline \# & \# \\ \hline *_2 & s_2 \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline *_1 & *_1 \\ \hline s & * \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline *_1 & *_1 \\ \hline s_1 & * \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline *_1 & *_2 \\ \hline s & * \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline *_1 & *_2 \\ \hline s_1 & * \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline *_2 & *_2 \\ \hline s & * \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline *_2 & *_2 \\ \hline s_2 & * \\ \hline \end{array},$$

$$\begin{array}{|c|c|}\hline *_2 & \# \\ \hline s & \# \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline *_2 & \# \\ \hline s_2 & \# \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & s \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & s_2 \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline *_1 & \# \\ \hline s & \# \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline *_1 & \# \\ \hline s_1 & \# \\ \hline \end{array}$$

We are using two indices: $_1$ for the left part of the picture and $_2$ for the right part. s stands for a specific symbol from the input alphabet and s_1 (resp. s_2) for its respective indexed counterpart. $*$ stands for any symbol from the input alphabet and $*_1$ (resp. $*_2$) for any symbol with respective index added. You can notice that the rewriting rules are defined in a way to allow indexing of whole picture with the same index (be it $_1$ or $_2$). This is because either language L_1 or language L_2 could contain an empty picture λ.

Assume that we are concatenating picture languages $\mathcal{L}$(H_LINES), $\mathcal{L}$(V_LINES) where V_LINES contains all pictures with vertical lines and H_LINES contains all pictures with horizontal lines. After the transformation, the picture could be:

#	#	#	#	#	#	#	#
#	0_1	1_1	1_1	0_2	0_2	0_2	#
#	0_1	0_1	0_1	0_2	1_2	0_2	#
#	0_1	0_1	0_1	0_2	1_2	0_2	#
#	1_1	1_1	1_1	0_2	1_2	0_2	#
#	0_1	0_1	0_1	0_2	1_2	0_2	#
#	0_1	0_1	0_1	0_2	0_2	0_2	#
#	#	#	#	#	#	#	#

We then define $A = (\Sigma, \Gamma, \Theta_f, \delta, \mu)$ as follows. Take $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Sigma_{11} \cup \Sigma_{22}$ where Σ_{11} contains all elements from Σ_1 with the new index $_1$ and Σ_{22} contains all elements from Σ_2 with the new index $_2$. Set of rewriting rules δ contains all elements from δ_1, all elements from δ_2 and the "transformation rules" mentioned above with the following modifications: All occurrences of the symbols from Σ_1 in δ_1 are replaced with their "indexed" counterparts from Γ same goes for the symbols from Σ_2 in δ_2.

Moreover because we possibly have no right border of the picture P_1 and no left border of the picture P_2, we need to modify the rewriting instructions further (in case either L_1 or L_2 contains an empty picture λ this modification is not needed for the rules from δ_1 or δ_2 or both). The rules from set δ_1 containing the right border symbols are replaced with tiles where the right side of the tile has been replaced with all the possible symbols from $\Gamma_2 \cup \Sigma_{22}$. Technically we can replace every tile with $|\Gamma_2 \cup \Sigma_{22}|^2$ tiles, standing for all possible "neighbors" from the picture P_2. Because of the disjoint alphabets this poses no problem in the recognition. The same modification is done with the set δ_2 (in this case we consider the left border and alphabet $\Gamma_1 \cup \Sigma_{11}$). More formally, we define the following sets of rules.

δ'_{11} = For every rule containing tile with right border (i.e. tile $\begin{array}{|c|c|}\hline a_1 & \# \\ \hline c_1 & \# \\ \hline\end{array}$) we create set of rules where the right border symbols are replaced with all possible pairs from $\Sigma_2 \cup \Gamma_2$. Note that in the case of upper right corner and lower right corner tiles, we replace only the lower right (upper right symbol) (i.e. in tiles $\begin{array}{|c|c|}\hline a_1 & \#_r \\ \hline \# & \# \\ \hline\end{array}$ only the $\#_r$ symbol is replaced).

δ'_{22} = For every rule containing tile with left border (i.e. tile $\begin{array}{|c|c|}\hline \# & a_2 \\ \hline \# & c_2 \\ \hline\end{array}$) we create set of rules where the left border symbols are replaced with all possible pairs from $\Sigma_1 \cup \Gamma_1$. Note that in the case of upper right corner and lower right corner tiles, we replace only the lower right (upper right symbol) (i.e. in tiles

$\#_r$	a_2
#	#

only the $\#_r$ symbol is replaced).

δ is then defined as $\delta = \delta_{11} \cup \delta'_{11} \cup \delta_{22} \cup \delta'_{22}$ where δ_{11} stands for set δ_1 with replaced symbols from Σ as stated above. Note that if L_2 does not contain an empty picture λ, then all rewriting rules from δ_1 containing the border symbol # in the right column will be removed from δ. Same goes for the picture L_1 and rewriting rules from δ_2. Accepting set $\Theta_f = \Theta_{f1} \cup \Theta_{f2}$ is modified in the same manner as rewriting rules (with the same note about empty pictures). Moreover all symbols from Σ in Θ_{f1} are replaced with their indexed counterparts from Σ_{11} and all symbols from Σ in Θ_{f2} are replaced with their indexed counterparts from Σ_{22}. Last thing remaining is to show that there exists the "weight function" μ which conforms with all rewritings mentioned above. If we join the functions μ_1, μ_2 and replace all the occurrences of symbols from Σ with their respective indexed counterparts (from Σ_{11} in case of μ_1 and from Σ_{22} in case of μ_2) we allow all rewritings of the original automata. The "transformation" rewritings taking care of the indexing of the picture can be allowed for example by assigning higher then current maximal weight to all symbols from the input alphabet Σ .

It is easy to show that if A_1 can accept P_1 and A_2 can accept P_2, then there exists an accepting computation of A on $P_1 \varobar P_2$. Hence, $L(A_1) \varobar L(A_2) \subseteq L(A)$. To show the opposite inclusion we must consider all the accepting computations of A.

As automaton is not forced to prefer some rewriting instructions over others we need to take a look how the successful computation will look like (and if the automaton does not accept some pictures it should rather reject). First we make an observation that if automaton uses some "transformation" instructions (i.e. the ones that change symbols from Σ to their indexed counterparts) it necessarily transforms whole "left part" rectangle. So situations like the following one are impossible:

#	#	#	#	#
#	$*_1$	$*_1$	$*$	#
#	$*_1$	$*_1$	$*$	#
#	$*_1$	$*_1$	$*$	#
#	$*$	$*$	$*$	#
#	#	#	#	#

This is because once an input symbol is indexed with either $_1$ or $_2$ there are no instructions to transform it back. And there are no accepting tiles of the former automaton A_1 with indexed symbols ($*_1$) on the top and other than indexed $*_1$ or # symbols on the bottom.

Therefore we have guaranteed that the transformed symbols form a rectangle that starts at the top of the picture, ends at the bottom and bounds to the left side. On the left part marked by the index $_1$, only the instructions working with the symbols from Γ_1 can be used (those are the rewriting instructions from A_1). On the right part marked by the index $_2$ on the other hand only instructions from A_2 can be used.

When automaton accepts the picture it must be in a state where left part can be recognized by acceptance set of Θ_{f1} and right part by Θ_{f2} (either left or right part can be empty!). The picture P therefore had to be concatenation of some $P_1 \oplus P_2$ where $P_1 \in L(A_1)$ and $P_2 \in L(A_2)$.

Proof can be easily modified to work for the row concatenation.

q.e.d.

Lemma 3.4 *The family $\mathcal{L}(2RA)$ is closed under union and intersection.*

Proof: Let L_1 and L_2 be two picture languages and let $A_1 = (\Sigma_1, \Gamma_1, \Theta_{f1}, \delta_1, \mu_1)$ and $A_2 = (\Sigma_2, \Gamma_2, \Theta_{f2}, \delta_2, \mu_2)$ be two two-dimensional restarting automata that recognize L_1 and L_2 respectively.

We assume w.l.o.g. that both the working alphabets Γ_1 and Γ_2 and the symbols in both accepting sets Θ_{f1} and Θ_{f2} are disjoint. Moreover we assume w.l.o.g. that the input alphabets Σ_1 and Σ_2 are the same.

A restarting automaton A_u for the "union language" $L = L_1 \cup L_2$ is quite easy to construct. We will use the same indexing "trick" we used in the proof of the previous lemma. Let Σ_{11} be an alphabet same as Σ_1 where every symbol had been attributed a new index $_1$ and Σ_{22} be an alphabet same as Σ_2 where every symbol had been attributed a new index $_2$. We also define δ_{11} which contains all the rules from δ_1 with the exception that every occurrence of symbol from Σ_1 is replaced with the respective symbol from Σ_{11}. Same holds for Θ_{f11}, δ_{22} and Θ_{f22}.

We then define $A_u = (\Sigma_u, \Gamma_u, \Theta_{fu}, \delta_u, \mu_u)$ where $\Sigma_u = \Sigma_1 \cup \Sigma_2$, $\Gamma_u = \Gamma_1 \cup \Sigma_{11} \cup \Gamma_2 \cup \Sigma_{22}$, $\Theta_{fu} = \Theta_{f11} \cup \Theta_{f22}$. $\delta_u = \delta_{11} \cup \delta_{22} \cup \delta_{work}$, where δ_{work} contains rewriting rules that index the whole input picture with the either index $_1$ or $_2$ by non-deterministically choosing the tile in the upper left corner and expanding the chosen index to the whole picture. μ_u is defined in a way, to allow all the rewriting rules defined in δ_{fu}. The automaton A_u then at the beginning of the computation non-deterministically chooses which set of the rewriting rules will be used (by choosing the first index), indexes the whole picture (as there are no other rewriting rules working with the input alphabet) and then follows the computation of either A_1 or A_2. Hence, $L_1 \cup L_2 \subseteq L(A_u)$. The opposite inclusion follows from the fact that the first rewriting of an accepting computation of the automaton A_u inserts a symbol indexed by x, $x \in \{1, 2\}$ into the picture. After that it can work only using the rewritings from $\delta_{xx} \cup \delta_{work}$ which allow only symbols indexed by x in the rewritings. Hence the last picture which is accepted by A_u corresponds to a picture from $L(\Theta_x)$. Hence the input word belongs to L_x.

We will now construct the two-dimensional restarting automaton A_i accepting the "intersection language" $L_1 \cap L_2$. The idea is to simulate runs of both restarting automata A_1, A_2 on the input picture and accept only if both automata would accept. We define two intermediate working alphabets $\Gamma_{1w} = \Gamma_1 \cup \Sigma_{11}$, $\Gamma_{2w} = \Gamma_2 \cup \Sigma_{22}$.

We define $A_i = (\Sigma_i, \Gamma_i, \Theta_{fi}, \delta_i, \mu_i)$ where $\Sigma_i = \Sigma_1 \cup \Sigma_2$, $\Gamma_i = \Gamma_{1w} \times \Gamma_{2w}$. Rewriting rules Θ_{fi} are defined so that every input symbol is firstly rewritten into the working alphabet in a way that every $a \in \Sigma$ is replaced with pair (a_1, a_2) where a_1 stands for symbol a from Γ_{1w} with index 1 and the same holds for a_2. Other rewriting rules are defined so that every rewriting simulates either rewriting instruction of A_1 or rewriting instruction of A_2. Formally for every instruction $\begin{array}{|c|c|}\hline a_1 & b_1 \\ \hline c_1 & d_1 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline e_1 & b_1 \\ \hline c_1 & d_1 \\ \hline\end{array}$ in δ_1 we add rewriting instruction set $\begin{array}{|c|c|}\hline (a_1,a_2) & (b_1,b_2) \\ \hline (c_1,c_2) & (d_1,d_2) \\ \hline\end{array} \rightarrow$ $\begin{array}{|c|c|}\hline (e_1,a_2) & (b_1,b_2) \\ \hline (c_1,c_2) & (d_1,d_2) \\ \hline\end{array}$ for all possible combinations of $a_2, b_2, c_2, d_2 \in \Gamma_{2w}$. Similar instructions are created for all rewriting instructions in δ_2. This way with every rewriting instruction of automaton A we simulate single rewriting instruction of either A_1 or A_2.

The accepting set Θ_{fi} is then defined as something like "$\Theta_{f1} \times \Theta_{f2}$" in a sense that for every tile $\begin{array}{|c|c|}\hline a_1 & b_1 \\ \hline c_1 & d_1 \\ \hline\end{array}$ in Θ_{f1} and every tile $\begin{array}{|c|c|}\hline a_2 & b_2 \\ \hline c_2 & d_2 \\ \hline\end{array}$ in Θ_{f2} we add to Θ_{fi} the tile $\begin{array}{|c|c|}\hline (a_1,a_2) & (b_1,b_2) \\ \hline (c_1,c_2) & (d_1,d_2) \\ \hline\end{array}$. In the case of border tiles we do so only with tiles of the "same type", i.e. tiles $\begin{array}{|c|c|}\hline \# & b_1 \\ \hline \# & d_1 \\ \hline\end{array}$ are combined only with tiles $\begin{array}{|c|c|}\hline \# & b_2 \\ \hline \# & d_2 \\ \hline\end{array}$. We can see that in this way picture is accepted if only if it is accepted by both A_1 and A_2.

q.e.d.

Lemma 3.5 *The family $\mathcal{L}(2RA)$ is closed under rotation.*

Proof: We prove closure under rotation by showing how a 2RA A' with the input picture P^R can simulate given 2RA A working on the input picture P. We can easily see that by simply rotating all tiles in both rewriting rules and accepting set we get automaton that will "almost" work. The only remaining issue to solve is the fact that our "new" automaton reads the original picture in the direction from the lower left corner to the upper right instead of the original direction from the upper left corner to the lower right. Because the automaton A' performs the first rewriting available in its reading direction it could perform some rewriting not available for the original automaton A and therefore accept (reject) picture it should rather reject (accept).

To solve this issue we first need to make an observation.

Observation 1: *When instruction rewrites a pixel in the picture there is no need to check all the tiles up from the beginning. It is enough to check just the tiles surrounding the rewritten pixel.*

Because each 2RA performs the FIRST rewriting available, when it passes some parts of the picture without rewriting we can be sure, there are no rewritings possible at the moment. Imagine for example, how would space bounded 4-way Turing machine simulate the restarting automaton. We can safely assume that the machine reads a complete tile instead of one pixel. Starting in the upper left corner the machine reads the input picture line by line and when it finds possible the rewriting position it performs the rewriting. After the rewriting the machine needs to check the positions surrounding the rewritten pixel because this change could allow some rewritings on "checked before" positions (i.e. positions above and left from the rewritten one). For example if there is a rewriting at the position (3,5) we can be sure there can be no rewriting at the position (1,1) at the moment.

When speaking about the rotated picture P^R, if pixel "r" has been rewritten, only pixels with '*' have to be checked.

#	#	#	#	#	#	#	#
#							#
#							#
#			$*$	$*$	$*$		#
#				r	$*$		#
#				$*$	$*$		#
#							#
#	#	#	#	#	#	#	#

We can mark the pixels surrounding the rewritten one and then check only those. If there is another rewriting on those marked pixels we mark some more and so on and so forth.

At this point we can make another observation. Let $C = |\Gamma| + |\Sigma|$

Observation 2: *Turing machine simulating restarting automata will rewrite each pixel at most* $10 * C + 3$ *times*

Every pixel can be rewritten at most C times (because of the weight function μ), moreover each pixel can be marked at most $8 * C$ - each time when one of its neighbors is rewritten. Because the number of maximum rewritings depends only on the size of the original working alphabet it gives us a good hope for simulating one restarting automaton with another.

More specifically we will simulate the run of the original restarting automaton on the rotated picture with restarting automaton with larger working alphabet.

First imagine that instead of the original working alphabet symbol "a" we get the set of symbols

-/r		$1 - (10 * C + 3)$
-/2/2t	**a**	-/rw/rw-t
-/1/1-t		-/c/c-t/cc

where upper left index has values "-" (as nothing) or "r" (as read) symbolizing whether this pixel was read before. Lower left index with values "-", "1" and "1-t" marks the last field read. The middle left index is used for backward marking and will be explained later. The lower right index with values "-", "c", "c-t" and "cc" marks the changed pixels and is used for marking the surrounding of rewritten pixel. The upper right index has values from 1 to $10 * C + 3$. For every symbol in original alphabet we create all possible combinations of those indices so the new automaton has working alphabet of size $2 * 3 * 3 * |\Gamma| * (10 * C + 3) * 3 * 4$.

The rewriting rules of the original automaton A are modified in the following way. We modify all rules of A' so they work only when their lower left corner symbol contains the index "1" (as actual pixel). The only exceptions are the tiles where lower left symbol is "#". In those cases we require that lower right pixel has the index. If it also contains the symbol "#" then the upper left pixel has the index and finally in case of lower left corner tile we require that the index "1" is in the upper right pixel.

A' then works as follows. First it marks the upper right corner with the "actual pixel" symbol "1" and then transports it in direction top to bottom right to left to bottom left corner (this is the read direction of the original automaton). Because the only rewriting rules we allow contain the actual pixel symbol "1" we can be sure that all rewritings are done in the right direction.

The transportation of the symbol has to be done in two stages. Because we are allowed to rewrite only one pixel at a time, we first have to mark the field below the symbol "1" and then unmark the original position. So we have rules like: $\begin{array}{|c|c|}\hline a_1 & b \\ \hline c & d \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_1 & b \\ \hline c_1 & d \\ \hline\end{array}$ and $\begin{array}{|c|c|}\hline a_1 & b \\ \hline c_1 & d \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a & b \\ \hline c_1 & d \\ \hline\end{array}$ (a_1 stands for symbol a with lower left index "1"). In this way when we move the index "1" we do so in direction top to bottom and can be therefore sure that we did not skip any possible rewriting before (because else it would be performed). Moreover when we are moving the actual index "1" we ensure that at first the index is copied on pixel below and in the following cycle the upper index is deleted. The only problem arises when we reach the bottom part of the picture P^R because we need to somehow transport the index "1" back to the top of next column to the left, but cannot do it in the same way as we did on the way down (it would lead to whole column having the index "1").

This is what we have the index "2" for. We create the rules

- $\begin{array}{|c|c|}\hline a & b_1 \\ \hline \# & \# \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_{2t} & b_1 \\ \hline \# & \# \\ \hline\end{array}$
- $\begin{array}{|c|c|}\hline a_{2t} & b_1 \\ \hline \# & \# \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_{2t} & b \\ \hline \# & \# \\ \hline\end{array}$
- $\begin{array}{|c|c|}\hline a_{2t} & \mathrm{b} \\ \hline \# & \# \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_2 & \mathrm{b} \\ \hline \# & \# \\ \hline\end{array}$
- $\begin{array}{|c|c|}\hline \mathrm{a} & \mathrm{b} \\ \hline c_2 & \mathrm{d} \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_2 & \mathrm{b} \\ \hline c_2 & \mathrm{d} \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline \# & \# \\ \hline a_2 & \mathrm{d} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline a_1 & \mathrm{d} \\ \hline\end{array}$

Those rules will allow us to transport the index "1" back to the top of the next column. The use of temporary index "2t" is necessary because otherwise the index "1" would be left on the bottom of the original column (as automaton would first find the rewriting position for transportation of the index "2" to the top). We presume that the symbols a, b, c, d stand for all possible choices from working alphabet as well as the symbol # in some cases. As a part of the index "1" movement we also mark the visited pixels with the "r" index (for example when we visit the position first time we can add both indices – "1" and "r"). This mark will be used later. For the lack of space we do not put here the whole set of the rules but rather present the idea.

When the rewriting occurs (i.e. the original automaton uses rewriting rule) we need to perform three steps to get the correct indexing. First, we rewrite the original symbol as the rewriting instruction does and mark it with "rw-t" (rewritten-temporary) index. Second, we "stop" the movement of the current index "1" as we rewrite it to index "1-t" (1-temporary). Third, we replace the "rw-t" index with "rw" index and continue with checking the surrounding of the rewriting.

The "rechecking" of the previously visited pixels is done in the following way. We can make an observation that the only pixels needed to be "rechecked" are the surrounding pixels that were already read (those we have not visited yet have no sense of rechecking). Moreover, we mark only the upper left pixels of the tiles that may contain the rewritten pixel (the same place where the pixel with index "1" is).

We need to mark the following pixels around the rewritten one ("rw" is the rewritten pixel and pixels containing "*" will be marked).

#	#	#	#	#	#	#
#						#
#						#
#		$*$	$*$			#
#		$*$	$rw*$			#
#						#
#						#
#	#	#	#	#	#	#

Note that the marking of the pixels on the left may not be necessary as they may be unread yet. This marking will be done by the following rules

- $\begin{array}{|c|c|}\hline \mathrm{a} & \mathrm{b} \\ \hline \mathrm{c} & d_{rw-c} \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \mathrm{a} & \mathrm{b} \\ \hline c_c & d_{rw-c} \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline \mathrm{a} & \mathrm{b} \\ \hline c_c & d_{rw-c} \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_{ct} & \mathrm{b} \\ \hline c_c & d_{rw-c} \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a_{ct} & \mathrm{b} \\ \hline c_c & d_{rw-c} \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_{ct} & b_{ct} \\ \hline c_c & d_{rw-c} \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a_{ct} & b_{ct} \\ \hline c_c & d_{rw-c} \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_c & b_{ct} \\ \hline c_c & d_{rw-c} \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a_c & b_{ct} \\ \hline c_c & d_{rw-c} \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_c & b_{ct} \\ \hline c_c & d_c \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a_c & b_{ct} \\ \hline c_c & d_c \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_c & b_c \\ \hline c_c & d_c \\ \hline\end{array}$

Careful reader probably noticed that we needed the temporary index “ct” again. This is because we will behave to the index “c” in the same way as to the index “1” in the sense that we allow rewritings when the lower left pixel of the rewritten tile contains the index “c” (and no other pixel on the tile has that index). In this way the pixels are checked in the original direction. When there was no rewriting rule for the checked tile in the original automaton *A*, we change the index “c” to “cc”. If there are some rewritings we mark more pixels and continue until we check all the pixels and remove all the marks “cc”. Because we also remove the “rw” indices in the process, the last index left will be the “1-t”. When it is alone (i.e. there are no “c”, “rw” or “cc” indices around) we rewrite it back to “1” and continue with the reading. To make this part a bit clearer we present the example of the rewriting rules of situation when there were no rewritings defined for the tiles in question in the original automaton *A*.

Rules taking care of unmarking the surroundings of the rewriting

- $\begin{array}{|c|c|}\hline a_c & b \\ \hline c_c & d \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_{cc} & b \\ \hline c_c & d \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a_{cc} & b \\ \hline c_c & d \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a_{cc} & b \\ \hline c_{cc} & d \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a_{cc} & b \\ \hline c & d \\ \hline\end{array} \to \begin{array}{|c|c|}\hline a & b \\ \hline c & d \\ \hline\end{array}$

And we are allowing following combination of indices for the original rewriting rules of the automaton A to work:

- $\begin{array}{|c|c|}\hline a & b \\ \hline c_1 & d \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a & b \\ \hline c_c & d \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a_{cc} & b \\ \hline c_c & d \\ \hline\end{array}$

In this way the automaton A' does the rewriting of the original symbols (not the indices) in exactly same order as the original automaton A.

To summarize the simulation, automaton A' uses several auxiliary indices around the original symbols to ensure the original automaton A reading (and rewriting) direction. The index "1" is used to represent the head of the automaton and moves in the original reading direction. All rewriting rules are modified to work only with certain combination of indices. In case A' simulates a rewriting by A, indices of the surrounding symbols are marked in a way that the automaton A' is forced to check the same tile the original automaton would. At the end of the simulation there will be some indices "2" left on the picture and lower left corner will contain index "1". All other indices are removed in the process of the simulation. The accepting set will be modified to work with those indices.

The last thing we need to check is the number of rewrites we will need for each pixel (including the marking!) as there must exist the weight function μ and therefore we can rewrite each pixel at most $|\Sigma' \cup \Gamma'|$ times (where Σ' and Γ' are the working and input alphabets of the automaton A' respectively). Let $C = |\Gamma \cup \Sigma|$. The original automaton A could rewrite each pixel at most C times. We do 2 rewrites in the process of adding and removing the index "1" and one more when we use the index "2" for transportation of index "1". Moreover every pixel can be marked by rewritings of its neighbors. We mark 4 surrounding neighbors (at most), use temporary indices in the process and remove them in the end which counts up to at most $4 * C * 5$ rewrites. Summarizing, every pixel can be rewritten at most $C + 3 + 4 * C * 5 = 10C + 3$ times. We construct the weight function μ in a way to allow all the rewritings needed. The last index we have not used yet is the "counter" index in the upper right corner of the working alphabet of A'. It is used for example when we need to "mark" and "unmark" the index "c" but leave all other indices (and the original symbol) intact. We do so by decreasing the counter index by 1. For example when we have a pixel with counter index 134 we can add an index "c" but when we want to remove it we need to decrease the counter index to 133 (as weight function does not allow us to return to the same symbol we used once). When we create the counter index sufficiently large (from one up to 10∗C+3) we can even lower it each time some rewriting occurs.

q.e.d.

We now show that the language family 2RA is also closed under mirroring. We choose to prove only closure under vertical mirroring as horizontal can be obtained as a combination of vertical mirroring and rotation.

Lemma 3.6 *The family $\mathcal{L}(2RA)$ is closed under mirroring.*

Proof: As stated above we will prove only closure under vertical mirroring. For the purpose of this proof we can see mirroring as a special type of rotation because our approach from the proof of closure under rotation (lemma 3.5) will work here as well. The only modifications needed are those dealing with the direction of readings/rewritings. As modification of the whole proof would be long and tiring we just show how to modify the movement of the actual pixel index "1".

The index "1" starts again in the upper right corner but this time moves from right to left and from top to bottom. The movement from right to left is done with following rules.

- $\begin{array}{|c|c|}\hline \mathrm{a} & \mathrm{b} \\ \hline \mathrm{c} & d_1 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \mathrm{a} & \mathrm{b} \\ \hline c_1 & d_1 \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline \mathrm{a} & \mathrm{b} \\ \hline c_1 & d_1 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \mathrm{a} & \mathrm{b} \\ \hline c_1 & \mathrm{d} \\ \hline\end{array}$

For the vertical movement from top to bottom we again use auxiliary index "2" and the following set of rules.

- $\begin{array}{|c|c|}\hline \# & a_1 \\ \hline \# & b \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & a_1 \\ \hline \# & b_2 \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline \# & a_1 \\ \hline \# & b_2 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & a \\ \hline \# & b_2 \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a & b \\ \hline c_2 & d \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline a & b \\ \hline c_2 & d_2 \\ \hline\end{array}$

- $\begin{array}{|c|c|}\hline a & b \\ \hline c_2 & d_2 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline a & b \\ \hline c & d_2 \\ \hline \end{array}$

- $\begin{array}{|c|c|}\hline a_2 & \# \\ \hline b & \# \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline a_1 & \# \\ \hline b & \# \\ \hline \end{array}$

We can see that this time, thanks to the reading direction of the new automaton the index "2" will even be removed in the process (aside from the last pixel where it will remain).

All other rewrites will be modified in the same way.

q.e.d.

In [4] D. Giammarresi and A. Restivo showed that $\mathcal{L}(TS)$ is not closed under complement. We conjecture that $\mathcal{L}(2RA)$ is also not closed under complement.

Conjecture 3.1 *The family $\mathcal{L}(2RA)$ is NOT closed under complement.*

Idea of the proof: We will use the idea and the language that was introduced by D. Giammarresi and A. Restivo in [4] when showing that $\mathcal{L}(TS)$ is not closed under complement. Let $\Sigma = \{a, b\}$ be an alphabet and let $L = \{P \in \Sigma^{*,*} | P = s \ominus s$ where s is a square$\}$. That is, language L contains pictures of size $(2n, n)$ for every $n \in N$ such that the top and the bottom square halves are identical. The proof of the conjecture could be given by showing that $L \notin \mathcal{L}(2RA)$ while its complement $\overline{L} = \{a, b\} \backslash L \in \mathcal{L}(2RA)$.

We presume that checking whether the two squares one above the next are exactly the same cannot be done in linear $O(n)$ (where $n = cols(P) * rows(P)$ is the size of the input picture) time yet we were unable to prove it formally. If this is the case, then two-dimensional restarting automaton would be unable to recognize such language as Turing machine simulating the 2RA is running in linear complexity and it would lead to contradiction. The result of Turing machine simulation could be obtained by slight modification of the proof of the closure by rotation.

The fact that $\overline{L} \in \mathcal{L}(2RA)$ can be shown in exactly the same way as in the original proof in [4].

We decompose $\overline{L} = L_1 \cup L_2$, where:
$L_1 = \{P \in \Sigma^{*,*} | rows(P) \neq 2 * cols(P)\}$
$L_2 = \{P \in \Sigma^{*,*} | rows(P) = 2 * cols(P)$ and top and bottom halves are different$\}$.

It is quite easy to show that L_1 is recognizable, using a restarting automaton which builds up a line, within a rectangle, that declines stepwise two squares by one, starting at the top left corner and missing the bottom right corner. On the other hand, L_2 can be written as:

$$L_2 = L_3 \cap (\Sigma^{*,*} \ominus (L_4 \cap (\Sigma^{*,*} \oplus L_5 \oplus \Sigma^{*,*})) \ominus \Sigma^{*,*})$$

where:
$L_3 = \{P \in \Sigma^{*,*} | rows(P) = 2 * cols(P)\}$
$L_4 = \{P \in \Sigma^{*,*} | rows(P) = cols(P) + 1\}$
$L_5 = \{P \in \Sigma^{*,*} | cols(P) = 1 \text{ and } P(1,1) \neq P(1, rows(P))\}$

Languages L_3 and L_4 can be recognized by the techniques similar to the one for L_1 described above. To see that L_5 is recognizable by the restarting automaton, we can imagine sending "index wave" from the first symbol to the last (where the index will be the actual symbol on the first position of the picture). This shows that the language $L_2 \in \mathcal{L}(2RA)$ and because of closure properties of $\mathcal{L}(2RA)$ $\overline{L} \in \mathcal{L}(2RA)$

We summarize all two dimensional restarting automaton closure properties in following theorem.

Theorem 3.1 *The family $\mathcal{L}(2RA)$ is closed under*

- *both row and column concatenation*
- *union and intersection*
- *rotation*
- *both horizontal and vertical mirroring*

3.0.4 Inserting 2RA into hierarchy of two dimensional languages

Lemma 3.7 *SDREC $\subseteq \mathcal{L}(2DRA)$.*

Proof: The class of sudoku-deterministically recognizable languages was introduced in Chapter 1. Given a domino-tiling system $T = (\Sigma, \Gamma, \Delta, \pi)$ and a picture Pin alphabet Σ our goal is to find the pre-image P_g in alphabet Γ for which holds $P = \pi(P_g)$ and $B_{1,2}(\hat{P_g}) \cup B_{1,2}(\hat{P_g}) \subseteq \Delta$. In case of sudoku-deterministic languages we do so by initializing all positions with all possible pre-image symbols and then by iteratively removing the implausible possibilities (in the same manner as sudoku-puzzle is solved).

We define the two dimensional restarting automaton $A = (\Sigma, \Gamma_A, \Theta_f, \delta, \mu)$ where input alphabet Σ is same as in T. Γ_A consists of all nonempty subsets of Γ. Θ_f is created to capture the exactly same set as Δ, which is possible due to the fact that domino-local languages are subset of local languages (Remark 1.3). Rewriting rules δ are created in order to remove one or more impossible choices for a pre-image symbol on a position (x,y). This can be done for example by taking all the possible tiles created from Γ_A and then checking if we could exclude some pre-image options on one or more positions by comparing the tile with the set Δ. In case there are more options how to "reduce" the tile we pick one randomly. B. Borchert and K. Reinhardt prove in [2] that the sudoku-deterministic process is successful regardless of which or how many possibilities are excluded in one step (which allows us to check the picture tile by tile as restarting automaton does).

q.e.d.

Theorem 3.2 *$\mathcal{L}(2DRA) \subseteq \mathcal{L}(2RA)$.*

Proof: The statement follows directly from the definitions of 2DRA and 2RA.

q.e.d.

Lemma 3.8 *$REC \subseteq \mathcal{L}(2RA)$.*

Proof: We can view this lemma from two perspectives. Probably the shortest proof is based on the closure on projection. Tiling systems are based upon local language and some projection. Local language is easily recognized by restarting automaton without any rewriting instructions and therefore any projection of such language is recognized by restarting automaton as well.

Second possible way to prove this lemma is directly from properties of restarting automaton.

Two-dimensional restarting automaton work as a reverse TS transformation. TS starts with local image and transforms it into generally non-local picture. 2RA on the other hand starts with general image and transforms it into some local picture.

Let $T = (\Sigma, \Gamma, \Theta, \pi)$ be a Tiling system. We will construct 2RA A that accepts exactly $\mathcal{L}(T)$.

Accepted local language is Θ. Rewriting instructions are generated in the following way. For every tile $t \in \Theta$ we generate tile $ttrans = \pi(t)$ and add rewriting instruction $ttrans \to t$. As π is generally not invertible, created 2RA is generally not deterministic as for single tile there could be multiple rewriting instructions and automaton has to non-deterministically choose between them. Moreover, we generate the rewriting instructions for partially inverted tiles, where for example the left part of the tile is already inverted into alphabet Γ but the right part is still in Σ. Those rules will be created by simply not projecting some parts of the original tiles.

q.e.d.

Theorem 3.3 *$SDREC \nsubseteq REC \Rightarrow REC \subset \mathcal{L}(2RA)$.*

Proof: Follows from the Lemma 3.7 and the Lemma 3.8.

q.e.d.

The fact that *$SDREC \nsubseteq REC$* was proven in [2]. Positioning of two-dimensional restarting automaton in hierarchy of two-dimensional languages that comes from results mentioned above is shown on following Figure 3.1.

Conjecture 3.2 *The class of $\mathcal{L}(2RA)$ when considered in 1-dimensional space equals to the class of regular languages.*

Idea of the proof: We can easily see that the *reg* $\subseteq$ 2RA (as 2RA can for example simulate the run of finite state automaton). On the other hand the simulation of restarting automaton by Turing machine which was used in proof of Lemma 3.5 gives us a result that two-dimensional restarting automaton is capable of recognizing only problems of linear time complexity (recall that each symbol of a picture can be rewritten at most k times, where k is the number of input and working symbols of the automaton). This combined with the result that palindromes are an example of context-free languages which cannot be recognized in better time than $O(n^2)$ ([15]) gives us 2RA $\subset$ context-free. We think that there is a way to prove that automaton created as one-dimensional version of two-dimensional restarting automaton (using for example only the horizontal dominoes instead of tiles) captures the class of regular languages yet we were unable to find such proof.

The result of this chapter leads us to the statement that languages recognized by two-dimensional restarting automata are an interesting class of languages as they have interesting closure properties, are stronger than the class of recognizable languages yet still collapse to the class of regular languages when taken into one dimension. The only inconvenient fact is that

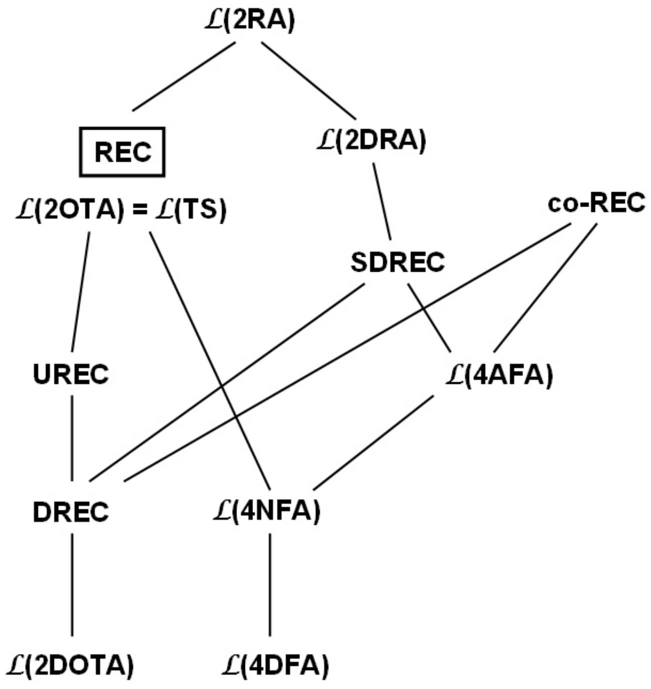

Figure 3.1: Two-dimensional restarting automata in the hierarchy of two-dimensional languages.Line between two classes means that the lower class is included in the higher. This inclusion need not to be strict. For example in the case of the classes DREC,UREC and REC the inclusions are strict, whereas in the case of 4ATA and SDREC we do not know yet.

the result $REC \subset 2RA$ heavily relies on the technical report [2] which was not published in any reliable source at the time of the publication of this thesis. The author was also unable to confirm the fact that SDREC class is outside REC from other sources. In the case this result turns out to be false we can assume that it is likely that two-dimensional restarting automata coincide with the class of recognizable languages (which would be an astonishing result as well).

Chapter 4

Usage of Two-dimensional Restarting automata

The last chapter concerns the practical usage of two-dimensional restarting automata. In previous chapter we have proved that the model we defined is successful when speaking about its properties. Here we would like to study how the model handles some basic two-dimensional objects both in a sense of capability to recognize them as well as complexity of creating such automaton for recognition. Chapter itself is divided in two sections. In the first one you can find recognition of some basic two-dimensional objects like line or cross. In the second section are presented 2RA accepting some more theoretical picture languages like palindromes, pictures with the same first and last column etc.

4.1 Basic pictures

In all following basic examples we assume two-letter input alphabet $\Sigma = \{0, 1\}$.

4.1.1 Line

Goal: Recognize picture containing only $0'$s with horizontal or vertical lines of $1'$s that do not cross or touch. We define line as a set of pixels that are next to each other all adjacent in the same direction.

Example 4.1 *Example of a line:*

#	#	#	#	#	#	#	#
#	*0*	*0*	*0*	*0*	*0*	*0*	#
#	*0*	***1***	*0*	*0*	*0*	*0*	#
#	*0*	***1***	*0*	*0*	*0*	*0*	#
#	*0*	***1***	*0*	*0*	*0*	*0*	#
#	*0*	***1***	*0*	*0*	*0*	*0*	#
#	*0*	*0*	*0*	*0*	*0*	*0*	#
#	#	#	#	#	#	#	#

Example of a non-line:

#	#	#	#	#	#	#	#
#	*0*	*0*	*0*	*0*	*0*	*0*	#
#	*0*	***1***	*0*	*0*	*0*	*0*	#
#	*0*	***1***	*0*	*0*	*0*	*0*	#
#	*0*	***1***	*0*	*0*	*0*	*0*	#
#	*0*	***1***	***1***	***1***	***1***	*0*	#
#	*0*	*0*	*0*	*0*	*0*	*0*	#
#	#	#	#	#	#	#	#

As picture language $\mathcal{L}$(LINES) is *local*, recognition is quite easy. Set of rewriting instruction is empty and set of accepting tiles corresponds to tiles of $\mathcal{L}$(LINES) which are:

$$\begin{array}{|c|c|}\hline \# & \# \\ \hline \# & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & 1 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline 0 & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline 1 & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & \# \\ \hline \# & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline 1 & \# \\ \hline \# & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & 0 \\ \hline \# & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & 1 \\ \hline \# & \# \\ \hline\end{array},$$

$$\begin{array}{|c|c|}\hline \# & 0 \\ \hline \# & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & 1 \\ \hline \# & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & 0 \\ \hline \# & 1 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & 1 \\ \hline \# & 1 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & \# \\ \hline 0 & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline 1 & \# \\ \hline 0 & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline 1 & \# \\ \hline 1 & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & \# \\ \hline 1 & \# \\ \hline\end{array},$$

$$\begin{array}{|c|c|}\hline \# & \# \\ \hline 0 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline 1 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline 1 & 1 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline 0 & 1 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & 0 \\ \hline \# & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline 1 & 0 \\ \hline \# & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline 1 & 1 \\ \hline \# & \# \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & 1 \\ \hline \# & \# \\ \hline\end{array},$$

$$\begin{array}{|c|c|}\hline 0 & 0 \\ \hline 0 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 0 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & 0 \\ \hline 0 & 1 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & 0 \\ \hline 1 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 0 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & 1 \\ \hline 0 & 1 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0 & 0 \\ \hline 1 & 1 \\ \hline\end{array},$$

$$\begin{array}{|c|c|}\hline 1 & 0 \\ \hline 1 & 0 \\ \hline\end{array}$$

Please note, that both the languages $\mathcal{L}$(H_LINES) (containing only the pictures with horizontal lines) and $\mathcal{L}$(V_LINES) (containing only the pictures with vertical lines) are *local* as well.

4.1.2 Multiple objects detection

A problem of different complexity would be to recognize picture containing SINGLE line (vertical or horizontal) in general position. Such language is not local and therefore rewriting instructions would be needed (to "count" the occurrences of lines in the image). Our goal would be to somehow "pass the information" about found line in some row/column. Because the problem is general (count number of occurrences of object XY) we show it here as a complete technique.

Let us assume w.l.o.g that we are trying to recognize language H_LINES1 containing only single horizontal line. Our goal is not to count the number of lines, but to simply check whether there is only single one. Therefore we can send information about the first line we find to whole image and mark every other line there.

Our automaton will have three-symbol working alphabet $\Gamma = \{0_1, 0_2, 0_3\}$ and μ is defined in a way allowing rewritings $0 \to 0_1 \to 0_2 \to 0_3 \to 1$. Rewriting instructions are defined to rewrite every 0 to 0_1 from left to right and from top to bottom until the first 1 is found. Let us assume the following situation:

#	#	#	#	#	#	#	#
#	0	0	0	0	0	0	#
#	0	0	0	0	0	0	#
#	0	0	0	0	0	0	#
#	0	0	**1**	**1**	**1**	0	#
#	0	0	0	0	0	0	#
#	0	0	0	0	0	0	#
#	0	0	0	0	0	0	#
#	0	0	0	0	0	0	#
#	#	#	#	#	#	#	#

Our rewriting instructions are defined in a following way:

$$\begin{array}{|c|c|}\hline \# & \# \\ \hline \# & 0 \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & 0_1 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \# & \# \\ \hline 0_1 & 0 \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \# & \# \\ \hline 0_1 & 0_1 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \# & 0_1 \\ \hline \# & 0 \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \# & 0_1 \\ \hline \# & 0_1 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0_1 & 0 \\ \hline\end{array} \to$$

$$\begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0_1 & 0_1 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 0 \\ \hline\end{array} \to \begin{array}{|c|c|}\hline 1 & 0_1 \\ \hline 0 & 0 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline 0_1 & 0 \\ \hline 0 & 0 \\ \hline\end{array} \to \begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0 & 0 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0 & 0 \\ \hline\end{array} \to \begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0_1 & 0 \\ \hline\end{array},$$

$$\begin{array}{|c|c|}\hline 0_1 & \# \\ \hline 0 & \# \\ \hline\end{array} \to \begin{array}{|c|c|}\hline 0_1 & \# \\ \hline 0_1 & \# \\ \hline\end{array}$$

In our situation, this set leads to following picture which is local and therefore acceptable by 2RA.

#	#	#	#	#	#	#	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_1	0_1	**1**	**1**	**1**	0_1	#
#	0_1	0_1	0	0	0	0_1	#
#	0_1	0_1	0	0	0	0_1	#
#	0_1	0_1	0	0	0	0_1	#
#	0_1	0_1	0	0	0	0_1	#
#	#	#	#	#	#	#	#

We now add rewriting instructions to "inform" rest of the picture of the occurrence of the first line. We do not have to deal with the lines below the first found, as the 0′s there would never again be rewritten to any 0_x symbol and therefore no line lying there would be accepted. The added set of instructions is:

$$\begin{array}{|c|c|}\hline 0_1 & 1 \\ \hline 0 & 0 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0_2 & 1 \\ \hline 0 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0_1 & 0_2 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0_2 & 0_2 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0_2 & 0_2 \\ \hline 0 & 0 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0_2 & 0_2 \\ \hline 0_2 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0_2 & 0_2 \\ \hline 0_2 & 0 \\ \hline\end{array}$$

$$\rightarrow \begin{array}{|c|c|}\hline 0_2 & 0_2 \\ \hline 0_2 & 0_2 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline 0_1 & 1 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline 0_2 & 1 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & \# \\ \hline 0_1 & 0_2 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline 0_2 & 0_2 \\ \hline\end{array}, \begin{array}{|c|c|}\hline \# & 0_2 \\ \hline \# & 0 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & 0_2 \\ \hline \# & 0_2 \\ \hline\end{array},$$

$$\begin{array}{|c|c|}\hline 1 & 0_1 \\ \hline 0 & 0 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 1 & 0_2 \\ \hline 0 & 0 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0_2 & 0_1 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0_1 & 0_1 \\ \hline 0_2 & 0_2 \\ \hline\end{array}, \begin{array}{|c|c|}\hline 0_2 & \# \\ \hline 0 & \# \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0_2 & \# \\ \hline 0_2 & \# \\ \hline\end{array}$$

the result then looks like:

#	#	#	#	#	#	#	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_2	0_2	**1**	**1**	**1**	0_2	#
#	0_2	0_2	0	0	0	0_2	#
#	0_2	0_2	0	0	0	0_2	#
#	0_2	0_2	0	0	0	0_2	#
#	0_2	0_2	0	0	0	0_2	#
#	#	#	#	#	#	#	#

which can be accepted by set of tiles that do not contain the tiles with unindexed 0′s above 1′s or 0_2′s above 1′s. In case there would be two lines present, result would be:

#	#	#	#	#	#	#	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_1	0_1	0_1	0_1	0_1	0_1	#
#	0_2	0_2	**1**	**1**	**1**	0_2	#
#	**1**	**1**	0	0	0	0_2	#
#	0	0	0	0	0	0_2	#
#	0	0	0	0	0	0_2	#
#	0	0	0	0	0	0_2	#
#	#	#	#	#	#	#	#

this picture will not be accepted as we would not allow $\begin{array}{|c|c|}\hline 0_2 & * \\ \hline 1 & * \\ \hline\end{array}$ tiles in acceptance set ($*$ stands for any symbol).

Careful reader probably noticed, that we still have one unused symbol in our working alphabet and that the solution mentioned above would not work for situation with two lines in one row (we would not be able to get the 0_2's "above" the second line found.

For this case we add/change the following instructions:

$$\begin{array}{|c|c|}\hline 1 & 0_1 \\ \hline 0 & 0 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 1 & 0_3 \\ \hline 0 & 0 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline 0_3 & 0_1 \\ \hline 0 & 0 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0_3 & 0_3 \\ \hline 0 & 0 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline 0_3 & \# \\ \hline 0 & \# \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0_3 & \# \\ \hline 0_2 & \# \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline 0_3 & 0_3 \\ \hline 0 & 0 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0_3 & 0_3 \\ \hline 0_2 & 0 \\ \hline\end{array}$$

and we refuse to accept any picture containing tile $\begin{array}{|c|c|}\hline 0_3 & 1 \\ \hline * & * \\ \hline\end{array}$. It is easy to see that all pictures with two (or more) lines fall into one of the following categories:

- Lines are below each other – we do not accept the picture because 0's above the second line are not rewritten to 0_1's.
- Lines are in the same row – we do not accept the picture because the second line contains 0_3 symbol on its left end.
- Lines in any other position – we do not accept the picture because the second line found will have 0_2's above itself.

Note: Please note, that this approach can be extended to count any fixed number of occurrences of lines (or generally any other symbol).

4.1.3 Tree

Goal: Recognize the picture containing series of curves forming a tree (i.e. curves that potentially cross, but does not form a closed curve). Furthermore we would like to recognize single tree (not forest). With closed curve we understand series of adjacent pixels that are connected either side by side or diagonally, where choosing a moving direction one can come a full circle.

Example of tree:

#	#	#	#	#	#	#	#
#	0	0	0	0	0	0	#
#	0	0	**1**	0	0	0	#
#	0	**1**	0	**1**	0	0	#
#	**1**	**1**	0	0	0	0	#
#	0	**1**	0	**1**	**1**	0	#
#	0	**1**	**1**	0	0	**1**	#
#	0	0	0	**1**	0	0	#
#	0	0	0	**1**	0	0	#
#	#	#	#	#	#	#	#

Example of not a tree:

#	#	#	#	#	#	#	#
#	0	0	0	0	0	0	#
#	0	0	**1**	0	0	0	#
#	0	**1**	0	**1**	0	0	#
#	**1**	**1**	0	0	**1**	0	#
#	0	**1**	0	**1**	**1**	0	#
#	0	**1**	**1**	0	0	**1**	#
#	0	0	0	**1**	0	0	#
#	0	0	0	**1**	0	0	#
#	#	#	#	#	#	#	#

We split the recognition process in the two separate tasks. First is the closed curve detection and second the single tree detection. Rather then trying to detect the closed curve by marking the 1′s in the image we "flood" the 0′s in the picture with special index (let it be 0_1). If there is a closed curve present than there will be some 0′s left unmarked. Detecting whether we have only single tree in the picture can be done with slightly modified approach from Subsection 4.1.2. We will index the first 1 we find with index $_1$ and all others with index $_2$. Moreover we allow rewriting 1_2 to 1_1 when those two indices meet. In this way if there is more then one tree present in the picture it will be completely marked with 1_2.

More specifically automaton recognizing single tree in the picture is defined as follows.

- $\Sigma = \{0, 1\}$
- $\Gamma = \{0_1, 0_2, 0_3, 1_1, 1_2\}$
- $\mu = 0 \rightarrow 3; 0_1 \rightarrow 2; 0_2 \rightarrow 1; 0_3 \rightarrow 1; 1 \rightarrow 3; 1_2 \rightarrow 2; 1_1 \rightarrow 1$
- δ = instructions are formed in a way to flood the $0'$s with index $_1$ horizontally and vertically from all sides (diagonal flooding is forbidden). Symbols 0_2 and 0_1 are used in the first row we find 1 - otherwise we would not be able to detect two (or more) $1'$s in single row. Concerning the $1'$s, first 1 found in a picture is marked with index $_1$ all others are marked with index $_2$. We then allow to rewrite the symbols 1_2 to 1_1 in case they appear on adjacent pixels (and therefore in the same tree).
- Θ_f = accepting set is defined to accept only the picture where all $0'$s are marked with some index (be it $_1$, $_2$ or $_3$) and the only $1'$s appearing there are marked with the index 1_1.

Rejected picture with one tree structure and one structure with closed curve then looks like (you can notice one 0 that was not indexed and structure of $1'$s that has index $_2$) :

#	#	#	#	#	#	#	#	#	#
#	0_2	$\mathbf{1_1}$	0_2	0_2	0_2	$\mathbf{1_2}$	0_3	0_3	#
#	0_2	$\mathbf{1_1}$	$\mathbf{1_1}$	0_2	$\mathbf{1_2}$	0	$\mathbf{1_2}$	0_2	#
#	0_2	$\mathbf{1_1}$	0_2	0_2	0_2	$\mathbf{1_2}$	$\mathbf{1_2}$	0_2	#
#	$\mathbf{1_1}$	0_2	0_2	0_2	0_2	0_2	0_2	0_2	#
#	0_2	$\mathbf{1_1}$	0_2	$\mathbf{1_1}$	$\mathbf{1_1}$	0_2	0_2	0_2	#
#	0_2	$\mathbf{1_1}$	$\mathbf{1_1}$	0_2	0_2	$\mathbf{1_1}$	0_2	0_2	#
#	0_2	0_2	0_2	$\mathbf{1_1}$	0_2	0_2	0_2	0_2	#
#	0_2	0_2	0_2	$\mathbf{1_1}$	0_2	0_2	0_2	0_2	#
#	#	#	#	#	#	#	#	#	#

4.1.4 Cross

Goal: Recognize the picture containing only $0'$s with cross made of $1'$s. We define cross as two lines of the same length (one horizontal and one vertical) that crosses in the middle. Note that the lines must have odd length.

#	#	#	#	#	#	#	#	#
#	0	0	0	**1**	0	0	0	#
#	0	0	0	**1**	0	0	0	#
#	0	0	0	**1**	0	0	0	#
#	**1**	**1**	**1**	**1**	**1**	**1**	**1**	#
#	0	0	0	**1**	0	0	0	#
#	0	0	0	**1**	0	0	0	#
#	0	0	0	**1**	0	0	0	#
#	0	0	0	0	0	0	0	#
#	#	#	#	#	#	#	0	#

We assume that the picture contains two lines, one horizontal one vertical which crosses somewhere (we can check this easily on accepting set). Our approach is then to check if all arms of the cross are of equal length. We do so in a manner where we mark all 1′s alternatively 1_1 and 1_2 in a way that if there is arm of the cross which is longer then the rest, then there will be 1′s which will not be marked. On the other hand if there are arms of the cross which are shorter then the others, then there will be 0′s marked as 0_1 or 0_2 outside the cross area.

In successful case, the picture will look like:

#	#	#	#	#	#	#	#	#
#	0	0	0	$\mathbf{1_1}$	0	0	0	#
#	0	0	0_1	$\mathbf{1_2}$	0_1	0	0	#
#	0	0_1	0_2	$\mathbf{1_1}$	0_2	0_1	0	#
#	$\mathbf{1_1}$	$\mathbf{1_2}$	$\mathbf{1_1}$	$\mathbf{1_2}$	$\mathbf{1_1}$	$\mathbf{1_2}$	$\mathbf{1_1}$	#
#	0	0_1	0_2	$\mathbf{1_1}$	0_2	0_1	0	#
#	0	0	0_1	$\mathbf{1_2}$	0_1	0	0	#
#	0	0	0	$\mathbf{1_1}$	0	0	0	#
#	0	0	0	0	0	0	0	#
#	#	#	#	#	#	#	#/	#

We do not write the whole instruction set, but you can probably imagine how it would look like. Its also easy to see that the final picture is local. We did not solve the case when there are more crosses in the picture, but as we already know the technique it would be easy to adapt it to this situation.

4.2 Basic theoretical languages

This section deals with recognition of more theoretical languages usually used in either examples or proofs that certain model is unable to recognize such language. We will deal with following languages:

- The first and the last column are the same
- The first and another column are the same
- Square picture with the same first row and column
- Any two columns are the same
- Palindromes

In all cases we assume rich input alphabet without further specification.

4.2.1 The first and the last column are the same

Goal: Recognize picture where first and last column are same.

We create working alphabet of the size $\Sigma \times \Sigma$ where every symbol will have added an index of another symbol so in case of $\Sigma = \{a, b\}$ we get $\Gamma = \{a_a, a_b, b_a, b_b\}$. We index each pixel of the first column with same index as symbol present on that pixel. Then we pass those indices from left to right on the last column.

The accepting set is then define to accept only if last column contains only pixels where the symbol and its index match. All other pixels can contain all possible combinations.

With this approach we did not destroy any information from the input picture. In case we do not care for this it would be satisfying to have working alphabet of exact same size as input alphabet and simply rewrite every row with a symbol found in the first column. Only the symbol in the last column would get rewritten if it matches the "send" symbol from the first column.

The only check we do with acceptance set is whether the symbols in the last column got rewritten.

4.2.2 The first and another column are the same

Goal: Check whether there is another column of the picture matching the first one.

This is slightly more complicated situation as the one before. We can still use the "indexing" trick to find the matching column, but that would not be enough. The problem is, that the accepting set cannot capture situation where "some" column is formed out of some special symbol (especially when all other are general). Therefore we will need to mark some special place (a corner for example) where we can easily check whether it contains some symbol or not.

We will use the automaton reading direction to our advantage and modify the solution from previous example in following way.

We send the "indexing wave" throughout the first row and mark all pixels that match the index with their symbol (we can for example rewrite them to some special symbol). In next row we send the "indexing wave" again but this time mark only the pixels that match the index and have the special symbol in pixel above them (i.e. we are selecting the columns that appeared to be the same as first one but differs on the present row). If we reach the last row with special symbol (somewhere) we send it (in a same way as we do in case of indices) to the right corner.

The accepting set then requires the right corner to contain the special symbol and the rest of the picture may be general.

4.2.3 Picture with the same first row and column

Goal: Check whether in the input square picture, first row matches its first column.

This task is easy as it is only combination of the previously used "indexing waves" and approach we used in case of measuring the arms of the cross. More specifically we send the indexing waves diagonally from the first row to the first column and check whether the in the first column all symbols matches their index.

4.2.4 Any two columns are the same

Goal: Check whether any two columns in the input picture match.

This is an example of a picture two-dimensional restarting automaton is probably unable to recognize. The problem is, that we need to check n columns each to each and to pass some information between the columns the only tool we have are the rewritings. But each pixel can be rewritten at most $|\Gamma|$ times therefore when we have fixed automaton (with fixed working alphabet) the number of rewritings needed for each pixel (to pass the information about surrounding columns) will necessary overgrow the size of the working alphabet.

4.2.5 Palindromes :

Goal: Recognize two-dimensional palindrome (i.e. picture $P \oplus P^{(v\_mirr)}$). We can define vertical palindromes (containing languages $P \oplus P^{(v\_mirr)}$) or horizontal palindromes (containing languages $P \ominus P^{(h\_mirr)}$) but since 2RA is closed under rotation it is not important which case we choose.

Let us first look on the vertical palindromes which resemble the string palindromes and on the picture of size $1 \times 2n$ are even equal.

Our first observation is that the picture language of all the palindromes $\mathcal{L}(PALI)$ is not *local*, therefore we will need some rewriting instructions (there is no local strategy of comparing the symbols on the opposite sides of the picture).

Assume we have the language $\mathcal{L}(PALI\_N)$ of all palindromes with pictures of the size $k \times 2N$ (i.e. palindromes of fixed width).

We can the proof the following lemma:

Lemma 4.1 *$\mathcal{L}(PALI\_N)$ is recognizable by 2DRA.*

Proof: We will verify the palindrome row by row from up to down. W.l.o.g. assume that we have palindrome of size $1 \times 2N$. We verify that the most left symbol matches the most right by sending a "wave" of rewritings from the left to the right, rewriting every symbol with its own representation with the index representing the symbol send through the "wave". Let us have an example:

$\Sigma = \{A, B, C\}, \Gamma = \{A_A, B_A, C_A, B_A, B_B, B_C, C_A, C_B, C_C\}$

Input image A =

#	#	#	#	#	#	#	#
#	A	C	B	B	C	A	#
#	#	#	#	#	#	#	#

In this case we send the wave of A's from the left to the right. Our rewriting rules would look like:

$$\begin{array}{|c|c|}\hline \# & \# \\ \hline \# & \mathrm{A} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline \# & A_A \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \# & \# \\ \hline A_A & \mathrm{C} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline A_A & C_A \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \# & \# \\ \hline C_A & \mathrm{B} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline C_A & B_A \\ \hline\end{array}$$

... and so on for all the possible combinations. After we reach the end of the line we verify whether the input symbol matches the one send through the wave. That is easily done by the rewriting rule

$$\begin{array}{|c|c|}\hline \# & \# \\ \hline C_C & \# \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \# \\ \hline \mathrm{V} & \# \\ \hline\end{array}$$

where "V" is a special symbol standing for "verified". We now send a "wave" back from the right to the left with information about successful verification. The first letter on the left is then rewritten with the "V" symbol and verification follows with the next pair.

We will need working alphabet of size $|\Sigma|^2 * 2 * N + 1$ where N stands for half the width of the palindrome. That is, because for each pair of the letters we verify we are sending two "waves" - that is two rewritings of all the letters between them. As we need to verify N pairs we are going to send $2*N$ waves. Moreover we cannot rewrite the symbols from one to another and then back, because of the μ function. Therefore we need $2*N$ of representations of all the possibilities of (original letter on the tape)*(letter passed in the wave). The last "+1" is for the special "V" letter.

It is easy to see that in successful case whole picture consists of "V" letter only (which is local language).

q.e.d.

Careful reader probably noticed, that our strategy works only because we know in advance the "size" of the palindrome and can prepare the working alphabet. For any algorithm imaginable we need to make N verifications of symbol "pairs". We certainly cannot make those verifications in the verification phase of 2DRA computing, because those verifications are not local – they are even very much everything else then local. The only option how to help ourselves is to make some kind of the rewriting rules that would allow us to finally verify the palindrome. But because the rewriting rules are limited by the μ function we can perform only limited number of rewritings on each position of the tape (specifically at most $|\Sigma| + |\Gamma|$) and this number is fixed for a fixed automaton. We feel that the number of rewritings needed on one field to pass the information about the palindrome is growing with the size of the palindrome and therefore would eventually overcome the possibilities of our automaton. Following lemma proves this hypothesis.

Lemma 4.2 *Language of palindromes of any size $\mathcal{L}$(PALI) cannot be recognized by 2RA.*

Proof: The proof is based on the result from [15, page 41-47] where Průša proves that the language of palindromes in two-dimensions cannot be recognized in better time that $O(\frac{n^2}{\log n})$. Průša is speaking about the one-dimensional palindromes recognized by the two-dimensional Turing machine, therefore n is the size of the one-dimensional palindrome in this case. Therefore we w.l.o.g. assume that we are working with the palindromes of size $1 \times n$.

The two-dimensional restarting automaton has, due to its limitation of rewriting on the tape, an upper bound for the number of "actions" it can make. We counted them in proof of Lemma 3.5 and the result was that it is less then $10 * |\Gamma + \Sigma| + 3$. This means that even the slowest algorithm working with two-dimensional restarting automaton is linear in time (as the number of actions for each position is fixed). If 2RA could recognize $\mathcal{L}$(PALI) it would be in direct contradiction with the result from Průša's PhD thesis.

4.3 Recognition with no working alphabet

In all of our examples we used working alphabet to recognize the picture. Working alphabet gives our automaton quite a strength – it allows to count occurrences, to measure whether two lines are of the "same length". It is natural to ask, what would not use any working alphabet at all. Right at the first glimpse we realize that the automaton instantly becomes weaker (for example it is unable to recognize the cross in two-letter alphabet or a square of in one-letter alphabet).

With two-letter alphabet we can easily show how weak the automaton becomes. The only rewrites automaton can perform are from one letter to another (w.l.o.g let us assume it to be 1 to 0). But at the moment of the rewriting, automaton restarts and therefore looses the information about previous state of the picture and pictures are rewritten from one possible input to another possible input. Therefore all of our rewritings MUST be error preserving, because if on is not, we are unable to decide whether our image is results of some rewriting or it is an input image and therefore potentially reject image which should be accepted (and vice-versa).

It is clear that we are unable to decide neither "single line" nor "cross" problem as we are unable to count the occurrences nor "measure" the equality of cross arms.

What problems are we then able to decide ? Local languages for example, because for those we do not need rewritings at all. In this thesis we did not study the recognition with no working alphabet any further but it would be interesting to find which class of languages is 2RA capable to recognize without the working alphabet.

Conclusion

We have successfully transformed the restarting automaton concept from one dimension into two dimensions. The resulting class of languages recognized by the two-dimensional restarting automaton proved to be closed under row and column concatenations, union, intersection, rotation, projection and both horizontal and vertical mirroring. Moreover we conjecture that despite the fact that the two-dimensional restarting automaton is likely stronger than the tiling systems, when restricted to one dimension it captures exactly class of regular languages. All those properties qualify the class of languages accepted by two-dimensional restarting automaton as an interesting member of two-dimensional languages hierarchy.

As a first work in this field we can say that this thesis was successful. Yet there is still much work that needs to be done both in exploring the power of two-dimensional restarting automaton itself as well as exploring the whole two-dimensional language hierarchy.

We would like to point out several questions which remain opened:

- Is the $\mathcal{L}(2RA)$ really not closed under complement ?
- Does the $\mathcal{L}(2RA)$ collapse to the class of regular languages when restricted to one dimension ?
- What are the properties of two-dimensional deterministic restarting automata ?
- Is the inclusion $\mathcal{L}(2DRA) \subseteq \mathcal{L}(2RA)$ strict ?
- Does the class $\mathcal{L}(2RA)$ contain class co-REC ?
- Are we able to confirm the strict inclusion $REC \subset \mathcal{L}(2RA)$?

Bibliography

[1] M. Blum and C. Hewitt. Automata on a two-dimensional tape. *IEEE Symposium on Switching and Automata Theory*, pages 155–160, 1967.

[2] B. Borchert and K. Reinhardt. Deterministically and sudoku-deterministically recognizable picture languages. 2006. http://www-fs.informatik.uni-tuebingen.de/ reinhard/depila.pdf.

[3] S. Eilenberg. *Automata, Languages, and Machines*. Academic Press, Inc., Orlando, FL, USA, 1974.

[4] D. Giammarresi and A.Restivo. *Two dimensional languages*, volume 3. Springer-Verlag, Berlin, handbook of formal languages edition, 1997.

[5] D. Giammarresi and A. Restivo. Recognizable picture languages. *International Journal Pattern Recognition and Artificial Intelligence*, pages 31–46, 241–256, 1992.

[6] K. Inoue and A.Nakamura. Some properties of two-dimensional on-line tesselation acceptors. *Information Sciences*, 6:95–121, 1977.

[7] K. Inoue and I. Takanami. A survey of two-dimensional automata theory. *Proc. 5th Int. Meeting of Young Computer Scientists*, pages 72–91, 1990.

[8] C. Moore J. Kari. New results on alternating and non-deterministic two-dimensional finite-state automata. *STACS*, pages 396–406, 2001.

[9] Petr Jančar, František Mráz, Martin Plátek, and Jörg Vogel. Restarting automata. In *Fundamentals of Computation Theory*, volume Volume 965/1995, pages 283–292, London, UK, 1995. Springer-Verlag.

[10] Petr Jančar, František Mráz, Martin Plátek, and Jörg Vogel. On monotonic automata with a restart operation. *Journal of Automata, Languages and Combinatorics*, 4(4):287–311, 1999.

[11] Tomasz Jurdziński and Friedrich Otto. Shrinking restarting automata. In *Mathematical Foundations of Computer Science 2005*, volume 3618/2005 of *Lecture Notes in Computer Science*, pages 532–543. Springer, Berlin, 2005.

[12] I. Takanami K. Inoue and A. Nakamura. A note on two-dimensional finite automata. *Information Processing Letters*, 7(1):49–52, 1978.

[13] D. Giammarresi M. Anselmo and M. Madonia. From determinism to non-determinism in recognizable two-dimensional languages. *Lecture Notes in Computer Science*, pages 36–47, 2007.

[14] Friedrich Otto. Restarting automata. In Zoltán Esik, Carlos Martin-Vide, and Victor Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 269–303. Springer, Berlin, 2006.

[15] D. Průša. Two dimensional languages. Phd thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2004.