January 2013

Modeling State Transitions with Automata

Egor Dolzhenko

University of South Florida, edolzhen@mail.usf.edu

Follow this and additional works at: http://scholarcommons.usf.edu/etd

Part of the Computer Sciences Commons, and the Mathematics Commons

Scholar Commons Citation


This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Modeling State Transitions with Automata

by

Egor Dolzhenko

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
Department of Mathematics and Statistics
College of Arts and Sciences
University of South Florida

Major Professor: Natasha Jonoska, Ph.D.
Brian Curtin, Ph.D.
Gregory McColm, Ph.D.
Brendan Nagle, Ph.D.
Masahiko Saito, Ph.D.

Date of Approval:
April 3, 2013

Keywords: Cellular Automata, Mandatory Results Automata, Symbolic Dynamics, Transducers, Two-Dimensional Languages

Copyright © 2013, Egor Dolzhenko
Dedication

To my family.
Acknowledgments

I would like to thank my advisor Natasha Jonoska for her help and guidance. I am also grateful to Jay Ligatti for introducing me to software security. I would like to thank my committee members, Brian Curtin, Gregory McColm, Brendan Nagle, Masahico Saito for their invaluable advice and help with making important career decisions.

Part of the work presented in this dissertation was done during the author’s visit to the Landweber Lab at Princeton University. I am grateful to all of the members of the lab, including Laura Landweber, Aaron Goldman, Leslie Beh, Kelsi Lindblad, John Bracht, and Xiao Chen. My stay at the Landweber Lab was funded by the grant DMS-0900671.

Also, I’d like to acknowledge the help of our math biology research group, including Ryan Arredondo, Jonathan Burns, Sarah Croome, Daria Karpenko, Maja Milosevic, Jamie Sprecher, Timothy Yeatman.

Special thanks to Pasha, Diana, Romain, Jonathan, Dimitris!

My graduate work has been supported by the NSF grants DMS-0900671 and CCF-0726396.
# Table of Contents

List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii

Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iv

Chapter 1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.1 Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    1.1.1 String Languages . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    1.1.2 Finite State Automata . . . . . . . . . . . . . . . . . . . . . . . . 8
    1.1.3 Shift Spaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

Chapter 2 Picture Languages . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.1 Definition and Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.2 Factorial-local Two-dimensional Languages . . . . . . . . . . . . . . . . . 13
  2.3 Other Classes of Picture Languages . . . . . . . . . . . . . . . . . . . . . 19

Chapter 3 Cellular Automata and Picture Languages . . . . . . . . . . . . . . . . 24
  3.1 Definition and some examples . . . . . . . . . . . . . . . . . . . . . . . . . 24
  3.2 Traces of Cellular Automata . . . . . . . . . . . . . . . . . . . . . . . . . 28
  3.3 CA-generated Two-dimensional Languages . . . . . . . . . . . . . . . . . . 29
  3.4 Factorial-local Cellular Automata . . . . . . . . . . . . . . . . . . . . . . 36
  3.5 Transducers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
  3.6 Transducer-Generated Picture Languages . . . . . . . . . . . . . . . . . . . 47

Chapter 4 Modeling Runtime Enforcement with Mandatory Results Automata . . 53
  4.1 Mandatory Results Automata . . . . . . . . . . . . . . . . . . . . . . . . . 55
    4.1.1 Example MRAs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
    4.1.2 Generalizing the Operational Semantics . . . . . . . . . . . . . . . . 60
  4.2 MRA-based Enforcement . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
    4.2.1 Policies and Properties . . . . . . . . . . . . . . . . . . . . . . . . . 61
    4.2.2 Enforcement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
  4.3 Analysis of MRA-enforceable Policies . . . . . . . . . . . . . . . . . . . . . 64
  4.4 Nondeterministic MRAs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
  4.5 A Hierarchy of Reasonable Properties . . . . . . . . . . . . . . . . . . . . . 73

Chapter 5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
  5.1 Open Questions and Future Work . . . . . . . . . . . . . . . . . . . . . . . 81

References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A finite-state automaton modeling an on/off switch</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>An example of a finite state automaton</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Transition of a right-check automaton</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>The right-check automaton for Example 2</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Recognition of blocks by right- and left-check automata</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>A space-time diagram of the shift cellular automaton $\sigma$</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>An $\delta$-chain (left) and a sequence $\epsilon$-shadowed by $\alpha$ (right)</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>Obtaining the block $B$ as a sub-block of the evolution of $\alpha$</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>The left-check automaton for the language defined by rule 192 CA</td>
<td>34</td>
</tr>
<tr>
<td>10</td>
<td>The right-check automaton for the language defined by rule 192 CA</td>
<td>35</td>
</tr>
<tr>
<td>11</td>
<td>Blocks used in the proof of Lemma 3.4</td>
<td>38</td>
</tr>
<tr>
<td>12</td>
<td>Blocks used in the proof of Proposition 3.3</td>
<td>40</td>
</tr>
<tr>
<td>13</td>
<td>The graph of the sofic shift which is the trace of rule 123</td>
<td>43</td>
</tr>
<tr>
<td>14</td>
<td>An example of a finite state transducer</td>
<td>48</td>
</tr>
<tr>
<td>15</td>
<td>An untrusted application and an executing system</td>
<td>57</td>
</tr>
<tr>
<td>16</td>
<td>Single-step operational semantics of mandatory results automata</td>
<td>58</td>
</tr>
<tr>
<td>17</td>
<td>The order in which Figure 16’s rules may be applied</td>
<td>64</td>
</tr>
<tr>
<td>18</td>
<td>A graph whose paths correspond to executions in $U$</td>
<td>65</td>
</tr>
<tr>
<td>19</td>
<td>A hierarchy of reasonable security properties</td>
<td>75</td>
</tr>
<tr>
<td>20</td>
<td>A hierarchy of cellular automata and a hierarchy of picture languages</td>
<td>79</td>
</tr>
<tr>
<td>21</td>
<td>An infographic describing a model to study horizontal gene transfer</td>
<td>83</td>
</tr>
</tbody>
</table>
Abstract

Models based on various types of automata are ubiquitous in modern science. These models allow reasoning about deep theoretical questions and provide a basis for the development of efficient algorithms to solve related computational problems. This work discusses several types of automata used in such models, including cellular automata and mandatory results automata.

The first part of this work is dedicated to cellular automata. These automata form an important class of discrete dynamical systems widely used to model physical, biological, and chemical processes. Here we discuss a way to study the dynamics of one-dimensional cellular automata through the theory of two-dimensional picture languages. The connection between cellular automata and picture languages stems from the fact that the set of all space-time diagrams of a cellular automaton defines a picture language. We will discuss a hierarchy of cellular automata based on the complexity of the picture languages that they define. In addition to this, we present a characterization of cellular automata that can be described by finite-state transducers.

The second part of this work presents a theory of runtime enforcement based on mechanism models called Mandatory Results Automata (MRAs). MRAs can monitor and transform security-relevant actions and their results. Because previous work could not model general security monitors transforming results, MRAs capture realistic behaviors outside the scope of previous models. MRAs also have a simple but realistic operational semantics that makes it straightforward to define concrete MRAs. Moreover, the definitions of policies and enforcement with MRAs are significantly simpler and more expressive than those of previous models. Putting all these features together, we argue that MRAs make
good general models of (synchronous) runtime mechanisms, upon which a theory of run-
time enforcement can be based. We develop some enforceability theory by characterizing
the policies deterministic and nondeterministic MRAs enforce.

This discussion represents a selection of our work in the general area of discrete mod-
eling which has appeared in [13, 14, 16, 25–28, 36]. Here we focus on [26, 28].
Chapter 1
Introduction

Much research in engineering and natural sciences concerns systems that change over time. Such systems are often modeled by a set of states, describing the system at various time points, and a function (or a relation) that governs the transitions of the system from one state to another. Automata form a broad class of abstract machines designed to formalize the above approach to modeling systems that change over time.

Perhaps the most widely used type of automata are the Finite State Automata (FSAs, formally defined in Subsection 1.1.2). An FSA consists of a finite set of states and a transition function that describes how various events (input symbols) change the state of the system modeled by the automaton. A simple example of an FSA describing a system consisting of an on/off switch and a unique click event is depicted in Figure 1. This system starts out in the on state (the starting state is indicated by the arrow pointing to it) and transitions between the on and off states with each click event. Sometimes it is convenient to consider a variant of finite state automata capable of outputting events (output symbols). These types of automata are called Finite-State Transducers (FSTs, discussed in Section 3.5).

Cellular Automata (CAs, studied in Chapter 2) is another type of automata which originated from Stanislaw Ulam’s and John von Neumann’s work in 1940s [15] that have since been a focus of much pure and applied research. Cellular automata are actively used for modeling various systems, in particular they are used in (a) physics to study Brownian motion [43, 45] and fluid dynamics [34, 53], (b) biology for predicting subcellular locations of proteins [61] and population dynamics [5], (c) engineering for traffic-flow simulations [6, 31, 54] and image processing [29, 55, 56], and they have many other applications in these
Figure 1.: A finite state automaton modeling an on/off switch. This automaton consists of the on and off states and changes its current state with each click event.

areas and beyond.

Cellular automata comprise an important and well-studied class of discrete dynamical systems [21]. CAs are functions that operate on bi-infinite sequences, called configurations, which play the role of states (in the sense described above). If $G$ is a cellular automaton and $\alpha$ is a bi-infinite configuration representing the current state of a system, then the configuration $G(\alpha)$ represents the next state of the system. One of the key features of cellular automata is that they can be completely specified by straightforward local rules and, at the same time, are capable of performing complex computations (indeed, some cellular automata can simulate arbitrary Turing machines [18]). Because cellular automata are easy to define, they are being actively used in computer simulations.

CAs are functions that map bi-infinite configurations of symbols to bi-infinite configurations of symbols, and so each CA $G$ and each configuration $\alpha$ gives rise to a sequence $\alpha, G(\alpha), G^2(\alpha), \ldots$. Stacking these configurations underneath one another, starting with $\alpha$, results in a half-plane array of symbols. We study the dynamics of cellular automata that give rise to arrays of symbols that can be described by a finite set of fixed-size blocks. We call such automata factorial-local cellular automata to underline their connection to the class of factorial-local picture languages. Computer calculations show that over half of the elementary CAs are factorial-local. We describe a characterization of the dynamics of these factorial-local CAs using the theory of shift spaces from symbolic dynamics. We
show that factorial-local CAs have the same characterization as one-sided cellular automata with SFT traces and discuss several other of their properties. We discuss the relationship between the hierarchy of classes of cellular automata and a corresponding hierarchy of picture languages.

The studies of cellular automata are mainly focused on understanding and predicting their long-term dynamics. Unfortunately, most of the questions regarding the dynamics are, in general, undecidable. Because of this, various classification schemes have been proposed in an attempt to distinguish automata with different complexity of dynamics. For example, Stephen Wolfram proposed a classification of cellular automata based on the dynamics of patterns observed in their space-time diagrams [60]. Other, more formal classifications followed.

One of the notions employed for understanding dynamics of one-dimensional cellular automata is called the trace. The trace was introduced by P. Kurka [42] and was further investigated by other researchers studying nilpotency of cellular automata (e.g., [17, 37]). For a cellular automaton, the trace (also called 1-trace) is the set of all infinite sequences \( \{\alpha^i_j\} (i = 1, 2, \ldots) \), where \( \alpha^i_j \) is the symbol at the \( j \)-th position in \( \alpha \) at the time-step \( i \) (i.e. \( G^i(\alpha)_j \)). Similarly, the \( k \)-traces (for \( k \geq 2 \)) are infinite sequences depicting evolution of \( k \) adjacent symbols instead of a single symbol.

In this work we concentrate on cellular automata that give rise to factorial-local picture languages because these languages form a class that can be considered foundational in the hierarchy of factorial picture languages [4, 35]. We show that it is decidable whether a factorial-local language is left- and/or right-extendable. Furthermore, we observe that factorial-local cellular automata have the same characterization as one-sided cellular automata previously studied by P. Di Lena [24]: All but finitely many of their traces are shifts of finite type (SFT). Example 9 describes a factorial-local cellular automaton having non-SFT traces and Example 8 shows a factorial-local cellular automaton that is not one-sided. Therefore, factorial-local cellular automata extend the class of cellular automata studied
by P. Di Lena [24]. Using the characterization of factorial-local cellular automata through SFT traces and nilpotency we show that it is not decidable whether a given cellular automaton is factorial-local. The undecidability, in fact, also follows from the corresponding characterization of one-sided cellular automata [24].

We investigate the relationship of factorial-local cellular automata to other classes of cellular automata. In particular, we observe that factorial-local cellular automata have the shadowing property and hence form a proper subset of regular cellular automata introduced by P. Kurka [42]. In addition, we show that there are cellular automata with the shadowing property that are not factorial-local (Example 10).

We introduce notations and background definitions in the next section, while Section 2.2 describes factorial-local two-dimensional picture languages and their properties. Picture languages associated with cellular automata are introduced in Section 3.3 and their properties are discussed in Section 3.4.

The final type of automata discussed in this work are the Mandatory Results Automata (MRAs, discussed in Chapter 4). MRAs are particular types of runtime enforcement mechanisms that work by monitoring untrusted applications to ensure that those applications obey desired policies. Runtime mechanisms, which are often called security or program monitors, are popular and can be seen in operating systems, web browsers, spam filters, intrusion-detection systems, firewalls, access-control systems, etc. Despite their popularity and some initial efforts at modeling monitors formally, we lack satisfactory models of monitors in general.

Not having general models of runtime mechanisms is problematic because it prevents us from developing an accurate and effective theory of runtime enforcement. On the other hand, if we can model runtime mechanisms accurately and generally, we should be able to use those models to better understand how real security mechanisms operate and what their limitations are, in terms of policies they can and cannot enforce.

It has been difficult to model runtime mechanisms generally. Most models (e.g., [1, 20,
are based on truncation automata [49, 57], which can only respond to policy (which consist of sets of allowed executions) violations by immediately halting the application being monitored (i.e., the target application). This constraint simplifies analyses but sacrifices generality. For example, real runtime mechanisms typically enforce policies that require the mechanisms to perform “remedial” actions, like popping up a window to confirm dangerous events with the user before they occur (to confirm a web-browser connection with a third-party site, to warn the user before downloading executable email attachments, etc). Although real mechanisms can perform these remedial actions, models based on truncation automata cannot—at the point where the target attempts to perform a dangerous action, truncation automata must immediately halt the target. Immediately halting the target application without performing some auxiliary actions (e.g., popping up a window or writing to a log file) to audit or otherwise explain the program termination is not user friendly. We know of no runtime mechanisms operating as true truncation automata in practice.

To address the limitations of truncation automata, an earlier work proposed edit automata which are models of monitors that can respond to dangerous actions by quietly suppressing them or by inserting other actions [49]. By inserting and suppressing actions, edit automata capture the ability of practical runtime mechanisms to transform invalid executions into valid executions, rather than the ability of truncation automata to only recognize and halt invalid executions. Edit automata have served as the basis for additional studies of runtime enforcement (e.g., [10, 58, 62]).

Unfortunately, while truncation automata are too limited to serve as general models of runtime mechanisms, edit automata are too powerful. The edit-automata model assumes monitors can predetermine the results of all actions without executing them, which enables edit automata to safely suppress any action. However, this assumption that monitors can predetermine the result of any action is impractical because the results of many actions are uncomputable, nondeterministic, and/or cannot tractably be predicted by a monitor (e.g.,
actions that return data in a network buffer, the cloud cover as read by a weather sensor, or spontaneous user input). Put in another way, the edit-automata model assumes monitors can buffer—without executing—an unbounded number of target-application actions, but such buffering is impractical in general because applications may require results for actions before producing new actions. For example, the echo program

\[
x = \text{input}(); \quad \text{output}(x)
\]

cannot produce its second action until receiving a result, which is unpredictable, for the first. Because the echo program invokes an action that edit automata cannot suppress (due to its result being unpredictable), this simple program, and any others whose actions may not return predictable results, are outside the edit-automata model.

Mandatory Results Automata (MRAs), introduced in [50] and further studied in [28], are designed to reason about systems involving an application and an executing system in which the application sends actions to the executing system and the executing system returns a result for each action it receives (see Figure 15).

The ability of MRAs to transform results of actions is novel among general runtime-enforcement models, as far as we are aware (although many papers have modeled monitors transforming results in particular domains, such as policy composition [9], without investigating the enforcement capabilities of such monitors in general). Yet this ability to transform results is crucial for enforcing many security policies, such as privacy, access-control, and information-flow policies, which may require (trusted) mechanisms to sanitize the results of actions before (untrusted) applications access those results. For example, policies may require that system files get hidden when user-level applications retrieve directory listings, that email messages flagged by spam filters do not get returned to clients, or that applications cannot infer secret data based on the results they receive. Because existing frameworks do not model monitors transforming results of actions, one cannot in general use existing models to specify or reason about enforcing such result-sanitization policies.

As automata models, MRAs resemble finite-state transducers. However, in general, an MRA could have an infinite set of states and can operate on systems involving an infinite
number of events. MRAs operate on (possibly infinite) strings called executions. The security policies are sets of executions. An MRA can enforce a security policy by accepting a subset (sound enforcement), a superset (complete enforcement), or the whole policy (precise enforcement). We describe a characterization of the security policies that MRAs can enforce for each type of enforcement.

1.1 Notation

In this section we define regular string languages, finite state automata, and shift spaces. All of these will be used extensively in the later chapters of the dissertation.

1.1.1 String Languages

Let $A$ denote a finite set called the alphabet. The elements of the alphabet are called symbols. For example, $A = \{a, b\}$ is an alphabet consisting of symbols $a$ and $b$. In general, we assume that the alphabets we deal with have at least two symbols.

A word is a finite sequence of symbols. The $i$-th symbol (starting from 0) of the word $w$ is denoted by $w_i$, and the number of symbols in $w$, or the length of $w$, is denoted by $|w|$. For example, $|aba| = 3$ and $(aba)_1 = b$. The empty word, denoted by $\lambda$, is the word that contains no symbols. The set of all words over an alphabet $A$ is denoted by $A^*$, while the set of all words of length $k$ is denoted by $A^k$ (so $A^* = \cup_{k \geq 0} A^k$). A word $v$ is called a factor of a word $w$ if $w = uvu'$ for some (possibly empty) words $u$ and $u'$. The set of all factors of a word $w$ is denoted by $F(w)$, and the set of all factors of length $k$ of $w$ is denoted by $F_k(w)$. Note that, in particular, $F(w)$ contains $w$ and $\lambda$. For example, $F(aba) = \{\epsilon, a, b, ab, ba, aba\}$, while $F_1(aba) = \{a, b\}$.

A string language over an alphabet $A$ is a set of words over $A$, i.e., any subset of $A^*$. The set of factors of a string language $L$, denoted $F(L)$, is the collection of all factors of all words in $L$. The set $F_k(L)$ denotes the collection of words in $F(L)$ of length $k$. 
1.1.2 Finite State Automata

String languages are often studied using abstract machines (e.g., finite state automata, push-down automata, Turing machines) that recognize them. Finite state automata are perhaps some of the simplest and most fundamental of such machines.

**Definition 1.1.1** A deterministic finite-state automaton is a tuple \((A, Q, \delta, q_0, F)\) where \(A\) is the alphabet of the automaton, \(Q\) is a finite set of states, \(\delta : Q \times A \rightarrow Q\) is a transition function, \(q_0\) is an initial state, and \(F \subseteq Q\) is the set of final states.

A finite-state automaton \(A = (A, Q, \delta, q_0, F)\) accepts a word \(w\) of length \(k\) if there is a sequence of states \(q_0, q_1, \ldots, q_k\) such that \(\delta(q_i, w_i) = q_{i+1}\) for \(i = 0, \ldots, k - 1\) and \(q_k \in F\). The set of all words accepted by an automation \(A\) is called the language of \(A\) and denoted by \(L(A)\). A language \(L\) is called regular if there exists a finite state automaton \(A\) such that \(L = L(A)\). Regular languages form an important foundational class of string languages. It is well-known that the class of regular languages is closed under the union, intersection, and complement operations.

Finite state automata are often depicted as directed graphs with labeled edges. The nodes in such a graph represent the set of automaton states \(Q\), while the labeled edges (or transitions) between the states are defined by \(\delta\). More specifically, there is a transition between the states \(q\) and \(q'\) labeled by the symbol \(a\) if \(\delta(q, a) = q'\). Furthermore, on such diagrams the initial states are indicated by arrows pointing to them and final states have extra circles around them.

Note that a word \(w\) is accepted by a finite state automaton \(A\) precisely when there is a path on such a diagram that begins at the initial state \(q_0\), ends at a final state \(q \in F\), and has a label \(w\).

**Example 1** Consider a finite state automaton \(A = (A, Q, \delta, q_0, F)\), where \(A = \{0, 1\}\), \(Q = \{q_0, q_1, q_2\}\), \(\delta = \{(q_0, 0, q_1), (q_1, 0, q_1), (q_1, 1, q_2), (q_2, 1, q_2)\}\), and \(F = \{q_2\}\). Automaton \(A\) is depicted in Figure 2. Observe that \(A\) accepts words in \(\{0, 1\}^*\) that consist of
a sequence of 0’s followed by a sequence of 1’s; i.e., the language of $A$ is defined by the regular expression $0^+1^+$.

For completeness, we point out another well-known characterization of regular languages.

A string language $L \subseteq A^*$ is called strictly locally testable if there are languages $B, M, E \subseteq A^*$ such that

$$L = (BA^* \cap A^*E) \setminus (A^*MA^*).$$

If $B = E = \{\lambda\}$ in the above definition, then the language $L$ is called factorial local. It can be shown that a language $S \subseteq \hat{A}^*$ is regular if and only if there exists a strictly locally testable language $L \subseteq A^*$ and a map $\pi : A \to \hat{A}^*$ such that $S = \pi(L)$.

1.1.3 Shift Spaces

Informally, shift spaces are sets that consist of infinite sequences of symbols that are closed under the shift-left operation. Some shift spaces are particularly well-behaved and can be characterized by finite sets of words.

The set of all infinite sequences of symbols from $A$ is denoted by $A^\mathbb{N}$ while the set of all bi-infinite sequences is denoted by $A^\mathbb{Z}$. For convenience, we will use $A^{\square}$ to denote either $A^\mathbb{N}$ or $A^\mathbb{Z}$ and refer to the sequences in $A^{\square}$ as configurations. Given a configuration $\alpha \in A^{\square}$ and integers $i, j$ with $i \leq j$, a word $w = \alpha[i, j] = \alpha_i\alpha_{i+1}\cdots\alpha_j$ is said to be a factor of
\(\alpha\). In this case, we write \(w \sqsubseteq \alpha\). The set of all factors of \(\alpha\) is denoted by \(F(\alpha)\). The set of factors of a configuration \(\alpha = \cdots aaaaaa \cdots\) (the bi-infinite sequence consisting of symbol \(a\)) is \(F(\alpha) = \{\epsilon, a, aa, aaa, \ldots\}\).

We equip \(A\) with the discrete topology and \(A^\square\) with the product topology. A \emph{shift map} \(\sigma\) on \(A^\square\) is defined by \(\sigma(\alpha)_i = \alpha_{i+1}\). A set \(X \subseteq A^\square\) is called a \emph{shift space} if it is closed in the topology on \(A^\square\) and \(\sigma(X) \subseteq X\) (i.e., \(X\) is \(\sigma\)-invariant).
This section describes sets of blocks of symbols called picture languages. Picture languages can be thought of as a natural generalization of string languages to two dimensions. However, despite numerous attempts, hierarchies of string languages (e.g., Chomsky hierarchy) proved themselves to be difficult to generalize. Currently, some of the most established classes in the hierarchy of picture languages are the so-called local and recognizable picture languages.

Local picture languages are a generalization of local string languages while recognizable picture languages are a generalization of regular string languages. In this work, we propose to use picture languages to study the dynamics of cellular automata through the complexity of space-time diagrams that they generate.

Parts of this chapter is taken from [26]. The abstract, introduction, and conclusion of this paper were incorporated into the abstract, introduction, and conclusion of the present work.

2.1 Definition and Examples

We start with a definition of a block which is a generalization of a word to two dimensions.

**Definition 2.1.1** Let \( S = \{1, \ldots, n\} \times \{1, \ldots, m\} \). A **block** of size \( n \times m \) or an \( n \times m \)-**block** over alphabet \( A \) is a map \( B : S \rightarrow A \). We say that \( n \) is the **number of rows** in \( B \) and \( m \) is the **number of columns** in \( B \). If at least one of \( m, n \) equals 0 then the block is **empty** and is denoted by \( \lambda \).
Blocks can be depicted as arrays of symbols: the first row of a block is the top row, and every successive row is located underneath the previous row. In particular, \( B(1, 1) \) is the top left corner of a block \( B \).

Next, we define a sub-block which is just a block contained inside a given block.

**Definition 2.1.2** If \( B \) is a block of size \( n \times m \), and \( x, y, n', m' \) are positive integers, such that \( x + n' \leq n + 1 \) and \( y + m' \leq m + 1 \), then a sub-block or a factor of \( B \) of size \( n' \times m' \) at position \( (x, y) \), denoted \( B |_{(x,y)}^{n' \times m'} \), is the \( n' \times m' \)-block such that \( B |_{(x,y)}^{n' \times m'} (i, j) = B(x + i - 1, y + j - 1) \) for \( 1 \leq i \leq n' \) and \( 1 \leq j \leq m' \). We say that a block \( B' \) is a sub-block (or factor) of \( B \) if there are \( x, y, n', m' \) such that \( B' = B |_{(x,y)}^{n' \times m'} \). We consider the empty block to be a sub-block of every block.

We illustrate the sub-block notation with the following example. Consider a \( 7 \times 10 \) block

\[
A = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\end{bmatrix}
\]

and \( 4 \times 4 \) and \( 4 \times 5 \) blocks

\[
B = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Then \( B = A |_{(1,1)}^{4 \times 4} \) is a \( 4 \times 4 \)-block at position \( (1, 1) \) in \( A \), and \( C = A |_{(4,5)}^{4 \times 5} = A |_{(1,2)}^{4 \times 5} \) is a \( 4 \times 5 \)-block that can be seen at positions \( (4, 5) \) or \( (1, 2) \).

Given an \( n \times m \)-block \( B \), the \( i \)th row of \( B \) is denoted with \( \mathcal{R}_B(i) \) and the \( j \)th column of \( B \) is denoted with \( \mathcal{C}_B(j) \). Observe that \( \mathcal{R}_B(i) = B |_{(i,1)}^{1 \times m} \) and \( \mathcal{C}_B(j) = B |_{(1,j)}^{n \times 1} \).
The set $A^\ast\ast$ denotes the set of all blocks over alphabet $A$. A subset $L$ of $A^\ast\ast$ is called a \textit{two-dimensional language} or a \textit{picture language}. We extend notations from one-dimensional languages to picture languages in a natural way: $A^{k \times t}$ is the set of all $k \times t$-blocks; $F_{k,t}(B)$ is the set of all $k \times t$-sub-blocks of a block $B$ and $F_{k,t}(L)$ is the set of all factors of size $k \times t$ of blocks in $L$. Similarly, the set of all sub-blocks of $B$ is denoted by $F(B)$ and the set of all factors of $L$ is denoted by $F(L)$.

If $F(L) = L$ then the two-dimensional language $L$ is called \textit{factorial}.

\textbf{Definition 2.1.3} Consider an $n \times m$-block $B$ and an $n \times m'$-block $B'$ in $A^\ast\ast$. The \textit{concatenation} of the block, $B$ and $B'$ is the $n \times (m + m')$-block $BB'$ defined by

$$BB'(i,j) = \begin{cases} B(i,j) & \text{if } 1 \leq j \leq m, \\ B'(i,j - m) & \text{if } m + 1 \leq j \leq m + m'. \end{cases}$$

For example, the concatenation of blocks $B$ and $C$ in the example above is the block

$$BC = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}.$$ 

Observe that concatenation of blocks of different height is undefined.

\subsection{Factorial-local Two-dimensional Languages}

A two-dimensional language $L \subseteq A^\ast\ast$ is \textit{(n,m)-factorial-local} if there exists a set $\Theta$ of $n \times m$-blocks over $A$ such that $L = F(L_\Theta)$ where,

$$L_\Theta = \{ B \in A^\ast\ast \mid \emptyset \neq F_{n,m}(B) \subseteq \Theta \}.$$
We write $L = L(\Theta)$, and the set $\Theta$ is called the *set of allowed blocks* of $L(\Theta)$. The two-dimensional language $L$ is called *factorial-local* if there are $n, m \in \mathbb{N}$ such that $L$ is $(n, m)$-factorial-local.

Note that $L_\Theta$ contains blocks with at least $n$ rows and at least $m$ columns whereas $L(\Theta)$ contains all factors (of arbitrary size) of blocks in $L_\Theta$.

The set $A^{\mathbb{Z} \times \mathbb{Z}}$ is the set of all configurations of the plane, it is the set of all possible placements of symbols from $A$ onto the lattice of the integers. Similarly to the one-dimensional case, we consider $F(\alpha)$ to be the set of all rectangular blocks that appear within $\alpha \in A^{\mathbb{Z} \times \mathbb{Z}}$. A set $\Sigma \subseteq A^{\mathbb{Z} \times \mathbb{Z}}$ is called *two-dimensional shift of finite type* if there is a set of $m \times n$-blocks $\Theta$ such that $F(\Sigma) = L(\Theta)$. In this case we say that $\Theta$ *defines* $\Sigma$ and write $\Sigma = \Sigma_\Theta$.

**Definition 2.2.1** A two-dimensional language $L$ is *right-extendable* if for every $n \times m$-block $B$ in $L$ there exists an $n \times 1$-block $B'$ such that $BB'$ is in $L$. *Left-, up-, and down-extendable* languages are defined analogously. The language $L$ is *horizontally* extendable if it is left- or right-extendable, and it is *vertically* extendable if it is up- or down-extendable. We say that $L$ is *extendable* if it is horizontally and vertically extendable.

**Lemma 2.1** Let $\Theta$ be a set of $k \times t$-blocks and $\Theta' = L(\Theta) \cap A^{n \times m}$ with $n \geq k$ and $m \geq t$. If $L(\Theta)$ is extendable, then $L(\Theta) = L(\Theta')$.

**Proof.** For a block $B \in L(\Theta)$ there are two possibilities. In the first case, if $B$ is “larger” than $n \times m$, i.e., with $F_{n,m}(B) \neq \emptyset$. Then $F_{n,m}(B) = F(B) \cap A^{n \times m} \cap L(\Theta) = F(B) \cap \Theta' \subseteq \Theta'$ and $B \in L(\Theta')$. Second, if $B$ “smaller” than $n \times m$, i.e., $F_{n,m}(B) = \emptyset$. In this case, since $L(\Theta)$ is extendable, there is a block $B'$ in $L(\Theta)$ such that $B$ is a sub-block of $B'$ and $F_{n,m}(B') \neq \emptyset$. By the above argument, $B' \in L(\Theta')$ and since $L(\Theta')$ is factorial, $B$ is also in $L(\Theta')$. As $L(\Theta') \subseteq L(\Theta)$ follows directly from the definitions, we have equality. \hfill \Box

The above lemma says that an extendable $(k, t)$-factorial-local language is also a $(n, m)$-factorial-local for any $n \geq k$ and $m \geq t$. This observation is part of the “folklore” and we include it here for completeness.
By definition, if \( \Sigma \) is a two-dimensional shift of finite type defined by \( \Theta \), \( F(\Sigma) = L(\Theta) \).

In general, given \( \Theta \), it is undecidable whether \( \Sigma_\Theta \) is empty [11] (see also the last chapter of [52]), and, for the same reason, given a block \( B \in L(\Theta) \) it is undecidable whether there is \( \alpha \in \Sigma_\Theta \) such that \( B \in F(\alpha) \). The former problem is known as the extendability problem.

Below we show that it is decidable whether every block can be extended just horizontally (or in a similar manner, just vertically).

We introduce two finite-state automata, right- and left-check automata. The states of these automata consist of blocks in \( \Theta \), the transitions are defined according to the overlaps within these blocks, and the labels of the transitions are words that appear at the bottom of the transition source state.

Let \( \Theta \) be a set of \( n \times m \)-blocks. The **right-check** automaton \( M_{\Theta}^R \) is a finite state automaton defined as follows. Besides the initial state \( q_0 \), the set of states \( Q \) consists of \( \Theta \cup \{ q_i^B | i = 1, \ldots, m - 2, B \in \Theta \} \). The set of transitions \( \delta \) is defined as a subset of \( Q \times A^{m-1} \times Q \) with \( \delta = \delta_1 \cup \delta_2 \) defined as follows. First, recall that \( R_B(i) \) denotes the \( i \)'th row of the block \( B \) and define

\[
\delta_1 = \left\{ (B, w, B) \mid B, B \in \Theta, B |^{(n-1) \times m}_{(1, 1)} = B |^{(n-1) \times m}_{(1, 1)}, w = R_B(n)[2, m] \right\}.
\]

Figure 3 depicts a transition in \( \delta_1 \). Setting \( q_0^B = q_0 \) and \( q_{n-1}^B = B \) for every block \( B \in \Theta \), define

\[
\delta_2 = \left\{ (q_i^B, R_B(i)[2, m], q_i^B) \mid \text{for } i = 1, \ldots, n - 1 \text{ and } B \in \Theta \right\}.
\]

The words accepted by the right-check automaton correspond to the \( m - 1 \) rightmost columns of blocks in \( L(\Theta) \). The paths consisting of transitions in \( \delta_2 \) correspond to the possible first \( n - 1 \) of rows of these columns and the paths consisting of transitions in \( \delta_1 \) correspond to the rest of the rows. Once a path uses a transition in \( \delta_1 \), all remaining transitions of the paths are in \( \delta_1 \).

The left-check automaton is defined through the right-check automaton constructed on
Figure 3.: Transition of a right-check automaton. The shaded area of $B$ coincides with the shaded area of $\bar{B}$. The word $w$, as indicated, is the label of the transition.

the reversed blocks in $\Theta$. Given an $n \times m$-block $B$, define $B^{\text{rev}}$ to be an $n \times m$-block in which columns of $B$ appear in the reversed order, i.e.,

$$B^{\text{rev}}(i, j) = B(m - i + 1, j).$$

We extend this notation to the set of blocks $\Theta = \{B_1, \ldots, B_k\}$ by $\Theta^{\text{rev}} = \{B_1^{\text{rev}}, \ldots, B_k^{\text{rev}}\}$.

Consider $\mathcal{M}^{R}_{\Theta^{\text{rev}}} = (A^{m-1}, Q, \delta, q_0, Q)$, and let $\delta' = \{(q_1, w^{\text{rev}}, q_2) \mid (q_1, w, q_2) \in \delta\}$. Then the left-check automaton is defined to be $\mathcal{M}^{L}_{\Theta} = (A^{m-1}, Q, \delta', q_0, Q)$.

Hence the definition of left-check automaton follows the same construction as the right-check automaton on the reversed blocks of $\Theta$. The paths in the left-check automaton correspond to the $m - 1$ leftmost columns of blocks in $L(\Theta)$.

**Example 2** Let

$$B_1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad B_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \quad B_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$
The language $L(\Theta)$ consists of all blocks of 0s and 1s where 1s appear diagonally and every two diagonals of 1s are separated by at least three diagonals of 0s.

Then $M^R_\Theta$ is depicted in Figure 4.

**Figure 4:** The right-check automaton for Example 2.

**Lemma 2.2** For a given $\Theta$, the right-check automaton $M^R_\Theta$ is graph-isomorphic (not preserving the labels) with the left-check automaton $M^L_\Theta$.

**Proof.** The claim follows directly from the construction of $M^L_\Theta$. Observe that $M^L_\Theta$ differs from $M^R_\Theta$ only up to the labeling of the transitions. Moreover, there is a transition from
$B$ to $B'$ in $\mathcal{M}_\Theta^R$ if and only if there is a transition from $B^R$ to $B'^R$ in $\mathcal{M}_\Theta^{RK}$.

**Lemma 2.3** Let $\Theta$ be a set of $n \times m$-blocks with $n \geq 2$ and $m \geq 2$. Then $w = w_1 w_2 \cdots w_k$ is in $L(\mathcal{M}_\Theta^R)$ if and only if there is a $k \times t$-block $B$ in $L(\Theta)$ of width $t \geq m$ such that $w_i = \mathcal{R}_{B(i)}[t - m + 1, t]$. Similarly, $w \in L(\mathcal{M}_\Theta^L)$ if and only if there is a block $B$ in $L(\Theta)$ of width $t \geq m$ such that $w_i = \mathcal{R}_{B(i)}[1, m]$.

**Proof.** The claim follows directly from the definitions of right-check and left-check automata (see Figure 5).

![Figure 5.: Recognition of blocks by right- and left-check automata. Reading the word $w = w_1 \cdots w_k$ within a block $B \in L(\Theta)$ by the right-check (to the left) and by the left-check automaton (to the right). The shaded area is a block $P$ in $\Theta$ which is a state in the right-check automaton.](image)

**Lemma 2.4** Let $\Theta$ be a set of $n \times m$-blocks over alphabet $A$.

- If $L(\mathcal{M}_\Theta^R) \subseteq L(\mathcal{M}_\Theta^L)$, then $L(\Theta)$ is right-extendable.
- If $L(\mathcal{M}_\Theta^L) \subseteq L(\mathcal{M}_\Theta^R)$, then $L(\Theta)$ is left-extendable.

**Proof.** We prove the first case since the case when $L(\Theta)$ is left-extendable is similar. Note that it is enough to prove the claim for blocks of width at least $m$ and height at least $n$ since
every block in \( L(\Theta) \) is a factor of such block. Let \( B \in L(\Theta) \) be a \( k \times t \)-block and \( P \) be the sub-block of \( B \) that consists of the last \( m - 1 \) columns of \( B \). Then, by Lemma 2.3, there exists a word \( w \) in \( L(M^R_{\Theta}) \) of length \( k \) such that \( R_P(i) = w_i \), for \( i = 1, \ldots, k \). Since \( w \) is also in \( L(M^L_{\Theta}) \) there exists a \( k \times m \)-block \( \bar{P} \in L(\Theta) \) with a sub-block \( P \) formed by the first \( m - 1 \) columns. Let \( P' \) be the last \((m-\text{th})\) column of \( \bar{P} \). Then \( F_{n \times m}(BP') \subseteq \Theta \), i.e., \( BP' \in L(\Theta) \). Hence \( L(\Theta) \) is right extendable.

\[ \square \]

2.3 Other Classes of Picture Languages

In this section, we use FLOC to denote the class of factorial-local picture languages. A factorial tiling system is a quadruple \((\Sigma, \Gamma, \Theta, \pi)\) where \( \Sigma \) and \( \Gamma \) are finite alphabets, \( \Theta \) is a finite set of \( 2 \times 2 \) blocks over \( \Gamma \) defining a factorial-local picture language \( L(\Theta) \) and \( \pi : \Gamma \mapsto \Sigma \) is a map called projection. A picture language \( L \subseteq \Sigma^{**} \) is factorial recognizable if there exists a tiling system \((\Sigma, \Gamma, \Theta, \pi)\) such that \( L = \pi(L(\Theta)) \) (extending \( \pi \) to the arrays). We denote by FREC(\( \Sigma \)) (or simply FREC) the family of all factorial-recognizable picture languages over \( \Sigma \).

For a block \( B \) over \( \Gamma \), let \( \hat{B} \) denote the block obtained by surrounding \( B \) with symbols \( \Box \). A two-dimensional language \( L \subseteq \Gamma^{**} \) is local if there exists a finite set \( \theta \) of \( 2 \times 2 \)-blocks over \( \Gamma \cup \{ \Box \} \) such that \( L = \left\{ B \in \Gamma^{**} \mid F_{2,2}(\hat{B}) \subseteq \Theta \right\} \) and we write \( L = L(\Theta) \). We denote LOC(\( \Gamma \)) (or simply LOC) the family of all local picture languages over \( \Gamma \).

A tiling system is a quadruple \((\Sigma, \Gamma, \Theta, \pi)\) where \( \Sigma \) and \( \Gamma \) are finite alphabets, \( \Theta \) is a finite set of \( 2 \times 2 \) blocks over \( \Gamma \cup \{ \Box \} \) defining a local language \( L(\Theta) \) and \( \pi : \Gamma \mapsto \Sigma \) is a projection. A two-dimensional language \( L \subseteq \Sigma^{**} \) is recognizable if there exists a tiling system \((\Sigma, \Gamma, \Theta, \pi)\) such that \( L = \pi(L(\Theta)) \). We denote by REC(\( \Sigma \)) (or simply REC) the family of all recognizable picture languages over \( \Sigma \).

In this section we discuss some properties of picture-language classes LOC, REC, FLOC, FREC, and row and column concatenation of regular and factorial string languages (as they
are defined in [4]). In particular, we show that for any factorial string language $L$, the set of its row and column concatenations belongs to FREC. The differences between framed (LOC, REC) and unframed (FLOC, FREC) languages have been studied in detail by A. Anselmo, N. Jonoska, and M. Madonia [4].

Let $B \ominus B'$ denote the vertical concatenation of blocks $B$ and $B'$ (having the same width) where the block $B'$ is placed directly underneath of the block $B$. This notation can be extended to sets, so that $L \ominus L' = \{B \ominus B' \mid B \in L \text{ and } B' \in L'\}$. Further, let $L_n = L \ominus \cdots \ominus L$ ($n$ times) and $L^* = \bigcup_{i \geq 0} L_i$.

**Lemma 2.5** For every string language $L$, $F(L^*)$ is a factorial-local picture language.

**Proof.** For a factorial string language $L$, consider a finite-state automaton recognizing it. That is, $\mathcal{M} = (A,Q,\delta,q_0,F)$ with $L(\mathcal{M}) = L$.

Given $n,m \in \mathbb{N}$, consider a relation $\propto_m^n$ on $Q$ defined by $q \propto_m^n q'$ whenever there are $u,u',v,$ and $v'$ in $A^*$ such that

$q_0u = q$ and $qv \in F$

and

$q_0u' = q'$ and $q'v' \in F$

with $|u| = |u'| = n$ and $|v| = |v'| = m$. In particular, if $q$ and $q'$ are in the relation $\propto_m^n$, then there are two accepting paths through these states of length $n + m$.

Let $X \in \mathcal{P}(Q)$ be an element of the power set of $Q$. Then $X$ is compatible, if for all $q$ and $p$ in $X$, $q \propto_m^n p$ for some $n,m \in \mathbb{N}$. Let $C_M \subseteq \mathcal{P}(Q)$ denote the set of all compatible subsets of $Q$.

Let $(q_1,a,q_2), (q_2,b,q_3), (p_1,d,p_2),$ and $(p_2,e,p_3)$ be the transitions of $M$. Then consider the following tile

<table>
<thead>
<tr>
<th>$(q_1,a,q_2)_c$</th>
<th>$(q_2,b,q_3)_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(p_1,d,p_2)_c$</td>
<td>$(p_2,e,p_3)_c$</td>
</tr>
</tbody>
</table>
where \( c \in C_M \) and \( q_i, p_i \in c \) for \( i = 1, \ldots, 3 \). Let \( \Theta \) consist of all possible tilings that can be constructed in this way. Define a tiling system \((\delta, A, \Theta, \pi)\), where \( \pi((q, a, q')) = a \). We show that \( \pi(L(\Theta)) = F(L^{*\otimes}) \).

Consider a block \( P \in \pi(L(\Theta)) \) and note that there exists a block \( B \in L(\Theta) \) with \( \pi(B) = P \) and

\[
B = \begin{pmatrix}
(q_1,1, a_1,1, q_1,2) c & \cdots & (q_1,m-1, a_1,m-1, q_1,m) c \\
\cdots & \cdots & \cdots \\
(q_n,1, a_n,1, q_n,2) c & \cdots & (q_n,m-1, a_n,m-1, q_n,m) c
\end{pmatrix}
\]

Observe that for all \( 1 \leq i \leq n \) and \( 1 \leq j \leq m \), \( q_{i,j} \) belongs to the same \( c \in C_M \).

This implies that for some \( n, m \in \mathbb{N} \), \( q \not\sim_k q' \) for every \( q, q' \in c \). Hence, there are blocks \( B' \) and \( B'' \) such that \( \pi(B'BB'') \in L^{*\otimes} \) and hence \( B \in F(L^{*\otimes}) \) as required.

Conversely, let \( P \) be a block in \( F(L(M)^{*\otimes}) \) then there is a block \( P' \in L(M)^{*\otimes} \) such that \( P \) is a sub-block of \( P' \). Then there is a block \( B' \in L(\Theta) \) such that \( \pi(B') = P' \), and hence there is a sub-block \( B \) of \( B' \) such that \( \pi(B) = P \). \( \square \)

Given two string languages \( L \) and \( L' \), let \( L \oplus L' \) denote the picture language consisting of blocks whose rows are words in \( L \) and columns are words in \( L' \).

**Lemma 2.6** Given a finite state automata \( M_1 \) and \( M_2 \) there exists \( L \) in FLOC and a projection \( \pi \) such that \( \pi(L) = L(M_1) \oplus L(M_2) \).

**Proof.** A construction similar to the one in Lemma 2.5 shows that \( L(M)^{*\otimes} \) is in FREC. The claim now follows by observing that \( L(M_1) \oplus L(M_2) = L(M)^{*\otimes} \cap L(M)^{*\otimes} \) and that FREC is closed under intersection. \( \square \)

**Corollary 2.6.1** Given a finite state automata \( M_1 \) and \( M_2 \) over an alphabet \( \Gamma \) and a projection \( \pi : \Gamma \rightarrow A \), then there exists an \( L \) in FREC such that \( L = F(\pi(L(M_1) \oplus L(M_2))) \)
DEFINITION 2.3.1 A deterministic Turing machine is a quadruple $M = (Q, \{q_0\}, T, \delta)$, where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $T \subseteq Q$ is the set of terminal states and $\delta : Q \times (A \cup \{B\}) \rightarrow (A \cup \{B\} \cup \{L, R\}) \times Q$ is a partial function of transitions.

A string language is called recursively enumerable (RE) if there exists Turing machine that accepts it.

Note that for every recursively-enumerable language $L$ over alphabet $A$ there is a language $L$ in LOC whose factors over alphabet $A$ are precisely the factors of words from $L$, i.e. $A^* \cap F(L) = F(L)$. Let $L$ denote an RE-language with $F(L)$ not regular (for example $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$) and $L \in$ LOC be such that $F(L) \cap A^* = F(L)$. Observe that if $F(L) \in$ REG we have that $F(L) \cap A^*$ is regular. Hence the $F(L)$ can not be in REC.

THEOREM 2.1 For every recursively enumerable language $L$ over $A$, there is $L$ in REC such that $F(L) = A^* \cap F(L)$

Proof. Let Turing machine $M = (Q, \{q_0\}, T, \delta)$ be given. Let $A$ be an alphabet of $M$ and denote $A' = \{a' \mid a \in A\}$. For every pair of symbols $a$ and $b$ in $A$ define

\[
\begin{array}{ccc}
\# & B & B \\
\# & \# & \# \\
\end{array}
\begin{array}{ccc}
B & B & B \\
# & \# & \# \\
\end{array}
\begin{array}{ccc}
B & a & B \\
# & \# & \# \\
\end{array}
\begin{array}{ccc}
B & a & B \\
# & \# & \# \\
\end{array}
\begin{array}{ccc}
B' & B' & B' \\
# & \# & \# \\
\end{array}
\begin{array}{ccc}
B' & B' & B' \\
# & \# & \# \\
\end{array}
\begin{array}{ccc}
B' & \# & \# \\
# & \# & \# \\
\end{array}
\begin{array}{ccc}
B & B & B \\
B & B & B \\
B & a & a \\
B & a & a \\
B & a & a \\
B & a & a \\
B & a & a \\
B & a & a \\
\end{array}
\begin{array}{ccc}
B & a & B \\
B & a & B \\
B & a & B \\
B & a & B \\
B & a & B \\
B & a & B \\
B & a & B \\
B & a & B \\
\end{array}
\begin{array}{ccc}
B & B & B \\
B & B & B \\
B & B & B \\
B & B & B \\
B & B & B \\
B & B & B \\
B & B & B \\
B & B & B \\
\end{array}
\begin{array}{ccc}
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
\end{array}
\begin{array}{ccc}
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
\end{array}
\begin{array}{ccc}
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
\end{array}
\begin{array}{ccc}
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
B' & B' & B' \\
\end{array}
\begin{array}{ccc}
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
B' & \# & \# \\
\end{array}
\]

For every transition $(q_i, a, R, q_j) \in \delta$ and $b, c, d \in A \cup \{B\}$ define

\[
\begin{array}{ccc}
c' & a' & a' \\
c' & q_i a' & q_i a' \\
\end{array}
\begin{array}{ccc}
a' & q_j b' & q_j b' \\
a' & q_j b' & q_j b' \\
\end{array}
\begin{array}{ccc}
q_j b' & d' & d' \\
q_j b' & d' & d' \\
\end{array}
\]

22
For every transition \((q_i, a, L, q_j) \in \delta\) and \(b, c, d \in A \cup \{B\}\) define
\[
\begin{array}{c|c|c|c|c|c}
q_i & a & L & q_j \\
\hline
q_i & a & d' & q_j & b' & c'
\end{array}
\]

For every transition \((q_i, a, b, q_j) \in \delta\) where \(b, c, d \in A \cup \{B\}\) define
\[
\begin{array}{c|c|c|c|c|c}
q_i & b' & c' & a' & d' & q_j \\
\hline
q_i & d' & a' & q_j & b' & c'
\end{array}
\]

and
\[
\begin{array}{c|c|c|c|c|c}
\# & B' & a' & b' & B' & \# \\
\hline
\# & B' & a' & B' & \# & \#
\end{array}
\]

For every \(q \in T\) define
\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\# & \# & \# & \# & \# & \# & \# & \# & \# \\
\hline
\# & B' & a' & b' & c' & q_a' & q_a' & B' & \#
\end{array}
\]

Let the collection of all such blocks be denoted by \(\Theta\). Let \(L = L(\Theta)\) be the local language defined by the set \(\Theta\). Then \(w \in L(M)\) if and only if there are positive integers \(n\) and \(m\) such that \(B^n w (B')^m\) is the first row of some block in \(L\). It follows that \(F(L(M)) = F(L) \cap A^*\). \(\square\)
Cellular automata originated from Stanislaw Ulam’s and John von Neumann’s work [15] in the 1940’s and have since been a focus of much pure and applied research. CAs can be considered a subclass of discrete dynamical systems that can perform complex computations. The defining feature of cellular automata is that they are completely specified by straightforward local rules and hence are being actively used in computer simulations. Parts of this chapter is taken from [26]. The abstract, introduction, and conclusion of this paper were incorporated into the abstract, introduction, and conclusion of the present work.

3.1 Definition and some examples

All the definitions in this subsection are well known, but we include them here for completeness.

Cellular automata can be thought of as functions that map bi-infinite sequences of symbols (or configurations) to themselves; i.e., they are mappings from $A^\mathbb{Z}$ to $A^\mathbb{Z}$. Each symbol of the output configuration is determined by a finite neighborhood of the input configuration. Formally, cellular automata are defined as follows.

**Definition 3.1.1** A one-dimensional cellular automaton is a triple $(A, r, \ell)$ where $A$ is the alphabet, $r$ is a natural number called the radius and $\ell : A^{2r+1} \rightarrow A$ is called the local function. The global function of a cellular automaton $(A, r, \ell)$ is the function $G : A^\mathbb{Z} \rightarrow A^\mathbb{Z}$ defined by $G(\alpha)_i = \ell(\alpha[i - r, i + r])$. We refer to the global function $G$ as the cellular automaton determined by the triple $(A, r, \ell)$ and write $G = G(A, r, \ell)$. 

24
EXAMPLE 3 (Shift Cellular Automaton) Given an alphabet $A$, the shift cellular automaton over $A$ is $\sigma = (A, 1, \ell)$, where the local function $\ell$ is defined by $\ell(a, b, c) = c$. Note that for any cellular automaton $G = (A, r, \ell')$ and a configuration $\alpha$, $G(\sigma(\alpha)) = \sigma(G(\alpha))$.

Cellular automata of the form $G = (\{0, 1\}, 1, \ell)$ are called elementary [60]. Consider an elementary cellular automaton $G$ with the local function $\ell$ defined by

\[
\begin{align*}
\ell(000) &= a_0 \\
\ell(001) &= a_1 \\
\ell(010) &= a_2 \\
\ell(011) &= a_3 \\
\ell(100) &= a_4 \\
\ell(101) &= a_5 \\
\ell(110) &= a_6 \\
\ell(111) &= a_7
\end{align*}
\]

order the output of the local function as follows

\[
\begin{array}{cccccccc}
111 & 110 & 101 & 100 & 011 & 010 & 001 & 000 & \text{input} \\
\hline
a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & \text{output}
\end{array}
\]

The decimal representation of the binary number $a_7a_6a_5a_4a_3a_2a_1a_0$ is called the rule number of the elementary cellular automaton. For example, for the shift cellular automaton, $a_7a_6a_5a_4a_3a_2a_1a_0 = 101010$. Because $1010102 = 42_{10}$, the shift cellular automaton is the rule 42 elementary cellular automaton.

Space-time diagrams of one-dimensional cellular automata are a half-plane arrays of symbols. For an automaton $G$, the first row of such an array is a configuration $\alpha \in A^\mathbb{Z}$, the second row is the configuration $G(\alpha)$, the third row is $G^2(\alpha)$, etc. Figure 6 shows the first seven rows of a space-time diagram of the shift cellular automaton $\sigma$ corresponding to the configuration $\alpha = \cdots 0000110110100 \cdots$.

According to the definition of cellular automata, the $i$-th symbol of a configuration output by a cellular automaton of radius $r$ is determined by the $i$-th symbol of the input configuration along with $2r$ symbols surrounding it. The interval $(i-r, i+r)$ is called the neighborhood of $i$ of radius $r$. There are, however, alternative definitions of cellular automata that allow arbitrary finite neighborhoods; i.e., neighborhoods of the form \{ $i + m$ | $m \in M$ \} where $M$ is a finite subset of $\mathbb{Z}$. Observe that our definition is comparable with these
\[
\cdots 0 0 0 0 1 1 0 1 1 0 1 0 0 \cdots \quad \alpha
\]
\[
\cdots 0 0 0 1 1 0 1 1 0 1 0 0 0 \cdots \quad \sigma(\alpha)
\]
\[
\cdots 0 0 1 1 0 1 1 0 1 0 0 0 0 \cdots \quad \sigma^2(\alpha)
\]
\[
\cdots 0 1 1 0 1 1 0 1 0 0 0 0 0 \cdots \quad \sigma^3(\alpha)
\]
\[
\cdots 1 1 0 1 1 0 1 0 0 0 0 0 0 \cdots \quad \sigma^4(\alpha)
\]
\[
\cdots 1 0 1 1 0 1 0 0 0 0 0 0 0 \cdots \quad \sigma^5(\alpha)
\]
\[
\cdots 0 1 1 0 1 0 0 0 0 0 0 0 0 \cdots \quad \sigma^6(\alpha)
\]

**Figure 6.** A space-time diagram of the shift cellular automaton \(\sigma\). The diagram corresponds to the initial configuration \(\alpha = \cdots 0000110110100 \cdots\) where all of the omitted symbols are 0’s.

other definitions because any such neighborhood \(\{i + m \mid m \in M\}\) is contained inside the neighborhood of \(i\) of radius \(r = \max \{|m| \mid m \in M\}\).

We equip \(A\) with the discrete topology and \(A^Z\) with the product topology. This topology is equivalent to the topology induced by the usual metric \(d\) on \(A^Z\) for which \(d(\alpha, \beta) = 2^{-n}\) where \(n = \max \{|m| \mid \alpha[-m, m] = \beta[-m, m]\}\). According to a well-known result by Curtis, Hedlund, and Lyndon [39], cellular automata on \(A^Z\) are exactly the continuous functions in \((A^Z, d)\) that commute with the shift cellular automaton \(\sigma\).

Next, we define the shadowing property of cellular automata. This property will help us to establish a hierarchy of certain classes of cellular automata.

**Definition 3.1.2** Given \(\delta > 0\), a sequence of configurations \(\alpha_1, \alpha_2, \ldots, \alpha_n\) over \(A\) is called an \(\delta\)-chain if

\[
 d(G(\alpha_i), \alpha_{i+1}) < \delta
\]

for \(1 \leq i < n\).

Intuitively, in an \(\delta\)-chain \(\alpha_1, \alpha_2, \ldots, \alpha_n\), \(\delta\) determines the size of the central portion of every configuration \(\alpha_i\) that must coincide with the image of its predecessor.

**Definition 3.1.3** Assume that \(\alpha_1, \alpha_2, \ldots, \alpha_n\) is a sequence of configurations in \(A^Z\). If,
for some configuration \( \alpha \in A^\Z \), we have \( d(G^i(\alpha), \alpha_i) < \epsilon \) for \( 0 \leq i \leq n \), then we say that \( \alpha_1, \ldots, \alpha_n \) is \( \epsilon \)-shadowed by \( \alpha \).

Intuitively, a sequence of configurations is \( \epsilon \)-shadowed by a configuration \( \alpha \) if the central portion (determined by \( \epsilon \)) of the \( i \)-th element of the sequence coincides with the central portion of the \( i \)-th image of \( \alpha \).

The diagram in Figure 7 (left) illustrates an \( \delta \)-chain, and the diagram in Figure 7 (right) illustrates the \( \epsilon \)-shadowing. The shaded region corresponds to the overlapping portion determined by \( \epsilon \).

**Definition 3.1.4** A cellular automaton \( G \) has the **shadowing property** if for every \( \epsilon > 0 \) there exists \( \delta > 0 \) such that every \( \delta \)-chain is \( \epsilon \)-shadowed by some configuration.

Let \( G(A, r, \ell) \) be a cellular automaton. If there is a symbol \( a \in A \) such that \( \ell(a^{2r+1}) = a \), then \( a \) is called a **quiescent state** for the automaton \( G \). Observe that if \( \alpha \in A^\Z \) is such that \( \alpha[i, i] = a \) for all \( i \in \N \), then \( G(\alpha) = \alpha \).

**Definition 3.1.5** A cellular automaton is said to be **nilpotent** if there is a configuration \( \alpha \in A^\Z \) and \( J \in \N \) such that for all \( j \geq J \) and \( \beta \in A^\Z \) it holds that \( G^j(\beta) = \alpha \).

It turns out that the above definition is equivalent to a seemingly more general requirement that the value of \( J \) depends on \( \alpha \) [19].
The configuration $\alpha$ in the above definition must be a constant configuration. To see this, note that $\alpha = G^J(\sigma(\alpha)) = \sigma(G^J(\alpha)) = \sigma(\alpha)$ because cellular automata commute with the shift (by the Curtis-Hedlund-Lyndon Theorem [39]). Furthermore, if $\alpha = \cdots aaaaa \cdots$, then $a$ must be a quiescent state because $G(\alpha) = \alpha$. To see this note that $\alpha = G^J(G(\alpha)) = G(G^J(\alpha)) = G(\alpha)$.

In 1992 Jarkko Kari showed that there is no algorithm capable of deciding whether a given cellular automaton is nilpotent [40] (i.e., the containment problem is undecidable). This result is often used to show the undecidability of the containment problem for other classes of cellular automata.

### 3.2 Traces of Cellular Automata

Given a cellular automaton $G = (A, r, \ell)$, consider all columns of width $k$ extracted from the space-time diagrams of $G$. Observe that these columns form a shift space over $(A^k)^N$. Below we precisely define these shift spaces and study their properties in the subsequent chapters.

**Definition 3.2.1** Given a cellular automaton $G$, the $k$-trace subshift (or just the $k$-trace) of $G$ is the set $T^k_G = \{(G^i(\alpha))[0, k-1])_{i \in \mathbb{N}} | \alpha \in A^Z\}$.

We refer to the 1-trace simply as the **trace** and write $T_G$ instead of $T^1_G$.

**Definition 3.2.2** A set $X \subseteq A^N$ is called a subshift of finite type (SFT) if there is $k \in \mathbb{N}$ and a set of words $B \subseteq A^k$ such that

$$X = X_B = \{x \in A^N | \forall i \in \mathbb{N} x_i x_{i+1} \cdots x_{i+k-1} \in B\}.$$ 

In this case we say that $B$ determines $X$ and that $k$ is the **order** of $X$.

A subshift $X$ is called sofic if the language $F(X) = \{w | w \sqsubseteq \alpha \text{ and } \alpha \in X\}$ is regular. A cellular automaton is called **regular** if it has a sofic $k$-trace for all $k \in \mathbb{N}$ [42].
3.3 CA-generated Two-dimensional Languages

Space-time diagrams of a one-dimensional cellular automaton can be visualized as half-plane arrays of symbols. The set of rectangular blocks extracted from such arrays forms a two-dimensional (picture) language. Such two-dimensional language contains the possible patterns (snapshots) that can be observed from the cellular automaton and are closely related to \( k \)-traces. In this section we introduce the notion of cellular automata-generated picture languages and focus on the the cellular automata that generate factorial-local languages.

**Definition 3.3.1** Given a cellular automaton \( G \), a CA-generated two-dimensional language, denoted \( F_G \), is the set of all \( n \times m \)-blocks \( B \) \((n, m \geq 0)\) such that

\[
\mathcal{R}_B(1) = G^k(\alpha)[i, i + m - 1] \\
\mathcal{R}_B(2) = G^{k+1}(\alpha)[i, i + m - 1] \\
\vdots \\
\mathcal{R}_B(n) = G^{k+n-1}(\alpha)[i, i + m - 1]
\]

for some \( i \in \mathbb{Z}, k \geq 0 \) and \( \alpha \in A^\mathbb{Z} \).

Observe that in the above definition \( i \) and \( k \) can be taken to be 0. Note that distinct cellular automata give rise to distinct picture languages because the \( 2 \times (2r + 1) \) blocks in the picture language \( F_G \) generated by a CA \( G \) contain the definition of \( G \)'s local function.

**Definition 3.3.2** A cellular automaton \( G \) is factorial-local if \( F_G \) is a factorial-local language. In this case we call the set of allowed blocks of \( F_G \) the set of allowed blocks of the cellular automaton \( G \).

**Example 4** Recall that the shift cellular automaton is a cellular automaton \( \sigma \) such that \( \sigma(\alpha)_i = \alpha_{i+1} \) for every \( \alpha \in A^\mathbb{Z} \). The shift cellular automaton is \((2,2)\)-factorial-local with
the following set of allowed blocks (the set of $2 \times 2$ blocks whose diagonal entries are constant).

$$
\begin{bmatrix}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}
$$

**Example 5** The rule 204 (inversion) is $(2, 1)$-factorial-local with the following set of allowed blocks.

$$
\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
$$

**Example 6** The set of all blocks $A^{**}$ is $(1, 1)$-factorial-local by setting $\Theta = A$. However, there isn’t a cellular automaton $G$ such that $F_G = A^{**}$ because the middle portion of a row of width at least $2r + 1$ (where $r$ is the radius of $G$) is uniquely determined by the preceding row.

The lemma below follows from the definition of CA-generated two-dimensional languages.

**Lemma 3.1** Let $G = (A, r, \ell)$ be a cellular automaton and $\Theta = F_G \cap A^{n \times m}$ with $n > 1$ and $m \geq 2r + 1$.

1. $F_G \subseteq L(\Theta)$.

2. $F_G$ is right-, left-, down-extendable.

3. $\bigcup_{i \geq m} (L(\Theta) \cap A^{n \times i})$ is right-extendable and left-extendable.

**Proof.** The first two properties are straight from the definition of $F_G$ as this set consists of blocks that can be extracted from the evolution of $G$ on bi-infinite configurations. The third condition follows from the first two. If $B \in L(\Theta)$ is of size $n \times i$ for $i \geq m$, then the left-most $n \times m$-sub-block of $B$, is in $\Theta \subseteq F_G$ and hence it is left-extendable. Similarly, the right most $n \times m$-sub-block of $B$ is in $\Theta$, implying that $B$ is right-extendable. □
For a configuration \( \alpha \) of a cellular automaton \( G(A, r, \ell) \) and a word \( w = \alpha[i, j] \sqsupset \alpha \) with \( |w| \geq 2r + 1 \) define (slightly abusing the notation) \( G(w) = G(\alpha_i \cdots \alpha_j) = G(\alpha)_{i+r} \cdots G(\alpha)_{j-r}. \)

Let \( G = (A, r, \ell) \) be a cellular automaton and let \( \Theta \) be the set that consist of \( 2 \times (2r+1) \) blocks in \( F_G \). Observe that these blocks “contain” the definition of the local function because the middle symbol of the bottom row is the result of applying the local function to the top row. Furthermore, any \( 2 \times (m+2r) \)-block in \( L(\Theta) \) contains \( 2 \times m \)-sub-block from \( F_G \) in the center. This observation can be generalized for “larger” blocks which would provide a sufficient condition for \( F_G \) to equal \( L(\Theta) \) for some \( \Theta \).

**Lemma 3.2** Let \( G = (A, r, \ell) \) be a cellular automaton and \( \Theta = F_G \cap A^{k \times t} \) with \( t \geq 2r + 1 \) and \( k \geq 2 \). Then an \( n \times m \)-block \( B \) is in \( F_G \) if and only if there are \( n \times (n-1)r \)-blocks \( B_1 \) and \( B_2 \) such that \( B_1BB_2 \in L(\Theta) \).

**Proof.** Note that \( A^{1 \times k} \cap F_G = A^k \), hence for \( n = 1 \) blocks \( B_1 \) and \( B_2 \) are empty so \( B_1BB_2 = B \). Assume \( n \geq 2 \).

If \( B \in F_G \) then the existence of blocks \( B_1 \) and \( B_2 \) with the desired property follows directly from the fact that \( F_G \) is both right- and left-extendable.

Conversely, if for some \( n \times m \)-block \( B \in L(\Theta) \) there are \( n \times (n-1)r \)-blocks \( B_1 \) and \( B_2 \) such that \( B_1BB_2 \in L(\Theta) \) then the claim follows from the following observations.

Note that every \( 2 \times (2r + 1) \)-sub-block of \( B_1BB_2 \) is a factor of \( \Theta \subseteq F_G \). Therefore, for every \( 2 \times (2r + 1) \)-sub-block \( P \) of \( B_1BB_2 \), \( \ell(R_P(1)) = P(2, r + 1) \). As a consequence, for every \( 2 \times (m + 2r) \)-sub-block \( \bar{P} \) of \( B_1BB_2 \), \( G(R_P(1)) = R_P(2)[r + 1, r + m] \), i.e., the result of applying \( G \) to the first row of \( \bar{P} \) coincides with the middle \( m \) symbols of the second row of \( \bar{P} \). This means that if \( \alpha \in A^2 \) is a configuration of \( G \) with \( \alpha_1 \cdots \alpha_{2(n-1)r+m} \) equal to the top row of \( B_1BB_2 \) (see Figure 8) then

31
Hence $B \in F_G$ (see Figure 8). □

The above lemma allows an algorithm that determines whether for a given set of blocks $\Theta$ and a given $r$ there is a cellular automaton $G$ with radius $r$ such that $F_G = L(\Theta)$. First we have the following characterization.

**Theorem 3.1** A cellular automaton $G = (A, r, \ell)$ is factorial-local if and only if there exists a set $\Theta$ of $n \times m$-blocks with $n > 1$ and $m \geq 2r + 1$ such that

- $\Theta = F_G \cap A^{n \times m}$,
- $L(\Theta)$ is left-extendable and right-extendable.

In this case $F_G = L(\Theta)$.

**Proof.** If $\Theta = F_G \cap A^{n \times m}$ and $L(\Theta)$ is left-extendable and right-extendable then every block $B$ in $L(\Theta)$ can be extended into block $B' = B_1 B B_2$ satisfying the hypothesis of
Lemma 3.2. Hence $B$ belongs to $F_G$. The converse follows from the definition of $F_G$. □

EXAMPLE 7 Consider a cellular automaton $G$ corresponding to the rule 192. Then define

$\Theta = F_G \cap A^{2\times3} = \{B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11}, B_{12}\}$, where

\[
\begin{align*}
B_1 &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, & B_2 &= \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}, & B_3 &= \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, & B_4 &= \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \\
B_5 &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, & B_6 &= \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}, & B_7 &= \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}, & B_8 &= \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \\
B_9 &= \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, & B_{10} &= \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, & B_{11} &= \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}, & B_{12} &= \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.
\end{align*}
\]

The finite-state automata $M^R_\Theta$ and $M^L_\Theta$ are depicted in Figures 9 and 10 respectively.

The languages of $M^R_\Theta$ and $M^L_\Theta$ can be defined as the set of words in $[(10)^+ \cup (11)^*(01) \cup \lambda](00)^+$ over alphabet $A^2$. Since $L(M^R_\Theta) = L(M^L_\Theta)$, $G$ is factorial-local by Lemma 2.4 and Proposition 3.1.

THEOREM 3.2 Given a cellular automaton $G = (A, r, \ell)$ and a set of $n \times (2r + 1)$-blocks $\Theta$ it is decidable whether $F_G = L(\Theta)$.

Proof. Using Lemma 3.2, the set $F_G \cap A^{n\times(2r+1)}$ can be computed by iterating $G$ on all words of length $2nr + 1$.

If $F_G \cap A^{n\times(2r+1)} = \Theta$, construct the right-check $M^R_\Theta$ and the left-check automaton $M^L_\Theta$. Then determine whether $L(\Theta)$ is left-extendable and right-extendable by comparing $L(M^R_\Theta)$ and $L(M^L_\Theta)$. If it is, then $L(\Theta) = F_G$ by Proposition 3.1, otherwise, conclude $F_G \neq L(\Theta)$ for any $G$ with radius $r$. □
Figure 9.: The left-check automaton for the language defined by rule 192 CA.
Figure 10.: The right-check automaton for the language defined by rule 192 CA.
Corollary 3.2.1 It is decidable whether a set of $n \times (2r+1)$-blocks $\Theta$ is a set of allowed blocks of a factorial-local cellular automaton of radius $r$.

Proof. Since there are finitely many cellular automata with radius $r$ over a fixed alphabet, it is sufficient to apply Theorem 3.2 to each of these automata. However, one can try to directly construct a local function and, if it exists, apply Theorem 3.2. This can be done as follows.

If $F_G = L(\Theta)$ for some cellular automaton $G$ then the top rows of blocks in $\Theta$ are exactly the factors of configurations of $G$ of length $2r+1$. Therefore all words of length $2r+1$ must appear as first rows of blocks in $\Theta$. In other words,

\[
\{ w \in A^{2r+1} \mid w = R_B(1) \text{ for some } B \in \Theta \} = A^{2r+1}.
\]

Given $\Theta$, this property can be decided. Assuming the set of all top rows of blocks in $\Theta$ is $A^{2r+1}$ we have the following. If $n = 1$ then, like in Example 6, there is no cellular automata $G$ with $F_G = L(\Theta)$. If $n \geq 2$, let $G(A, r, \ell)$ be the cellular automaton with the local function $\ell$ defined as follows. For every $w \in A^{2r+1}$ there exists a block $P \in L(\Theta)$ with $w$ as its top row. Define, if possible, $\ell : A^{2r+1} \to A$ with $\ell(w) = P(2, r+1)$. If this doesn’t define a function (the image of some $w$ is not unique) then conclude $F_G \neq L(\Theta)$ for any $G$ with radius $r$. If $\ell$ is defined, use Proposition 3.2 to decide whether $F_G = L_{\Theta}$. □

3.4 Factorial-local Cellular Automata

In this section we study the properties of factorial-local cellular automata. In particular, we prove that they have the same characterization as a class of one-sided cellular automata previously investigated by P. Di Lena [24], and give an example of factorial-local cellular automata that is not one-sided.

In addition we show that factorial-local cellular automata have the shadowing property which implies that they are regular. We describe a cellular automaton (originally defined
by P. Kurka [42]) that has the shadowing property, but is not factorial-local.

**Lemma 3.3** Let $G$ be a cellular automaton with a local rule of radius $r$ and $n \geq 2r + 1$. Then the $n$-trace of $G$ is an SFT of order $m$ if and only if $G$ is $(m, n)$-factorial-local.

**Proof.** Let $G$ be a cellular automaton with a local rule of radius $r$ and assume that the $n$-trace of $G$ is SFT $X_\Theta$ for some $n \geq 2r + 1$. Observe that words in $\Theta$ can be interpreted as $m \times n$-blocks. Let $L(\Theta)$ be the factorial-local language defined by these blocks. Notice that $F_G \subseteq L(\Theta)$ because every $m \times n$ factor of a block in $F_G$ belongs to $\Theta$.

We now prove that $L(\Theta)$ is both right- and left-extendable. Consider an $k \times \ell$ block $B$ in $L(\Theta)$. If $k < m$ or $\ell < n$, then $B$ is necessarily a sub-block of a larger block $P \in L(\Theta)$ and if $P$ is extendable, then so is $B$. Thus without loss of generality we assume that $k \geq m$ and $\ell \geq n$. Note that $B = Q'Q$ where $Q$ is an $k \times n$ sub-block of $B$. Then $Q \in F(X_\Theta)$ and hence $Q \in F_G$. Since $F_G$ is extendable, there is an $k \times n$ block $C$ such that $QC \in F_G$. And because $F_{k,n}(QC) \subseteq \Theta$, we have $F_{k,n}(BC) \subseteq \Theta$ and $BC \in L(\Theta)$. Thus $B$ is right-extendable. Symmetric argument shows that $B$ is left-extendable. Hence $G$ is $(m, n)$-factorial-local by Proposition 3.1.

Conversely, assume that $G$ is $(m, n)$-factorial-local cellular automaton and let $\Theta$ be its set of allowed $m \times n$ blocks. Note that $T^n_G \subseteq X_\Theta$ and hence to prove that $X_\Theta$ is $T^n_G$ it is enough to show that every factor of a configuration in $X_\Theta$ defines a block in $F_G$. Given a word (block) $B \in X_\Theta$, note that $F_{m,n}(B) \subseteq \Theta$ and hence $B \in F_G$ as required. □

**Definition 3.4.1** Given two $n \times m$-blocks $P$ and $Q$ and an integer $r < m/2$, $P \sqcap_r Q$ is the $n \times m$-block defined by

$$(P \sqcap_r Q)(i, j) = \begin{cases} P(i, j) & \text{if } i \neq n \text{ or } j \leq m - r, \\ Q(n, j) & j > m - r. \end{cases}$$

According to the definition, the last $r$ symbols of the last row of $P \sqcap_r Q$ are the same as the last $r$ symbols of the last row of $Q$. All remaining symbols of $P \sqcap_r Q$ are the same as
Figure 11.: Blocks used in the proof of Lemma 3.4. If the block in the middle and on the left are in $F_G$ then so is the block on the right.

in $P$. In other words, we have substituted the “bottom right corner” of $P$ with the bottom right corner of $Q$.

**Lemma 3.4** Let $G$ be a cellular automaton of radius $r$. Let $B'$ and $B''$ be two $k \times \ell$-blocks in $F_G$ whose first $k - 1$ rows coincide. If $\ell \geq 2r + 1$ and $B = B' \cap_r B''$, then for all blocks $A', A'', C', C''$ of height $k$ (possibly empty),

\[ A'BC', A''B''C'' \in F_G \text{ implies } A'BC'' \in F_G. \]

**Proof.** Note that all symbols in $B'$, $B''$ coincide, except possibly the leftmost and the rightmost $r$ symbols of the last row. This is because the middle $\ell - 2r$ symbols of the last rows of these blocks are determined by the previous rows (which are the same).

The claim follows from the fact that every $2r + 1$-sub-word of the first $k - 1$ rows of $A'(B' \cap_r B'')C''$ is a sub-word of the corresponding row of either $A'B'$ or $B''C''$ (see Figure...
THEOREM 3.3 If a cellular automaton of radius $r$ is $(n, m)$-factorial-local, then it is also $(n, 2r + 1)$-factorial-local.

Proof. Assume $G$ is an $(n, m)$-factorial-local cellular automaton with local function of radius $r$ and the set of allowed blocks $\Theta$ of size $n \times m$.

If $m < 2r + 1$, then by Lemma 3.1, $F_G = L(\Theta)$ is extendable and according to Lemma 2.1, $G$ is also $(n, 2r + 1)$-factorial-local.

Suppose that $m > 2r + 1$. Let $\Theta' = F_G \cap A^{n \times (2r+1)}$. We show that the $(2r+1)$-trace of $G$ equals the SFT $X_{\Theta'}$. According to Lemma 3.3, the equality implies that $G$ is $(n, 2r + 1)$-factorial-local. Note that the $(2r+1)$-trace of $G$ is a subset of $X_{\Theta'}$ by construction of $\Theta'$.

We use induction to prove that every factor of a configuration in $X_{\Theta'}$ is also a factor of a configuration in $(2r+1)$-trace of $G$.

A block $B$ defined by a factor of $X_{\Theta'}$ of height $k \leq n$ is a factor of a block in $\Theta'$ and hence belongs to $F_G$. For the inductive step, suppose all $s \times (2r+1)$-blocks in $L(\Theta')$ belong to $F_G$ for all $s < k$. Let $B$ be a $k \times (2r+1)$-block in $X_{\Theta'}$. Observe that by the inductive hypothesis, the block comprised of the top $k-1$ rows of $B$ belongs to $F_G$. Since blocks in $F_G$ are down-extendable by Lemma 3.1, there exists a $k \times (2r+1)$-block $C$ in $F_G$ whose first $k-1$ rows equal to the first $k-1$ rows of $B$.

Let $B_L = B \cap_r C$, $B_R = C \cap_r B$ and note that $B_L$ and $B_R$ are the same as $C$ except for the left and right bottom corners respectively which are taken from $B$. Let $D_L$, $D_R$, and $D$ denote the blocks comprised of the bottom $n$ rows of $B_L$, $B_R$, and $C$ respectively. Note that by Lemma 3.4 applied to $D$ and the $n \times (2r+1)$-sub-block of $B$ comprised of the bottom $n$ rows of $B$, both $D_L$ and $D_R$ are in $F_G$ and hence in $\Theta'$. All other $n \times (2r+1)$-sub-blocks of $B_L$ and $B_R$ are sub-blocks of $C$ (and also $B$) and hence also belong to $\Theta'$. This shows that both $B_L$ and $B_R$ are in $L(\Theta')$.

We show that $B_L$ is in $F_G$. Let $Q = CC'$ be a $k \times m$-block in $F_G$ whose left-most
$k \times (2r + 1)$-sub-block is $C$ ($C'$ exists since $C \in F_G$ and $F_G$ is right-extendable by Lemma 3.1). Denote the block comprised of the bottom $n$ rows of $C'$ with $D'$ (see Figure 12).

Let $P = B_L C'$, then the bottommost sub-block of $P$ of height $n$ is $D_L D'$. The leftmost $n \times (2r + 1)$-sub-block of $D_L D$ is $D_L$ (also sub-block of $B_L$) and hence belongs to $\Theta'$ and consequently to $F_G$.

Applying Lemma 3.4 to both $D_L$ and $D' \cdot D'$, we obtain that $D_L D$ belongs to $F_G$. All other $n \times m$ sub-blocks of $P$ are also sub-blocks of $Q$ and hence belong to $\Theta$. Thus $P \in L(\Theta) = F_G$. Since $B_L$ is a sub-block of a block in $F_G$, $B_L$ is also in $F_G$.

A similar argument shows that $B_R$ is in $F_G$. Furthermore, by Lemma 3.4, $B = B_L \cap_r B_R$ is in $F_G$ as needed. □

Lemma 3.3 and Proposition 3.3 show that factorial-local cellular automata have the same characterization in terms of traces as a class of one-sided cellular automata with SFT traces investigated by P. Di Lena [24].

**Theorem 3.4** (P. Di Lena, [24]) Let $G$ be a one-sided cellular automaton of radius $r$. The
following conditions are equivalent:

- $k$-trace of $G$ is SFT ($k \geq r$),
- $r$-trace of $G$ is SFT,
- all of the $k$-traces ($k \geq r$) of $G$ are SFT.

Replacing $r$ with $2r+1$ in Proposition 3.4 (Proposition 3.6 in [24]) gives the corresponding characterization of factorial-local cellular automata.

**Corollary 3.4.1** Let $G$ be a cellular automaton of radius $r$. The following conditions are equivalent:

1. $G$ is factorial-local,
2. $k$-trace of $G$ is SFT ($k \geq 2r+1$),
3. $(2r+1)$-trace of $G$ is SFT,
4. all of the $k$-traces ($k \geq 2r+1$) of $G$ are SFT.

**Proof.** Lemma 3.3 proves that (1) implies (2). If $k$-trace of $G$ ($k \geq 2r+1$) is SFT, then according to Lemma 3.3 and Proposition 3.3, $G$ is $(n, 2r+1)$-factorial-local for some $n$ and by Lemma 3.3, the $(2r+1)$-trace of $G$ is SFT. Hence (2) implies (3). If $(2r+1)$-trace of $G$ is SFT, then by Lemma 3.3, $G$ is $(n, 2r+1)$-factorial-local for some $n$, and hence according to Lemma 2.1, it is also $(n, k)$-factorial-local for every $k \geq 2r+1$. Again, by Lemma 3.3, all of the $k$-traces ($k \geq 2r+1$) of $G$ are SFT. Thus (3) implies (4). The fact that (4) implies (1) follows directly from Lemma 3.3.

**Example 8** Note that there are factorial-local cellular automata which are not one-sided. Cellular automaton $G$ defined by rule 128 is one example. This automaton is factorial-local
with the following allowed blocks of size $2 \times 3$.

\[
\begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
\end{array}
\]

However, it is not one-sided since

\[
\begin{array}{cccc}
1 & 1 & 1 & \ldots & 1 \\
0 & 1 & 1 & \ldots & 1 \\
\end{array} \in F_G, \quad \begin{array}{cccc}
1 & \ldots & 1 & 1 & 1 \\
1 & \ldots & 1 & 1 & 0 \\
\end{array} \in F_G, \quad \begin{array}{cccc}
1 & 1 & \ldots & 1 & 1 \\
1 & 1 & \ldots & 1 & 1 \\
\end{array} \in F_G
\]

We point out that there are factorial-local cellular automata that have some of their traces being sofic and not shift of finite type.

**Example 9** Rule 123 is an example of a cellular automaton that is factorial-local whose trace is not a shift of finite type. Recall that this cellular automaton is defined by the following local function.

\[
\begin{array}{cccccccccccc}
111 & 110 & 101 & 100 & 011 & 010 & 001 & 000 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
\end{array}
\]

By using a computer program we obtained a set of allowed blocks for this cellular automaton which consists of blocks of size $5 \times 3$. The deterministic graph-representation of the trace is depicted in Figure 13. The trace is not a shift of finite type because the automaton in Figure 13 because the infinite sequence $(01)\omega$ has multiplicity 2 [32] (or because the corresponding finite-state automaton has two cycles with the same label). By Corollary 3.2.1 for every $k \geq 3$, the $k$-trace of rule 123 is a shift of finite type. This cellular automaton is an example showing traces to be both strictly sofic and shift of finite type.
Below we investigate the relationship between factorial-local, shadowing, and nilpotent cellular automata.

**Corollary 3.4.2** Factorial-local cellular automata have the shadowing property.

**Proof.** Kurka [42] proves that every cellular automaton whose every $k$-trace, $k \geq 1$ is a subshift of finite type has the shadowing property. In fact, the same argument (taking $k \geq n$ and decreasing appropriately $\delta$) proves our claim.

However, the example below shows that the class of cellular automata with the shadowing property is larger than the class of factorial-local cellular automata.

**Example 10** Consider Example 17 in [42] which has the shadowing property. It is defined on alphabet $\{0, 1, 2, 3, 4\}$ with radius $r = 1$ and a local function determined with the following table (in all other cases, the local function does not change the symbol).

<table>
<thead>
<tr>
<th></th>
<th>02</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>2</th>
<th>34</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

For every $k$, the $k$-trace of this cellular automaton contains $m \times k$-blocks that are all 1’s except the last column which has 0’s separated by even number of 1’s. This is true because
every configuration that has a right infinite portion after a 0 or a 1 which consists of symbols 2, 3, and 4 evolves such that the 4’s propagate to the left and the 2 and 3 become 1’s. When a 4 is next to 1 it becomes a 0. The alternation between 2 and 3 ensures that there are even number of 1’s between any two 0’s. Note that the only way that a 1 shows up after a 0 in the trace is when 1 is adjacent to 4. Therefore, for all \( k \), the \( k \)-trace of this cellular automaton is not a shift of finite type and therefore, this cellular automaton is not factorial-local.

The following corollary is immediate from Corollary 3.4.2.

**Corollary 3.4.3**  Factorial-local cellular automata are regular.

*Proof.* Proposition 1 in [42] proves that cellular automata with shadowing property are regular. \( \square \)

The following observation follows from the fact that every nilpotent cellular automata has all its \( k \)-traces shifts of finite type [24], and therefore every nilpotent cellular automaton is factorial-local. The proof included here shows that this fact can be deduced independently.

**Theorem 3.5**  Every nilpotent cellular automaton is factorial-local.

*Proof.* Let \( G \) be nilpotent and \( J \in \mathbb{N} \) and \( z \in A^\mathbb{Z} \) be such that \( G^J(\alpha) = z \) for all \( \alpha \in A^\mathbb{Z} \). Then \( z \) is a constant configuration having all states equal to a quiescent state \( q \).

Let \( s = 2(J - 1)r + 1 \) and define \( \Theta \) to be the collection of \( J \times s \)-blocks (\( J \) rows and \( s = 2(J - 1)r + 1 \) columns) in \( F_G \).

Let \( B \in F_G \) be an \( n \times m \)-block. If \( J \leq n \) and \( s \leq m \) then \( \emptyset \neq F_{J,s}(B) \subseteq \Theta \), i.e., \( B \in L(\Theta) \). If \( n < J \) or \( m < s \), (recall that \( F_G \) is right-extendable and down-extendable) there exists a \( n' \times m' \)-block \( B' \) in \( F_G \) such that \( J \leq n' \), \( s \leq m' \) and \( B \) is a sub-block of \( B' \). By the first part of the proof, \( B' \in L(\Theta) \), therefore \( B \in L(\Theta) \) since \( L(\Theta) \) is factorial. Hence \( F_G \subseteq L(\Theta) \).

44
Let \( n \times m \)-block \( B \) be in \( L(\Theta) \). If \( n \geq J \) then all rows in \( B \) starting from \( J \)th, up till the \( n \)th, equal \( q^m \), because for each \( P \) in \( \Theta \), the bottom row of \( P \) is \( q^s \). Hence, to prove that \( B \in F_G \) it is sufficient to prove that \( B' \), the \( J \times m \)-sub-block of \( B \) at position \( (1, 1) \) is in \( F_G \), i.e., that there is a configuration that can generate the first \( J \) rows of \( B \). By Lemma 3.1, \( L(\Theta) \) is left-extendable and right-extendable when restricted to the blocks of height \( J \) and therefore, by Lemma 3.2, \( B' \in F_G \). □

In [23], it was shown that it is decidable whether a regular cellular automaton is nilpotent. Therefore Proposition 3.6 below is a consequence of [23] and Corollary 3.4.3. Here we observe that one can use another algorithm in the case when the cellular automaton is factorial-local. Unfortunately, both algorithms have very high complexity.

**Theorem 3.6** If \( G \) is a factorial-local cellular automaton, then it is decidable whether \( G \) is nilpotent.

**Proof.** Let \( G \) be a factorial-local cellular automaton. Consider a list of all possible sets of \( n \times m \) blocks and for each set \( \Theta \) from this list use Proposition 3.2 to determine if \( L(\Theta) = F_G \). The equality will be eventually obtained since \( G \) is factorial-local. Assume \( G \) is factorial-local and \( \Theta \subseteq A^{n \times m} \) is such that \( F_G = L(\Theta) \). Consider the right-check automaton \( M^R_\Theta \) and let \( k \) denote an integer such that every word in \( L(M^R_\Theta) \) of length greater than \( k \) can be “pumped”. Such \( k \) always exists because \( F_G \) is always down-extendable. Let \( M' \) denote an automaton over alphabet \( \tilde{A} = A^{m-1} \) that recognizes the language defined by the regular expression \((\lambda + \tilde{A} + \tilde{A}^2 + \cdots + \tilde{A}^k)(\tilde{q})^*\), where \( \tilde{q} = q^{m-1} \in \tilde{A} \) is the quiescent state (if there is no quiescent state, or there is more than one quiescent state then \( G \) is not nilpotent and checking for quiescent state is decidable). If \( L(M^R_\Theta) \setminus L(M') \) is not empty, then there is a word in \( L(M^R_\Theta) \) containing a symbol other than \( \tilde{q} \) at position greater than \( k \) that can be “pumped”. So, \( G \) is not nilpotent. However, if \( L(M^R_\Theta) \setminus L(M') \) is empty, then we can take \( J = k \). Then for all \( j \geq J \) and \( \alpha \in A^Z \) we have \( G^j(\alpha) = z \) (where \( z \) is the constant configuration whose every cell is in the state \( q \)). Hence \( G \) is nilpotent. □
Corollary 3.4.4 It is undecidable whether a cellular automaton is factorial-local.

Proof. The claim follows from the Proposition 3.6 and the fact that all nilpotent cellular automata are factorial-local. Nilpotency problem has been shown to be undecidable in [40]. □

3.5 Transducers

In this remaining sections of this chapter we consider a problem of simulating cellular automata with symbol-to-symbol transducers (further referred to simply as transducers).

A class of two-dimensional picture languages that falls in between local and recognizable (see below) two-dimensional picture languages called transducer generated languages was introduced by us in [25]. A transducer generated language is the set of all rectangular blocks that are obtained through the successive iterations of some transducer, such that each row in a block is an output of the transducer from the preceding row. In [27] it was shown how this class of languages determines the set of arrays that can be generated by triple cross-over DNA molecules simulating Wang tiles and a use of appropriate molecular device that sets up the input of the transducer. The De-Bruijn graph of a cellular automaton can be seen as a transducer and therefore, the picture languages generated by cellular automata are in close relationship to the transducer generated languages. We observe that not every transducer generated language is also a picture language generated by a cellular automaton, and we give necessary conditions in which a given transducer simulates a cellular automaton.

Definition 3.5.1 A nondeterministic transducer is a five-tuple

$$\tau = (\Sigma, Q, \delta, q_0, T),$$

where $\Sigma$ is a finite alphabet, $Q$ is a finite set of states, $q_0$ is an initial state ($q_0 \in Q$), $T$ is
a set of final (terminal) states \((T \subseteq Q)\), and \(\delta \subseteq Q \times \Sigma \times \Sigma \times Q\) is the set of transitions. A transducer is called deterministic if its set of transitions defines a function (which is also denoted with \(\delta\)) \(\delta : Q \times \Sigma \rightarrow \Sigma \times Q\).

To a transducer we associate a directed labeled graph in the standard way: the set of vertices is the set of states \(Q\) and the directed edges are transitions in \(\delta\), such that an edge \(e = (q, a, a', q')\) starts at \(q\) and terminates at \(q'\). To each edge we associate labels \(I : \delta \rightarrow \Sigma\) and \(O : \delta \rightarrow \Sigma\) being the input and the output labels such that for \(e = (q, a, a', q')\) we have \(I(e) = a\) and \(O(e) = a'\). The input and the output labels are naturally extended to paths in the transducer.

We say that a word \(w\) is accepted by a transducer \(\tau = (\Sigma, Q, \delta, q_0, T)\) if there is a path \(p = e_1 \cdots e_k\) which starts at \(q_0\), terminates with a state in \(T\) and \(I(p) = I(e_1) \cdots I(e_k) = w\). The path \(p\) in this case is called an accepting path for \(w\). The language which consists of all words that are accepted by \(\tau\) is called the input language of \(\tau\) and is denoted \(I(\tau)\). A word \(v\) is said to be an output of \(\tau\) if there is \(w \in I(\tau)\) and an accepting path \(p\) for \(w\) such that \(O(p) = v\). In this case we also write \(v \in O(w)\). The language which consists of all outputs of \(\tau\) is the output language for \(\tau\) denoted with \(O(\tau)\).

### 3.6 Transducer-Generated Picture Languages

If \(\tau\) is a transducer (deterministic or nondeterministic), we define inductively: \(\tau^0(w) = \{w\}\) if \(w \in I(\tau)\) and \(\tau^r(w) = \{u \mid \text{there is } v \in \tau^{r-1}(w), u \in O(v)\}\).

**Definition 3.6.1** An \(n \times m\)-block \(B\) is generated by a transducer \(\tau\) if \(B_{(1,1)}^{n \times 1} \in I(\tau)\) and \(B_{(1,i+1)}^{n \times 1} \in \tau(B_{(1,i)}^{n \times 1})\) for all \(1 \leq i < m\).

**Example 11** Consider a finite-state transducer \(\tau = (\Sigma, Q, \delta, q_0, T)\) where \(\Sigma = \{0, 1\}\), \(Q = \{q_0, q_1, q_2\}\), \(T = \{q_2\}\), and \(\delta = \{(q_0, 0, 1, q_1), (q_1, 0, 1, q_1), (q_1, 1, 0, q_2), (q_2, 1, 0, q_2)\}\).
Figure 14.: An example of a finite state transducer. The transducer accepts words consisting of 0’s followed by 1’s and outputs a word obtained by swapping 1s and 0s in the original word.

This transducer is depicted in Figure 14. Observe that \( \tau \) generates the block

\[
B = \begin{bmatrix}
0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0
\end{bmatrix}
\]

Note that the requirement \( B_{(1,i+1)}^{n \times 1} \in \tau(B_{(1,i)}^{n \times 1}) \) is not equivalent to the more general property \( B_{(1,1)}^{1 \times m} \in \tau^i(B_{(1,1)}^{1 \times m}) \), since the latter does not require a row to be an output of \( \tau \) on a previous row (unless \( \tau \) is deterministic). Moreover the empty block \( \epsilon \) is generated by a transducer \( \tau \) if the initial state is also a terminal state. A \( n \times 1 \)-block is generated by a transducer \( \tau \) if it belongs to \( I(\tau) \). Every transducer with non-empty input language generates blocks of size \( n \times i \) for \( i = 1, 2 \).

**Definition 3.6.2** A picture language which consists of all blocks that are generated by a deterministic (nondeterministic) transducer \( \tau \) is called the picture language generated by \( \tau \) and is denoted with \( L_\tau \). A picture language \( L \) is said to be transducer generated if there is a transducer \( \tau \) such that \( L = L_\tau \).

We restrict the following discussion to the one-dimensional cellular automata with a local rule of radius \( r = 1 \) and the set of states \( A = \{0, 1, q\} \), where \( q \) is a quiescent state. In addition, assume that the local rule, \( \ell : A^{2r+1} \rightarrow A \), of every cellular automaton has the following property. Given \( a_0, a_2 \in A \), \( \ell(a_0, a_1, a_2) \neq q \) if \( a_1 \in \{0, 1\} \). We will also restrict cellular automata to finite configurations that contain symbols 0 and 1 contiguously.
DEFINITION 3.6.3 A transducer $\tau$ and a cellular automaton $G$ are equivalent if $m \times n$ block $B \in L_\tau$ if and only if

$$\alpha = \ldots qqqB_{1,1}^m \ldots$$

$$G(\alpha) = \ldots qqqB_{1,2}^m \ldots$$

$$\ldots$$

$$G^n(\alpha) = \ldots qqqB_{1,n}^m \ldots$$

THEOREM 3.7 For every cellular automaton $G$ there is a transducer $\tau$ equivalent to it.

Recall that a transducer $\tau$ is called functional if it defines a function on $I(\tau)$.

THEOREM 3.8 Let $\tau$ be a functional transducer with $I(\tau) = A^*$. Then there exists a cellular automaton equivalent to $\tau$ if and only if there exists an integer $r$ such that any two accepting paths

$$p_1 = e_1e_2\ldots e_n$$

$$p_2 = e'_1e'_2\ldots e'_n$$

satisfy the following conditions

- for $1 \leq i < r$, if $I(e_1\ldots e_{i+r}) = I(e'_1\ldots e'_{i+r})$ then $O(e_i) = O(e'_i)$
- for $r \leq i, j \leq n - r$, if $I(e_i\ldots e_{i+r}) = I(e'_i\ldots e'_{j+r})$ then $O(e_i) = O(e'_j)$
- for $n - r < i \leq n$, if $I(e_i\ldots e_n) = I(e'_i\ldots e'_n)$ then $O(e_i) = O(e'_i)$

COROLLARY 3.4.5 The $r$ in the above proposition can be taken to be $n^2$, where $n$ is the number of states in the transducer $\tau$.

Proof. This follows from the fact that any two paths in $\tau$ of length $n^2$ with the same input label can be extended to a larger paths with the same input label. \qed

49
The next theorem (from [27]) shows that transducer-generated languages are recognizable.

**Theorem 3.9 Transducer generated languages are in REC.**

**Proof.** Assume that the transducer \( \tau = (\Sigma, Q, \delta, q_0, T) \) is given. Let \( \Gamma = Q \Gamma = \{ qa | q \in Q \text{ and } a \in \Sigma \} \) and projection \( \pi : \Gamma \mapsto \Sigma \) be defined by \( \pi(qa) = a \). To prove the claim we define a set of blocks \( \Theta \) such that \( \pi(L(\Theta)) = L_\tau \). The set \( \Theta \) consists of the union of input blocks, transition blocks, and output blocks defined below.

- **Input blocks.** The collection of input blocks \( \Theta_1 \) is the union of \( \Theta_1, \Theta_2, \) and \( \Theta_3 \) defined as follows.

\[
\Theta_1 = \left\{ qa q' c \left| (q, a, b, q') \in \delta \text{ and } c \in \Sigma \right. \right\}
\]

\[
\Theta_2 = \left\{ qa \left| q_0 \text{ is the initial state and } a \in \Sigma \right. \right\}
\]

\[
\Theta_3 = \left\{ qa \left| (q, a, b, q') \in \delta, c \in \Sigma, q' \in T \right. \right\}
\]

- **Transition blocks.** The collection of transition blocks \( \Theta_t \) is the union of \( \Theta_4, \Theta_5, \) and \( \Theta_6 \) defined as follows.
\[ \Theta_4 = \begin{cases} \begin{array}{c|c} pb & p'c \\ \hline qa & q'd \end{array} \end{cases} | (q, a, b, q') \in \delta \text{ and } p, p' \in Q, c, d \in \Sigma \]

\[ \Theta_5 = \begin{cases} \begin{array}{c|c} q_0b \\ \hline q_0a \end{array} \end{cases} | (q_0, a, b, q) \in \delta \]

\[ \Theta_6 = \begin{cases} \begin{array}{c|c} pb \\ \hline qa \end{array} \end{cases} | (q, a, b, q') \in \delta \text{ and } q' \in T, p \in Q \]

- **Output blocks.** The collection of output blocks \( \Theta_o \) is the union of blocks \( \Theta_7, \Theta_8, \) and \( \Theta_9 \) defined below. Since the last row of a block generated by a transducer is not necessarily accepted by the transducer (i.e., it may not be in \( I(\tau) \)) no transition type control is imposed on these tiles.

\[ \Theta_7 = \begin{cases} \begin{array}{c|c} qa & q' \end{array} \end{cases} | qa, q'b \in \Gamma \]

\[ \Theta_8 = \begin{cases} \begin{array}{c|c} qa & \end{array} \end{cases} | qa \in \Gamma \]

\[ \Theta_9 = \begin{cases} \begin{array}{c|c} qa & \end{array} \end{cases} | qa \in \Gamma \]

Assume that \( B \in \pi(L(\Theta)) \). Then \( B = \pi(P) \) for some \( n \times m \)-block \( P \in L(\Theta) \). Let \( B_j \) and \( P_j \) denote the \( j \)-th rows of \( B \) and \( P \) respectively for \( j = 1, \ldots, m-1 \). Then \( B_j = j = a_1 \cdots a_n \) and there are states \( q_1, \ldots, q_{n-1} \) such that \( P_j = q_0 a_1 \cdots q_{n-1} a_n \). By construction of \( L(\Theta) \) we have that \( (q_{i-1}, a_i, b_i, q_i) \in \delta \) for some \( b_i \in \Sigma \) for every \( i = 0, \ldots, n-1 \), and the state \( q_n \in T \). Hence the sequence \( p_j = (q_0, a_1, b_1, q_1) \cdots (q_{n-1}, a_n, b_n, q_n) \), is
an accepting path in $\tau$ such that $B_j = \mathcal{I}(p_j) \in \mathcal{I}(\tau)$. Moreover, by construction of the transition blocks $\Theta_t$ we have that $B_{j+1} = b_1 \cdots b_n$ and so, $B_{j+1} = \mathcal{O}(p_j)$. Thus, $B \in L_\tau$, i.e., $\pi(L(\Theta)) \subseteq L_\tau$.

Converse, if $B \in L_\tau$, then for each $1 \leq j < m - 1$ there is an accepting path $p_j = (q_0, a_1, b_1, q_1)(q_1, a_2, b_2, q_3) \cdots (q_{n-1}, a_n, b_n, q_n)$ of $\tau$ such that $\pi(p_j) = B_j$ and $\mathcal{O}(p_j) = B_{j+1}$. Consider a $n \times m$ block $P$ such that $P_j = q_0 a_1 \cdot q_1 a_2 \cdots q_{n-1} a_n$ for all $1 \leq j < m - 1$ and choose $P_m$ to be a word over the alphabet $\Gamma$ such that $\pi(P_m) = \mathcal{O}(p_{m-1}) = B_m$. Hence $\pi(P) = B$ and by construction $B \subseteq F_{2,2}(\Theta)$. Thus $B \in \pi(L(\Theta))$ and $\pi(L(\Theta)) = L_\tau$. □
Chapter 4

Modeling Runtime Enforcement with Mandatory Results Automata

This chapter presents a theory of runtime enforcement based on mechanism models called MRAs (Mandatory Results Automata). Their name alludes to the requirement that MRAs are obligated to return a result to the target application before seeing the next action it wishes to execute. In the MRA model, results of actions may or may not be predeterminable.

This chapter is taken from [28], which builds on [50, 51], which is in turn built on earlier work on security automata [7, 8, 46–49].

Conceptually, we wish to secure a system organized as in Figure 15a, where an application produces actions, and for every action produced, the underlying executing system (e.g., an operating system, virtual machine, or CPU) returns a result to the target application. Results may be exceptions or void or unit values, so all actions can be considered to produce results. For simplicity, the MRA model assumes all actions are synchronous; after the application produces an action \(a\), it cannot produce another action until receiving a result for \(a\). In contrast, the edit-automata model can be viewed as one in which all actions are fully asynchronous (because edit automata can buffer, without executing, an unbounded number of actions). Hence, MRAs and edit automata are extremes on a spectrum of runtime execution transformers: MRAs operate on systems having fully synchronous actions, while edit automata operate on systems having fully asynchronous actions. It may be possible to combine the semantics of MRAs and edit automata to model runtime enforcement on systems having both synchronous and asynchronous actions.

Figure 15b shows how we think of a monitor securing the system of Figure 15a. In
Figure 15b, the monitor interposes on and transforms actions and results to ensure that the actions actually executed, and the results actually returned to the application, are valid (i.e., satisfy the desired policy). The monitor may or may not be inlined into the target application.

The semantics of MRAs enables simple and flexible definitions of policies and enforcement, significantly simpler and more flexible than those of previous work. In particular, the definition of executions presented here allows policies to make arbitrary requirements on how monitors must transform actions and results. Consequently, this work’s definition of enforcement does not need an explicit notion of transparency, which previous work has considered essential for enforcement [30, 38, 49]. Transparency constrains mechanisms, forcing them to permit already-valid actions to be executed. The MRA model enables policies to specify strictly more and finer-grained constraints than transparency, thus freeing the definition of enforcement from having to hardcode a transparency requirement.

After defining MRAs and the precise circumstances under which they can be said to enforce policies, this work characterizes the policies MRAs can enforce soundly, completely, and precisely. We then generalize MRAs by introducing nondeterministic MRAs (NMRAs) and analyze their enforcement powers as well. NMRAs model situations in which external factors (such as a thread scheduler) influence a monitor’s behavior. Finally, we compare the analyses of MRAs and NMRAs to derive a hierarchy of policies they can enforce soundly, completely, and precisely.

*Summary of Contributions*  
This work develops a theory of runtime enforcement, in which monitors may transform both actions and results. It contributes:

- A simple but general model of runtime mechanisms called MRAs. MRAs appear to be the first general model of runtime mechanisms that can transform results and enforce result-sanitization policies.

- Definitions of policies and enforcement that, because they can reason about how mon-
itors transform actions and results, are significantly simpler and more expressive than existing definitions.

- A generalization of MRAs to NMRAs, which model runtime mechanisms that may execute nondeterministically (e.g., because they’re implemented in multiple threads or processes).
- An analysis of the policies MRAs and NMRAs can enforce soundly, completely, and precisely.
- A hierarchy of runtime-enforceable policies based on the previously mentioned analysis.

4.1 Mandatory Results Automata

This section builds up definitions of actions, results, executions, and operations on them.

Given an alphabet $\Sigma$, we use a nonempty set of words $A \subset \Sigma^+$ to represent the set of actions a system can execute and another nonempty set of words $R \subset \Sigma^+$ to represent the set of those actions’ results. The sets $A$ and $R$ are assumed to be computable and disjoint. An event is either an action or a result. We use $E$ to denote the set of events; $E = A \cup R$. An exchange $\xi$ is a pair of events $\langle e, e'\rangle$, consisting of an input event $e$ (i.e., an event input to the monitor) and an output event $e'$ (i.e., an event output from the monitor).

The set of all exchanges over $E$ is denoted by $\mathcal{E}$. Given sets $E$ and $E'$ of events, define $\langle E, E' \rangle = \{\langle e, e'\rangle \mid e \in E \text{ and } e' \in E'\}$ to be the set of all exchanges with an input event from $E$ and output event from $E'$.

An execution or trace is a possibly infinite sequence of exchanges. The empty execution is an execution that contains no exchanges and is denoted by $\epsilon$. The length of an execution $x$ is the number of exchanges in $x$. The binary concatenation operator $\cdot$ for finite-length executions is defined as usual. For a finite-length execution $x$ and an execution $y$, let $x \cdot y = xy$ (i.e., first exchanges in $y$ directly following the exchanges in $x$). However, if the
length of $x$ is infinite then $x \cdot y = x$.

Let $X$ and $Y$ be two sets of executions over $\mathcal{E}$. Then $XY$ is the set of all executions obtained by concatenating every execution in $X$ to every execution in $Y$. More formally, we have $XY = \{xy \mid x \in X, y \in Y\}$. Given a set of executions $X$, also define $X^0 = \{\epsilon\}$, $X^1 = X$, and $X^n = XX^{n-1}$. Further,

$$X^* = \bigcup_{i \geq 0} X^i, \quad X^+ = \bigcup_{i > 0} X^i,$$

$$X^N = \{x_1x_2 \ldots \mid x_i \in X, x_i \neq \epsilon\}, \text{ and}$$

$$X^\infty = X^* \cup X^N.$$

As usual, we identify a singleton set with the element it contains, e.g., we write $((e, e'))^N$ instead of $(\{e, e'\})^N$. Given a set of executions $X \subseteq \mathcal{E}^\infty$, define $X_f$ to be the set of finite executions in $X$, i.e. $X_f = X \cap \mathcal{E}^*$. An execution $y$ is a prefix of an (infinite) execution $x$ if $x = y$ or $x = yx'$ for some (infinite) execution $x'$. The set of all prefixes of $x$ is denoted by $\mathcal{P}(x)$. We write $y \preceq x$ if $y$ is finite prefix of $x$, and $y \prec x$ if, in addition, $x \neq y$. The relations $\succ$ and $\succeq$ are defined symmetrically.

Given a set $X \subseteq \mathcal{E}^\infty$, $\mathcal{P}(X)$ is the set of prefixes of every execution in $X$, i.e. $\mathcal{P}(X) = \bigcup_{x \in X} \mathcal{P}(x)$. For all executions $x$ and sets of executions $X$, if $x \in \mathcal{P}(X)$ then we say that $x$ is alive in $X$, or just that $x$ is alive (when $X$ is clear from context). Otherwise we say that $x$ is dead.

We model monitors that behave as in Figure 15b as MRAs.

**Definition 4.1.1** A mandatory results automaton (MRA) $M$ is a tuple $(E, Q, q_0, \delta)$, where $E$ is the event set over which $M$ operates, $Q$ is a recursively enumerable set of automaton states, $q_0$ is $M$’s initial state, and $\delta : Q \times E \rightarrow Q \times E$ is $M$’s Turing-computable (partial) transition function, which takes $M$’s current state and input event and returns $M$’s next state and output event.
Figure 15.: The interaction between an untrusted application and an executing system. In (a), an untrusted application executes actions on a system and receives results for those actions. In (b), an MRA interposes on, and enforces the validity of, the actions executed and the results returned.

We call $q_\alpha$ a configuration of MRA $M$, where $q$ is $M$’s current state and $\alpha$ is either $t$ or $s$ depending on whether $M$’s next input can come from the target application ($t$) or the executing system ($s$). The starting configuration of an MRA is $(q_0,t)$ because the monitor begins executing in its initial state and receives its first input event from the target application.

We define the operational semantics of MRAs with a labeled single-step judgment whose form is $C \xrightarrow{\xi}_M C'$. This judgment indicates that MRA $M$ takes a single step from configuration $C$ to configuration $C'$ while extending the current trace by an exchange $\xi$. Because $M$ will always be clear from context, we henceforth omit it from the judgment.

The definition of MRAs’ single-step semantics appears in Figure 16. Four inference rules define all possible MRA transitions:

1. **In-Act-Out-Act** enables the MRA to receive a new input action from the target ($\text{next}_t$ is the next action generated by the target) and, in response, output an action to the executing system.

2. **In-Act-Out-Res** enables the MRA, immediately after inputting an action $a$, to return a result $r$ for $a$ to the target.
\[ C \xrightarrow{\xi} C' \]

\[
\begin{align*}
\text{next}_t &= a \quad \delta(q,a) = (q',a') \quad \text{(In-Act-Out-Act)} \\
q_t \xrightarrow{\langle a,a' \rangle} q'_t \\
\text{next}_s &= r \quad \delta(q,r) = (q',a) \quad \text{(In-Res-Out-Act)} \\
q_s \xrightarrow{\langle r,a \rangle} q'_s \\
\text{next}_t &= a \quad \delta(q,a) = (q',r) \quad \text{(In-Act-Out-Res)} \\
q_t \xrightarrow{\langle a,r \rangle} q'_t \\
\text{next}_s &= r \quad \delta(q,r) = (q',r') \quad \text{(In-Res-Out-Res)} \\
q_s \xrightarrow{\langle r,r' \rangle} q'_s
\end{align*}
\]

Figure 16.: Single-step operational semantics of mandatory results automata.

3. **In-Res-Out-Act** enables the MRA to receive a new input result from the executing system (\(\text{next}_s\) is the next result generated by the system) and, in response, output another action to the executing system.

4. **In-Res-Out-Res** enables the MRA, immediately after inputting a result \(r\), to return a possibly different result \(r'\) to the target for the action it most recently tried to execute.

### 4.1.1 Example MRAs

We next consider a couple of example MRAs exhibiting simple, everyday sorts of behaviors found in practical monitors. The behaviors are so simple that they may seem trivial; nonetheless, the behaviors are outside existing runtime-enforcement models because they involve monitors acting on unpredictable results of actions (something neither truncation nor edit automata can do).

**EXAMPLE 12 [Spam-filtering (Result-sanitizing) MRA]** In this example, we construct an MRA to secure the interaction of an email client (the target application) with an email server (the executing system). MRA \(M\) sanitizes the results of \texttt{getMail} actions to filter out spam emails. \(M\)'s states consist of a boolean flag indicating whether \(M\) is in the process.
of obtaining email messages; $M$ begins in state 0. $M$'s transition function $\delta$ is defined by:

$$\delta(q,e) = \begin{cases} 
(0,e) & \text{if } q = 0 \text{ and } e \neq \text{getMail} \\
(1,e) & \text{if } q = 0 \text{ and } e = \text{getMail} \\
(0,\text{filter}(e)) & \text{if } q = 1
\end{cases}$$

That is, $M$ outputs its inputs verbatim and does not change its state as long as it does not input a getMail action. When $M$ does input getMail, it sets its boolean flag and allows getMail to execute. If $M$ then inputs a result $r$ for getMail (i.e., a list of messages), it outputs the spam-filtered version of $r$ and returns to its initial state. With similar techniques, $M$ could sanitize results in other ways (e.g., to remove system files from directory listings).

**EXAMPLE 13 [Dangerous-action-confirming MRA]**

This second example MRA pops up a window to confirm a dangerous action $d$ with the user before allowing $d$ to execute. We assume $d$ has a default return value $r_d$, which must be returned when the user decides not to allow $d$ to execute ($r_d$ would typically be a null pointer or a value indicating an exception). We also assume a popupConfirm action that works like a JOptionPane.showConfirmDialog method in Java, returning either an OK or cancel result. $M$ uses a boolean flag, again initially set to 0, for its state, and the following transition function.

$$\delta(q,e) = \begin{cases} 
(0,e) & \text{if } q = 0 \text{ and } e \neq d \\
(1,\text{popupConfirm}) & \text{if } q = 0 \text{ and } e = d \\
(0,r_d) & \text{if } q = 1 \text{ and } e = \text{cancel} \\
(0,d) & \text{if } q = 1 \text{ and } e = \text{OK}
\end{cases}$$

This function works as expected: $M$ outputs non-$d$ input events verbatim. Once $M$ inputs a $d$ action, it outputs a popupConfirm action and waits for a result. If the user cancels the execution of $d$, $M$ outputs result $r_d$; if the user OKs $d$, $M$ outputs and allows $d$ to execute.
Because of the simplicity in MRAs’ operational semantics, and in concrete MRA transition functions, plus the fact that MRA behaviors match our understanding of the essential behaviors of real runtime monitors, we believe that MRAs serve as a good basis for developing a theory of runtime enforcement.

4.1.2 Generalizing the Operational Semantics

Before we can formally define what it means for an MRA to enforce a policy, we need to generalize the single-step semantics to account for multiple steps. First, we define the (finite) multi-step relation, with judgment form $C \xrightarrow{x} C'$, in the standard way as the reflexive, transitive closure of the single-step relation. The trace above the arrow in the multi-step judgment gets built by concatenating, in order, every exchange labeled in the single-step judgments. Hence, $C \xrightarrow{x} C'$ means that the MRA builds execution $x$ while transitioning, using any finite number of single steps, from configuration $C$ to configuration $C'$.

We also define a judgment $M \Downarrow x$ to mean that MRA $M$, when its input events match the sequence of input events in $x$, in total produces the possibly infinite-length execution $x$. Formally, judgment $M \Downarrow x$ is defined as follows: for a possibly infinite-length execution $x \in \mathcal{E}^\infty$, $M \Downarrow x$ iff there exists a sequence of $M$-configurations $C_0, C_1, C_2, \ldots$ such that for any $n$-length prefix $x'$ of $x$ we have

$$C_0 \xrightarrow{\xi_1} C_1 \xrightarrow{\xi_2} \cdots \xrightarrow{\xi_n} C_n = C_0 \xrightarrow{x'} C_n.$$

The above-mentioned sequence of $M$-configurations $C_0, C_1, C_2, \ldots$ is called the run of $M$ producing $x$. When $M$ is clear from the context, the run of $M$ producing $x$ is denoted by $R_x$. Furthermore, if $C_n = (q)_{\alpha}$ for some $\alpha \in \{t, s\}$ then we write $q_0x' = q$. Finally, given an MRA $M$, the language of $M$, written $\mathcal{L}(M)$, is defined to be $\{x \mid M \Downarrow x\}$. 

4.2 MRA-based Enforcement

This section defines what it means for an MRA to enforce a policy.

4.2.1 Policies and Properties

A policy is a predicate on sets of executions [57]. Policy $P$ is a property iff there exists a predicate $\hat{P}$ over $E^\infty$ such that $\forall X \subseteq E^\infty : (P(X) \iff \forall x \in X : \hat{P}(x))$. Because there is a one-to-one correspondence between a property $P$, its predicate $\hat{P}$, and the set of executions that satisfy $P$ (i.e., the set $X$ such that $\forall x \in X : \hat{P}(x)$), the rest of this work uses $P$ unambiguously to refer to any of the three depending on the context.

Intuitively, policies can determine whether a set of target executions is valid based on the executions’ relationships with one another, but properties cannot take such inter-execution relationships into account. It is sometimes possible for runtime mechanisms to enforce nonproperty policies: a monitor could refer to earlier traces (e.g., saved in files) when deciding how to transform the current execution, or it could monitor multiple executions of a program concurrently [22]. For simplicity, this work analyzes only the properties MRAs can enforce; we assume monitors make decisions about a single execution at a time.

There are two important differences between this work’s definition of policies and the definitions in previous models. The differences arise from the way executions are modeled here: instead of modeling executions as just the actions a monitor outputs, the MRA model also includes (1) output results, and (2) all input events, in executions. Because policies here may take output results into account, they can specify constraints on which results may be returned to targets; policies here may require results to be sanitized. For example, the spam-filtering MRA from Section 4.1.1 enforces a policy requiring all results of getMail actions to be filtered (this policy is a property because it is satisfied iff every execution in a set $X$ has exactly zero spam-containing results of getMail actions).

Moreover, because policies in the MRA model can take input events into account, poli-
cies here can require arbitrary relationships to hold between input and output events. For example, a property $P$ could be dissatisfied by execution $⟨\text{shutdown}, e⟩$ (i.e., $⟨\text{shutdown}, e⟩ \not\in P$) unless $e = \text{popupConfirm}$. To enforce this $P$, an MRA may have no choice but to output $\text{popupConfirm}$ upon inputting a $\text{shutdown}$ action. Policies in previous models (e.g., truncation and edit automata) could not specify such relationships between input and output events because the policies were predicates over output executions only. The primary relationship allowed between input and output events in previous models was transparency, which was hardcoded into the definition of enforcement [38, 49] and required monitors to output valid inputs unchanged. Transparency can be encoded in policies in the MRA model (by defining policies to be satisfied only by executions in which valid inputs get output unchanged), but policies here are strictly more expressive than transparency because they can specify arbitrary input-output relationships. For example, the popup-confirmation policy above specifies a relationship that is outside the scope of transparency (because there is no requirement for $\text{shutdown}$ to be output unchanged).

4.2.2 Enforcement

Unlike previous models of security automata, the current model defines enforcement in terms of soundness and completeness, which are standard principles in the broader fields of security and verification. An MRA $M$ is sound with respect to property $P$ whenever $M$ only produces traces satisfying $P$; $M$ is complete with respect to $P$ whenever it produces all traces satisfying $P$; and $M$ is precise with respect to $P$ whenever it is sound and complete with respect to $P$.

**Definition 4.2.1** An MRA $M$

- *soundly enforces* $P$ whenever $\mathcal{L}(M) \subseteq P$;

- *completely enforces* $P$ whenever $P \subseteq \mathcal{L}(M)$, and

- *precisely enforces* $P$ whenever $\mathcal{L}(M) = P$. 
Definition 4.2.1 is significantly simpler and more flexible than definitions of enforcement in related work, because it (1) does not hardcode transparency-style requirements and (2) defines complete and precise, in addition to sound, enforcement.

As is standard, sound enforcement permits false positives (i.e., false alarms) but not false negatives. Conversely, complete enforcement permits false negatives but not false positives. Precise enforcement permits neither false positives nor false negatives.

For some examples of MRA enforcement, let’s consider the policies enforced by the example MRAs from Section 4.1.1. Because these policies match behaviors of MRAs that automata in previous models cannot mimic, the policies defined below cannot be enforced by previously studied security automata.

EXAMPLE 14 [Policies enforced by the MRA from Example 12] Consider the spam-filtering MRA M from Example 12. If
- \(X = \{\langle \text{getMail}, \text{getMail} \rangle \langle r, \text{filter}(r) \rangle \mid r \in R\}\) and
- \(Y = \{\langle a, a \rangle \langle r, r \rangle \mid a \in A, r \in R, \text{ and } a \neq \text{getMail}\}\),

then the set of the executions produced by M is \(P = \mathcal{P}((X \cup Y)\infty)\), so M precisely enforces \(P\), completely enforces every subset of \(P\), and soundly enforces every superset of \(P\).

EXAMPLE 15 [Policies enforced by the MRA from Example 13] Let M be the dangerous-action-confirming MRA from Example 13. If
- \(X = \{\langle d, \text{popupConfirm} \rangle \langle \text{OK}, d \rangle \langle r, r \rangle \mid r \in R\}\),
- \(Y = \{\langle d, \text{popupConfirm} \rangle \langle \text{cancel}, r_d \rangle \}\), and
- \(Z = \{\langle a, a \rangle \langle r, r \rangle \mid a \in A, r \in R, \text{ and } a \neq d\}\),

then the set of executions produced by M is \(P = \mathcal{P}((X \cup Y \cup Z)\infty)\). Again, M precisely enforces \(P\), completely enforces every subset of \(P\), and soundly enforces every superset of \(P\).
4.3 Analysis of MRA-enforceable Policies

This section analyzes the properties that MRAs can enforce soundly, completely, and precisely. Throughout this section, and for the remainder of the work, we assume that we’re dealing with a system having action set $A$ and result set $R$.

The following definition and lemma describe the structure of executions that can be observed during a run of an MRA. Executions are constrained in part because MRAs may only input a result (action) after outputting an action (result or nothing, initially).

**Definition 4.3.1** The set

$$\mathcal{P}((\langle A, R \rangle^\infty (A, A) \langle R, A \rangle^\infty (R, R))^\infty)$$

is called the **MRA execution universe** and is denoted by $U$.

MRAs can only produce executions in $U$.

**Lemma 4.1** $\forall M : \mathcal{L}(M) \subseteq U$

**Proof.** The claim is a direct consequence of MRAs’ single-step semantics. The initial transition of an MRA must be either In-Act-Out-Act or In-Act-Out-Res, and subsequent transitions must be ordered as described in Figure 17. The executions produced by making
Figure 18.: A graph whose paths correspond to executions in $U$.

transitions in the orders depicted in Figure 17 correspond to paths in the graph depicted in Figure 18, which in turn correspond to executions satisfying the $\infty$-regular expression defining $U$. Hence, every trace built by an MRA must be an element of $U$. □

Definition 4.3.2 presents some interesting classes of properties, including exchange-based versions of safety [44] and liveness [2].

**Definition 4.3.2** A property $P$ is called

- **exchange prefix closed**, whenever $\forall x \in E^\infty : (x \in P \Rightarrow P(x) \subseteq P)$.
- **exchange omega closed**, whenever $\forall x \in E^\omega : (P(x)_f \subseteq P \Rightarrow x \in P)$.
- **exchange safety**, whenever $\forall x \in E^\infty : (x \notin P \Rightarrow \exists x' \preceq x : \forall y \succeq x' : y \notin P)$.
- **MRA liveness**, whenever $\forall x \in U_f : \exists y \succeq x : y \in P$.

Let PC be the set of all exchange-prefix-closed properties, OC the set of exchange-omega-closed properties, ES the set of exchange-safety properties, and ML the set of MRA-liveness properties.

Safety properties can alternatively be defined as the intersection of the prefix-closed and omega-closed properties.

**Lemma 4.2** $ES = PC \cap OC$
Proof. (⊆) Let \( P \in ES \) be an exchange-safety property. We show that \( P \in PC \). Assume that \( x \in P \), but \( P(x) \not\subseteq P \). Then there exists a prefix \( x' \) of \( x \) such that \( x' \notin P \). Because \( P \in ES \), there is \( x'' \preceq x' \) such that \( \forall y \succeq x'', y \notin P \). Notice that \( x \succeq x'' \), and so \( x \notin P \) which is a contradiction. This shows that \( P \in PC \).

We next show that \( P \in OC \). Assume that for some \( x \in E_\omega \), \( P(x) \not\subseteq P \) but \( x \notin P \). Then because \( P \) is in ES, there exists \( x' \preceq x \) such that \( x' \notin P \), which contradicts the assumption that \( P(x) \not\subseteq P \). This shows that \( P \in OC \).

(⊇) Let \( P \in PC \cap OC \). Assume that \( x \in E_\infty \) and \( x \notin P \). If \( x \in E^* \), then for all \( y \succeq x \), \( y \notin P \); if a \( y \succeq x \) were in \( P \), then because \( P \in PC \), we would have \( x \in P \) (a contradiction). On the other hand, if \( x \in E_\omega \), then because \( P \in OC \) and \( x \notin P \), \( \exists x' \preceq x \) such that \( x' \notin P \), so by the same reasoning used in the previous sentence, \( \forall y \succeq x', y \notin P \). Hence, \( P \in ES \).

As in previous enforcement models, the execution universe here is a safety property.

**Lemma 4.3** \( U \in ES \)

**Proof.** The traces of MRA execution universe \( U \) correspond to possibly infinite paths in the graph \( G \) depicted in Figure 18. \( U \) is prefix closed because for every path \( x \) in \( G \), all prefixes of \( x \) must also be paths in \( G \). \( U \) is omega closed because for all infinite-length executions \( x \), if all the finite prefixes of \( x \) are paths in \( G \) then so must be \( x \). \( U \) is therefore in \( PC \cap OC \), which equals \( ES \) by Lemma 4.2.

Let MS (MC, MP) denote the set of security properties soundly (completely, precisely) enforceable by MRAs. To establish succinct characterizations of MS, MC, and MP, we define two additional sets of properties as follows.

**Definition 4.3.3** A property \( P \) is called *reasonable* when (a) \( \epsilon \in P \), (b) \( P \subseteq U \), and (c) \( \mathcal{P}(P)_f \) is a recursively enumerable set. The set of all reasonable properties is denoted by \( RS \).
A property $P$ is called deterministic when, for all finite-length executions $x$ and events $e$, there exists at most one exchange $\xi = \langle e, e' \rangle$ such that $x\xi$ is alive in $P$. The set of all deterministic properties is denoted by $DT$.

The next theorem shows that MRAs precisely enforce exactly those properties that are reasonable, deterministic, and exchange safety. The proof is constructive; it shows how to create an MRA to precisely enforce any such property. For example, given the spam-filtering property defined in Example 14 of Section 4.2.2, the proof constructs an MRA $M = (E, Q, q_0, \delta)$, where $\delta(q, e)$ is defined to be $(q, \text{filter}(e))$ when $q$ ends with $\langle \text{getMail}, \text{getMail} \rangle$ and $(q, e, e)$ otherwise. This $M$ precisely enforces $P$ and is functionally equivalent to the spam-filtering MRA defined in Example 12 of Section 4.1.1.

**Theorem 4.1** $MP = RS \cap DT \cap ES$

**Proof.** ($\subseteq$) Suppose that $P$ is precisely enforceable and let $M = (E, Q, q_0, \delta)$ be an MRA with $L(M) = P$.

- **$P$ is reasonable.** If the target never outputs an action, then $M$ never makes a transition and so produces the empty execution. This shows that $\epsilon$ is always in $L(M) = P$. Also, according to Lemma 4.1, $P \subseteq U$. Finally, $P_f = P(L(M))$ can be enumerated because (1) the events in $E$ can be enumerated, and (2) $\delta$ is Turing-computable.

- **$P$ is deterministic.** For all finite-length executions $x$ and events $e$ and $e'$, if $q = q_0, x$ and $x\langle e, e' \rangle$ is alive in $P$, then $\delta(q, e) = (q', e')$ (for some state $q'$), implying that $P$ is deterministic.

- **$P$ is exchange-safety.** $L(M) = P$ is in PC because for all executions $x$ in $L(M)$, $M$ produces all prefixes of $x$ on its way to producing $x$ itself. $L(M) = P$ is also in OC because $M$ is deterministic, so if $M$ produces all the finite prefixes of an infinite
execution $x$, then $M$ must also produce $x$ (because $M$, being deterministic, always passes through the same configurations used to produce $x'$ when producing $x$, if $x' \preceq x$). Hence, $P$ is in $PC \cap OC = ES$.

(⊇) Consider a property $P$ that is reasonable, deterministic, and exchange-safety (meaning prefix and omega closed). Let $M = (E, E^*, \epsilon, \delta)$ be an MRA with

$$\delta(q, e) = (q\langle e, e'\rangle, e')$$

if $q\langle e, e'\rangle$ is alive in $P$.

Observe that $\delta$ is a Turning-computable function because $P$ is reasonable, prefix closed, and deterministic. Thus $M$ is well defined. Next we show that $\mathcal{L}(M) = P$.

First observe that $\mathcal{L}(M) \subseteq P$ because (1) the definition of $\delta$ ensures that all finite executions produced by $M$ are alive in $P$, (2) $P$ is prefixed closed, and (3) $P$ is omega closed.

To see that $P \subseteq \mathcal{L}(M)$, first consider an infinite execution $x = \xi_0\xi_1\xi_2 \ldots$ in $P$. For all $n \geq 0$, let $C_n = (x_n)_\alpha$, where (1) $x_n$ is the $n$-length prefix of $x$, and (2) $\alpha$ is $t$ if $\xi_n$ begins with an action and $s$ otherwise. Because $\forall n \geq 0 : C_n \xrightarrow{\xi_n} C_{n+1}$ (by the definition of $\delta$ and the fact that $P \subseteq U$), we have that $M$ produces $x$, so $x \in \mathcal{L}(M)$. A similar argument shows that all finite $x$ in $P$ are also in $\mathcal{L}(M)$, so $P \subseteq \mathcal{L}(M)$.

□

The next corollary shows that a property $P$ is soundly enforceable by an MRA iff $P$ contains $\epsilon$.

**Corollary 4.3.1** $MS = \{P \mid \epsilon \in P\}$

**Proof.** (⊆) If $P$ is soundly enforceable, then there exists an MRA $M$ such that $\mathcal{L}(M) \subseteq P$. By Theorem 4.1, $\mathcal{L}(M)$ is reasonable, so $\epsilon \in \mathcal{L}(M)$. Hence, $\epsilon \in P$ as required.

(⊇) Consider a property $P$ containing $\epsilon$. Observe that the property $\{\epsilon\}$ is reasonable, deterministic, and exchange safety, and hence precisely enforceable by Theorem 4.1. Because every superset of a precisely enforceable property is soundly enforceable, $P$ is soundly enforceable. □
In terms of complete enforcement, a property $P$ is enforceable by MRAs iff $P$ is a subset of a reasonable and deterministic property.

**Corollary 4.3.2** $MC = \{ P \mid \exists P' \supseteq P : P' \in RS \cap DT \}$

**Proof.** $(\subseteq)$ If $P$ is completely enforceable, then there exists an MRA $M$ such that $P \subseteq L(M)$. Note that by Theorem 4.1, $L(M)$ is deterministic and reasonable.

$(\supseteq)$ Let $P$ be a subset of a deterministic and reasonable property $P'$. Define

$$\hat{P} = \mathcal{P}(P') \cup \{ x \in E^\omega \mid \mathcal{P}(x)_f \subseteq \mathcal{P}(P') \}.$$ 

Note that if $\hat{P}$ is precisely enforceable, then $P$ is completely enforceable because $P \subseteq P' \subseteq \hat{P}$. Thus we finish the proof by showing that $\hat{P}$ is deterministic, reasonable, and exchange safety, and hence precisely enforceable by Theorem 4.1.

- $\hat{P}$ is deterministic. Suppose that for a finite execution $x$ and an event $e$, there are two events $e', e''$ such that $x(e, e')$ and $x(e, e'')$ are alive in $\hat{P}$. By the definition of $\hat{P}$, these executions are prefixes of executions in $P'$ and so are alive in $P'$. Because $P'$ is deterministic, $e' = e''$. This proves determinism of $\hat{P}$.

- $\hat{P}$ is reasonable. First note that $\hat{P}$ is a superset of $P'$, which is reasonable and therefore contains $\epsilon$. In addition, $P' \subseteq U$, and because $U$ is omega closed (by Lemmas 4.2–4.3), $\{ x \in E^\omega \mid \mathcal{P}(x)_f \subseteq U \} \subseteq U$. Using these observations and the definition of $\hat{P}$,

$$\hat{P} = \mathcal{P}(P') \cup \{ x \in E^\omega \mid \mathcal{P}(x)_f \subseteq \mathcal{P}(P') \}$$

$$\subseteq \mathcal{P}(U) \cup \{ x \in E^\omega \mid \mathcal{P}(x)_f \subseteq \mathcal{P}(U) \}$$

$$\subseteq U \cup \{ x \in E^\omega \mid \mathcal{P}(x)_f \subseteq U \}$$

$$\subseteq U.$$ 

Finally, the set $\mathcal{P}(\hat{P})_f$ equals $\mathcal{P}(P')_f$ and so is recursively enumerable.
4.4 Non-deterministic MRAs

Since the behavior of an MRA is determined by its transition function, MRAs model deterministic security monitors. Although this is an adequate constraint for most real-life security monitors, there are situations when it is beneficial to consider some redundancy (i.e., non-determinism) in properties and monitors. Such situations arise naturally when external factors may influence whether an execution is valid, or how a monitor behaves. For example, a multi-threaded monitor’s behavior may be subject to an external thread scheduler; the properties enforceable by multi-threaded monitors may allow for such non-determinism. As another example, monitors may be influenced by auxiliary, unpredictable inputs, such as readings from weather sensors or spontaneous human input.

The aim of this section is to introduce a model of generic non-deterministic security monitors and investigate the properties they can enforce.

**Definition 4.4.1** A non-deterministic MRA (NMRA) $N$ is a tuple $(E, Q, I, \delta)$, where $E$ is the event set over which $N$ operates, $Q$ is a recursively enumerable set of automaton states, $I \subseteq Q$ is a recursively enumerable set of $N$’s initial states, and $\delta \subseteq Q \times E \times Q \times E$ is a recursively enumerable transition relation.

The single-step ($\rightarrow$), multi-step ($\rightarrow^*$) and production ($\Downarrow$) relations are defined for NM-RAs in the same ways as for MRAs (though for NM-RAs, the single-step rules of Figure 16 would have to replace premises of the form $\delta(q, e) = (q', e')$ with premises of the form $(q, e, q', e') \in \delta$).

Let NS (NC, NP) denote the collection of security properties soundly (completely, precisely) enforced by NM-RAs. We now investigate the security properties NM-RAs soundly,
completely, and precisely enforce.

First, as with MRAs, NMRAs can only produce traces in the universe $U$.

**Lemma 4.4** $\forall N : \mathcal{L}(N) \subseteq U$

**Proof.** Observe that the proof of Lemma 4.1 is applicable to NMRAs, and so also proves the claim of this lemma. $\square$

We next consider NMRAs as precise enforcers and find that for every reasonable, prefix- and omega-closed property $P$, there exists an NMRA that precisely enforces $P$. Moreover, if there exists an NMRA that precisely enforces $P$, then $P$ is reasonable and prefix closed. These inclusions are strict, so we can write this result formally as $(RS \cap PC \cap OC) \subset NP \subset (RS \cap PC)$. By Lemma 4.2, an equivalent way to state this result is as follows.

**Theorem 4.2** $RS \cap ES \subset NP \subset RS \cap PC$

**Proof.** We start by proving $RS \cap ES \subseteq NP$. Let $P$ be a property in $RS \cap ES$, $T$ be a Turing Machine that enumerates $\mathcal{P}(P)_f$ (which must exist because $P \in RS$), and $N = (E, Q, I, \delta)$ be an NMRA such that

- $Q = \{x_i \mid x \in \mathcal{E}^* \text{ and } i \in \mathbb{N}\}$,
- $I = \{\epsilon_i \mid i \in \mathbb{N}\}$, and
- $\delta$ is defined as follows. Given $x_i$ and $e$, run $T$ for $i$ steps. If the final execution output by $T$ is $x\langle e, e'\rangle$, for some event $e'$, then $(x_i, e, x\langle e, e'\rangle_j, e') \in \delta$, for all $j \in \mathbb{N}$.

Note that $\delta$’s definition implies that it is recursively enumerable, so $N$ is well defined. Next we show that $\mathcal{L}(N) = P$.

First observe that $\mathcal{L}(N) \subseteq P$ because (1) the definition of $\delta$ ensures that all finite executions produced by $N$ are alive in $P$, (2) $P$ is prefixed closed, and (3) $P$ is omega closed.

To see that $P \subseteq \mathcal{L}(N)$, first consider an infinite execution $x = \xi_0\xi_1\xi_2 \ldots$ in $P$. For all $n \geq 0$, let $C_n = ((x_n)i)_{\alpha}$, where (1) $x_n$ is the $n$-length prefix of $x$, (2) $i$ is such that $T$
outputs the \((n + 1)\)-length prefix of \(x\) after \(i\) iterations, and (3) \(\alpha\) is \(t\) if \(\xi_n\) begins with an action and \(s\) otherwise. Because \(\forall n \geq 0 : C_n \xrightarrow{\xi_n} C_{n+1}\) (by the definition of \(\delta\) and the fact that \(P \subseteq U\)), we have that \(N\) produces \(x\), so \(x \in \mathcal{L}(N)\). A similar argument shows that all finite \(x\) in \(P\) are also in \(\mathcal{L}(N)\), so \(P \subseteq \mathcal{L}(N)\). This completes the proof that \(RS \cap ES \subseteq NP\).

To prove that \(NP \subseteq RS \cap PC\), suppose that \(P\) is precisely enforceable and let \(N = (E, Q, I, \delta)\) be an NMRA with \(\mathcal{L}(N) = P\).

- **P is reasonable.** If the target never outputs an action, then \(N\) never makes a transition and so produces the empty execution. This shows that \(\epsilon\) is always in \(\mathcal{L}(N) = P\). Also, according to Lemma 4.4, \(P \subseteq U\). Finally, \(\mathcal{P}(P)_f = \mathcal{P}(\mathcal{L}(N))_f\) can be enumerated because (1) the events in \(E\) can be enumerated, and (2) \(\delta\) is recursively enumerable.

- **P is prefix closed.** \(\mathcal{L}(N) = P\) is in \(PC\) because for all executions \(x\) in \(\mathcal{L}(N)\), \(N\) produces all prefixes of \(x\) on its way to producing \(x\) itself.

We now prove that both inclusions are strict. First we give an example of a property \(\bar{P}\) precisely enforceable by an NMRA that is not exchange-safety. Define \(N = (E, \mathbb{N}, \mathbb{N}, \delta)\) such that \(\delta = \{(i, e, i-1, e') \mid e, e' \in E \text{ and } i > 1\}\). Then \(\bar{P} = \mathcal{L}(N) = U_f\), but \(U_f\) is not omega closed and therefore not in \(ES\).

Finally, because (1) \(RS \cap PC\) is uncountable, and (2) \(NP\) is countable, we have that \((RS \cap PC) \setminus NP\) is nonempty (and uncountable). (1) holds because properties in \(RS \cap PC\) need not be omega closed and may therefore contain arbitrary subsets of infinite-length executions (as long as the subsets don’t violate reasonableness or prefix closure). (2) holds because the set of NMRA\'s is countable (events, states, initial states, and transitions are all recursively enumerable).

As a corollary of Theorem 4.2, there exists an NMRA that produces \(U\). In contrast, although all MRAs produce subsets of \(U\), no MRA can produce \(U\) itself (because \(U\) is nondeterministic).

**Corollary 4.4.1** \(\exists N : \mathcal{L}(N) = U\)
Proof. \( U \) is reasonable (by the definitions of reasonable and \( U \)) and exchange safety (by Lemma 4.3), so \( U \) is precisely enforceable by an NMRA (by Theorem 4.2).

As with MRAs, a property \( P \) is soundly enforceable by an NMRA iff \( P \) contains \( \epsilon \).

**Corollary 4.4.2** \( \text{NS} = \{P \mid \epsilon \in P\} \)

**Proof.** \((\subseteq)\) If \( P \) is soundly enforceable, then there exists an NMRA \( N \) such that \( \mathcal{L}(N) \subseteq P \). By Theorem 4.2, \( \mathcal{L}(N) \) is reasonable, so \( \epsilon \in \mathcal{L}(N) \). Hence, \( \epsilon \in P \) as required.

\((\supseteq)\) If \( \epsilon \in P \), then by Corollary 4.3.1, \( P \in \text{MS} \). Because every MRA is an NMRA, \( P \) is also in NS. \( \square \)

In terms of complete enforcement, a property \( P \) is enforceable by NMRAs iff \( P \) is a subset of a reasonable property.

**Corollary 4.4.3** \( \text{NC} = \{P \mid \exists P' \supseteq P : P' \in \text{RS}\} \)

**Proof.** \((\subseteq)\) If \( P \) is completely enforceable, then there exists an NMRA \( N \) such that \( P \subseteq \mathcal{L}(N) \). Note that by Theorem 4.2, \( \mathcal{L}(N) \) is reasonable.

\((\supseteq)\) Assume that \( P \) is a subset of a reasonable property and is therefore also a subset of \( U \). By Corollary 4.4.1, there exists an NMRA that precisely enforces \( U \), so that same NMRA completely enforces \( P \). \( \square \)

### 4.5 A Hierarchy of Reasonable Properties

This section ties together the results of previous sections, to establish relationships between various classes of reasonable properties. In particular, Corollaries 4.4.4–4.4.6 cast the results of earlier theorems and corollaries into the domain of the reasonable properties.

For notational convenience, let \( \text{RX}X \) denote \( \text{RS} \cap X.X \), where \( X.X \) is a class of properties. For example, RMS is \( \text{RS} \cap \text{MS} \), the set of reasonable properties that can be soundly enforced by MRAs.
**Corollary 4.4.4** $RS = RMS = RNS = RNC$ and $RMC = RDT$

**Proof.** Because all reasonable properties contain $\epsilon$, Corollaries 4.3.1 and 4.4.2 imply that $RS = RMS = RNS$. Also, Corollary 4.4.3 implies that $RS \subseteq NC$, so $RS = RNC$ as well. Finally, Corollary 4.3.2 implies that $RDT \subseteq MC \subseteq DT$ (the latter inclusion because subsets of deterministic properties must be deterministic), so $RMC = RDT$. □

**Corollary 4.4.5** $MP = RDT \cap RES$ and $RES \subset NP \subset RPC$

**Proof.** Immediate by Theorems 4.1–4.2. □

**Corollary 4.4.6** $RMC \cap RML = \emptyset$ and $RES \cap RML = \{U\}$

**Proof.** As shown in the proof of Corollary 4.4.4, $MC \subseteq DT$. Also note that for all properties $P \in ML$, all executions in $U_f$ must be alive in $P$, implying that $DT \cap ML = \emptyset$. Hence, $MC \cap ML = \emptyset$, so $RMC \cap RML = \emptyset$. In addition, if $P \in ES \cap ML$, then every execution in $U_f$ must be alive in $P$, yet $P$ must be prefix closed (hence $U_f \subseteq P$) and omega closed (hence $U \subseteq P$). The only reasonable property $P$ such that $U \subseteq P$ is $U$ itself, so $RES \cap RML = \{U\}$. □

Figure 19 summarizes the results of Corollaries 4.4.4–4.4.6. The nine example properties depicted in Figure 19 can be defined as follows.

*Property 1.* Property 1 could be any nondeterministic, non-prefix-closed, non-liveness property. For example, consider the following sets of executions.

- $X = \{\langle d, \text{popupConfirmError}\rangle \langle OK, d \rangle \langle r, r \rangle \mid r \in R\}$
- $Y = \{\langle d, \text{popupConfirmError}\rangle \langle \text{cancel}, r_d \rangle\}$
- $X' = \{\langle d, \text{popupConfirmWarning}\rangle \langle OK, d \rangle \langle r, r \rangle \mid r \in R\}$
- $Y' = \{\langle d, \text{popupConfirmWarning}\rangle \langle \text{cancel}, r_d \rangle\}$

Figure 19 summarizes the results of Corollaries 4.4.4–4.4.6. The nine example properties depicted in Figure 19 can be defined as follows.
Let $P = (X \cup Y \cup X' \cup Y')^\infty$. This property allows monitors to respond to dangerous events by confirming them with either error or warning messages; according to the property, users must respond to the confirmation messages. Note that $P$ is reasonable but nondeterministic (because both $\langle d, \text{popupConfirmError} \rangle$ and $\langle d, \text{popupConfirmWarning} \rangle$ are alive), non-prefix-closed (because $\langle d, \text{popupConfirmError} \rangle$ is invalid but alive), and non-liveness (because execution $\langle d, d \rangle$ is not alive).

*Property 2. Reusing the definitions of $X$ and $Y$ from Property 1, let $P = (X \cup Y)^\infty$. As before, $P$ is reasonable, non-prefix-closed, and non-liveness. However, $P$ is now deterministic because the choice of error versus warning messages has been removed.

*Property 3. Consider the NMRA $N$ used to define $\bar{P}$ in Theorem 4.2 (i.e., $\mathcal{L}(N) = U_f$). Removing from $N$‘s transition relation all tuples of the form $(i, e, i-1, e')$ such that $e \neq e'$ guarantees that $\mathcal{L}(N)$ is (1) deterministic (because there is exactly one valid output for every input), (2) in NP (because $N$ is an NMRA), (3) not in RES (because $\mathcal{L}(N)$ remains non-omega-closed), and (4) not in RML (because the execution $\langle e, e' \rangle$, where $e \neq e'$, is dead).
*Property 4.* We have already seen properties in MP: Examples 14 and 15 defined properties precisely enforced by the MRAs constructed in Examples 12 and 13.

*Property 5.* Property 5 could be any nondeterministic exchange-safety property. For example,

\[ P = \{ \epsilon, \langle \text{fopen, fopen} \rangle, \langle \text{fopen, logfopen} \rangle \} \]

is in RES \ MP.

*Property 6.* Per Corollary 4.4.6, Property 6 is \( U \).

*Property 7.* Property 7 could be \( U_f \), which is a nondeterministic, non-safety, liveness property that is in NP (as shown in the proof of Theorem 4.2).

*Property 8.* Property 8 could be any non-prefix-closed liveness property, e.g., a minimum-activity property forbidding exactly those executions of length between 1 and \( i \) (for some fixed \( i > 0 \)).

*Property 9.* Returning to the NMRA \( N \) used to define \( \bar{P} \) in Theorem 4.2 (i.e., \( L(N) = U_f \)), remove from \( N \)'s transition relation all tuples of the form \( (i, e, i-1, e) \). Then assuming that \( E \) contains multiple actions and results, \( L(N) \) is nondeterministic, non-safety (because \( L(N) \) remains non-omega-closed), and non-liveness (because \( \langle e, e \rangle \) is dead), but still in NP (because \( N \) is an NMRA).

To summarize the enforcement capabilities of MRAs and NMRAs:

- MRAs precisely enforce deterministic properties that are reasonable and prefix and omega closed, while NMRAs can precisely enforce all properties that are reasonable and prefix and omega closed. NMRAs can also precisely enforce countably many of the uncountable properties that are reasonable and prefix closed but not omega closed.
Both MRAs and NMRAs soundly enforce all reasonable properties.

Although NMRAs completely enforce all reasonable properties, MRAs completely enforce only the deterministic reasonable properties.

In terms of (reasonable) safety and liveness properties, we’ve found that MRAs precisely enforce a strict subset of safety properties (i.e., the ones that are also deterministic), while NMRAs precisely enforce a strict superset of safety properties. However, this analysis would change under alternative definitions of safety and liveness. For example, one could define safety and liveness on an event-by-event basis, rather than the exchange-based definitions used in this work. Under an event-based definition of safety, the property $P = \{e, (a, a)\}$ could be considered non-safety because it specifies that just inputting $a$, without also outputting $a$, is invalid but can be rectified by outputting $a$. MRAs can precisely enforce event-based non-safety properties like this $P$, as long as the non-safety is always within exchanges.

More important than the low-level analysis of properties enforceable by (N)MRAs are the higher-level ideas of including monitors’ input and output events on traces and defining enforcement in terms of soundness and completeness. With these techniques, policies become monitor centric (i.e., they specify valid/invalid monitor behaviors), versus the target-centric policies of earlier security-automata and formal-verification frameworks (which specify valid/invalid target behaviors). In the domain of runtime enforcement, monitor-centric policies have two primary advantages over target-centric policies: (1) monitor-centric policies are more expressive because they can require arbitrary relationships to hold between monitors’ inputs and outputs, and (2) monitor-centric policies simplify definitions of enforcement because the definitions no longer have to hardcode particular input-output relationships.
Chapter 5

Conclusion

With this work we initiate studies of cellular automata through the sets of two-dimensional blocks extracted from their space-time diagrams. We started the investigation with a class of cellular automata (factorial-local cellular automata) that give rise to factorial-local two-dimensional languages (a foundational class in hierarchy of two-dimensional factorial languages [35]). The class of factorial-local cellular automata falls in between the class of cellular automata with the shadowing property and cellular automata with SFT traces. For example, there are factorial-local cellular automata that have both, strictly sofic and SFT traces (Example 9). Further, rule 192 cellular automaton (Example 7) is factorial-local, but not nilpotent, as it has at least two fixed-point configurations (all 0s and all 1s). It turns out that factorial-local cellular automata have the same characterization as one-sided automata with SFT traces (Proposition 3.3, Proposition 3.6 in [24] or Proposition 3.4). In [12] it is proven that there are regular cellular automata without shadowing property and Example 10 shows a cellular automaton that has the shadowing property but is not factorial-local.

We believe that a large number of regular cellular automata are factorial-local. We used the algorithm outlined in Proposition 3.2.1 to write a program capable of enumerating factorial-local elementary cellular automata (with radius 1 over the alphabet \{0, 1\}). We found that 144 out of 256 elementary cellular automata are factorial-local with the size of allowed blocks at most $5 \times 3$. Our results and the source code of the program are available at http://edolzhen.myweb.usf.edu/.

On the other hand, two-dimensional picture languages generated by transducers (introduced in [25] and further studied in [27]) form a subclass of recognizable picture languages
which in turn include the factorial-local languages. It turns out that there are factorial-local languages that are not generated by cellular automata. Consider for example the language that consists of all rectangular blocks of 0s and 1s where each appearance of symbol 1 is surrounded by 0s. Such a language cannot be generated by a cellular automaton because it would require any such automaton to output (a) a sequence of 0s and also (b) a sequence of 0s with scattered 1s on input consisting of a bi-infinite sequence of 0s. In addition to this, Examples 2 and 6 present factorial-local languages that are not generated by cellular automata.

Further, there are cellular automata generated languages which are not transducer generated because every row in a block of the picture language $F_G$ generated by a cellular automaton $G$ depends not just on the previous row of the block (as is the case in transducer generated languages), but on the previous row extended on both sides by the size of the radius.

Our findings are summarized in Figure 20. The diagram on the left of this figure shows the relationship between different classes of cellular automata. The diagram on the right shows the relationship between classes of two-dimensional picture languages.

In the second part of this work we’ve studied an automata-model of runtime security monitors called Mandatory Results Automata. To summarize the enforcement capabilities
of MRAs and NMRAs:

- MRAs precisely enforce deterministic properties that are reasonable and prefix and omega closed, while NMRAs can precisely enforce all properties that are reasonable and prefix and omega closed. NMRAs can also precisely enforce countably many of the uncountable properties that are reasonable and prefix closed but not omega closed.
- Both MRAs and NMRAs soundly enforce all reasonable properties.
- Although NMRAs completely enforce all reasonable properties, MRAs completely enforce only the deterministic reasonable properties.

In terms of (reasonable) safety and liveness properties, we’ve found that MRAs precisely enforce a strict subset of safety properties (i.e., the ones that are also deterministic), while NMRAs precisely enforce a strict superset of safety properties. However, this analysis would change under alternative definitions of safety and liveness. For example, one could define safety and liveness on an event-by-event basis, rather than the exchange-based definitions used in this work. Under an event-based definition of safety, the property \( P = \{ \epsilon, \langle a, a \rangle \} \) could be considered non-safety because it specifies that just inputting \( a \), without also outputting \( a \), is invalid but can be rectified by outputting \( a \). MRAs can precisely enforce event-based non-safety properties like this \( P \), as long as the non-safety is always within exchanges. Our findings are summarized by the hierarchy of security properties presented in Figure 19.

More important than the low-level analysis of properties enforceable by (N)MRAs are the higher-level ideas of including monitors’ input and output events on traces and defining enforcement in terms of soundness and completeness. With these techniques, policies become monitor centric (i.e., they specify valid/invalid monitor behaviors), versus the target-centric policies of earlier security-automata and formal-verification frameworks (which specify valid/invalid target behaviors). In the domain of runtime enforcement, monitor-centric policies have two primary advantages over target-centric policies: (1) monitor-
centric policies are more expressive because they can require arbitrary relationships to hold between monitors’ inputs and outputs, and (2) monitor-centric policies simplify definitions of enforcement because the definitions no longer have to hardcode particular input-output relationships.

5.1 Open Questions and Future Work

There are many open questions in the theory of two-dimensional picture languages concerning transitivity, pattern generation, etc. All of these questions remain to be investigated in the realm of the picture languages generated by cellular automata. For example, characterization of blocks in picture languages generated by cellular automata that can appear in the same space-time diagrams could help to further our understanding of the CA dynamics. Furthermore, it would be interesting to know whether there is a connection between the topological entropy of CA-generated languages and the complexity of computations that the corresponding cellular automata can perform.

Currently, the author is collaborating with Aaron Goldman (postdoctoral fellow at Laura Landweber’s lab in Princeton) on a cellular automata based model to study the dynamics of horizontal gene transfer and its effects on diverse populations of single-celled organisms. The infographic describing the project is depicted on Figure 21. The ultimate goal of this project is to develop a model that could be used for studying currently available data sets. For example, the ones contained in LUCApedia: a database for the study of ancient life [36]. One way to analyze the dynamics of such (probabilistic) cellular automata is to study blocks that have high probability of occurring and the picture languages generated by these blocks.

It may be worth applying cellular automata models to other biological systems. For instance, it is conceivable that the development of the somatic genome form its germline precursor in binucleate ciliates [13] can be described by a dynamical system representable by a cellular automaton. This dynamical system could be used to reason about the multitude
of “states” that the germline genome assumes during its transformation into soma. This approach could complement the existing topological and graph-theoretic models [3, 14, 16].

Finally, we would like to extend Mandatory Results Automata (MRAs) model for runtime enforcement monitors. Currently MRAs only allow one to explicitly reason about synchronous actions which limits their applicability for modeling asynchronous systems.
Cell CA (static case)

The initial grid is randomly populated with cells, the initial amount of cells is controlled by the CREATE_CELL_PROB parameter. The cells are divided into four quadrants; cells in each quadrant have the same set of genes (10 genes with copy number 1); cells from different quadrants have distinct genes, so, altogether, there are 40 genes initially.

The cellular automaton iterates on a grid, starting from the initial grid until a stable state is reached.

UPDATING THE GRID
The state of every cell in the updated grid is based on its state in the old grid and the states of its eight neighbors.

UPDATING COPY NUMBER
The copy number of a gene is updated as follows:

1. previous copy number is assigned
2. increased based on GROWTH_RATE
3. decreased based on LOSS_RATE
4. increased using GENE_FROM_NEIGHBOR_RATE from each neighbor that has the gene

If, say, GROWTH_RATE = 0.15 and a copy number of gene G is 1 then after step 2 the copy number becomes 1.15 (which is still 1 for an external observer).

Figure 21.: An infographic describing a model to study horizontal gene transfer. This project is a joint work with Aaron Goldman.
References


Appendices

Appendix A: Copyright and Permissions

Egor Dolzhenko <edolzhen@mail.usf.edu>

Mon, Apr 1, 2013 at 11:12 PM

Dear Prof. Natasha,

Thanks for your email. Of course we would be happy to grant the permission for your student to include this journal article "Two-dimensional languages and cellular automata" to her / his dissertation. Full acknowledgment should be made to the original source with the following format:

Title of the Work, Author(s) and/or Editor(s) Name(s), Title of the Journal, Vol and Issue No., Copyright @ year and name of the publisher

Best regards

Wendy

Rights Department
About the Author

Egor Dolzhenko was born in Sochi, Russia in 1984. He studied applied Mathematics and Computer Science in the Kuban State University (Russia). In 2005, he transferred to the University of South Florida where he completed M.A./B.A. and Ph.D. programs in Mathematics. His research interests are Formal Language Theory, Computational Biology/Bioinformatics, and Software Security.