Santa Fe Institute Working Paper 04-09-027
arxiv.org/abs/cs.CV/0410017

Automated Pattern Detection—An Algorithm for Constructing Optimally Synchronizing Multi-Regular Language Filters

Carl S. McTague[1,2,*] and James P. Crutchfield[1,†]
[1]Santa Fe Institute, 1399 Hyde Park Road, Santa Fe, NM 87501, USA
[2]Department of Mathematical Sciences, University of Cincinnati, Cincinnati, OH 45221-0025, USA
(Dated: October 7, 2004)

In the computational-mechanics structural analysis of one-dimensional cellular automata the following automata-theoretic analogue of the *change-point problem* from time series analysis arises: *Given a string σ and a collection $\{\mathcal{D}_i\}$ of finite automata, identify the regions of σ that belong to each $\mathcal{D}_i$ and, in particular, the boundaries separating them.* We present two methods for solving this *multi-regular language filtering problem*. The first, although providing the ideal solution, requires a stack, has a worst-case compute time that grows quadratically in σ's length and conditions its output at any point on arbitrarily long windows of future input. The second method is to algorithmically construct a transducer that approximates the first algorithm. In contrast to the stack-based algorithm, however, the transducer requires only a finite amount of memory, runs in linear time, and gives immediate output for each letter read; it is, moreover, the best possible finite-state approximation with these three features.

PACS numbers: 05.45.Tp, 89.70.+c, 05.45.-a, 89.75.Kd
Keywords: cellular automata; regular languages; computational mechanics; domains; particles; pattern detection; transducer; filter; synchronization; change-point problem

I. INTRODUCTION

Imagine you are confronted with an immense one-dimensional dataset in the form of a string σ of letters from a finite alphabet Σ. Suppose moreover that you discover that vast expanses of σ are regular in the sense that they are recognized by simple finite automata $\mathcal{D}_1, \ldots, \mathcal{D}_n$. You might wish to bleach out these regular substrings so that only the boundaries separating them remain, for this reduced presentation might illuminate σ's more subtle, larger-scale structure.

This *multi-regular language filtering problem* is the automata-theoretic analogue of several, more statistical, problems that arise in a wide range of disciplines. Examples include estimating stationary epochs within time series (known as the *change-point problem* [1]), distinguishing gene sequences and promoter regions from enveloping junk DNA [2], detecting phonemes in sampled speech [3], and identifying regular segments within line-drawings [4], to mention a few.

The multi-regular language filtering problem arises directly in the computational-mechanics structural analysis of cellular automata [5]. There, finite automata recognizing temporally invariant sets of strings are identified and then filtered from space-time diagrams to reveal systems of particles whose interactions capture the essence of how a cellular automaton processes spatially distributed information.

We present two methods for solving the multi-regular language filtering problem. The first covers σ with maximal substrings recognized by the automata $\{\mathcal{D}_i\}$. The interesting parts of σ are then located where these segments overlap or abut. Although this approach provides the ideal solution to the problem, it unfortunately requires an arbitrarily deep stack to compute, has a worst-case compute time that grows quadratically in σ's length, and conditions its output at any point on arbitrarily long windows of future input. As a result, this method becomes extremely expensive to compute for large data sets, including the expansive space-time diagrams that researchers of cellular automata often scrutinize.

The second method—and our primary focus—is to algorithmically construct a finite transducer that approximates the first, stack-based algorithm by printing sequences of labels i over segments of σ recognized by the automaton $\mathcal{D}_i$. When, at the end of such a segment, the transducer encounters a letter forbidden by the prevailing automaton $\mathcal{D}_i$, it prints special symbols until it resynchronizes to a new automaton $\mathcal{D}_j$. In this way, the transducer approximates the stack-based algorithm by jumping from one maximal substring to the next, printing a few special symbols in between. Since it does not jump to a new maximal substring until the preceding one ends, however, the transducer can miss the true beginning of any maximal substring that overlaps with the preceding one. Typically, the benefits of the finite transducer outweigh the occurrence of such errors.

*URL: www.mctague.org/carl; Electronic address: mctague@santafe.edu
†URL: www.santafe.edu/~chaos; Electronic address: chaos@santafe.edu

In contrast with the stack-based algorithm it approximates, however, the transducer requires only a finite amount of memory, runs in linear time, and gives immediate output for each letter read—significant improvements for cellular automata structural analysis and, we suspect, for other applications as well. Put more precisely, the transducer is Lipschitz-continuous (with Lipschitz constant one) under the cylinder-set topology, whereas the stack-based algorithm, which conditions its output on arbitrarily long windows of future input, is generally not even continuous.

It is also worth noting that the transducers thus produced are the best possible approximations with these three features and are identical to those that researchers have historically constructed by hand. Our algorithm thus relieves researchers of the tedium of constructing ever more complicated transducers.

Cellular Automata

Before presenting our two filtering methods, we introduce cellular automata in order to highlight an important setting where the multi-regular language filtering problem arises, as well as to give some visual intuition to our approach.

Let Σ be a discrete alphabet of k symbols. A *local update rule* of radius r is any function $\phi : \Sigma^{2r+1} \to \Sigma$. Given such a function, we can construct a global mapping of bi-infinite strings $\Phi : \Sigma^{\mathbb{Z}} \to \Sigma^{\mathbb{Z}}$, called a *one-dimensional cellular automaton* (CA), by setting:

$$\Phi(\sigma)_i := \phi(\sigma_{i-r} \ldots \sigma_i \ldots \sigma_{i+r}) ,$$

where σ_i denotes the ith letter of the string σ. Since the image under Φ of any period-N bi-infinite string also has period N, it is common to regard Φ as a mapping of finite strings, $\Sigma^N \to \Sigma^N$. When regarded in this way, a CA is said to have *periodic boundary conditions*.

For k=2 and r=1, there are precisely 256 local update rules, and the resulting CAs are called the *elementary* CAs (or ECAs). Wolfram [6] introduced a numbering scheme for them: Order the neighborhoods Σ^3 lexicographically and interpret the symbols $\{\phi(\eta) : \eta \in \Sigma^3\}$ as the binary representation of an integer between 0 and 255.

By interpreting a string's letters as values assumed by the *sites* of a discrete *lattice*, a CA can be viewed as a spatially extended dynamical system—discrete in time, space, and local state. Its behavior as such is often illustrated through so-called *space-time diagrams*, in which the iterates $\{\Phi^t(\sigma^0)\}_{t=0,1,2,\ldots}$ of an initial string σ^0 are plotted as a function of time. Figure 1, for example, depicts ECA 110 acting iteratively on an initial string of length $N = 150$.

Due to their appealingly simple architecture, researchers have studied CAs not only as abstract mathematical objects, but as models for physical, chemical, biological, and social phenomena such as fluid flow, galaxy formation, earthquakes, chemical pattern formation, biological morphogenesis, and vehicular traffic dynamics. Additionally, they have been used as parallel computing devices, both for the high-speed simulation of scientific models and for computational tasks such as image processing. More generally, CAs have provided a simplified setting for studying the "emergence" of cooperative or collective behavior in complex systems. The literature for all these applications is vast and includes Refs. [7, 8, 9, 10, 11, 12, 13, 14, 15, 16].

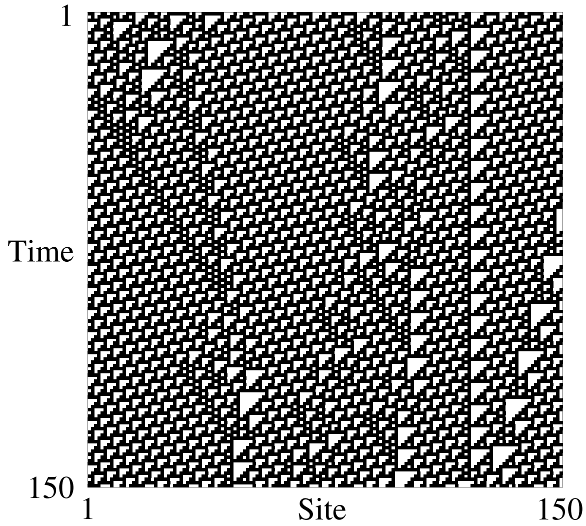

FIG. 1: A space-time diagram illustrating the typical behavior of ECA 110. Black squares correspond to 1s, and white squares to 0s.

Computational-Mechanics Structural Analysis of CAs

The computational-mechanics [17, 18] structural analysis of a CA rests on the discovery of a "pattern basis"—a collection $\{\mathcal{D}_i\}$ of automata that describe the emergent structural components in the CA's space-time behavior [19, 20]. Once such a pattern basis is found, conforming regions of space-time can be seen as background *domains* through which coherent structures not fitting the basis move. In this way, structural features set against the domains can be identified and analyzed.

More formally, Crutchfield and Hanson define a *regular domain* $\mathcal{D}$ to be a regular language (the collection of strings recognized by some finite automaton) that is:

1. *temporally invariant*—the CA maps $\mathcal{D}$ *onto* itself; that is, $\Phi^n[\mathcal{D}] = \mathcal{D}$ for some $n > 0$ —and
2. *spatially homogeneous*—the same pattern can

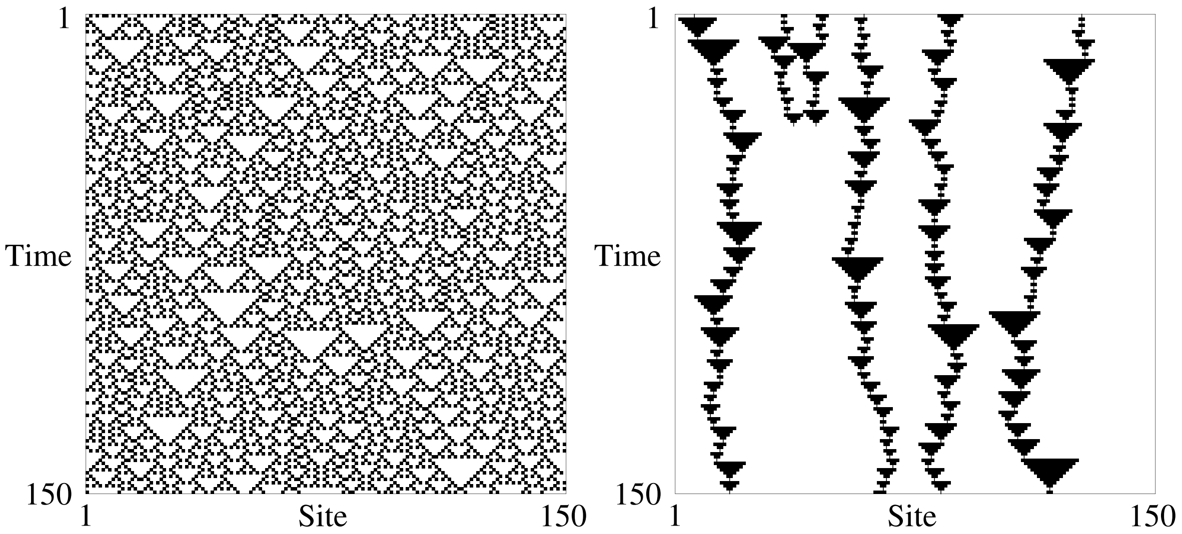

FIG. 2: (Left) Space-time diagram illustrating the typical behavior of ECA 18—a CA exhibiting apparently random behavior, i.e., the set of length-L spatial strings has a positive entropy density as $L \to \infty$. (Right) The same space-time diagram filtered with the regular domain $\mathcal{D} = \mathrm{sub}\left([0(0+1)]^*\right)$. (After Ref. [21].)

occur at any letter: the recurrent states in the minimal finite automaton recognizing $\mathcal{D}$ are strongly connected.

Once we discover a CA's regular domains—either through visual inspection or by an automated induction method such as the ϵ-machine reconstruction algorithm [5]—the corresponding space-time regions are, in a sense, understood. Given this level of discovered regularity, we bleach out the domain-conforming regions from space-time diagrams, leaving only "unmodeled" deviations, whose dynamics can then be studied. Sometimes, as is the case for the CAs we exhibit here, these deviations resemble particles and, by studying the characteristics of these particle-like deviations—how they move and what happens when they collide, we hope to understand the CA's (possibly hidden) computational capabilities.

Consider, for example, the apparently random behavior of ECA 18, illustrated in Fig. 2. Although no coherent structures present themselves to the eye, computational-mechanics structural analysis lays bare particles hidden within its output: Filtering its space-time diagrams with the regular domain $\mathcal{D} = \mathrm{sub}([0(0{+}1)]^*)$—where $\mathrm{sub}(\mathcal{L})$ denotes the regular language consisting of all subwords of strings belonging to the regular language $\mathcal{L}$—reveals a system of particles that follow random walks and pairwise annihilate whenever they touch [20, 21, 22]. Thus, by blurring the CA's deterministic behavior on strings, we discover higher-level stochastic particle dynamics. Although this loss of deterministic detail may at first seem conceptually unsatisfying, the resulting view is more structurally detailed than the vague classification of ECA 18 as "chaotic".

Thus, discovering domains and filtering them from space-time diagrams is essential to understanding the information processing embedded within a CA's output.

II. METHOD 1—FILTERING WITH A STACK

We now present the first method for solving the general multi-regular language filtering problem with which we began. Although the following method is perhaps the most thorough and easiest to describe, it requires an arbitrarily deep stack to compute. Its description will rest upon a few basic ideas from automata theory. (Please refer to the first few paragraphs of App. A, up to and including Lemma 2, where these preliminaries are reviewed.)

To filter a string σ, this method identifies the collection of its maximal substrings that the automata $\{\mathcal{D}_i\}$ accept. More formally, given a string σ, let $\sigma_{a,b}$ denote the substring $\sigma_a \sigma_{a+1} \cdots \sigma_b$ for $a, b \in \mathbb{Z}$. If σ is bi-infinite, extend this notation so that $a = -\infty$ and $b = \infty$ denote the intuitive infinite substrings. Place a partial ordering $\prec$ on all such substrings by setting $\sigma_{a,b} \prec \sigma_{a',b'}$ if $a' \leq a \leq b \leq b'$. Then let $\mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma)$ denote the collection of maximal substrings $\sigma_{a,b}$ (with respect to $\prec$) that the $\{\mathcal{D}_i\}$ accept—or, in symbols, let:

$$\mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma) := \{\sigma_{a,b} \in \mathcal{P} : \text{there is no } \sigma' \in \mathcal{P} \text{ with } \sigma_{a,b} \prec \sigma'\} ,$$

where $\mathcal{P} := \{\sigma_{a,b} : \mathcal{D}_i \text{ accepts } \sigma_{a,b} \text{ for some } i\}$.

The following algorithm can be used to compute $\mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma)$.

Algorithm 1. ***Input:*** *The automata $\mathcal{D}_1, \ldots, \mathcal{D}_n$ and the length-N string σ.*

> ***Let*** $\mathcal{A} := \mathrm{Det}(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$.
> ***Let*** s_0 *be $\mathcal{A}$'s unique start state.*
> ***Let*** $\mathbf{S}$ *and* $\mathbf{M}$ *be empty stacks.*
> ***For*** $j = 1 \ldots N$ ***do***
>   *Push* (s_0, j) *onto* $\mathbf{S}$.
>   ***For*** *each* $(s, i) \in \mathbf{S}$ ***do***
>   ***If*** *there is a transition* $(s, \sigma_j, s') \in T(\mathcal{A})$
>   ***then*** *replace* (s, i) *with* (s', i) *in* $\mathbf{S}$.
>   ***Otherwise***, *remove* (s, i) *from* $\mathbf{S}$.
>   ***If***, *in addition,* (s, i) *was at the bottom of* $\mathbf{S}$
>   ***then*** *push the pair* $(i, j-1)$ *onto* $\mathbf{M}$.
> ***Let*** (s_f, i_f) *be the pair at the bottom of* $\mathbf{S}$.
> *Push* (i_f, N) *onto* $\mathbf{M}$.
> ***Output:*** $\mathbf{M}$.

The following proposition is easily verified, and we state it without proof.

Proposition 1. *If σ is a* finite string *and if $\mathbf{M}_\sigma$ is the output of the above algorithm when applied to σ, then* $\mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma) = \{\sigma_{a,b} : (a, b) \in \mathbf{M}_\sigma\}$.

We summarize Prop. 1 by saying that Algorithm 1 solves the *local* filtering problem in the sense that it can compute $\mathcal{P}_{\max}(\{\mathcal{D}_i\}, w)$ over a finite, contractible window w. (By *contractible* we mean that periodic boundary conditions along the boundary of w are ignored.)

The *global* filtering problem, which takes into account periodic boundary conditions, is considerably more subtle. A somewhat pedantic example is filtering the bi-infinite string $0^{\mathbb{Z}}$ consisting entirely of 0s with the language $\mathrm{sub}[(0^m 1)^*]$. (Recall that $\mathrm{sub}(\mathcal{L})$ is our notation for the collection of substrings of strings belonging to $\mathcal{L}$.) The local approach applied to a finite length-N window 0^N, where $N < m$, will return 0^N itself as its single maximal substring; i.e., $\mathcal{P}_{\max}(\{\mathrm{sub}[(0^m 1)^*]\}, 0^N) = \{0^N\}$. In contrast, the global filter of $0^{\mathbb{Z}}$ will consist of heavily overlapping length-m substrings beginning and ending at every position within $0^{\mathbb{Z}}$:

$$\mathcal{P}_{\max}(\{\mathrm{sub}[(0^m 1)^*]\}, 0^{\mathbb{Z}}) = \{0^{\mathbb{Z}}_{a+1} 0^{\mathbb{Z}}_{a+2} \cdots 0^{\mathbb{Z}}_{a+m} : a \in \mathbb{Z}\}.$$

Fortunately, by examining sufficiently large finite windows, Algorithm 1 can also be used to solve this more subtle global filtering problem in the case of a bi-infinite string that is periodic. The following Lemma captures the essential observation.

Lemma 1. *Suppose σ is a* period-N bi-infinite string. *Then every maximal substring $\sigma_{a,b} \in \mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma)$ must have length $\leq m \cdot N$, where $m := \max\{|S(\mathcal{D}_i)|\}_i$, or else $\mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma)$ must consist of $\sigma_{-\infty,\infty} = \sigma$, alone.*

Proof. Our argument is a variation on the proof of the classical Pumping Lemma from automata theory. Suppose that $\sigma_{a,b} \in \mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma)$, a and b are finite, and $b - a + 1 > m \cdot N$. Then one of the domains, say $\mathcal{D}_i$, accepts $\sigma_{a,b}$. By definition, this means there is a sequence of transitions in $T(\mathcal{D}_i)$ of the form $(s_a, \sigma_a, s_{a+1}), (s_{a+1}, \sigma_{a+1}, s_{a+2}), \ldots, (s_b, \sigma_b, s_{b+1})$. Consider the sequence of pairs:

$$\{(s_i, i \bmod N)\}_{i=a}^{b} \subset S(\mathcal{D}_i) \times \mathbb{Z}_N.$$

Since:

$$b - a + 1 > m \cdot N \geq |S(\mathcal{D}_i) \times \mathbb{Z}_N|,$$

the Pigeonhole Principle implies that this sequence must repeat—say $(s_l, l \bmod N) = (s_{l'}, l' \bmod N)$ for integers $l < l'$. But then $\mathcal{D}_i$ must also accept any string of the form:

$$\sigma_a \sigma_{a+1} \cdots \sigma_l (\sigma_{l+1} \cdots \sigma_{l'})^* \sigma_{l'+1} \cdots \sigma_b.$$

Since $l \bmod N = l' \bmod N$, such strings correspond to arbitrarily long substrings of the original bi-infinite string σ. As a result, $\sigma_{a,b}$ cannot be maximal. This contradiction implies that either (i) a and b are not both finite or (ii) $b - a + 1 \leq m \cdot N$. A straightforward generalization of our argument in fact shows that either (i) *both a and b are infinite* or (ii) $b - a + 1 \leq m \cdot N$. □

A consequence of Lemma 1 is that we can solve the global filtering problem by applying Algorithm 1 to a window of length $mN + 1$.

Proposition 2. *Suppose σ is a* period-N bi-infinite string *and that $\mathbf{M}_{\sigma'}$ is the output of Algorithm 1 when applied to the finite string $\sigma' := \sigma_1 \sigma_2 \cdots \sigma_{mN+1}$, where $m := \max\{|S(\mathcal{D}_i)|\}_i$. Then:*

$$\mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma) = \{\sigma_{a+qN, b+qN} : (a, b) \in \mathbf{M}_{\sigma'}, q \in \mathbb{Z}\},$$

unless $\mathbf{M}_{\sigma'}$ consists of $(1, mN+1)$ alone, in which case $\mathcal{P}_{\max}(\{\mathcal{D}_i\}, \sigma) = \{\sigma_{-\infty,\infty} = \sigma\}$.

The major drawback of Algorithm 1, however, is its worst-case compute time.

Proposition 3. *The worst-case performance of the stack-based filtering algorithm (Algorithm 1) has order $\mathcal{O}(N^2)$, where N is the length of the input string σ.*

Proof. For each $j = 1 \ldots N$, the algorithm pushes a new pair (s_0, j) onto the stack $\mathbf{S}$ and then advances each pair on $\mathbf{S}$. In the case that $\mathcal{A}$ accepts the entire string σ, the algorithm will never remove any pairs from $\mathbf{S}$ and will thus advance a total of $\sum_{j=1}^{N} j = \frac{1}{2}N(N+1)$ pairs. The proposition follows since it is possible to advance each pair in constant time. □

III. METHOD 2—FILTERING WITH A TRANSDUCER

The second method—and our primary focus—is to algorithmically construct a finite transducer that approximates the stack-based Algorithm 1 by printing sequences of labels i over segments of σ recognized by the automaton $\mathcal{D}_i$. When, at the end of such a segment, the transducer encounters a letter forbidden by the prevailing automaton $\mathcal{D}_i$, it prints special symbols until it resynchronizes to a new automaton $\mathcal{D}_j$. The special symbols consist of labels for the kinds of domain-to-domain transition and λ, which indicates that classification is ambiguous.

In this way, the transducer approximates the stack-based algorithm by jumping from one maximal substring to the next, printing a few special symbols in between. Because it does not jump to a new maximal substring until the preceding one ends, however, the transducer can miss the true beginning of any maximal substring that overlaps with the preceding one. But if no more than two maximal substrings overlap at any given point of σ, then it is possible to combine the output of two transducers, one reading left-to-right and the other reading right-to-left, to obtain the same output as the stack-based algorithm.

These shortcomings are minor, and in exchange the transducer gains several significant advantages over the stack-based algorithm it approximates: It requires only a finite amount of memory, runs in linear time, and gives immediate output for each letter read.

Although finite transducers are generally considered less sophisticated than stack-based algorithms in the sense of computational complexity, the construction of this transducer is considerably more intricate than the preceding stack-based algorithm and is, in fact, our principal aim in the following.

Our approach will be to construct a transducer $\mathrm{Filter}(\{\mathcal{D}_i\})$ by 'filling in' the forbidden transitions of the automaton $\mathcal{A} := \mathrm{Det}(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$. We will thus tie our hands behind our backs at the outset by permitting the transducer to remember *only* as much about past input as does the automaton $\mathcal{A}$ while recognizing domain strings.

Unfortunately, $\mathcal{A}$'s states will generally preserve too little information to facilitate optimal resynchronization. It is possible, however, to begin with elaborately constructed, equivalent, non-minimal domains $\mathcal{D}'_i$ that yield an automaton $\mathcal{A}' := \mathrm{Det}(\mathcal{D}'_1 \sqcup \cdots \sqcup \mathcal{D}'_n)$ whose states do preserve just enough information to facilitate optimal resynchronization. The transducer obtained by 'filling in' the forbidden transitions of this automaton $\mathcal{A}'$ represents the best possible (transducer) approximation of the stack-based algorithm. We present a preprocessing algorithm which produces these equivalent, non-minimal domains $\{\mathcal{D}'_i\} = \mathrm{Optimize}(\{\mathcal{D}_i\})$ at the end of our discussion of Method-2 filtering.

The idea underlying our construction is the following. Suppose that while reading the string σ we are recognizing an increasingly long string accepted by $\mathcal{D}_i$ when we encounter a forbidden letter a. In accepting σ up to this point, the automaton $\mathcal{A}$ will have reached a certain state $s \in S(\mathcal{A})$ that has no outgoing transition corresponding to the letter a. Our goal is to create such a transition by examining the collection of all possible strings that could have placed us in the state s and to resynchronize to the state of $\mathcal{A}$ that is most compatible with the potentially foreign strings obtained by appending to these strings the forbidden letter a.

In this situation there will be two natural desires. On the one hand, we wish to unambiguously resynchronize to as *specific* a domain state as possible; but, on the other, we wish to rely on as little of the imagined past as possible. (We use the term *imagined* because our transducer remembers only the state $s \in S(\mathcal{A})$ we have reached—not the particular string that placed us there.) To reflect these desires, we introduce a partial ordering on the collection of potential resynchronization states $\{S_{i,l}\}$, where i measures the specificity of resynchronization and l the length of imagined past.

We now implement this intuition in full detail. Our exposition relies heavily on ideas from automata theory. (We now urge reading App. A in its entirety.)

As above, let $\mathcal{A} := \mathrm{Det}(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$ and let $S(\mathcal{A}) \overset{\psi_{\mathcal{A}}}{\hookrightarrow} S(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$ be the canonical injection provided by Lemma 2 in App. A. Assume that there is a canonical injection $S(\mathcal{D}_1) \sqcup \cdots \sqcup S(\mathcal{D}_n) \hookrightarrow S(\mathcal{A})$ and that we can therefore regard the sets $S(\mathcal{D}_i)$ as subsets of $S(\mathcal{A})$. An example of this situation is depicted in Fig. 3. A sufficient condition for the existence of such an injection is that each $\mathcal{D}_i$ is minimal and that $\mathrm{Lang}(\mathcal{D}_i) \not\subset \mathrm{Lang}(\mathcal{D}_l)$ for $i \neq l$. Minimality is far from required, however, and the assumption is valid for a much larger class of domains. (Put informally, it suffices if we can associate to each state $s \in S(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$ a string that corresponds to a unique path through $\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n$—one that leads to s.)

Let $\mathcal{T}$ be a transducer with the same states, start state, and final states as $\mathcal{A}$, but with the transitions:

$$T(\mathcal{T}) := \{(s, a|f(s'), s') : (s, a, s') \in T(\mathcal{A})\} ,$$

where:

$$f(s') = \begin{cases} i & \text{if } \phi_{\mathcal{A}}(s') \subset S(\mathcal{D}_i) , \\ \lambda & \text{otherwise} , \end{cases}$$

and where λ is a new symbol in the output alphabet Σ' indicating that domain labeling was not possible, for example, because the partial string read so far belongs to more than one or none of the automata $\{\mathcal{D}_i\}$. To recapitulate, the transducer's output alphabet Σ' consists of three kinds of symbol: domain labels $\{1 \ldots n\}$, domain-domain transition types $\{1, 2, \ldots, p\}$, and ambiguity λ.

The transducer $\mathcal{T}$'s input, $\mathrm{In}(\mathcal{T})$, recognizes precisely those strings recognized by the given domains. Our goal is to extend $\mathcal{T}$ by introducing transitions of

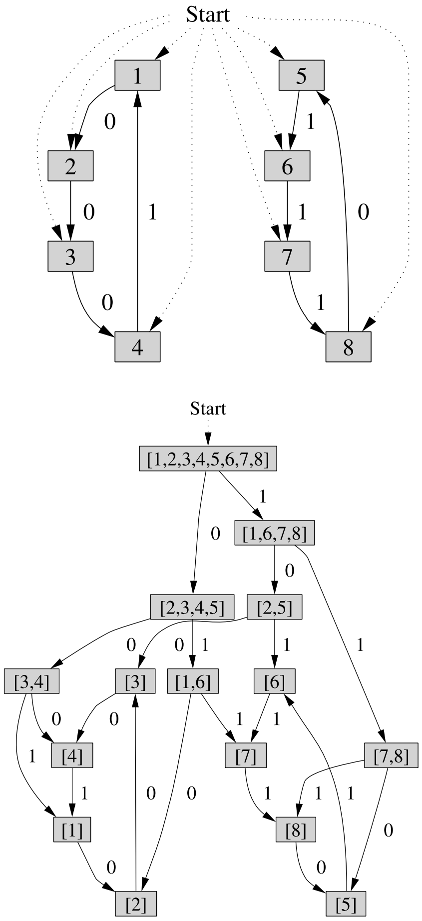

FIG. 3: The domains $\mathcal{D}_1$ and $\mathcal{D}_2$ (top) and the automaton $\mathcal{A} = \mathrm{Det}(\mathcal{D}_1 \sqcup \mathcal{D}_2)$ (bottom). Start states are indicated by dotted arrows from the word "Start", and final states are darkened. Notice that the states of $\mathcal{A}$ correspond to collections of states of $\mathcal{D}_1$ and $\mathcal{D}_2$ and that the former are canonically injected into the latter, here by the map $n \mapsto [n]$.

the form:

$$\{(s, a|h(s,a), g(s,a)) \ : s \in S(\mathcal{T}) = S(\mathcal{A}), a \in \Sigma, \text{ and there are no transitions of the form } (s,a,\cdot) \in T(\mathcal{A})\} \,,$$

where the functions $g(s,a)$ and $h(s,a)$ are defined in the following paragraphs. The transducer $\mathrm{Filter}(\{\mathcal{D}_i\})$ obtained by adding these transitions to $\mathcal{T}$ will then have the desired property that its input $\mathrm{In}(\mathrm{Filter}(\{\mathcal{D}_i\}))$ will accept all strings [29].

Let W_l denote the collection of strings corresponding to length-l paths through $\mathcal{A}$ beginning in any of its states, but *ending* in state s, and let W'_{l+1} denote the collection of strings obtained by appending the letter a to the strings of W_l. The strings $\bigcup_{l\geq 0} W'_l$ are accepted by the finite automaton $\mathcal{A}^{s,a}$ obtained by adding a new state f and a transition (s,a,f) to $\mathcal{A}$, and by setting $\mathrm{Start}(\mathcal{A}^{s,a}) := S(\mathcal{A}^{s,a})$ and $\mathrm{Final}(\mathcal{A}^{s,a}) := \{f\}$. An example is shown in Fig. 4, where the four-state domain has a transition added from state [2] on symbol 1, which was originally forbidden.

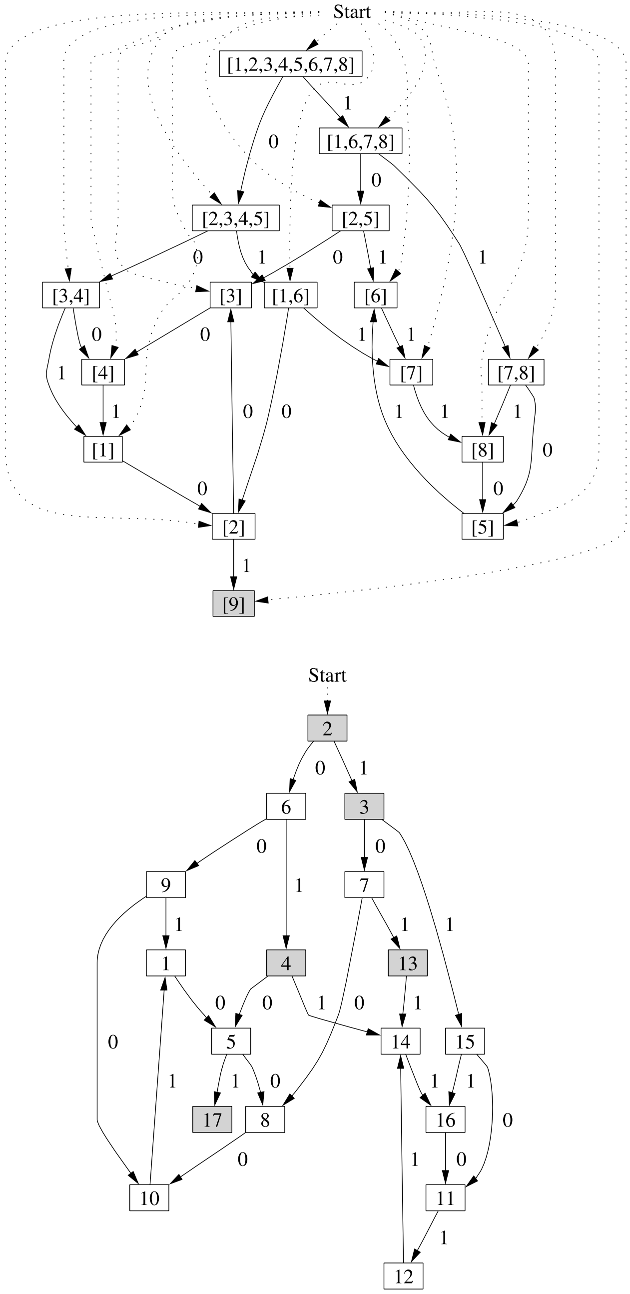

FIG. 4: The semi-deterministic automaton $\mathcal{A}^{[2],1}$ (top) obtained by adding a state $f = [9]$ and its deterministic version $\mathrm{Det}(\mathcal{A}^{[2],1})$ (bottom) with states relabeled with the integers $1 \ldots 17$ in order to simplify later diagrams.

In order to choose the resynchronization state $g(s,a)$ for the forbidden transition (s,a), we examine the strings of $\bigcup_{l\geq 0} W'_l$ that also belong to one or more of the domains $\{\mathcal{D}_i\}$. We do this by constructing the automaton $\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}$, which we call the *resynchro-*

nization automaton. By Lemma 3, there is a canonical, although not necessarily injective, association:

$$\phi : S(\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}) \to S(\mathcal{A})$$

given by the composition:

$$S(\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}) \quad \hookrightarrow \quad S(\mathrm{Det}(\mathcal{A}^{s,a})) \times S(\mathcal{A}) \quad \to \quad S(\mathcal{A}) \ ,$$

where the right-most map is the second-factor projection, $(s, s') \mapsto s'$.

The resynchronization automaton $\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}$ may reveal several possible resynchronization states. To help distinguish among them, we put them into sets $\{S_{i,l}\}$ where i measures the *specificity* of resynchronization and l the length of imagined past. More precisely, let $S_{i,l}$ denote those states $s \in S(\mathcal{A})$ to which ϕ associates at least one state $s' \in \mathrm{Final}(\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A})$ (i.e. $s = \phi(s')$) satisfying the following two conditions: (1) s corresponds, under Lemma 2, to precisely i states of $\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n$ and (2) there is a length-l path from the unique start state of $\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}$ to s'.

Give the sets $\{S_{i,l}\}$ the dictionary ordering; that is, let $S_{i,l} < S_{i',l'}$ if $i < i'$ or if $i = i' \wedge l < l'$. The set $S_{|S(\mathcal{A})|,0}$ consists of the unique start state of $\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}$. Thus, by the well ordering principle, there must be a unique, least set among the sets $\{S_{i,l}\}$ that consist of a single state, say $\{s'\}$. Let $g(s, a) := s'$, and let $h(s, a) := h'(s, s') = h'(s, g(s, a))$, where h' is any injection $S(\mathcal{T}) \times S(\mathcal{T}) \hookrightarrow \Sigma'$ (chosen independent of s and a). An example of this construction is shown in Fig. 5.

The transducer is completed by repeating the above steps for all forbidden transitions.

Computability of the transducer $\mathrm{Filter}(\{\mathcal{D}_i\})$

Although the transducer $\mathrm{Filter}(\{\mathcal{D}_i\})$ is well defined, it is perhaps not immediately clear that it is computable. After all, we appealed to the well ordering principle to obtain a least singleton set $\{s'\}$ among the sets $\{S_{i,l}\}$. In fact, infinitely many sets $S_{i,l}$ precede the stated upper bound $S_{|S(\mathcal{A})|,0}$—for instance, all of the sets $S_{1,\mathbb{N}}$ do, provided $|S(\mathcal{A})| > 1$.

The construction is nevertheless computable, because for each i the sequence of sets $S_{i,\mathbb{N}}$ must eventually repeat. In fact, we can compute this sequence of sets exactly by automata-theoretic means.

Proposition 4. *The transducer* $\mathrm{Filter}(\{\mathcal{D}_i\})$ *is computable.*

Proof. Let $\mathcal{Z}[\mathcal{C}]$ denote the automaton obtained by relabeling all of the automaton $\mathcal{C}$'s transitions with 0s. This automaton will almost certainly be nondeterministic. The equivalent deterministic automaton $\mathrm{Det}(\mathcal{Z}[\mathcal{C}])$ is useful, because the state it reaches when accepting the string 0^l corresponds precisely,

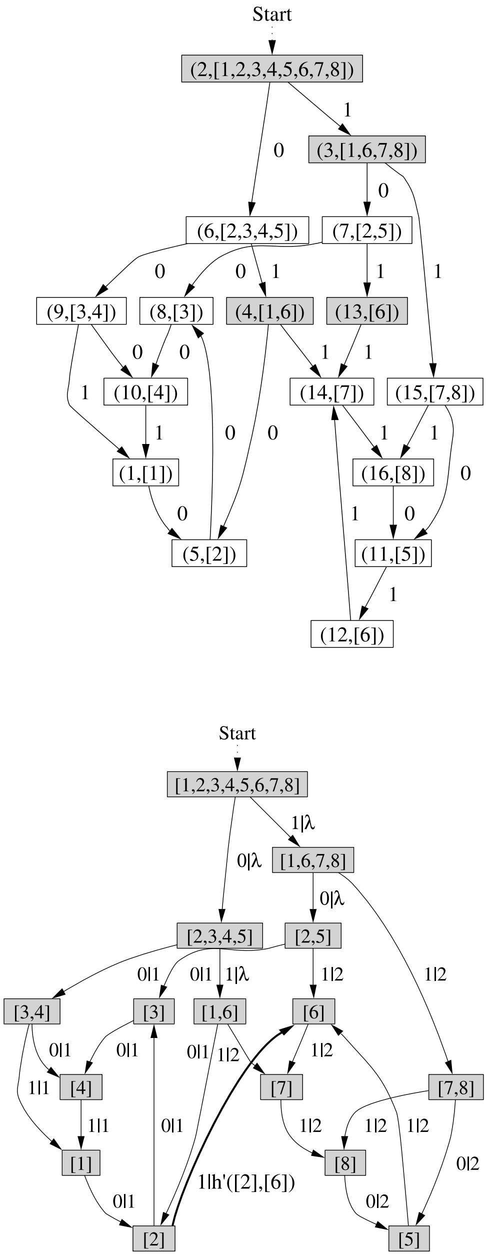

FIG. 5: The resynchronization automaton $\mathrm{Det}(\mathcal{A}^{[2],1}) \cap \mathcal{A}$ (top). Here $S_{1,3}$ consists of the state $(13, [6])$ alone, and all other $S_{1,\bullet}$ are empty. So we choose $s' = [6]$ and add a transition $([2], 1|h'([2], [6]), [6])$ to $\mathcal{T}$ (bottom).

under Lemma 2, to the collection of states that can be reached by length-l paths through $\mathcal{C}$.

Moreover, since $\mathrm{Det}(\mathcal{Z}[\mathcal{C}])$ is defined over a single letter, yet deterministic and finite, it must have a special graphical structure: its single start state s_0 must lead to a finite loop after a finite chain of nonrecurrent states. (Actually, if $\mathcal{C}$ has no loops whatsoever, there will not even be a loop.) Thus, its states have a linear ordering: $s_0 \xrightarrow{0} s_1 \xrightarrow{0} \cdots \xrightarrow{0} s_m \xrightarrow{0} s_{m+1} \xrightarrow{0} \cdots \xrightarrow{0} s_{m+m'} \xrightarrow{0} s_m$. An example is illustrated in Fig. 6, where $m = 4$ and $m' = 0$.

By Lemma 2 the states $\{s_k\}$ correspond to collections of states of $\mathcal{C}$ under an injection:

$$\begin{aligned}\psi_{\mathcal{Z}[\mathcal{C}]} : S(\mathrm{Det}(\mathcal{Z}[\mathcal{C}])) \hookrightarrow &\{S \subset S(\mathcal{Z}[\mathcal{C}])\} \\ = &\{S \subset S(\mathcal{C})\}.\end{aligned}$$

Let $\mathcal{C} := \mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}$ in the preceding discussion. As before, by Lemma 3, there is a function:

$$\phi : S(\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}) \to S(\mathcal{A}) \ .$$

Let $S_{*,l} \subset S(\mathcal{A})$ denote those states defined by the formula:

$$S_{*,l} := \phi[\psi_{\mathcal{Z}[\mathcal{C}]}(s_l) \cap \mathrm{Final}(\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A})] \ .$$

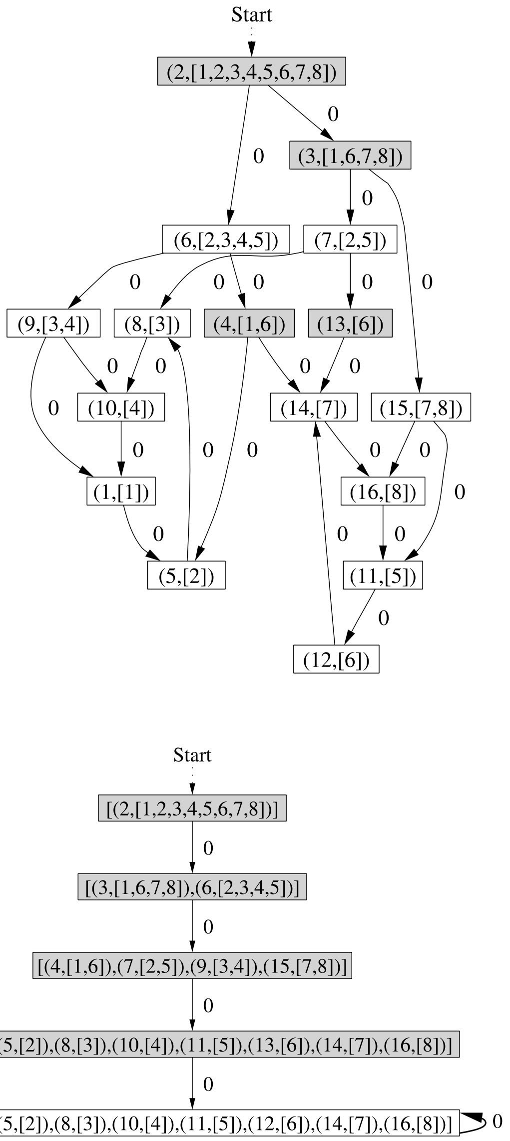

FIG. 6: The automaton $\mathcal{Z}[\mathrm{Det}(\mathcal{D}_1^{[2],1}) \cap \mathcal{A}]$ (top) and its deterministic version $\mathrm{Det}(\mathcal{Z}[\mathrm{Det}(\mathcal{D}_1^{[2],1}) \cap \mathcal{A}])$ (bottom).

Finally, let $S_{i,*}$ denote those states of $\mathcal{A}$ that correspond to precisely i states of $\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n$; that is, let:

$$S_{i,*} := \{s \in S(\mathcal{A}) : |\phi_{\mathcal{A}}(s)| = i\} \ ,$$

where $\phi_{\mathcal{A}} : S(\mathcal{A}) \hookrightarrow \{S \subset S(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)\}$ is the injection provided by Lemma 2.

The sets $S_{i,l}$ can then be computed as the intersections $S_{i,*} \cap S_{*,l}$, and we need only examine these for $1 \leq i \leq |S(\mathcal{D}_1)| + \cdots + |S(\mathcal{D}_n)|$ and $0 \leq l \leq m + m'$ to discover the least one under the dictionary ordering that is a singleton $\{s'\}$. □

We summarize the entire algorithm.

Algorithm 2. ***Input:*** *The regular domains* $\mathcal{D}_1, \ldots, \mathcal{D}_n$.

- *Let* $\mathcal{A} := \mathrm{Det}(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$.
- *Choose any injection* $h' : S(\mathcal{A}) \times S(\mathcal{A}) \hookrightarrow \Sigma'$.
- *Make* $\mathcal{A}$ *into a transducer* $\mathcal{T}$ *by adding the symbol* i *as output to any transition ending in a state corresponding to states of only one domain* $\mathcal{D}_i$ *and by adding* λ*s as output symbols to all other transitions.*
- ***For each*** *forbidden transition* $(s, a) \in S(\mathcal{A}) \times \Sigma(\mathcal{A})$, *add a transition to* $\mathcal{T}$ *through the following procedure* ***do***
  - *Construct the automaton* $\mathcal{A}^{s,a}$ *by adding to* $\mathcal{A}$ *the transition* (s, a, f), *where* f *is a new state, and by letting* f *be its only final state.*
  - *Construct the automaton* $\mathrm{Det}(\mathcal{Z}[\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}])$, *where* $\mathcal{Z}[\mathcal{C}]$ *is the automaton obtained by relabeling all of* $\mathcal{C}$*'s transitions with* 0*s. Its states will have a natural linear ordering* $s_0 \to s_1 \to \cdots \to s_{m+m'}$.
  - *Let* $S_{i,*}$ *and* $S_{*,l}$ *be the subsets of* $S(\mathcal{A})$ *defined by:*

    $$\begin{aligned}S_{*,l} := \phi[&\psi_{\mathrm{Det}(\mathcal{Z}[\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}])}(s_l) \\ &\cap \mathrm{Final}(\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A})] \ \textit{and} \\ S_{i,*} := \{&s \in S(\mathcal{A}) : |\psi_{\mathcal{A}}(s)| = i\}.\end{aligned}$$

  - *Find the singleton set* $\{s'\}$ *among the sets:*

    $$\begin{aligned}\{S_{i,*} \cap S_{*,l} : 1 \leq i \leq |\mathcal{D}_1| + \cdots + |\mathcal{D}_n|, \\ 0 \leq l \leq m + m'\}\end{aligned}$$

    that occurs first under the dictionary ordering.
  - *Add the transition* $(s, a|h'(s, s'), s')$ *to* $\mathcal{T}$.

Output: $\mathrm{Filter}(\{\mathcal{D}_i\}) := \mathcal{T}$.

Algorithmic Complexity

Proposition 5. *The worst-case performance of the transducer-constructing algorithm (Algorithm 2) has order no greater than:*

$$|\mathcal{A}| \cdot (|\Sigma| - 1) \cdot \exp \circ \exp\left(2 \cdot |\mathcal{A}| + 1\right) \ ,$$

where $|\mathcal{A}|$ *has order* $\exp(|\mathcal{D}_1| + \cdots + |\mathcal{D}_n|)$.

Proof. The algorithm's most expensive step is the computation of $\mathrm{Det}(\mathcal{Z}[\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}])$. Unfortunately, because computing $\mathrm{Det}(\mathcal{G})$ has order $\exp(|\mathcal{G}|)$, and because computing $\mathcal{G} \cap \mathcal{H}$ has order $|\mathcal{G}| \cdot |\mathcal{H}|$, this computation has order $\exp \circ \exp(2 \cdot |\mathcal{A}| + 1)$.

Finally, recall that the algorithm computes $\mathrm{Det}(\mathcal{Z}[\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}])$ for every forbidden transition (s, a) of $\mathcal{A}$. A rough upper bound for the number of such transitions is $|\mathcal{A}| \cdot (|\Sigma| - 1)$. From these two upper bounds the proposition follows. □

Although this analysis may at first seem to objurgate the transducer-constructing algorithm, the reader should realize that, once computed, $\mathcal{T}$ can be very efficiently used to filter arbitrarily long strings. That is, unlike the stack-based algorithm, its performance is linear in string length. Thus, one pays during the filter design phase for an efficient run-time algorithm—a trade-off familiar, for example, in data compression.

Constructing optimal transducers from non-minimal domains, a preprocessing step to Algorithm 2

Recall that we constructed the transducer $\mathrm{Filter}(\{\mathcal{D}_i\})$ by 'filling in' the forbidden transitions of the automaton $\mathcal{A} := \mathrm{Det}(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$. This proved somewhat problematic, however, because $\mathcal{A}$'s states do not always preserve enough information about past input to unambiguously resynchronize to a unique, recurrent domain state. In order to help discriminate among the several possible resynchronization states, we introduced the partially ordered sets $\{S_{i,l}\}$. But even so, several attractive resynchronization states often fell into the same set $S_{i,l}$. So, lacking any objective way to choose among them, we resigned ourselves to a less attractive resynchronization state occurring in a later set $S_{i',l'}$, simply because it appeared alone there, making our choice unambiguous. If only the states of the automaton $\mathcal{A}$ preserved slightly more information about past input, then such compromises could be avoided.

In this section we present an algorithm that splits the states of a given collection $\{\mathcal{D}_i\}$ of domains to obtain an equivalent collection $\{\mathcal{D}'_i\} = \mathrm{Optimize}(\{\mathcal{D}_i\})$ of domains that preserve just enough information about past input to enable unambiguous resynchronization in the transducer obtained by *filling in* the forbidden transitions of the automaton $\mathcal{A}' := \mathrm{Det}(\mathcal{D}'_1 \sqcup \cdots \sqcup \mathcal{D}'_n)$.

We will accomplish this by associating to each state of the original domains $\mathcal{D}_i$ a collection of automata that partition past input strings into equivalence classes corresponding to individual resynchronization states. We will then refine these partitions so that $\mathcal{D}_i$'s transition structures can be lifted to them and thus obtain the desired domains $\{\mathcal{D}'_i\} = \mathrm{Optimize}(\{\mathcal{D}_i\})$.

This procedure, taken as a preprocessing step to Algorithm 2, will thus produce the best possible transducer for Method-2 multi-regular language filtering.

We now state our construction formally. If $s' \in S(\mathcal{A})$, then let $\mathcal{A}_{s'}$ denote the automaton that is identical to the automaton $\mathcal{A}$ except that its only final state is s'. Additionally, if (s, a) is a forbidden transition of the automaton $\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n$, then let $\mathcal{B}(s, a, s')$ denote the automaton satisfying the formula:

$$\mathcal{B}(s, a, s') \cdot a = \mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}_{s'} \ ,$$

where $\cdot$ denotes concatenation. That is, let $\mathcal{B}(s, a, s')$ denote the automaton that is identical to the automaton $\mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}_{s'}$ except that its final states are given by $\{s_f : (s_f, a, s'_f) \in T(\diamond), s'_f \in \mathrm{Final}(\diamond)\}$, where $\diamond := \mathrm{Det}(\mathcal{A}^{s,a}) \cap \mathcal{A}_{s'}$. Note that in most cases $\mathrm{Lang}(\mathcal{B}(s, a, s'))$ will be empty.

Next we associate to each state $s \in S(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$ a collection $\Gamma(s)$ of automata. If the state s has no forbidden transitions, let $\Gamma(s) := \{\Sigma^*\}$. If the state s has at least one forbidden transition, however, then let $\Gamma(s)$ denote the collection of automata:

$$\begin{aligned}\Gamma(s) := \mathrm{Disjoin}(\{\Sigma^* \cdot \mathcal{B}(s, a, s') :& \\ &(s, a, \cdot) \notin T(\mathcal{A}), s' \in S(\mathcal{A})\}) \ ,\end{aligned}$$

where $\mathrm{Disjoin}(\{\mathcal{C}_\gamma\})$ denotes the coarsest partition of $\bigcup_\gamma \mathrm{Lang}(\mathcal{C}_\gamma)$ by automata $\{\mathcal{E}_\epsilon\}$ that is compatible with the automata $\{\mathcal{C}_\gamma\}$. That is, $\mathrm{Disjoin}(\{\mathcal{C}_\gamma\})$ denotes the smallest collection $\{\mathcal{E}_\epsilon\}$ of automata satisfying (i) $\bigcup_\epsilon \mathrm{Lang}(\mathcal{E}_\epsilon) = \bigcup_\gamma \mathrm{Lang}(C_\gamma)$ and (ii) $\mathrm{Lang}(\mathcal{C}_\gamma) \cap \mathrm{Lang}(\mathcal{E}_\epsilon)$ is either empty or equal to $\mathrm{Lang}(\mathcal{E}_\epsilon)$ for all γ and ϵ.

It is possible to compute $\mathrm{Disjoin}(\{\mathcal{C}_\gamma\})$ inductively with the formula:

$$\begin{aligned}&\mathrm{Disjoin}(\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m\}) = \\ &\qquad \{\mathcal{C}_1 \setminus (\mathcal{C}_2 \sqcup \cdots \sqcup \mathcal{C}_m)\} \cup \\ &\qquad \{\mathcal{C}_1 \cap \mathcal{C}' : \mathcal{C}' \in \mathrm{Disjoin}(\{\mathcal{C}_2, \ldots, \mathcal{C}_m\})\} \cup \\ &\qquad \{\mathcal{C}_1 \setminus \mathcal{C}' : \mathcal{C}' \in \mathrm{Disjoin}(\{\mathcal{C}_2, \ldots, \mathcal{C}_m\})\}.\end{aligned}$$

Note that $\bigcup_{\mathcal{E} \in \Gamma(s)} \mathrm{Lang}(\mathcal{E}) = \Sigma^*$ for all states $s \in S(\mathcal{D}_1 \sqcup \cdots \sqcup \mathcal{D}_n)$. This is because $\mathrm{Lang}(\mathcal{B}(s, a, s'))$ contains only the empty string if $s' \in S(\mathcal{A})$ is the unique state reached on input a from $\mathcal{A}$'s starting state—that is, if $(s_0, a, s') \in T(\mathcal{A})$, where $\{s_0\} = \mathrm{Start}(\mathcal{A})$.

Our goal is to create for each original domain $\mathcal{D}_i$ an equivalent domain $\mathcal{D}'_i$ by splitting each state $s \in S(\mathcal{D}_i)$ into states of the form $(s, \mathcal{E})$, where $\mathcal{E} \in \Gamma(s)$. But to endow these split states with a transition structure equivalent to $\mathcal{D}_i$'s, we typically must refine the sets $\Gamma(s)$ further. We must construct a refinement $\Gamma'(s)$ of each $\Gamma(s)$ with the property that if (s, a, s') is a transition of $\mathcal{D}_i$, then to each $\mathcal{E} \in \Gamma'(s)$ there corresponds a unique $\mathcal{E}' \in \Gamma'(s')$ with $\mathrm{Lang}(\mathcal{E} \cdot a) \subset \mathrm{Lang}(\mathcal{E}')$. Given such refinements $\Gamma'(s)$, we can take the pairs $\{(s, \mathcal{E}) : s \in S(\mathcal{D}_i), \mathcal{E} \in \Gamma'(s)\}$ as the states of $\mathcal{D}'_i$ and equip them with transitions of the form $(s, \mathcal{E}) \xrightarrow{a} (s', \mathcal{E}')$, and thus obtain an equivalent, but non-minimal, domain $\mathcal{D}'_i$.

The following algorithm can be used to compute the desired refinements $\Gamma'(s)$.

Algorithm 3. ***Input:*** *The domain $\mathcal{D}$ and the function Γ that assigns to each state $s \in S(\mathcal{D})$ a collection $\Gamma(s)$ of automata that partition Σ^*.*

– ***For each*** *state $s \in S(\mathcal{D})$,* ***let:***

$$\Gamma'(s) := \mathrm{Disjoin}\left(\bigcup\{\Gamma'(s, a, s') : (s, a, s') \in T(\mathcal{D})\}\right) ,$$

where:

$$\Gamma'(s, a, s') := \{\mathcal{E}''_\alpha\}_\alpha \text{ and}$$
$$\{\mathcal{E}''_\alpha \cdot a\}_\alpha := \{(\mathcal{E} \cdot a) \cap \mathcal{E}' : \mathcal{E} \in \Gamma(s), \mathcal{E}' \in \Gamma(s')\}.$$

– *If $\Gamma'(s) \neq \Gamma(s)$ for some state $s \in S(\mathcal{D})$, then* ***repeat*** *with Γ' in place of Γ. Otherwise:*

Output: Γ'.

Proposition 6. *Algorithm 3 eventually terminates, producing the coarsest possible refinements $\Gamma'(s)$ of $\Gamma(s)$ compatible with $\mathcal{D}$'s transition structure.*

Proof. We construct fine, but finite, refinements that are compatible with $\mathcal{D}$'s transition structure, then use this result to conclude that Algorithm 3 must eventually terminate. Moreover, we also conclude that, when Algorithm 3 terminates, it produces the coarsest possible refinements that are compatible with $\mathcal{D}$'s transition structure.

Let $\{\mathcal{E}_i\}$ denote the potentially large, but finite, collection of automata:

$$\{\mathcal{E}_i\}_{i=1}^N := \mathrm{Disjoin}\left(\bigcup_{s \in S(\mathcal{D})} \Gamma(s)\right) ,$$

which partition Σ^*.

We refine the partition $\{\mathcal{E}_i\}$ to make it compatible with $\mathcal{D}$'s transitions by examining the automaton $\mathcal{F} := \mathrm{Det}(\mathcal{E}_1 \sqcup \cdots \sqcup \mathcal{E}_N)$. Since the automata $\{\mathcal{E}_i\}$ cover Σ^*, the deterministic automaton $\mathcal{F}$ can have no forbidden transitions, and all its states must be final. Moreover, because the automata $\{\mathcal{E}_i\}$ are disjoint, each of $\mathcal{F}$'s states must correspond (under the canonical injection $\psi_{\mathcal{F}}$ of Lemma 2) to final states of precisely one automaton $\mathcal{E}_i$. In this way, the automata $\{\mathcal{E}_i\}$ correspond to a partition of the states of $\mathcal{F}$.

Since each automaton $\mathcal{E}_i$ is equivalent to the automaton obtained by restricting $\mathcal{F}$'s final states to those states corresponding (under $\psi_{\mathcal{F}}$) to final states of $\mathcal{E}_i$, we can refine the partition $\{\mathcal{E}_i\}$ by refining this partition of $\mathcal{F}$'s states.

Although a coarser refinement may suffice, we can always choose the partition consisting of single states. That is, if $s \in S(\mathcal{F})$, let $\mathcal{F}_s$ denote the automaton that is identical to the automaton $\mathcal{F}$ except that its only final state is s. Then $\{\mathcal{F}_s : s \in S(\mathcal{F})\}$ is a refinement of the partition $\{\mathcal{E}_i\}$ with the special property that for each automaton $\mathcal{F}_s$ and $a \in \Sigma(\mathcal{F})$, there is a unique automaton $\mathcal{F}_{s'}$ such that $\mathcal{F}_s \cdot a = \mathcal{F}_{s'}$. Indeed, since $\mathcal{F}$ is deterministic, s' is the unique state corresponding to a transition $(s, a, s') \in T(\mathcal{F})$.

If we let $\Gamma''(s) := \{\mathcal{F}_{s'} : s' \in S(\mathcal{F})\}$ for each state $s \in S(\mathcal{D})$, then we obtain finite refinements of $\Gamma(s)$ compatible with $\mathcal{D}$'s transition structure, as desired.

This result implies that Algorithm 3 must eventually terminate. After all, every refinement that Algorithm 3 performs must already be reflected in $\Gamma''(s)$. Moreover, since every refinement that the algorithm performs is essential to compatibility with $\mathcal{D}$'s transition structure, the algorithm must, upon termination, produce the coarsest (smallest) compatible refinement possible. □

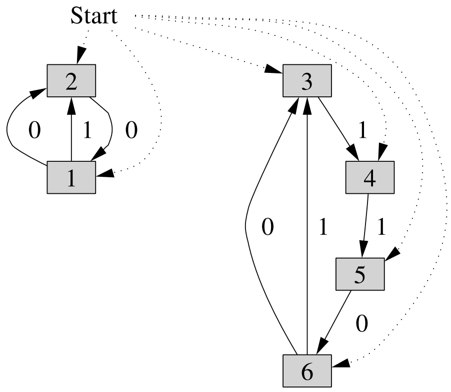

FIG. 7: The positive-entropy domains $\mathcal{D}_1$ and $\mathcal{D}_2$ of the binary, next-to-nearest neighbor CA 2614700074. (After Ref. [19].)

When applied to the domains $\mathcal{D}_1$ and $\mathcal{D}_2$ in Fig. 7, for example, Algorithm 3 produces the equivalent, non-minimal domains $\{\mathcal{D}'_1, \mathcal{D}'_2\} = \mathrm{Optimize}(\{\mathcal{D}_1, \mathcal{D}_2\})$ shown in Fig. 8. Notice these domains' many nonrecurrent states. These have almost no effect on the automaton $\mathcal{A}' := \mathrm{Det}(\bigsqcup_i \mathcal{D}'_i)$.

IV. APPLICATIONS

We now present four applications to illustrate how the stack-based Algorithm 1 and its transducer approximation (Algorithms 2 and 3) solve the multi-regular language filtering problem. The first is the cellular automaton ECA 110, shown previously. Its rather large filtering transducer is quite tedious to

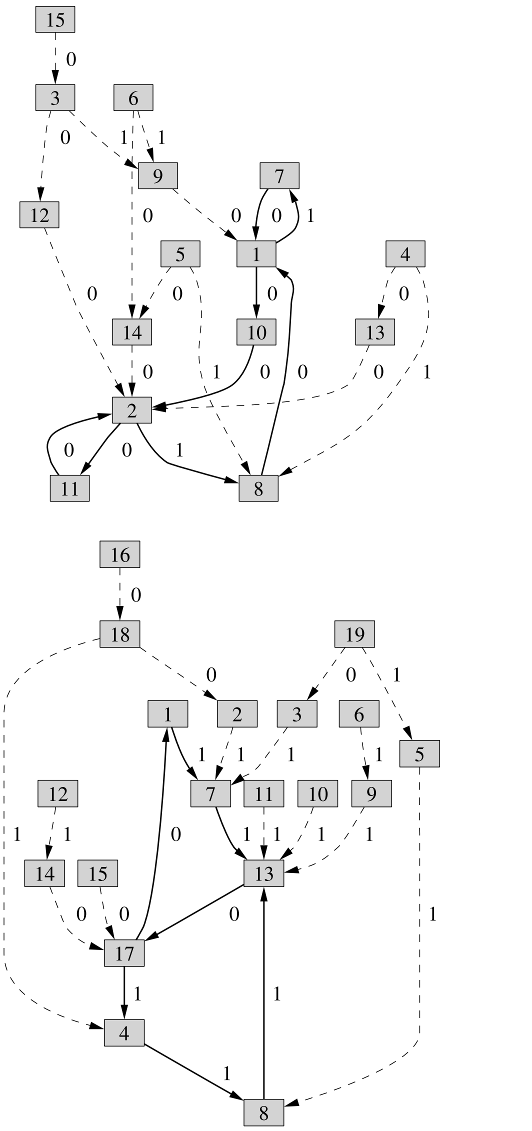

FIG. 8: The equivalent, non-minimal domains $\{\mathcal{D}_1', \mathcal{D}_2'\} = \text{Optimize}(\{\mathcal{D}_1, \mathcal{D}_2\})$ obtained by applying Algorithm 3 to the positive-entropy domains $\mathcal{D}_1$ and $\mathcal{D}_2$ in Fig. 7. ($\mathcal{D}_1'$ (top) and $\mathcal{D}_2'$ (bottom).) The "Start" arrows are omitted for clarity (all states are starting), and some of the transitions are drawn with dashed arrows to help the reader distinguish the recurrent states.

construct by hand, but Algorithm 2 produces it handily. The second example, ECA 18, which we have also already seen, illustrates the stack-based Algorithm 1's ability to detect overlapping domains. The third example shows our methods' power to detect structures in the midst of apparent randomness: the domains and sharp boundaries between them are identified easily despite the fact that the domains themselves have positive entropy and their boundaries move stochastically. The example shows the use of—and need for—domain-preprocessing (Algorithm 3). That is, rapid resynchronization is achieved using a filter built from optimized, non-minimal domains. The final example demonstrates the transducer (constructed by Algorithms 2 and 3) detecting domains in a multi-stationary process—what is called the change-point problem in statistical time-series analysis. This example emphasizes that the methods developed here are not limited to cellular automata. More importantly, it highlights several of the subtleties of multi-regular language filtering and clearly illustrates the need for the domain-preprocessing Algorithm 3.

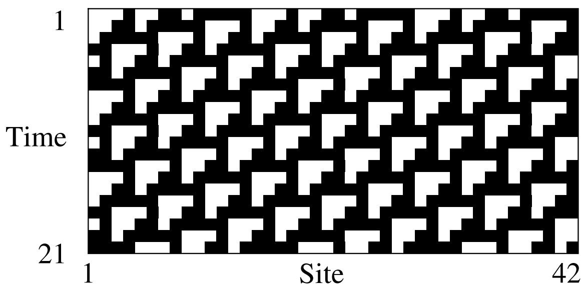

FIG. 9: ECA 110's principal domain, $\text{sub}[(00010011011111)^*]$.

ECA 110

First consider ECA 110, illustrated earlier in Fig. 1. Its domains are easy to see visually; they have the form $\text{sub}(w^*)$ for some finite word w. Its dominant domain is $\text{sub}(w^*) = \text{sub}[(00010011011111)^*]$, illustrated in Fig. 9. In fact, the transducer $\text{Filter}(\{\text{sub}[(00010011011111)^*]\})$, constructed from this single domain, filters ECA 110's space-time behavior well; see Fig. 10.

Notice, in that figure, the wide variety of particle-like domain defects that the filtered version lays bare. Note, moreover, how these particles move and collide according to consistent rules. These particles are important to ECA 110's computational properties; a subset can be used to implement a Post Tag system [23] and thus simulate arbitrary Turing machines [24].

ECA 18

Next, consider ECA 18, illustrated earlier in Fig. 2. It is somewhat more challenging to filter, because its domain $\mathcal{D} = \text{sub}\left([0(0+1)]^*\right)$ has positive entropy. As a result, its particles are difficult—although by no means impossible—to see with the naked eye. Nevertheless, the stack-based algorithm filters its space-time diagrams extremely well, as illustrated in Fig. 2 (right). There, black rectangles are drawn where maximal substrings overlap, and vertical bars are drawn where maximal substrings abut. As men-

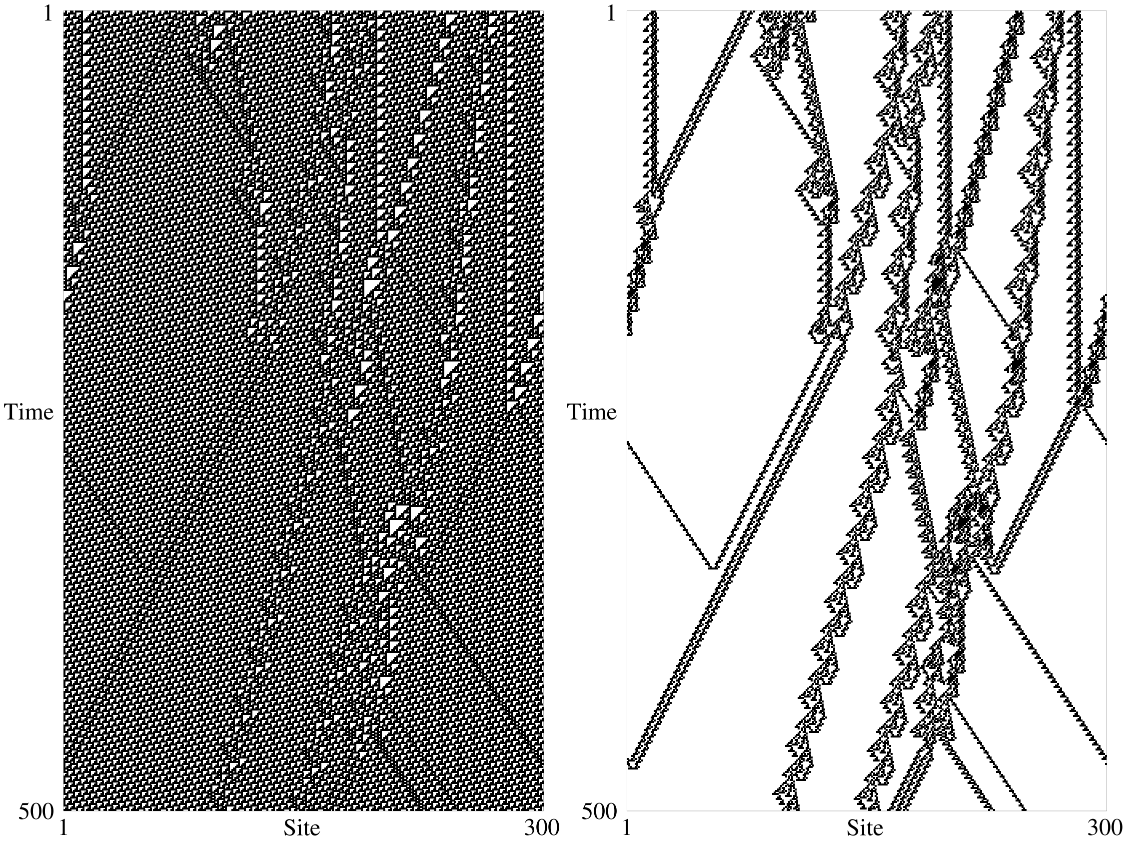

FIG. 10: An ECA 110 space-time diagram (left) filtered by the transducer Filter($\{\text{sub}[(00010011011111)^*]\}$) (right).

tioned earlier, these particles, whose precise location is somewhat ambiguous, follow random walks and pairwise annihilate whenever they touch [20, 21, 22].

It is worth mentioning that the transducer Filter($\{\mathcal{D}\}$) produces a less precise filtrate in this case—and that Filter(Optimize($\{\mathcal{D}\}$)) does no better. Indeed, since breaks in ECA 18's domain have the form $\cdots 1(0^{2n})1\cdots$, the precise location of the domain break is ambiguous: if reading left-to-right, it does not occur until the 1 on the right of 0^{2n} is read; whereas, if reading right-to-left, it does not occur until the 1 on the left is read. In other words, if reading left-to-right, the transducer Filter($\{\mathcal{D}\}$) detects only the right edges of the black triangles of Fig. 2 (right). Similarly, if reading right-to-left, it detects only the left edges of these triangles. In this case it is possible to fill in the space between these pairs of edges to obtain the output of the stack-based algorithm.

CA 2614700074

Now consider the binary, next-to-nearest neighbor (i.e. k=r=2) CA 2614700074, shown in Fig. 11. Crutchfield and Hanson constructed it expressly to have the positive-entropy domains $\mathcal{D}_1$ and $\mathcal{D}_2$ in Fig. 7 [19].

As illustrated in Fig. 11, the optimal transducer Filter(Optimize($\{\mathcal{D}_1, \mathcal{D}_2\}$)) filters this CA's output well. This illustrates a practical advantage of multi-regular language filtering: it can detect structure embedded in randomness. Notice how the filter easily identifies the domains and sharp boundaries separating them, even though the domains themselves have positive entropy and their boundaries move stochastically.

It is worth noting that in place of the gray regions of Fig. 11 so clearly identified by the optimal transducer as corresponding to the second domain $\mathcal{D}_2$, the simpler transducer Filter($\{\mathcal{D}_1, \mathcal{D}_2\}$) produces a regular checkering of false domain breaks (not pictured). This is because, when examining the sole forbidden transition $(s, a) = (2, 1)$ of the first domain $\mathcal{D}_1$, Algorithm 2 discovers that the first non-empty set $S_{i=1,l=4} = \{2, 4, 5\}$ contains three resynchronization states. It unfortunately abandons both states 4 and 5, which belong to the second domain, instead choosing to resynchronize to the original state 2 itself, because it occurs alone in the next set $S_{1,5}$. As a result, the transducer Filter($\{\mathcal{D}_1, \mathcal{D}_2\}$) has no transitions leaving the first domain whatsoever and is therefore incapable of detecting jumps from the first domain to the second. This is why it prints a checkering of domain breaks instead of correctly resynchronizing to the second domain. The optimal transducer does not suffer from this problem, because Algorithm 3 splits

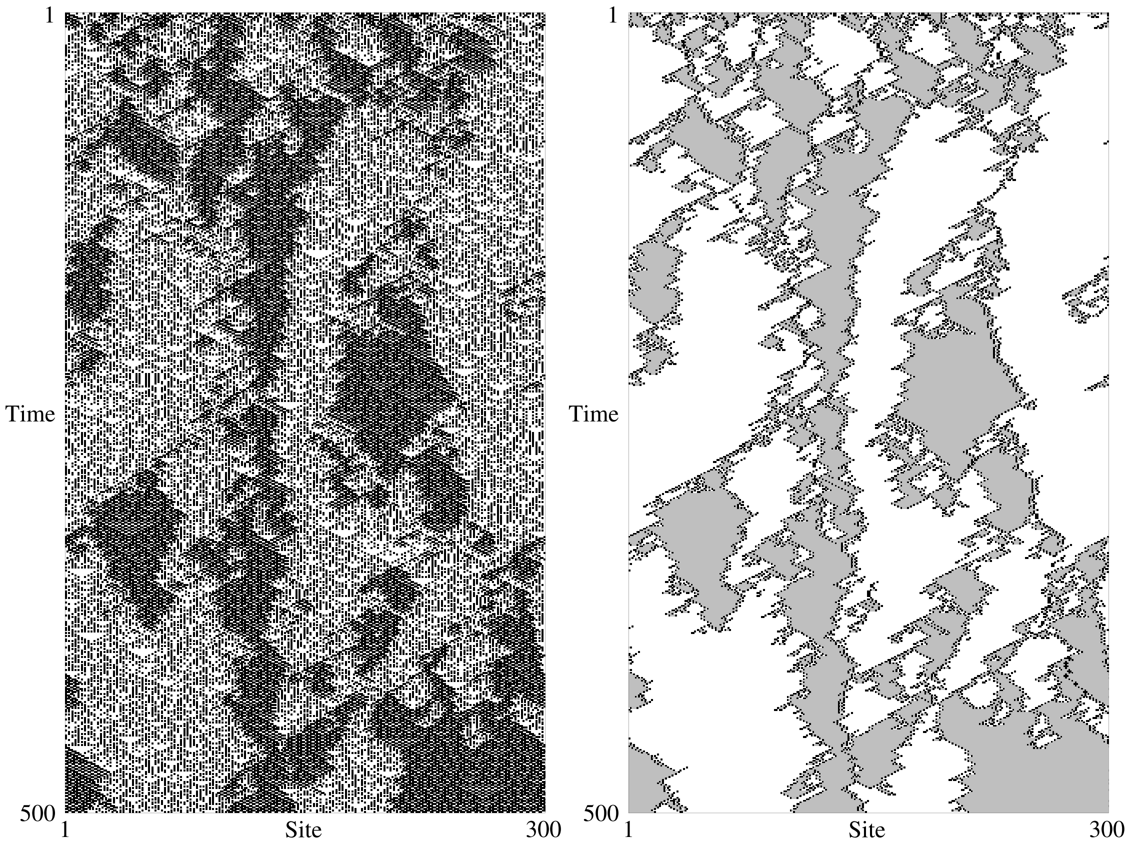

FIG. 11: Binary, next-to-nearest neighbor CA 2614700074 space-time diagram (left) filtered by the transducer Filter(Optimize($\{\mathcal{D}_1, \mathcal{D}_2\}$)) (right). The white regions on the right correspond to the domain $\mathcal{D}_1$, the gray to the domain $\mathcal{D}_2$. The black squares separating these regions correspond to the interruption symbols $h'(s, s')$ that the transducer emits between domains.

state 2 into several new ones, from which unambiguous resynchronization to the appropriate state—2, 4, or 5—is possible.

Change-Point Problem: Filtering Multi-Stationary Sources

Leaving cellular automata behind, consider a binary information source that hops with low probability between the two three-state domains $\mathcal{D}_1$ and $\mathcal{D}_2$ in Fig. 12 (top). This source allows us to illustrate subtleties in multi-regular language filtering and, in particular, in the construction of the optimal transducer Filter(Optimize($\{\mathcal{D}_i\}$) can be.

To appreciate how subtle filtering with the domains $\mathcal{D}_1$ and $\mathcal{D}_2$ is—and why the extra states of Optimize($\{\mathcal{D}_1, \mathcal{D}_2\}$) are needed to do it—consider the following. First choose any finite word w of the form:

$$(0^6 + 0^3 1^2)^* 0^3 1^2 0.$$

As the ambitious reader can verify, both of the strings $101111w$ and $110w$ belong to the domain $\mathcal{D}_2$. In fact, both correspond to unique paths through $\mathcal{D}_1 \sqcup \mathcal{D}_2$ ending in state 5 of Fig. 12 (top).

On the other hand, the strings $01111w1$ and $10w1$ are also domain words—the first belonging to $\mathcal{D}_2$, but the second belonging to $\mathcal{D}_1$. In fact, $01111w1$ corresponds to a unique path through $\mathcal{D}_1 \sqcup \mathcal{D}_2$ ending in *state 6*, while $10w1$ corresponds to a unique path ending in *state 3*.

As a result, these four strings are the maximal substrings of the non-domain strings $101111w1$ and $110w1$, as indicated by the brackets below:

corresponds to a unique path
through $\mathcal{D}_2$ ending in state 5

$$\rlap{$\overbrace{1\,0\,1\,1\,1\,1\,w}$}1\,\underbrace{0\,1\,1\,1\,1\,w\,1}$$

corresponds to a unique path
through $\mathcal{D}_2$ ending in *state 6*

corresponds to a unique path
through $\mathcal{D}_2$ ending in state 5

$$\rlap{$\overbrace{1\,1\,0\,w}$}1\,\underbrace{1\,0\,w\,1}$$

corresponds to a unique path
through $\mathcal{D}_1$ ending in *state 3*

This example illustrates several important points. First of all, it shows that when the naive transducer

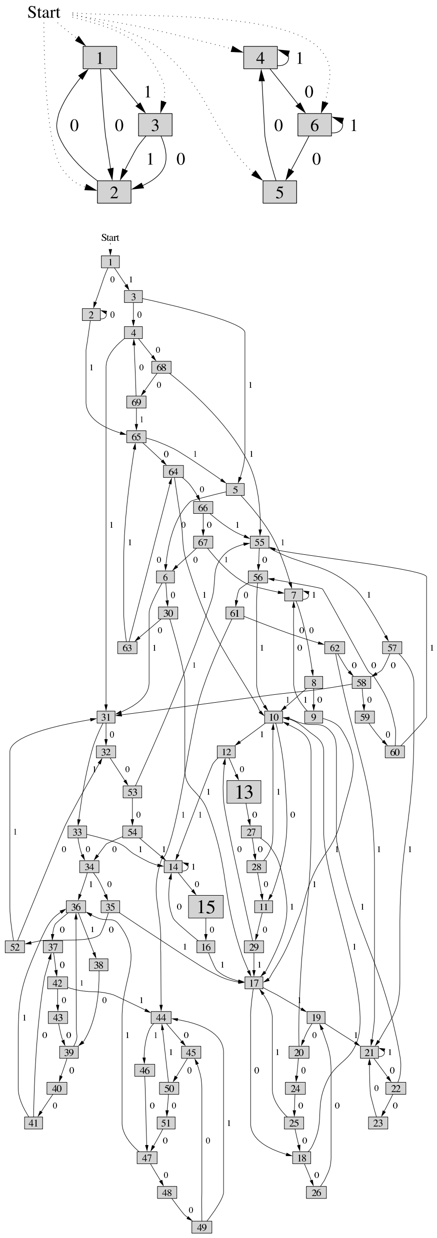

FIG. 12: Two similar three-state domains $\mathcal{D}_1$ (top left) and $\mathcal{D}_2$ (top right) illustrate how subtle the construction of the optimal transducer $\text{Filter}(\text{Optimize}(\{\mathcal{D}_i\}))$ can be: the automaton $\mathcal{A}' := \text{Det}(\bigsqcup \text{Optimize}(\{\mathcal{D}_1, \mathcal{D}_2\}))$ (below), from which the optimal transducer is constructed, has 69 states—the unoptimized automaton $\mathcal{A} := \text{Det}(\mathcal{D}_1 \sqcup \mathcal{D}_2)$ (not pictured) has 30.

$\text{Filter}(\{\mathcal{D}_1, \mathcal{D}_2\})$ reaches the forbidden letter 1 at the end of either of these two strings, the state 2 reached does not preserve enough information to resynchronize to the appropriate state—3 or 6, respectively. As a result, it must either make a guess—at the risk of choosing incorrectly and then later reporting an artificial domain break (as in the preceding cellular automaton example)—or else jump to one of its nonrecurrent states, emitting a potentially long chain of λs until it can re-infer from future input what was already determined by past input.

As unsettling as this may be, the example illustrates something far more nefarious. Since an arbitrarily long word w can be chosen, it is impossible to fix the problem by splitting the states of $\text{Filter}(\{\mathcal{D}_1, \mathcal{D}_2\})$ so as to buffer finite windows of past input. In fact, because w is chosen from a language with positive entropy, the number of windows that would need to be buffered grows exponentially.

At this point achieving optimal resynchronization might seem hopeless, but it actually is possible. This is what makes Algorithm 3—and in particular the proof that it terminates (Prop. 6)—not only surprising, but extremely useful.

Indeed, recall that instead of splitting states according to finite windows, Algorithm 3 splits them according to entire regular languages of past input and that, by Prop. 6, a finite number of these regular languages will always suffice to achieve optimal resynchronization. And so, instead of reaching the same original state 2 when reading the strings $101111w$ and $110w$, the optimal transducer $\text{Filter}(\text{Optimize}(\{\mathcal{D}_1, \mathcal{D}_2\}))$ reaches two distinct states $(2, \mathcal{E})$ and $(2, \mathcal{E}')$, where $101111w \in \text{Lang}(\mathcal{E})$ and $110w \in \text{Lang}(\mathcal{E}')$. These two split states are labeled with the enlarged integers 15 and 13, respectively, in Fig. 12 (bottom), which shows $\mathcal{A}' := \text{Det}(\bigsqcup \text{Optimize}(\{\mathcal{D}_1, \mathcal{D}_2\}))$—the automaton from which $\text{Filter}(\text{Optimize}(\{\mathcal{D}_1, \mathcal{D}_2\}))$ is constructed. As illustrated in that figure, the optimal transducer has 69 states—the unoptimized automaton $\mathcal{A} := \text{Det}(\mathcal{D}_1 \sqcup \mathcal{D}_2)$ (not pictured) has 30.

V. CONCLUSION

We posed the multi-regular language filtering problem and presented two methods for solving it. The first, although providing the ideal solution, requires a stack, has a worst-case compute time that grows quadratically in string length and conditions its output at any point on arbitrarily long windows of future input. The second method was to algorithmically construct a transducer that approximates the first algorithm. In contrast to the stack-based algorithm it approximates, however, the transducer requires only a finite amount of memory, runs in linear time, and gives immediate output for each letter read—significant improvements for cellular automata structural analysis and, we suspect, for other applications as well. It is, moreover, the best possible approximation with these three features. Finally, we applied both methods to the computational-mechanics structural analysis of cellular automata and to a version of the change-point problem from time-series analysis.

Future directions for this work include generalization both to probabilistic patterns and transducers and to higher dimensions. Although both seem difficult, the latter seems most daunting—at least from the standpoint of transducer construction—because there is as yet no consensus on how to approach the subtleties of high-dimensional automata theory. (See, for example, Refs. [25] and [26] for discussions of two-dimensional generalizations of regular languages and patterns.) Note, however, that the basic notion of maximal substrings underlying the stack-based algorithm is easily generalized to a broader notion of higher-dimensional maximal connected subregions, although we suspect that this generalization will be much more difficult to compute.

In the introduction we alluded to a range of additional applications of multi-regular language filtering. Segmenting time series into structural components was illustrated by the change-point example. This type of time series problem occurs in many areas, however, such as in speech processing where the structural components are hidden Markov models of phonemes, for example, and in image segmentation where the structural components are objects or even textures. One of the more promising areas, though, is genomics. In genomics there is often quite a bit of prior biochemical knowledge about structural regions in biosequences. Finally, when coupled with statistical inference of stationary domains, so that the structural components are estimated from a data stream, multi-regular language filtering should provide a powerful and broadly applicable pattern detection tool.

Acknowledgments

This work was supported at the Santa Fe Institute under the Networks Dynamics Program funded by the Intel Corporation and under the Computation, Dynamics, and Inference Program via SFI's core grants from the National Science and MacArthur Foundations. Direct support was provided by DARPA Agreement F30602-00-2-0583.

APPENDIX A: AUTOMATA THEORY PRELIMINARIES

In this appendix we review the definitions and results from automata theory that are essential to our exposition. A good source for these preliminaries is Ref. [27], although its authors employ altogether different notation, which does not suit our needs.

Automata

An *automaton* $\mathcal{A}$ over an alphabet $\Sigma(\mathcal{A})$ is a collection of states $S(\mathcal{A})$, together with subsets $\mathrm{Start}(\mathcal{A}), \mathrm{Final}(\mathcal{A}) \subset S(\mathcal{A})$, and a collection of transitions $T(\mathcal{A}) \subset S(\mathcal{A}) \times \Sigma(\mathcal{A}) \times S(\mathcal{A})$. We call an automaton *finite* if both $S(\mathcal{A})$ and $T(\mathcal{A})$ are.

An automaton $\mathcal{A}$ *accepts* a string $\sigma = a_1 a_2 \cdots a_n$ if there is a sequence of transitions $(s_1, a_1, s_2), (s_2, a_2, s_3), \ldots, (s_{n-1}, a_n, s_n) \in T(\mathcal{A})$ such that $s_1 \in \mathrm{Start}(\mathcal{A})$ and $s_n \in \mathrm{Final}(\mathcal{A})$. Denote the collection of all strings that $\mathcal{A}$ accepts by $\mathrm{Lang}(\mathcal{A})$. Two automata $\mathcal{A}$ and $\mathcal{B}$ are said to be *equivalent* if $\mathrm{Lang}(\mathcal{A}) = \mathrm{Lang}(\mathcal{B})$.

We can think of an automaton as a directed graph whose edges are labeled with symbols from $\Sigma(\mathcal{A})$. In this view, an automaton accepts precisely those strings that correspond to paths through its graph beginning in its start states and ending in its final ones.

An automaton $\mathcal{A}$ is said to be *semi-deterministic* if any pair of its transitions that agree in the first two slots are identical, that is, any pair of transitions of the form (s_1, a, s_2) and $(s_1, a, s_2') \in T(\mathcal{A})$ satisfy $s_2 = s_2'$. A *deterministic* automaton is one that is semi-deterministic and that has a single start state. If $\mathcal{A}$ is deterministic, then each string of $\mathrm{Lang}(\mathcal{A})$ corresponds to precisely one path through $\mathcal{A}$'s graph.

For two automata $\mathcal{A}$ and $\mathcal{B}$, let $\mathcal{A} \sqcup \mathcal{B}$ denote their *disjoint union*—the automaton over the alphabet $\Sigma(\mathcal{A}) \cup \Sigma(\mathcal{B})$ whose states are the disjoint union of the states of $\mathcal{A}$ and $\mathcal{B}$, i.e. $S(\mathcal{A} \sqcup \mathcal{B}) = S(\mathcal{A}) \sqcup S(\mathcal{B})$ (and similarly for its start and final states) and whose transitions are the union of the transitions of $\mathcal{A}$ and $\mathcal{B}$. In this way, $\mathrm{Lang}(\mathcal{A} \sqcup \mathcal{B}) = \mathrm{Lang}(\mathcal{A}) \cup \mathrm{Lang}(\mathcal{B})$.

In this terminology, a *domain* is a semi-deterministic finite automaton $\mathcal{D}$ whose states are all start and final states, i.e. $\mathrm{Start}(\mathcal{D}) = S(\mathcal{D}) = \mathrm{Final}(\mathcal{D})$, and whose graph is strongly connected—i.e., there is a path from any one state to any other.

Finally, a domain $\mathcal{D}$ is said to be *minimal* if all equivalent domains $\mathcal{D}'$ satisfy $|S(\mathcal{D})| \leq |S(\mathcal{D}')|$.

Standard Results

Lemma 2. *Every automaton $\mathcal{A}$ is equivalent to a deterministic automaton* $\mathrm{Det}(\mathcal{A})$. *Moreover,* $\mathrm{Det}(\mathcal{A})$*'s states correspond uniquely to collections of $\mathcal{A}$'s states; in other words, there is a canonical injection* $S(\mathrm{Det}(\mathcal{A})) \overset{\psi_{\mathrm{Det}(\mathcal{A})}}{\hookrightarrow} \{S : S \subset S(\mathcal{A})\}$.

Lemma 3. *If $\mathcal{A}$ and $\mathcal{B}$ are automata, then there is an automaton $\mathcal{A} \cap \mathcal{B}$ that accepts precisely those strings accepted by both $\mathcal{A}$ and $\mathcal{B}$; that is,* $\mathrm{Lang}(\mathcal{A} \cap \mathcal{B}) = \mathrm{Lang}(\mathcal{A}) \cap \mathrm{Lang}(\mathcal{B})$. *If $\mathcal{A}$ and $\mathcal{B}$ are deterministic, then so is $\mathcal{A} \cap \mathcal{B}$. Moreover, there is a canonical injection $S(\mathcal{A} \cap \mathcal{B}) \hookrightarrow S(\mathcal{A}) \times S(\mathcal{B})$, which restricts to* *injections* $\mathrm{Start}(\mathcal{A} \cap \mathcal{B}) \hookrightarrow \mathrm{Start}(\mathcal{A}) \times \mathrm{Start}(\mathcal{B})$ *and* $\mathrm{Final}(\mathcal{A} \cap \mathcal{B}) \hookrightarrow \mathrm{Final}(\mathcal{A}) \times \mathrm{Final}(\mathcal{B})$.

Transducers

A *transducer* $\mathcal{T}$ from an alphabet $\Sigma(\mathcal{T})$ to an alphabet $\Sigma'(\mathcal{T})$ is an automaton on the alphabet $\Sigma(\mathcal{T}) \times \Sigma'(\mathcal{T})$. We will use the more traditional notation $(s, b|c, s')$ in place of $(s, (b, c), s') \in T(\mathcal{T})$.

The *input* of a transducer $\mathcal{T}$ is the automaton $\mathrm{In}(\mathcal{T})$ whose states, start states, and final states are the same as $\mathcal{T}$'s, but whose transitions are given by $T(\mathrm{In}(\mathcal{T})) := \{(s, b, s') : (s, b|c, s') \in T(\mathcal{T})\}$. Similarly, the *output* of a transducer $\mathcal{T}$ is the automaton $\mathrm{Out}(\mathcal{T})$ whose transitions are given by $T(\mathrm{Out}(\mathcal{T})) := \{(s, c, s') : (s, b|c, s') \in T(\mathcal{T})\}$.

A transducer $\mathcal{T}$ is said to be *well defined* if $\mathrm{In}(\mathcal{T})$ is deterministic, because such a transducer determines a function from $\mathrm{Lang}(\mathrm{In}(\mathcal{T}))$ onto $\mathrm{Lang}(\mathrm{Out}(\mathcal{T}))$.

APPENDIX B: IMPLEMENTATION

In order to give the reader a sense for how the algorithms can be implemented, we rigorously implement Algorithm 2 here in the programming language Haskell [28]. Haskell represents the state of the art in polymorphicly typed, lazy, purely functional programming language design. Its concise syntax enables us to implement the algorithm in less than a page. Haskell compilers and interpreters are freely available for almost any computer [30].

We emulate our exposition in the preceding sections by representing a finite automaton as a list of starting states, a list of transitions, and a list of final states, and a transducer as a finite automaton whose alphabet consists of pairs of symbols:

```
data FA s i = FA{faStarts :: [s],
                 faTrans :: [(s, i, s)],
                 faFinals :: [s]}

type Transducer s i o = FA s (i, o)
```

We need the following simple functions, which compute the list of symbols and states present in an automaton:

```
faAlphabet :: Eq i ⇒ FA s i → [i]
faAlphabet fa = nub [a | (_, a, _) ← faTrans fa]

transStates :: Eq s ⇒ [(s, i, s)] → [s]
transStates trans =
    nub $ foldl (λss (s, _, s') → s : s' : ss) [] trans

faStates :: Eq s ⇒ FA s i → [s]
faStates fa = foldl union [] [faStarts fa,
                              transStates $ faTrans fa,
                              faFinals fa]
```

We also require the following three functions: the first two implement Lemmas 2 and 3, and the third computes the disjoint union of a list of automata. To expedite our exposition, we provide only their type signatures:

```
faDet :: (Ord s, Eq i) ⇒ FA s i → FA [s] i
faIntersect :: (Eq s, Eq s', Eq i) ⇒
                          FA s i → FA s' i → FA (s, s') i
faDisjointUnion :: [FA s i] → FA (Int, s) i
```

Notice that the first two functions return automata whose states are represented as lists and pairs of the argument automata's states; these representations intrinsically encode the Lemmas' canonical injections. The function *faDisjointUnion* returns an automaton whose states are represented as pairs (i, s) where s is a state of the ith argument automaton $\mathcal{D}_i$.

We use these representations frequently in the following implementation of Algorithm 2:

```
transducerFilterFromDomains :: (Eq s, Ord s, Eq i) ⇒ [FA s i] → Transducer [(Int, s)] i Int
transducerFilterFromDomains faDs =
    FA (faStarts faA) (baseTTrans ++ newTTrans) (faFinals faA)
    where faA = faDet $ faDisjointUnion faDs      -- 𝒜 := Det(𝒟₁ ⊔ ··· ⊔ 𝒟ₙ)
          baseTTrans = [(s, (a, f s'), s') | (s, a, s') ← faTrans faA]
              where f ss | length is ≡ 1 = head is      -- transition ending in 𝒟ᵢ
                         | otherwise = 0      -- synchronization, λ
                         where is = nub $ map fst ss
          forbiddenPairs = [(s, a) | s ← faStates faA, a ← faAlphabet faA] \\ [(s, a) | (s, a, _) ← faTrans faA]
          newTTrans = map newTransition forbiddenPairs
          newTransition (s, a) = (s, (a, o), s')
              where faAsa = (FA (f : faStates faA) ((s, a, f) : faTrans faA) [f])      -- the automaton 𝒜^{s,a}
                    f = [(1 + length faDs, head $ faStarts $ head faDs)]      -- the fresh state f used to build 𝒜^{s,a}
                    faDetAsaCapA = (faDet faAsa) `faIntersect` faA      -- the automaton Det(𝒜^{s,a} ∩ 𝒜)
                    faZ = faDet $ zero faDetAsaCapA      -- the automaton Det(𝒵[Det(𝒜^{s,a} ∩ 𝒜)])
```

$$
\begin{aligned}
&\quad\quad reachableStateSeq = take\ (length\ \$\ faStates\ faZ) \quad \text{-- the states } s_1, \ldots, s_{m+m'} \\
&\quad\quad\quad\quad (iterate\ nextState\ \$\ head\ \$\ faStarts\ faZ) \\
&\quad\quad\quad \mathbf{where}\ nextState\ s = head\ [s'' \mid (s', \_, s'') \leftarrow faTrans\ faZ, s' \equiv s] \\
&\quad\quad sStarLs = map\ ((map\ snd) \circ \quad \text{-- the sets } \{S_{*,l}\}_{l=1}^{m+m'} \\
&\quad\quad\quad\quad (intersect\ \$\ faFinals\ faDetAsaCapA))\ reachableStateSeq \\
&\quad\quad siStars = [nub \quad \text{-- the sets } \{S_{i,*}\}_{i=1} \\
&\quad\quad\quad\quad [s \mid s \leftarrow map\ snd\ \$\ faFinals\ faDetAsaCapA, length\ s \equiv i] \\
&\quad\quad\quad\quad \mid i \leftarrow [1\,..\,length\ \$\ head\ \$\ faStarts\ faA]] \\
&\quad\quad sijs = concatMap\ (\lambda siStar \rightarrow map\ (intersect\ siStar)\ sStarLs)\ siStars \quad \text{-- the sets } \{S_{i,j}\}_{i,j} \\
&\quad\quad [s'] = head\ \$\ filter\ (\lambda sij \rightarrow length\ sij \equiv 1)\ sijs \quad \text{-- the state } s' \text{ to which to synchronize} \\
&\quad\quad o = -1 \quad \text{-- domain break} \\
&zero\ fa = FA\ (faStarts\ fa)\ [(s, 0, s') \mid (s, \_, s') \leftarrow faTrans\ fa]\ (faFinals\ fa)
\end{aligned}
$$

To implement the domain optimization algorithm Optimize(•) is somewhat more—but not overwhelmingly—complicated. In fact, we generated the examples here by computer, rather than by hand.

[1] S. Zacks. Survey of classical and bayesian approach to the change-point problem: Fixed sample and sequential procedures of testing and estimation. In *Recent Advances in Statistics*, pages 245–269. Academic Press, 1983.

[2] P. Baldi and S. Brunak. *Bioinformatics—The Machine Learning Approach*. Bradford Books, MIT Press, Cambridge, Massachusetts, 2nd edition, 1997.

[3] S. Furui. *Digital Speech Processing, Synthesis, and Recognition*. Marcel Dekker, New York, 1989.

[4] H. Freeman. Computer processing of line-drawing images. *Computing Surveys*, 6(1):57–97, 1974.

[5] J. P. Crutchfield. The calculi of emergence: Computation, dynamics, and induction. *Physica D*, 75:11–54, 1994.

[6] S. Wolfram. Statistical mechanics of cellular automata. *Review of Modern Physics*, 55:601–644, 1983.

[7] A. W. Burks, editor. *Essays on Cellular Automata*. Univerity of Illinois Press, Urbana, IL, 1970.

[8] J. D. Farmer, T. Toffoli, and S. Wolfram, editors. *Cellular Automata: Proceedings of an Interdisciplinary Workshop*. North Holland, Amsterdam, 1984.

[9] F. Fogelman-Soulie, Y. Robert, and M. Tchuente, editors. *Automata Networks in Computer Science: Theory and Applications*. Manchester University Press, Manchester, UK, 1987.

[10] H. A. Gutowitz, editor. *Cellular Automata*. MIT Press, Cambridge, MA, 1990.

[11] T. Karapiperis and B. Blankleider. Cellular automaton model of reaction-transport processes. *Physica D*, 78:30–64, 1994.

[12] K. Nagel. Particle hopping models and traffic flow theory. *Phys. Rev. E*, 53:4655–4672, 1996.

[13] C. Jesshope, V. Jossifov, and W. Wilhelmi, editors. *International Workshop on Parallel Processing by Cellular Automata and Arrays (Parcella '94)*. Akademie Verlag, Berlin, 1994.

[14] J. P. Crutchfield, M. Mitchell, and R. Das. The evolutionary design of collective computation in cellular automata. In *Evolutionary Dynamics—Exploring the Interplay of Selection, Neutrality, Accident, and Function*, Santa Fe Institute Series in the Sciences of Complexity, pages 361–411. Oxford University Press, 2001.

[15] T. Toffoli and N. Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, Cambridge, MA, 1987.

[16] S. Wolfram, editor. *Theory and Applications of Cellular Automata*. World Scientific, Singapore, 1986.

[17] J. P. Crutchfield and K. Young. Inferring statistical complexity. *Physical Review Letters*, 63:105, 1989.

[18] J. P. Crutchfield and C. R. Shalizi. Thermodynamic depth of causal states: Objective complexity via minimal representations. *Physical Review E*, 59(1):275–283, 1999.

[19] J. P. Crutchfield and J. E. Hanson. Turbulent pattern bases for cellular automata. *Physica D*, 69:279–301, 1993.

[20] J. E. Hanson and J. P. Crutchfield. The attractor-basin portrait of a cellular automaton. *J. Stat. Phys.*, 66:1415–1462, 1992.

[21] J. P. Crutchfield and J. E. Hanson. Attractor vicinity decay for a cellular automaton. *CHAOS*, 3(2):215–224, 1993.

[22] K. Eloranta. The dynamics of defect ensembles in one-dimesional cellular automata. *Journal of Statistical Physics*, 76(5/6):1377, 1994.

[23] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

[24] M. Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[25] D. P. Feldman and J. P. Crutchfield. Structural information in two-dimensional patterns: Entropy convergence and excess entropy. *Phys. Rev. E*, 67(5):051103, 2003.

[26] K. Lindgren, C. Moore, and M. Nordahl. Complexity of two-dimensional patterns. *Journal of Statistical Physics 91*, 91(5-6):909–951, 1998.

[27] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, 1979.

[28] S. L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003.

[29] Accepting Σ^* is rather generous: the filter will take any string and label its symbols according to the hypothesized patterns $\{\mathcal{D}_i\}$. This is not required for cellular automata, since their configurations often contract to strict subsets of Σ^* over time.

[30] For Haskell compilers and interpreters, please visit www.haskell.org.