If, under the action of f, the volumes of regions in state space contract, it is **dissipative**; the term is used in analogy with physical systems in which energy is not conserved. It is a well-known result from the theory of differential equations that the flow $\phi(t, \mathrm{x}(0))$ on a compact set $\mathcal{C} \subseteq \mathcal{X}$ exists and is unique for each $\mathrm{x}(0) \in \mathcal{C}$ and can be uniquely extended backward and forward in time up to the boundary of $\mathcal{C}$.[51]

A dynamical system is **linear** if the vector sum of two orbits, $\mathrm{x}_t + \mathrm{y}_t$ for maps or $\phi(t, \mathrm{x}(0)) + \phi(t, \mathrm{y}(0))$ for flows, is itself an orbit. A system that is not linear is **nonlinear**. For maps, the dynamic of a linear system can be written $\mathrm{x}_t = F\mathrm{x}_{t-1}$ where F is an $N \times N$ matrix with constant entries. For flows, a linear system is expressed by $\dot{\mathrm{x}} = F\mathrm{x}$, where again F is an $N \times N$ matrix with constant entries.

The number of independent quantities necessary to exactly specify the state x is the **dimension** N of the state space. The dimension is related to the number d of **degrees of freedom** as defined in energy-conserving systems by $N = 2d$.

The close connection of flows and maps is further demonstrated by a construction called the **Poincaré surface of section**. This is defined for some flows, especially oscillations and flows near periodic orbits (see below). For such a flow in $\mathcal{X} \subset \mathbb{R}^N$, the Poincaré surface of section is a hyperplane $\Gamma \subset \mathbb{R}^{N-1}$ that cuts $\mathcal{X}$ in such a way that all the orbits $\mathrm{x}(t)$ cross Γ transversely. For example, in $\mathbb{R}^3$, the surface of section is a plane, and an orbit intersects Γ in a single point. The timeseries of successive crossings (x_t, y_t) of Γ is then a discrete-time mapping that gives a summary of the systems's progress along the orbit. Finding a Poincaré surface of section for a given flow can be difficult in general and is sometimes impossible. On the other hand, any mapping f can be embedded in some flow g in such a way that f is a Poincaré surface of section of g. The flow g is called the **suspension** of f.

3.2 Invariant Sets

This section presents a short listing of some of the more important structural features in the state space of a dynamical system: the points and sets that are invariant under the dynamic. These are a large part of the flora and fauna of qualitative dynamics.

A general invariant set U is defined by the property of being mapped exactly onto itself by the dynamic:

$$\begin{cases} U = f(U) & (maps) \\ U = \phi(t, U), \ \forall\, t \geq 0 & (flows) \end{cases} \tag{3.5}$$

Some of the different kinds of invariant sets are listed below.

1. **Fixed point**: for maps, a state $\mathrm{x} \in \mathcal{X}$ is a fixed point if $f(\mathrm{x}) = \mathrm{x}$; for flows, $f(\mathrm{x}) = 0$ so that $\mathrm{x}(t) = \mathrm{x}(0) = \mathrm{x}$ for all $t > 0$.
2. **Periodic orbit**: for maps, $\mathrm{x} \in \mathcal{X}$ is a periodic point if $f^p(\mathrm{x}) = \mathrm{x}$ for some integer p. The orbit $\{f^t(\mathrm{x}), 0 \leq t < p\}$ is a periodic orbit. For flows, a periodic point has $\mathrm{x}(t + \tau) = \mathrm{x}(t)$ for some $\tau > 0$.

3. **Quasiperiodic motion.** In the simplest case, flow on a two dimensional torus, the states are given by $\mathrm{x} = (x, y)$, and the flow is periodic in both x and y: $x(t + \tau_x) = x(t)$ and $y(t + \tau_y) = y(t)$ with τ_x and τ_y incommensurate—that is, τ_x/τ_y is an irrational number. If τ_x and τ_y are commensurate, orbits are periodic. However, for τ_x/τ_y irrational, an orbit spirals around the torus infinitely many times, never quite meeting up with itself to form a periodic orbit. For maps, consider a Poincaré surface of section of a system undergoing quasiperiodic motion on a 2–dimensional torus: the sequence of successive intersections of the flow with the surface of section fills out a 1–dimensional curve.
4. **Nonwandering set.** A state x is a **nonwandering point** if, for any neighborhood U of x, there exists some $t \neq 0$ such that

$$\begin{cases} \phi(t, U) \cap U \neq \emptyset & (flows) \\ f^t(U) \cap U \neq \emptyset & (maps) \end{cases} \tag{3.6}$$

   The set of all nonwandering points in the state space is the **nonwandering set** of the system. The orbit of a nonwandering point returns to the neighborhood of the point infinitely often. Fixed points and periodic points are nonwandering.
5. **Limit set.** A state x_∞ is called an ***ω*** **limit point** of $\mathrm{x} \in \mathcal{X}$, if there exists a sequence $\{t_i\}$, $t_i \to \infty$ such that:

$$\begin{cases} \phi(t_i, \mathrm{x}) \to \mathrm{x}_\infty & (flows) \\ f^{t_i}(\mathrm{x}) \to \mathrm{x}_\infty & (maps) \end{cases} \tag{3.7}$$

   For a set $V \subseteq \mathcal{X}$, the ***ω*** **limit set** of V, denoted by $L_\omega(V)$, is the set of ω limit points of all states in V.
6. **Attracting set.** An attracting set is a closed invariant set $A \subset \mathcal{X}$ for which there is some neighborhood U of A such that:

$$\begin{cases} \forall \mathrm{x} \in U, \forall t \geq 0, \phi(t, \mathrm{x}) \in U \text{ and } \phi(t, \mathrm{x}) \underset{t \to \infty}{\longrightarrow} A & (flows) \\ \forall \mathrm{x} \in U, \forall t \geq 0, f^t(\mathrm{x}) \in U \text{ and } f^t(\mathrm{x}) \underset{t \to \infty}{\longrightarrow} A & (maps) \end{cases} \tag{3.8}$$

3.3 Stability

One of the basic properties of an invariant set is its stability. There are a number of different senses in which a fixed point (or periodic point, etc.) may be stable. Two of these are listed below in terms of maps; the definitions for flows are analogous.

Definition. A solution x_t is **Lyapunov stable** if a solution x'_t based nearby remains close to x_t for all time, i.e., if for *every* neighborhood V of x_0 there is a neighborhood $V_1 \subset V$ such that every solution $f^t(\mathrm{x}')$ with $\mathrm{x}' \in V_1$ is defined and lies in V for all $t > 0$. Thus for $\mathrm{x}_t \in \mathbb{R}^N$, for example, for any given $\varepsilon > 0$ there exists a $\delta > 0$ such that for any other solution x'_t satisfying $|\mathrm{x}'_0 - \mathrm{x}_0| < \delta$, then $|\mathrm{x}'_t - \mathrm{x}_t| < \varepsilon$ for all $t > 0$.

Definition. A solution x_t is **asymptotically stable** if it is Lyapunov stable and if there exists a constant $b > 0$ such that for any other solution x'_t satisfying $|\mathrm{x}'_0 - \mathrm{x}_0| < b$, then $\lim_{t \to \infty} |\mathrm{x}'_t - \mathrm{x}_t| = 0$.

A similar notion to stability appropriate to ensemble evolution is afforded by the definition of trapping set, as follows.

Definition. A region $U \subset \mathcal{X}$ is a **trapping set** (or trapping region) if

$$\begin{cases} \phi(t, U) \subset U & (flows) \\ f^t(U) \subset U & (maps) \end{cases} \tag{3.9}$$

for all $t > 0$.

3.4 Attractors

Informally, an "attractor" is a region of state space into which the system eventually falls, for some neighborhood of initial conditions about it. The attractors of a system describe its asymptotic behavior, and are accordingly given a central place in the qualitative study of dynamical systems. The precise formulation of this notion has been done in several ways, however. Ref. [31], for example, lists a number of different definitions before offering two new ones.

Most definitions of attractor contain two basic conditions: *stability* and *transitivity.* The former was the subject of the last section. The latter is basically a requirement that an attractor must be minimal, i.e., that the union of two separate attractors is not itself an attractor. This is expressed formally in the definition below.

Definition. An invariant set A is **topologically transitive** if, for any two open sets $U, V \subset A$ the following condition is satisfied:

$$\begin{cases} \exists t \in \mathbb{R} \text{ such that } f(t, U) \cap V \neq \emptyset & (flows) \\ \exists t \in \mathbb{Z} \text{ such that } f^t(U) \cap V \neq \emptyset & (maps) \end{cases} \tag{3.10}$$

Topological transitivity is equivalent to the statement that A possesses an orbit which is **dense** in the set, so that it passes close to every point in the set. This condition can be in general quite difficult to prove.

A brief and incomplete listing of some of the more recent definitions of attractor follows.

Definition (1). Conley, 1978.[14] A is an attractor if and only if:

1. A is closed and invariant.
2. There exists a neighborhood V of A such that A is the ω limit set of V.

Definition (2). Guckenheimer and Holmes, 1983.[44] A is an attractor if and only if:

1. There exists a dense orbit in A.
2. There exists a neighborhood V of A such that $L_\omega(V) \subset A$, and for all $t > 0$,

$$\begin{cases} \phi(t, V) \subset V & (flows) \\ f^t(V) \subset V & (maps) \end{cases} \tag{3.11}$$

Definition (3). Milnor, 1985.[77,78] A is an attractor if and only if:

1. A is closed.
2. For measure μ and "realm of attraction" $\rho(A) = \{\mathrm{x} : \mathrm{x} \in \mathcal{X}, L_\omega(\mathrm{x}) \subset A\}$, $\mu(\rho(A)) > 0$.
3. There is no closed set $A' \subset A$ such that $\mu(\rho(A) - \rho(A')) > 0$, where the minus sign denotes the set-theoretic difference.

Definition (4). Temam, 1988.[97] A is an attractor if and only if:

1. A is invariant.
2. There exists some open neighborhood V of A such that, for every $v \in V$, $d\big(f^t(v), A\big) \to 0$ as $t \to \infty$, where $d(\mathrm{x}, A)$ is the distance from point x to the nearest point in A.

Definition (5). Wiggins, 1990.[102] A is an attractor if and only if:

1. A is an attracting set (see section 3.2).
2. A is topologically transitive.

Definition (6). Schori, 1991.[87] A is an attractor if and only if:

1. A is closed.
2. There exists an open neighborhood V of A such that $f\big(\overline{V}\big) \subset V$ and $\bigcap_{t \geq 0} f^t(V) = A$, where $\overline{V}$ is the closure of V.

As the above list shows, attractors are typically defined as closed sets. They always satisfy some stability criterion, though sometimes only in a measure-theoretic sense, as in definition (3). Often, though not universally, they possess transitivity or some equivalent.

3.5 Basins and Separatrices

Once the attractors of a system have been identified, it is natural to examine the properties of the evolution onto the attractors.

The **basin** of an attractor is the set of all states that evolve onto it. This is expressed as follows:

$$\begin{cases} \mathcal{B}(A) = \bigcup_{t \leq 0} \phi(t, U) & (flows) \\ \mathcal{B}(A) = \bigcup_{t \leq 0} f^t(U) & (maps) \end{cases} \tag{3.12}$$

where, as above, U is some neighborhood of A.

The **separatrices** or basin boundaries are the complement of the basins. As the name implies, they are sets that *separate* basins of different attractors. For a system having a given set of attractors $\{A_i\}$ and basins $\{\mathcal{B}(A_i)\}$, the global separatrix $\mathcal{S}$ is given by $\mathcal{S} = \mathcal{X} - \bigcup\limits_i \mathcal{B}(A_i)$. For a state on a separatrix, an infinitesimal perturbation in one direction will move the state into the one basin, say $\mathcal{B}(A_0)$, while a perturbation in a different direction moves it into another basin $\mathcal{B}(A_1)$.

Investigations of Hayashi[49] and later by Grebogi, MacDonald *et al.*[42,74] into the structure of separatrices have indicated that they can be extremely complex. In some low dimensional systems possessing more than one attractor, the boundary between adjacent basins can itself be a fractal object. This means that it can be arbitrarily difficult to predict which attractor a given state on the separatrix will eventually fall into. This is one form of **sensitivity to initial conditions** (SIC); another form is discussed below. One mechanism for this is the existence of a *chaotic repeller*, which is an *unstable* invariant set on which the behavior is chaotic (see below).

The attractors, basins, and separatrices of a dynamical system are together called its **attractor-basin portrait**.

3.6 Deterministic Chaos

Many nonlinear systems exhibit the phenomenon of **deterministic chaos**. This refers to the properties of the system's asymptotic behavior—i.e., of the motion on its attractors. Chaos has been variously defined, but can at least roughly be characterized by the following features:

1. *Infinite number of periodic orbits.* An infinite number of periodic orbits of different periods are embedded in the attractor. This is now called "topological chaos."
2. *Bounded aperiodic motion.* The motion of an orbit on the attractor is both confined to a bounded region of $\mathcal{X}$ and is aperiodic.
3. *Instability under infinitesimal perturbations.* This is a second type of "sensitivity to initial conditions" (the first type has to do with basin boundaries, see above). Two trajectories $\mathrm{x}(t)$ and $\mathrm{x}'(t)$ that are arbitrarily close at $t = 0$ diverge exponentially, at least while they remain close.

Two quantitative statistics taken from the ergodic theory of dynamical systems help to characterize chaotic behavior. The first of these is the Kolmogorov entropy or **metric entropy rate**, which is informally the rate at which information about the system's location in state space is lost. The entropy rate is most conveniently defined in terms of symbol sequences obtained from a *generating partition*, as introduced in section 1.4. A generating partition is a discretization of the state space $\mathbb{R}^N$ into boxes or cells of linear size ϵ in such a way that there is a finite-to-one correspondence between orbits $\mathrm{x}(t)$ in state space and infinite sequences of cells $i_0, i_1, i_2, ...$ visited by the system. The idea of a generating partition is naturally motivated by the fact that any measuring instrument has finite precision, and so discretizes the state space into resolvable cells. In effect, a measuring instrument projects continuous orbits in the state space $\mathcal{X}$ onto sequences of cells in a discrete *measurement space* $\mathcal{M}$.

To define the metric entropy, we consider the evolution of the set of states in $\mathcal{X}$ that maps onto a given cell i_0 in $\mathcal{M}$. Let $P_{i_0,...,i_n}$ denote the fraction of these states that traverse the particular sequence of cells $i_0, ..., i_n$. Then the metric entropy h_μ is given by

$$h_\mu = \lim_{n\to\infty} \frac{-1}{n} \sum_{i_0,...,i_n} P_{i_0,...,i_n} \log P_{i_0,...,i_n} \tag{3.13}$$

where the logarithm is base 2. In this way, h_μ measures the average diversity of orbits starting from a single cell in $\mathcal{M}$. Another way to say this is that h_μ measures the average information lost about the system's state between measurements. If the system has $h_\mu > 0$, the initial measurement uncertainty is magnified during the evolution, and two orbits $\mathrm{x}(t)$ and $\mathrm{x}'(t)$ that start out in the same cell i_0—i.e., that are not resolved by the measuring instrument—may at some later time τ end up in different cells i_τ and i'_τ, and so be resolved by the instrument. Conversely, the predictive power of the measurement at $t = 0$ decays in time, since it cannot identify which of cells i_τ or i'_τ the system will later be in. Chaotic evolution is characterized by a positive entropy rate.

A related set of quantities are the **Lyapunov exponents**. These are defined by the response of the system to infinitesimal perturbations. Consider a flow in one dimension, with nearby trajectories $\mathrm{x}(t)$ and $\mathrm{x}'(t)$ such that $|\mathrm{x}(0) - \mathrm{x}'(0)| = \epsilon$. If the system is chaotic, these trajectories diverge exponentially fast, so that for t small,

$$\left|\mathrm{x}'(t) - \mathrm{x}(t)\right| \propto e^{\lambda(\mathrm{x})t} \tag{3.14}$$

The exponent $\lambda(\mathrm{x})$ is the Lyapunov characteristic exponent of the orbit $\mathrm{x}(t)$. Higher-dimensional systems have a set of Lyapunov exponents describing the effects of perturbations in different directions. Directions in which the perturbed orbit diverges from the original orbit are *unstable* directions, and have positive Lyapunov exponents. *Stable* directions, in which the perturbed orbit relaxes back down onto the original, have negative Lyapunov exponents. If the divergence or convergence of the perturbed orbit is slower than exponential in some direction, then that direction is associated with a Lyapunov exponent of 0. The Lyapunov exponents can be calculated from the eigenvalues of the integral of the Jacobian matrix defining the flow.[88] If the orbit $\mathrm{x}(t)$ is on an ergodic attractor, so that time averages over the orbit are equal to state-space averages over the attractor, then the Lyapunov exponents can be used to characterize the attractor as a whole. This is done by averaging over all initial conditions $\mathrm{x}(0)$ on the attractor. An attractor with at least one positive Lyapunov exponent is chaotic.

3.7 Spatially-Extended Systems

A spatially extended system occupies a region $\Omega \subset \mathbb{R}^d$ of physical space, and each spatial point $\mathrm{x} \in \Omega$ is considered the seat of a degree of freedom. A state of the system consists of the specification of the value of some quantity (e.g., velocity of fluid flow) at *each point* in Ω. If space is continuous, as is the case for PDEs, the state x is given by a function, and so the dimensions of state space form a continuum (e.g., $\mathcal{X}$ is a Hilbert space). For spatially-discrete systems such as CA or coupled maps lattices, the dimension of the state space scales with lattice size; an infinite lattice has a countable infinity of dimensions.

In some closed, dissipative systems, where the state space is continually contracting, the effective dimension of the asymptotic evolution can be finite. For example, Temam[97] analyzes a variety of infinite-dimensional dynamical systems, showing that each possesses a global attracting set A of finite dimension; unfortunately little is known about the attractor other than that it has finite dimension. Estimates of the actual dimension range from $\dim A \sim 10^9$ for wind-tunnel experiments up to $\dim A \sim 10^{20}$ for geophysical flows (weather). In either case, such a high value of $\dim A$ poses a serious obstacle to determining the geometric structure of the orbits.

Another equally serious problem with the state-space analysis of a spatially-extended system is that there is no explicit notion of *pattern*. Two states may both have the same pattern but may be arbitrarily distant from each other in the state space—for example, two sine waves in one spatial dimension with the same wavelength but different phase.

Both of these difficulties are evident in previous investigations of attractors in CA. Hurley[57–60] has proven a number of interesting results, taking as the CA state space the bi-infinite lattice, and as the definition of "attractor" those of Conley and Milnor above. Notably missing, however, is any notion whatever of the spatial pattern of states on the attractor. The "transients and cycles" view described in section 2.3 also suffers from this deficiency, in addition to being unwieldy for even moderately large lattices.

These difficulties were in fact one of the prime motivations behind the analysis developed in part II.

Temam[97] also lists two more "radically new" aspects present in spatially extended systems, which our analysis must ultimately address:

1. *Existence of chaos in space.* Turbulence can develop in some regions of the system, but not in others. This is another ramification of the fact that there are now *spatial* patterns, as well as *temporal* ones.
2. *Possible existence of singularities on some small sets.* For example, in fluid mechanics it is not known whether or not in three spatial dimensions the vorticity can become infinite in some parts of the region Ω occupied by the flow. Such singularities, if they are present, must considerably complicate the analysis, possibly even leading to a breakdown in the validity of the equation of motion.

4 Finite State Machines

This chapter outlines those parts of the theory of formal languages and automata that will be needed in later chapters. For a thorough presentation of this topic, see any text on the subject.[53] Much of what is said here is standard; however, a few ideas were formulated expressly for use in computational mechanics (such as, for example, statistical complexity and machine composition; see below).

The central notion is that of a **machine**, which in this context is an idealized object for sequentially reading, generating, or processing strings of symbols. There are two basic types of machines defined here, automata and transducers, which will be used to represent sets of strings and mappings between sets, respectively. Finite state machines will be the apparatus used in part II to express the notion of spatial structure or pattern in a set of strings—e.g., in the CA state space. It is important not to confuse the automata defined here, which will be used to represent ensembles of symbol strings, with cellular automata, which are the dynamical systems we are studying. While there is in fact a close relation between CA and finite state machines—which we will be exploring in part II—at this point it is helpful to keep the distinction between them as clear as possible.

4.1 Symbols, Strings and Formal Languages

A **symbol** σ is an abstract entity taking on values in some finite set Σ, the **alphabet**. The size of the alphabet will be denoted by $\|\Sigma\|$.

A **string** or **word** is a finite sequence of symbols. The length of a string ω, denoted by $|\omega|$, is the number of symbols in ω. The empty string, denoted by λ, is the string consisting of zero symbols. A **substring** or **subword** is a contiguous part of a string or word. The **concatenation** of two strings x and y is denoted xy. A **prefix** x of a string y is a substring consisting of any number of the leading symbols of y; that is, $y = xz$ for some z. A **suffix** x of a string y is a substring such that $y = zx$.

The ith symbol of a word ω is denoted by ω_i. By convention, we take $i \geq 0$, so that the first symbol in ω is ω_0. For a given word ω, the substring $\omega_i ... \omega_j$, $j \geq i$, is denoted by $\omega_{i,j}$. Note that in this notation $\omega_{i,i} = \omega_i$.

A simple statistic for comparing equal-length strings is the **Hamming distance** $|\cdot|_H$, which counts the number of sites as which two strings differ. For example, given the two strings $x = 0010$ and $y = 0110$, the Hamming distance between them is $|x - y|_H = 1$.

A **formal language** is a set of finite strings of symbols in some alphabet. The empty set $\emptyset$ and the set $\{\lambda\}$ are formal languages. The set of all length L strings of symbols in alphabet Σ is denoted by Σ^L. The set of all strings of any length, including the empty string, is denoted by Σ^*; i.e.,

$$\Sigma^* = \bigcup_{L \geq 0} \Sigma^L . \tag{4.1}$$

The **subword closure** of a formal language $\mathcal{L}$, denoted by $\mathrm{sub}(\mathcal{L})$, is the set of all substrings of strings in $\mathcal{L}$:

$$\mathrm{sub}(\mathcal{L}) = \{\omega_{i,j} : \omega \in \mathcal{L}, 0 \leq i \leq j \leq |\omega|\}. \tag{4.2}$$

Clearly, $\mathrm{sub}(\mathcal{L}) \supseteq \mathcal{L}$. A formal language for which $\mathrm{sub}(\mathcal{L}) = \mathcal{L}$ is **subword closed**.

4.2 Finite Automata and Regular Languages

Definition. An **automaton** is formally defined as a quintuple

$$\mathrm{M} = \{Q, \Sigma, \delta, Q_0, F\} \tag{4.3}$$

> in which Q is a set of **states**; Σ is an **alphabet** of tape symbols; $\delta : Q \times \Sigma \rightarrow Q$ is the **transition rule**, written $\delta(q, a) = p$, that takes the current state q and a symbol a read from a tape into a new state p; $Q_0 \subseteq Q$ is the set of **initial states** in one of which the machine starts; and F is the set of **final** or **accepting states**. If the set Q of states is finite, $\|Q\| < \infty$, then M is a **finite automaton**, abbreviated FA.

The transitions of a finite automaton are also called **edges**, in accordance with the graphical representation of the machine (see fig. 4.1 below). For a given transition $\delta(q, a) = p$, state q is the **source**, state p is the **destination**, and a is the **label**.

Finite automata can be used in two ways: *input* (or *reading*) mode and *output* (or *writing*) mode. In input mode, the symbols of an existing string are read in sequence from a tape and used to drive the machine from state to state according to the transition rule. In output mode, a transition from a given state is chosen at random, and the symbol labeling each selected transition is written to a tape (i.e., *emitted*) as that transition is taken. The sequence of emitted symbols is the string generated by the machine.

For a given state q and symbol a the transition $\delta(q, a)$ may or may not be defined. In the latter case, it is said to be a **disallowed** transition. On encountering a disallowed transition while reading a string, the machine immediately stops operation. The transition function may be extended to act on strings using the notation $\delta(q, s)$ where s is any string of symbols. An automaton is said to **recognize** a string if, upon attempting to read it from the tape, no disallowed transitions are seen. An automaton **accepts** a string if, after recognizing it, the automaton is in one of its accepting states. Strings that are not accepted are **rejected**. Two automata are **equivalent** if they accept identical sets of strings. Two automata are **disjoint** if they accept disjoint sets of strings.

A finite automaton is **deterministic** if (i) there is a unique start state q_0, so that $Q_0 = \{q_0\}$, and (ii) for all states $q \in Q$, each different transition from q to another state is labeled with a unique tape symbol. A finite automaton that is not deterministic is **nondeterministic**. These two automaton classes are abbreviated DFA and NFA, respectively.

The designation of "deterministic" is based on the fact that when a DFA is in input mode, the symbols read from the tape uniquely determine the sequence of transitions taken. This notion of determinism is different from the usual one in physics, in which a dynamical system is deterministic if its current state completely determines the sequence of future states. This is not generally the case for either NFAs or DFAs. In a DFA there may be more than one transition (each with a different symbol) from a given state. This means that when the DFA is in output mode the particular transition taken is chosen at random, i.e., nondeterministically. A deterministic finite automaton in output mode can be therefore be viewed as a "nondeterministic" data source in the sense of the term used in physics.

In addition to NFAs and DFAs, we will have occasion to use a third type of finite automaton, the **finite automaton with $\boldsymbol{\lambda}$-moves**.* This is an extension of FA to allow transitions on the null symbol λ. When one of these transitions is taken by the machine, no symbol is read or emitted. Finite automata with λ-moves are by definition nondeterministic. One use of FA with λ-moves will be to implement the common restriction (imposed in ref. [53], for example) that FA have a unique start state. This is done by adding a new start state q_0' to the machine, drawing λ-transitions from this state to each state in Q_0 of the original machine, and finally making q_0' the *only* start state. All states that can be reached by λ-transitions from q_0' are effectively "starting" states.

The language $\mathcal{L}(\mathrm{M})$ of a machine M is the set of all strings accepted by that machine. This definition is independent of whether or not M is finite. An automaton M is said to accept a formal language $\mathcal{L}$ if it accepts all strings $\mathcal{L}$ and rejects all strings not in $\mathcal{L}$.

The notion of an automaton accepting a language is based on the use of the machine in input mode. The analogous notion for machines in output mode is that of **generating** a language: the language generated by an automaton M is the set of all strings generated by taking all possible paths—i.e., sequences of transitions—in M. The language generated by a finite automaton is identical to the language accepted by that automaton.

Definition. A **regular language** is that $\mathcal{L}(\mathrm{M})$ accepted by some finite automaton M. **Regular language space**, or just "language space", is the collection $\mathcal{R}$ of all regular languages.

As is shown in section 4.6 below, for each regular language $\mathcal{L}$ there is a unique deterministic finite automaton $\mathrm{M}_{\mathrm{min}}$ with the minimal number $\|Q\|$ of states that accepts $\mathcal{L}$. In most cases the subscript will be dropped, so that a machine M will be understood to be the minimal machine $\mathrm{M}_{\mathrm{min}}$ unless it is explicitly stated otherwise.

---

*These have also been called finite automata with ϵ-moves, e.g. in ref. [53].

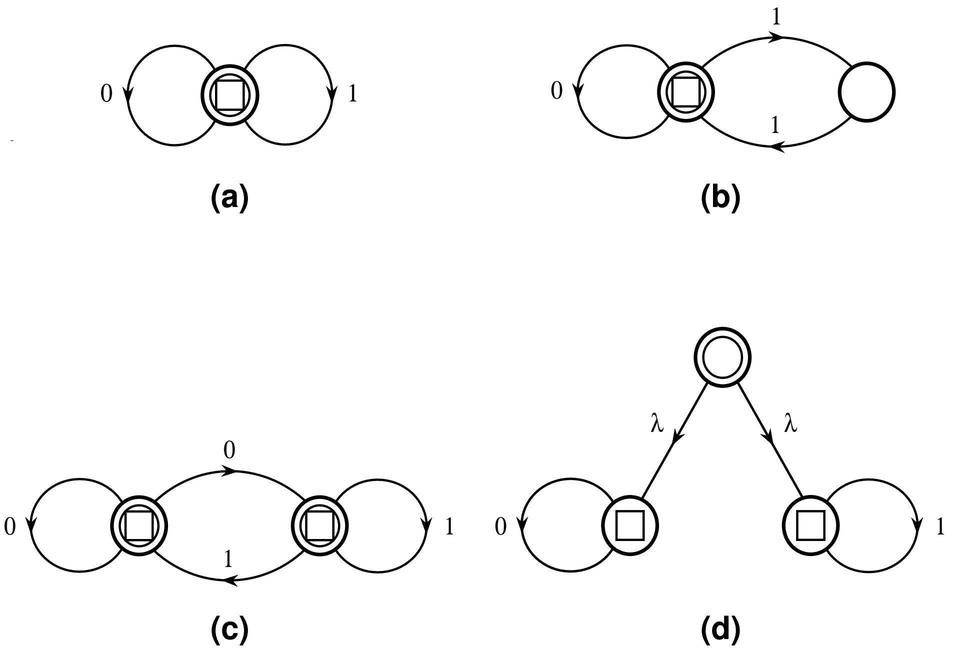


Fig. 4.1 Examples of finite automata. (a) A DFA with binary alphabet $\Sigma = \{0, 1\}$ and a single state, that accepts the set Σ^* of all strings of any length. (b) A DFA accepting the language $(0 + 11)^*$. (c) An NFA accepting the same language as in (a). (d) An NFA with λ-moves accepting the language $(0^* + 1^*)$. In this and all succeeding graphs of finite state machines, edges denote transitions, and nodes with inscribed circles and squares denote start and accept states, respectively.

Consistent with this, the minimal DFA accepting $\mathcal{L}$ will be denoted $\mathrm{M}(\mathcal{L})$. This allows the finite automaton and regular language representations to be used interchangeably.

We will refer to the number of states $\|Q\|$ of a machine M as the **size** of the machine, also denoted by $\|\mathrm{M}\|$. Note that this is not at all the same as the size or cardinality $\|\mathcal{L}(\mathrm{M})\|$ of the language accepted by M. It is quite often the case that $\|\mathrm{M}\|$ is a small finite number but $\|\mathcal{L}(\mathrm{M})\|$ is infinite.

Another way of representing a regular language, besides as a finite automaton, is with a **regular expression**. While we will not need a complete description of the rules for writing regular expressions, a few of them will turn out to be useful. For our purposes, a regular expression is an expression that can be constructed out of the following elements:

1. The alphabet symbols $\sigma \in \Sigma$.
2. (x), where x is a regular expression: denotes grouping.
3. xy: concatenation of x and y.
4. $x + y$ denotes *either* x or y.
5. x^*: denotes any number of concatenations of x (including zero).
6. x^+: one or more concatenations of x.

Thus, for example, the regular expressions for the languages accepted by the machines of fig. 4.1 are as follows: (a) and (c): $(0 + 1)^*$; (b) $(0 + 11)^*$; and (d) $(0^* + 1^*)$.

An important subclass of regular languages is the class of regular process languages.

Definition. A **regular process language** is a regular language $\mathcal{L}$ that is subword closed: $\mathcal{L} = \mathrm{sub}(\mathcal{L})$. A **regular process** is a finite state machine that generates a regular process language.

The terminology comes from the notion of the machine as a continually operating process in output mode, i.e., choosing transitions and emitting symbols. A word $\omega \in \mathcal{L}$ is then a finite subsequence of a bi-infinite stream of symbols emitted by the process. Since the subsequence can start and stop at any point, all subwords must be allowed. Finite automata for regular process languages have the property that all states are accepting: $F = Q$.[29]

For any language $\mathcal{L}$ that is *not* a regular process language, at least one state in $\mathrm{M}(\mathcal{L})$ is not accepting. The machine $\mathrm{M}(\mathrm{sub}(\mathcal{L}))$ for the subword closure of $\mathcal{L}$ can be constructed from $\mathrm{M}(\mathcal{L})$ by making all states in $\mathrm{M}(\mathcal{L})$ accept states. The new machine may or may not be minimal; if not, then when it is minimized, the number of states will necessarily decrease. Therefore we have $\|\mathrm{M}(\mathrm{sub}(\mathcal{L}))\| \leq \|\mathrm{M}(\mathcal{L})\|$, where $\|\mathrm{M}\|$ denotes the number of states in the *minimal* machine M. Note that the inequality in machine sizes is in the opposite sense as the inclusion relation between the languages, $\mathrm{sub}\mathcal{L} \supseteq \mathcal{L}$.

Definition. The **process graph** of a process language $\mathcal{L}$ is the graph formed by allowing the associated machine $\mathrm{M}(\mathcal{L})$ to start in any state, and minimizing the result.

The process graph is the "state transition graph" of [104]. All states are both start and accept states. The process graph emphasizes certain asymptotic properties of the language by eliminating transient states devoted to "phase-locking" of individual strings to the machine's recurrent states.[20]

4.3 Finite-State Transducers

As we have seen, automata can be used either as input devices for classifying existing strings or as output devices that generate strings belonging to a given language. These two modes of operation are easily combined by modifying the machine so that on each transition a symbol is emitted as well as read. The resulting "transducer" machines, also called Mealy machines,[53] are functions from strings to strings. In the information theory literature the process versions are referred to as channels.[7] In ergodic theory, they are endomorphisms of subshifts.[70] Transducers give a very compact notation for functions that map complicated sets, e.g. Cantor sets, to complicated sets. Of equal importance, they are constructive in the sense that they indicate literally how the function is to be implemented and so computed.

Definition. A **transducer**

$$\mathrm{T} = \{Q, \Sigma_{in}, \delta, Q_0, F, \Sigma_{out}, \beta\} \tag{4.4}$$

is an automaton where Q, δ, Q_0, F are defined as for finite automata and where Σ_{in} is the FA input alphabet Σ. Σ_{out} is an output alphabet. The observation function β maps $Q \times \Sigma_{in}$ to Σ_{out}. A transducer operates as a finite automaton with the extra feature of writing a symbol in Σ_{out} to an output tape upon reading a symbol in Σ_{in} from the input tape. As in the case of finite automata, a transducer with a finite number of states is called a **finite state transducer**, abbreviated FST.

The effect of running a transducer over a space-time pattern, say, is to recode the data by replacing original symbols with those emitted by the transducer. Formally, the transducer induces a mapping f_{T} from strings over Σ_{in} to strings over Σ_{out}.

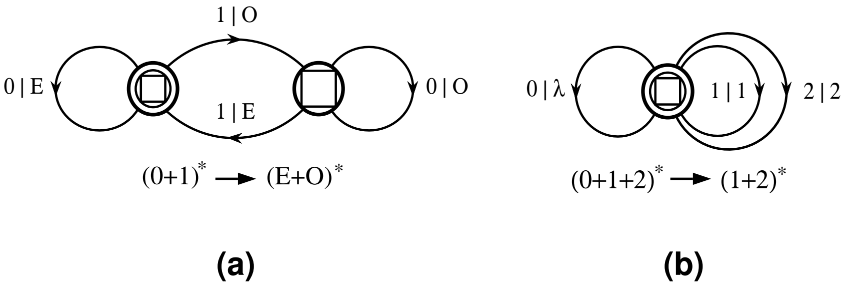


Fig. 4.2 Some simple FSTs. (a) A deterministic FST for converting an arbitrary string of symbols to a sequence of "even" (E) and "odd" (O) symbols giving the parity of the input string read so far. (b) An FST with λ-outputs that filters out all the 0s from the input string. Edges are labeled $\sigma_{in}|\sigma_{out}$ denoting input and output symbols, respectively.

Like finite automata, FSTs can be deterministic or nondeterministic. In a deterministic FST there is a unique start state q_0, and each state has a unique transition for any input symbol σ_{in}. There are two types of nondeterministic FSTs. The first type, an **irreducibly** nondeterministic FST, has one or more states in which at least two transitions have the same input symbol σ_{in} but different output symbols σ_{out}. This means that the induced mapping f_{T} is not a function: a single input string is mapped onto more than one output string. The second type of nondeterministic FST is **reducibly** nondeterministic, and has at least one state in which more than one transition is labeled with the same input/output pair $(\sigma_{in}, \sigma_{out})$, but is not irreducibly nondeterministic. A machine of this type can be converted to a deterministic FST by the same algorithm used to convert an NFA into a DFA, with the symbol pairs $(\sigma_{in}, \sigma_{out})$ treated as single labels.

A simple variation of FSTs is the **finite state transducer with $\boldsymbol{\lambda}$-outputs**. In close analogy to FA with λ-moves, this is defined as an FST in which the alphabet of legal outputs includes the null symbol λ, indicating that no output symbol is generated. One use of the null-output transitions is to allow the transducer to "synchronize" with the input pattern before starting to emit symbols. In chapter 6, we will see that cellular automaton evolution can be implemented as the operation of an FST with λ-outputs.

Another type of transducer, called a **Moore machine**, has output symbols attached to states rather than edges, so that the machine emits a symbol on entering a given state. In ref. [53], it is shown that for every Moore machine there is an equivalent finite state transducer (Mealy machine), and vice-versa.

4.4 Other Machines

There are several other variants of the finite automata defined above. Some of the more relevant ones are listed here.

1. **Sofic systems** are much studied in the extensive field of symbolic dynamics,[101] which deals with certain properties of bi-infinite strings. The fundamental object in symbolic dynamics is the *subshift*, which is a set of bi-infinite strings that is invariant under the *shift operator* that shifts the spatial position of a string one cell to the left (or equivalently, shifts the spatial origin one cell to the right). Roughly speaking, a sofic system is a subshift that is expressible as the set of bi-infinite strings generated by some regular process.
2. A **subshift of finite type** (SSFT) is a sofic system in which the restrictions imposed by the machine structure can be summarized by a finite set of minimal-length subwords that are not allowed in any word in the language. This is the set of *irreducible forbidden words*. Any string that contains a forbidden word is itself forbidden.
3. **Strictly sofic systems** (SSS) are the complement of SSFTs in the class of sofic systems. Another characterization of strictly sofic systems is that the set of irreducible forbidden words is infinite.
4. **Stochastic DFAs** are a variant of DFAs in which each transition is labeled with a probability as well as a symbol. The sum of the weights of all edges leaving a given state is necessarily normalized to 1. The transition probabilities also induce a probability distribution on the states. Stochastic DFAs (and stochastic NFAs, see below) describe a probability distribution over the set of strings accepted by the machine.
5. **Stochastic NFAs**, as the name implies, are NFAs with probabilities on the transitions. Unlike ordinary NFAs, it is not necessarily true that a stochastic NFA has an equivalent stochastic DFA. Examples of stochastic NFAs exist for which constructing the equivalent stochastic DFA yields an infinite number of states or even a nondenumerable infinity of states.[27]
6. **Hidden Markov models** and **functions of Markov chains** are similar to stochastic NFAs. The name comes from the Markov chains of probability theory,[38] which are stochastic processes in which states are defined such that the future behavior of the process is conditioned only on the current state, independent of the past behavior that drove the machine into that state. Unlike stochastic FAs,

the transitions between states are not labeled. A function of a Markov chain has a stochastic *output function* $\beta(v)$ defined at each state v such that on entering the state a given symbol is emitted with a given probability. A hidden Markov model is essentially the same thing.

4.5 Language and Machine Quantifiers

Of the many quantifiers of finite state machines and regular languages, a few which we will find useful in later chapters are discussed here.

Relative Size

One simple comparison of languages is the relative size. For finite languages ($\|\mathcal{L}\| < \infty$), the relative size of language $\mathcal{L}_1$ with respect to another language $\mathcal{L}_2$ is just $\|\mathcal{L}_1\| / \|\mathcal{L}_2\|$. For infinite languages, the relative size is defined as follows.

Definition. Let $N_{\mathcal{L}}(L)$ be the number of strings $\omega \in \mathcal{L}$ of length $|\omega| = L$. The **relative size** of a language $\mathcal{L}_1$ with respect to another language $\mathcal{L}_2$ such that $\mathcal{L}_1 \subseteq \mathcal{L}_2$, is defined as

$$S(\mathcal{L}_1|\mathcal{L}_2) \equiv \lim_{L\to\infty} \frac{N_{\mathcal{L}_1}(L)}{N_{\mathcal{L}_2}(L)} \tag{4.5}$$

$S(\mathcal{L}_1|\mathcal{L}_2)$ is the fraction of words in $\mathcal{L}_2$ that are also in $\mathcal{L}_1$, in the limit large word length.

Matrix Representation

An alternative to the graphical representation of machines, as shown in figs. 4.1 and 4.2, is the specification of the machine architecture by a set of transition matrices. These are defined as follows.

The **transition matrix**[†] $T_\sigma(\mathrm{M})$ of a finite automaton M with $\|Q\| = V$ states is a square matrix of size $V \times V$ in which the elements t_{ij} are given by

$$t_{ij} = \begin{cases} 1 & \text{if } \delta(q_i, \sigma) = q_j \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

In other words, $T_\sigma(\mathrm{M})$ has a 1 in each element t_{ij} for which the machine has a corresponding transition from state q_i to q_j on symbol σ, and a 0 everywhere else.

The **connection matrix** $T(\mathrm{M})$ is the sum of the transition matrices:

$$T(\mathrm{M}) = \sum_{\sigma\in\Sigma} T_\sigma(\mathrm{M}) \tag{4.7}$$

As such, it disregards the symbols labeling transitions, and an given element only records the number of transitions between two states.

[†] Note the italic symbol T, to differentiate the transition matrix from the roman symbol T used for transducers. The use of T for transition matrix is confined to this subsection and the one following it.

For stochastic machines, there is a corresponding stochastic transition matrix for which the elements t_{ij} give the probability that the machine will make the given transition. This is the probability $p_\sigma(q_j|q_i)$ of the transition from q_i to q_j on symbol σ, conditioned on the probability $p(q_i)$ that the machine is in the state q_i:

$$t_{ij} = p_\sigma(q_j|q_i)p(q_i) \tag{4.8}$$

Again, the stochastic connection matrix is the sum of the stochastic transition matrices for each symbol in Σ.

Machine Entropy and Complexity

The **topological entropy** per symbol $h_0(\mathcal{L})$ of a regular language $\mathcal{L}$ is the exponential growth rate in number of distinct strings in $\mathcal{L}$:

$$h_0(\mathcal{L}) = \lim_{L\to\infty} \frac{\log N_{\mathcal{L}}(L)}{L} \tag{4.9}$$

where $N_{\mathcal{L}}(L)$ is the number of distinct words of length L in $\mathcal{L}$. In terms of the connection matrix $T(\mathrm{M}(\mathcal{L}))$, $h_0(\mathcal{L})$ is given by

$$h_0(\mathcal{L}) = \log \lambda_{\max} \tag{4.10}$$

where $\lambda_{\max}$ is the largest eigenvalue of $T(\mathrm{M}(\mathcal{L}))$.

The **metric entropy**, which takes into account the probability of occurrence of words, is defined as

$$h_1(\mathcal{L}) = \lim_{L\to\infty} -\frac{1}{L} \sum_{\substack{\omega\in\mathcal{L}\\ |\omega|=L}} p(\omega)\log p(\omega) \tag{4.11}$$

where $p(\omega)$ is the probability of occurrence of the word ω and the sum is taken over all distinct words of length L. For a stochastic finite state machine, the metric entropy is given by

$$h_1(\mathcal{L}) = -\sum_{q_i\in Q} p(q_i) \sum_{\substack{q_j\in Q\\ \sigma\in\Sigma}} p_\sigma(q_j|q_i)\log p_\sigma(q_j|q_i) \tag{4.12}$$

where, as above, Q is the set of states in the minimal machine $\mathrm{M}(\mathcal{L})$ (or the process graph, if $\mathcal{L}$ is a process language), $p(q_i)$ is the asymptotic probability that the machine is in state q_i and $p_\sigma(q_j|q_i)$ is the transition probability from state q_i to state q_j on symbol σ.

Entropy is a measure of the *diversity* of words in a language $\mathcal{L}$. It is the direct application of the Shannon entropy[89] to formal languages. For a given alphabet Σ, the range of both entropies is restricted to the closed interval $0 \le h_0, h_1 \le \log k$, where $k = \|\Sigma\|$. In general, the metric and topological entropies are related by $h_1 \le h_0$.

If all transitions from any given state in the stochastic machine $\mathrm{M}(\mathcal{L})$ have equal probability, then $h_1 = h_0$.

The (finitary) **topological complexity** and **statistical complexity**[28,29] of a regular language, $C_0(\mathcal{L})$ and $C_1(\mathcal{L})$ respectively, are related to the amount of "memory" in the machine that generates that language, and are given by

$$\begin{aligned} C_0(\mathcal{L}) &= \log \|Q\| \\ C_1(\mathcal{L}) &= -\sum_{q_i \in Q} p(q_i) \log p(q_i) \end{aligned} \tag{4.13}$$

The statistical (or metric) and topological complexities are related by $C_1 \leq C_0$, with equality if and only if all state probabilities are equal. Unlike the entropy, the complexity can be arbitrarily large. Since C_0 is the logarithm of an integer, it takes on values in a discrete spectrum. C_1 can attain any positive real value.

4.6 Machine Operations

This section details a number of specific operations involving FAs and FSTs will be needed in the sequel.

Conversion to DFA

In ref. [53], the following two theorems are proved:

Theorem. Let $\mathcal{L}$ be a language accepted by an NFA $\mathrm{M}(\mathcal{L})$. Then there exists a DFA M' that accepts $\mathcal{L}$.

Theorem. If $\mathcal{L}$ is accepted by an NFA with λ-moves, then $\mathcal{L}$ is accepted by an NFA without λ-moves.

Together these imply that the class of DFAs, the class of NFAs, and the class of NFAs with λ-moves are all equivalent. In practice this means that we may convert any given finite automaton into a DFA before measuring its properties. The algorithm for converting an NFA to a DFA is given here; the conversion of an NFA with λ-moves to an ordinary NFA is similar and can be found in ref. [53].

The essence of the algorithm for converting an NFA $\mathrm{M} = \{Q, \Sigma, \delta, Q_0, F\}$ into an equivalent DFA $\mathrm{M}' = \{Q', \Sigma, \delta', \{q_0'\}, F'\}$ is to construct M' such that its states represent *sets* of the states of M. That is, $Q' \subseteq 2^Q$ where 2^Q is the collection of subsets of Q. In effect, the state of M' reached upon reading some word ω will be constructed to correspond to the set of states of M that can be reached by any sequence of transitions in M labeled with the symbols of ω.

An element of Q' will be denoted $q' = [q_i, q_j, ..., q_z]$ where $q_i, q_j, ..., q_z$ are all in Q. The unique start state of M' is just the set of start states of M, that is, $q_0' = [Q_0]$. The transition function $\delta'(q', a)$ is defined by

$$\delta'([q_i, q_j, ..., q_z], a) = [p_i, p_j, ...p_z] \tag{4.14}$$

if and only if

$$\delta(\{q_i, q_j, ..., q_z\}, a) = \{p_i, p_j, ...p_z\}. \tag{4.15}$$

That is, δ' applied to an element $q' = [q_i, q_j, ..., q_z]$ of Q' is computed by applying δ to each element of Q in the set $\{q_i, q_j, ..., q_z\}$ that q' represents. The resulting set $\{p_i, p_j, ..., p_z\}$ of elements of Q is used to define a new element in Q', namely $p' = [p_i, p_j, ..., p_z]$. The construction of Q' thus proceeds simultaneously with the process of defining the transition function δ'. For example, from the start state $q'_0 = [Q_0]$, a new "child" state $p' = \delta'(q'_0, a)$ is constructed for each symbol a. Each of these states in turn is used to construct *its* child state, and so on until all states have been constructed and all transitions have been defined. The process is guaranteed to terminate because 2^Q is a finite set. Finally, we note that a state $q' = [q_i, q_j, ..., q_z]$ is in F' if and only if any of the $\{q_i, q_j, ..., q_z\}$ is in F.

As the above algorithm makes clear, the process of converting an NFA to a DFA can have a dramatic effect on the size of the machine. In the worst case, an exponential increase in the number of states is possible.

Minimization

For any given regular language $\mathcal{L}$, there can be many DFAs of many different sizes that accept $\mathcal{L}$. Of these, there is at least one that is **minimal**, i.e., that has the fewest states. The Myhill-Nerode theorem[53] guarantees that any minimal DFA accepting a given language is unique up to a trivial relabeling of states. The proof of this theorem involves the construction of the minimal machine. One such construction algorithm, taken from ref. [53], is summarized here; a more efficient algorithm can be found in ref. [52]. The minimization algorithm given here for DFAs can easily be modified to apply to transducers are well. Minimizability is a curious, notable feature of finite state machines; for other types of machine (e.g. stack-based automata or Turing machines), analogous minimization algorithms to not exist.

The process of minimizing a DFA consists of identifying pairs of states that are *equivalent* to each other, in the following sense: A state q_1 is equivalent to a state q_2 if and only if for each input string s, $p_1 = \delta(q_1, s)$ is an accepting state if and only if $p_2 = \delta(q_2, s)$ is accepting. This has two parts: (i) $\delta(q_1, s)$ and $\delta(q_2, s)$ must both be defined, so that s is recognized, and (ii) p_1 and p_2 must be both accepting or both nonaccepting. Once two such equivalent states q_1 and q_2 are found, they are *merged* by moving all of q_1's input transitions to q_2 (i.e., for all transitions with q_1 as destination, change the destination to q_2) and deleting q_1, or vice-versa. In merging q_1 and q_2, there is no need to replicate q_1's output transitions on q_2; the output transitions of q_1 and q_2 must necessarily be equivalent as a precondition for equivalence of q_1 and q_2.

The identification of equivalent states actually proceeds by identifying all pairs of states that are *distinct*, i.e., not equivalent. Finding distinct pairs is done iteratively. In each iteration, all pairs are tested for distinctness. A pair (q_1, q_2) is labeled as distinct if one of the following properties holds:

i. q_1 is accepting but q_2 is not, or vice-versa.
ii. For some symbol a, q_1 has a transition on a but q_2 does not, or vice-versa.
iii. There exists a symbol a such that $p_1 = \delta(q_1, a)$ and $p_2 = \delta(q_2, a)$ are distinct.

As case (iii) indicates, each new iteration will identify new distinct pairs based on pairs that were identified in previous iterations. This process continues until no new distinct pairs are identified. Once every distinct pair has been found, all remaining pairs are necessarily equivalent.

Output language

Definition. Let T be a transducer. The **output machine**, which we denote $[\mathrm{T}]_{\mathrm{out}}$, is the minimal DFA accepting the set of all strings that can be emitted from T, on all possible inputs. $[\mathrm{T}]_{\mathrm{out}}$ can be constructed by the following series of steps:

   i. The input symbols on all transitions of T are disregarded, leaving only the output symbols labeling transitions.
   ii. The resulting machine is typically an NFA with λ-moves. This is converted to an NFA without λ-moves, and then to a DFA, using the standard algorithms.
   iii. The DFA is then minimized to obtain the unique machine with the minimal number $\|Q\|$ of states.

This construction is called **minimization with respect to the output**.

Machine Composition

Given a machine $\mathrm{M}(\mathcal{L})$ and a transducer T, we can define the composition operator $\mathrm{T} \circ \mathrm{M}(\mathcal{L})$ such that the resulting machine, itself a transducer, accepts as input exactly the set of strings in $\mathcal{L}$, and for each such input string $\omega \in \mathcal{L}$, it emits the encoded string $f_{\mathrm{T}}(\omega)$. In effect, $\mathrm{M}(\mathcal{L})$ and T are connected in series, with $\mathrm{M}(\mathcal{L})$ emitting strings in $\mathcal{L}$ which are then used as inputs to T. This corresponds to restricting the domain of f_{T} to the language $\mathcal{L}$. The operator is formally expressed in the following.

Definition. Let M and T be a finite automaton and a transducer, respectively:

$$\begin{aligned} \mathrm{M} &= \left\{ Q^{\mathrm{M}}, \Sigma^{\mathrm{M}}, \delta^{\mathrm{M}}, Q_0^{\mathrm{M}}, F^{\mathrm{M}} \right\} \\ \mathrm{T} &= \left\{ Q^{\mathrm{T}}, \Sigma_{in}^{\mathrm{T}}, \delta^{\mathrm{T}}, Q_0^{\mathrm{T}}, F^{\mathrm{T}}, \Sigma_{out}^{\mathrm{T}}, \beta^{\mathrm{T}} \right\} \text{ with } \Sigma_{in}^{\mathrm{T}} = \Sigma^{\mathrm{M}} \end{aligned} \tag{4.16}$$

Then the **composition** of T and M, denoted $\mathrm{T}' = \mathrm{T} \circ \mathrm{M}$ is a transducer

$$\mathrm{T}' = \left\{ Q', \Sigma'_{in}, \delta', Q'_0, F', \Sigma'_{out}, \beta' \right\} \tag{4.17}$$

such that: Q' are ordered pairs of the states in Q^{T} and Q^{M}: $Q' = Q^{\mathrm{T}} \times Q^{\mathrm{M}}$; the input alphabet is $\Sigma'_{in} = \Sigma_{in}^{\mathrm{T}} = \Sigma^{\mathrm{M}}$; the set of start states is $Q'_0 = \{(q_0^{\mathrm{T}}, q_0^{\mathrm{M}}) : q_0^{\mathrm{T}} \in Q_0^{\mathrm{T}}, q_0^{\mathrm{M}} \in Q_0^{\mathrm{M}}\}$; the accepting states are $F' = \{(q^{\mathrm{T}}, q^{\mathrm{M}}) : q^{\mathrm{T}} \in F^{\mathrm{T}}, q^{\mathrm{M}} \in F^{\mathrm{M}}\}$; the output alphabet is $\Sigma'_{out} = \Sigma^{\mathrm{T}}_{out}$; the transition rule $\delta' : Q' \times \Sigma'_{in} \to Q'$ is given by

$$\delta'(q', \sigma_{in}) = \left(\delta^{\mathrm{T}}\left(q^{\mathrm{T}}, \sigma_{in}\right), \delta^{\mathrm{M}}\left(q^{\mathrm{M}}, \sigma_{in}\right)\right) \text{ , where } q' = \left(q^{\mathrm{T}}, q^{\mathrm{M}}\right); \tag{4.18}$$

and the observation function $\beta' : Q' \times \Sigma'_{in} \to \Sigma'_{out}$ is

$$\beta'(q', \sigma_{in}) = \beta^{\mathrm{T}}\left(q^{\mathrm{T}}, \sigma_{in}\right), \text{ again with } q' = \left(q^{\mathrm{T}}, q^{\mathrm{M}}\right). \tag{4.19}$$

Machine Metric

A transducer can be used to define a **machine metric** that measures the distance of a string from a given language. The action of this transducer is to encode a given pattern into a string of 1s that represents in unary form the number of disallowed transitions observed in the pattern. We will represent the fiducial language by its corresponding automaton M. Then the metric with respect to this language is defined as follows.

Let $\mathrm{T_M}$ be the transducer formed from automaton M by augmenting it to output the null symbol λ on allowed transitions and adding new transitions that emit a 1 on previously disallowed transitions. The new transitions are "reset" transitions, connected to the state representing the effect of having read the disallowed symbol when in the start state.

Definition. Let M be a machine, $\mathrm{T_M}$ be the transducer described above, and ω be any string, not necessarily in $\mathcal{L}(\mathrm{M})$. The **machine metric** is

$$\|\omega\|_{\mathrm{M}} = \begin{cases} |f_{\mathrm{T_M}}(\omega)| & |\omega| < \infty \\ \lim\limits_{n \to \infty} \frac{|f_{\mathrm{T_M}}(\omega_{0,n})|}{|\omega_{0,n}|} & \text{otherwise} \end{cases} \tag{4.20}$$

where $f_{\mathrm{T_M}}(\omega)$ is the string emitted by $\mathrm{T_M}$ upon reading the input ω. In the infinite case, $\omega_{0,n}$ denotes the string consisting of the first n symbols of ω. The finite and infinite cases are related by dividing the former by $|\omega|$.

The machine metric is an alternative to the **Hamming metric**, which measures the distance between two strings by their Hamming distance. Recall that this latter quantity is defined as the number of sites in which two strings (assumed of equal length) differ. One significant flaw in the Hamming metric is that strings which are shifted in spatial origin, but are otherwise identical, can have arbitrarily large Hamming distance. The Hamming distance completely fails to take into account any structure or pattern that may be present in the two strings. The machine metric, on the other hand, is expressly designed to compare two strings by their relation to a given pattern.

5 Machine Reconstruction

One of the most important tools in computational mechanics is *machine reconstruction*. This is the inductive construction, from a sample of a dynamical system's behavior, of a finite state machine representing the computational structure of the system. This chapter reviews the machine reconstruction method for finite automata developed by Crutchfield and Young[19,20,28].

The procedure outlined here is *topological*, i.e., it disregards the statistical properties of the data. Stochastic reconstruction, in which the data's probabilistic structure is taken into account, is a straightforward extension. Reconstruction of transducers is another straightforward extension. Transducer reconstruction can be used, for example, to find the effective dynamic of an arbitrary spatially extended system, either a discrete system such as CA or a continuous system measured with finite precision.

The context of machine reconstruction presupposes a dynamical system evolving according to its equation of motion. From this system, a data stream is taken by some sort of measuring instrument. A *measuring instrument* is a device that maps states of the dynamical system onto symbols in some finite alphabet $\mathcal{A}$ such as, for example, the binary alphabet $\mathcal{A} = \{0, 1\}$. In continuum systems, this necessarily entails partitioning the state space into discrete cells according to the precision of the measurements; this fact is often called *measurement distortion*. Denoting the linear size of such a cell by ϵ, the machines reconstructed from the data in this way are called **ϵ-machines**. For discrete systems such as cellular automata, this type of measurement distortion is not a factor, since the value of a cell can be determined exactly.

The reconstruction is accomplished in three steps: (i) gathering the *data*; (ii) building a *tree* from the data; and (iii) building a *machine* from the tree. These can be regarded as the first levels in a hierarchy of increasingly sophisticated models of the system.[20]

5.1 The Reconstruction Procedure

Step 1: Gathering the data

The first step in machine reconstruction is to gather the data by means of one or more measuring instruments. There are a number of ways to do this for a spatially-extended system, each of which provides a different type of information about the system. For example, a single instrument with fixed spatial position will output a time sequence of measurements of the system referenced to that position; this is a temporal data stream. Another scheme uses a row of instruments, each of which measures the system at the same time, but at a different location: a spatial data stream. Oblique data streams, for which both spatial position and measurement time change along the sequence, are obvious possibilities as well; these would be used to trace the evolution of a travelling disturbance, for example.

A given symbol sequence of length N is converted to a list of subwords of length D, the **tree depth**, obviously with $D \leq N$. The simplest way to make this conversion is to parse the data stream with a sliding window of width D, as shown in fig. 5.1. The sliding window disregards the positions of the subwords in the data stream, and so implicitly assumes translation invariance in the data. Two alternatives are to take a number of sequences for a number of different initial conditions or a number of different time steps, each of which assumes a different type of invariance.

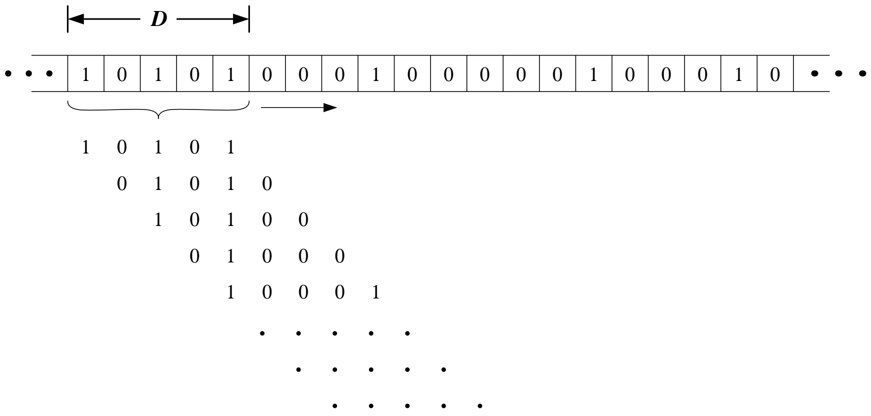


Fig. 5.1 Gathering the data: a sample data stream and the first few subwords obtained from it via a sliding window of width $D = 5$.

Step 2: Building the tree

The subwords are then used to construct tree of depth D (hence the name tree depth), as shown in fig. 5.2. The branches of the tree are labeled with measurement symbols. Each subword is put down on the tree such that the top level of branches corresponds to the first symbol in the word, the second level corresponds to the second symbol, and so forth. A path of successive branches in the tree, starting from the root node and moving downward, then corresponds to a subword. At each node, an integer count measures the number of subwords that have been laid down over that node. For topological reconstruction, the count is ignored. Stochastic reconstruction uses the count to measure the relative probabilities of the different subwords.

The tree depth is an approximation parameter, in that it imposes a finite horizon over which structural features of the data can be detected. For a given amount of data (i.e., fixed and finite N), the tree depth is made as large as possible to capture the greatest amount of structure in the data, without making it so large that statistical fluctuations become significant. The trade-off is that only very short-range structure will be detected if D is small, whereas for D large the number of subwords obtained from the data will be insufficient to overcome empirical fluctuations. If a sequence of length N is parsed with a sliding block as above, $N - D$ subwords of length D are obtained, which are then laid down on the tree. If the data source is a fair coin emitting 0 for "heads" and 1 for "tails", the subwords are uniformly distributed over

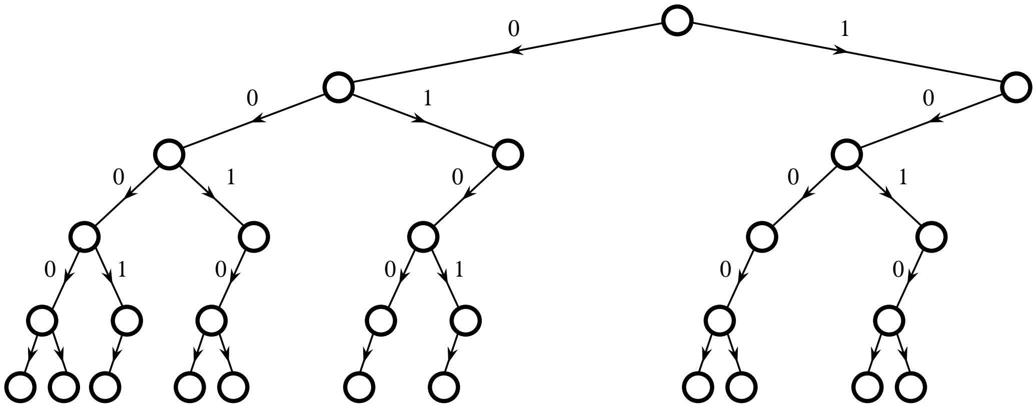


Fig. 5.2 Building the tree: the tree of depth $D = 5$ constructed from the subwords in the data stream of fig. 5.1.

all of the 2^D distinct possibilities. The tree would then be a fully branched binary tree having 2^D nodes (or "leaves") on the bottom level. Denoting the count of the nth leaf by c_n, we would expect to have

$$c_n \approx \frac{N - D}{2^D} \tag{5.1}$$

for all n. To reduce statistical fluctuations to the order of, say, $\sqrt{c_n}/c_n = \delta$ for some fixed δ we would therefore want to choose D such that

$$\sqrt{\frac{2^D}{N - D}} = \delta. \tag{5.2}$$

For $N = 1 \times 10^6$ and $\delta = 0.01$ this formula gives $D = 6$. For processes with lower topological entropy rate than the fair coin process, the number of distinct leaves in the tree is lower, and so the count of each leaf is correspondingly higher. This reduces the fluctuations and makes it possible to use a deeper tree. If the entropy rate is h, the above estimate of the optimal tree depth is modified slightly to read

$$\sqrt{\frac{2^{Dh}}{N - D}} = \delta. \tag{5.3}$$

Step 3: Building the machine

The machine is constructed by identifying sets of **future equivalent** nodes in the tree. Two nodes are future equivalent if and only if the branching structure of the subtrees rooted at the nodes is identical. A **subtree** is a part of a tree branching beneath a given tree node, the subtree's root. The term "future equivalence" comes from the fact that the branching structure beneath a node is a graphical representation of the set of possible symbol sequences that can follow a partial subword terminating on that node.

A **morph** is a possible shape—i.e., a branching structure—of a subtree. Each morph defines an equivalence class of subtrees having that morph's shape. The morphs

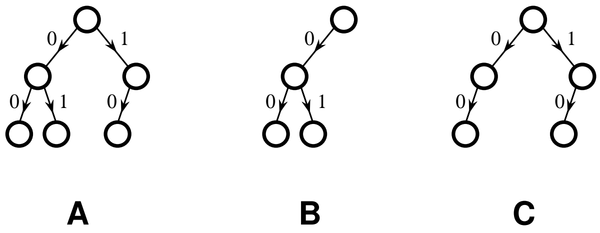


Fig. 5.3 The three distinct 2–morphs present in the tree of fig. 5.2.

occurring in a given tree, or in other words the classes of future equivalent tree nodes, are the *states* of the reconstructed machine.

Because the tree has finite depth D, the morphs also must be finite. A morph of depth L is called an **L-morph**, and L is called the **morph depth**. The morph depth is an approximation parameter imposed on the reconstruction by the finite tree depth D, which was in turn imposed by the limitations of finite data, as noted above.

In a given tree, there may be some L-morphs that do not occur. This is the effect of *restrictions* in the data stream—i.e., some of the k^D ($k = \|\mathcal{A}\|$) possible subwords of length D were not found. As we will see, it is these restrictions that determine the structure of the reconstructed machine: they comprise the (topological) regularity detected in the data stream.

In choosing the optimal value of L, two competing considerations must be balanced. The first of these is that the number of morphs should be as great as possible—i.e., morph depth should be large. The second is that the number of occurrences of each morph should be maximized—which means that the morph depth should be *small*. A good compromise value is obtained by the choosing the morph depth of

$$L_0 = \left\lfloor \frac{D-1}{2} \right\rfloor \tag{5.4}$$

where $\lfloor x \rfloor$ is the integer part of x.[19] In the tree of fig. 5.2, with $D = 5$, the optimal morph depth is therefore $L = 2$. Fig. 5.3 shows the three distinct 2–morphs present in the tree of fig. 5.2.

Each L-morph is assigned a unique index. The depth L subtree rooted at each node is assigned the index of the morph with matching shape. In fig. 5.3, the morphs are labeled with the letters A, B, and C. Fig. 5.4 shows the tree with each node labeled by the index of the matching morph. Note that the bottom two levels of the tree cannot be labeled, since nodes in those levels do not have complete depth L subtree below them.

As we have stated, the morphs define the states of the reconstructed machine. For each morph occurring anywhere in the tree, therefore, a state is added to the machine; the state is given the same index as the morph. Transitions between states are then read directly off the tree. For any tree node with index u with a branch labeled with

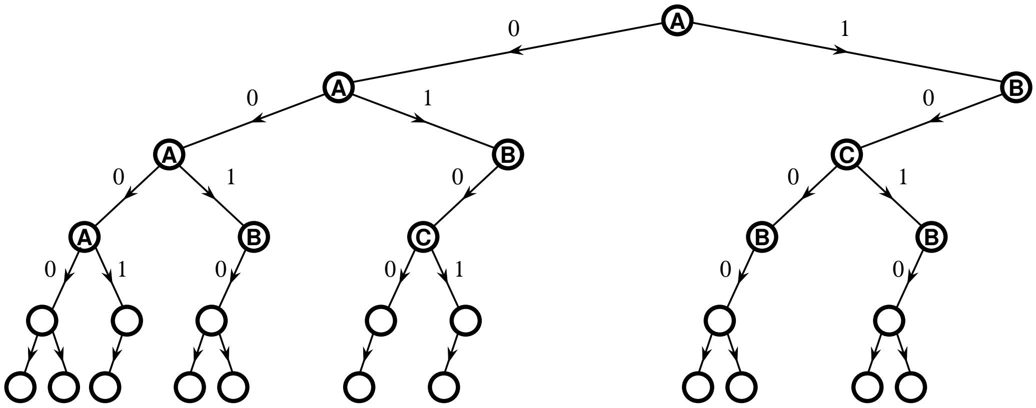


Fig. 5.4 The tree of fig. 5.2, with nodes labeled by morph.

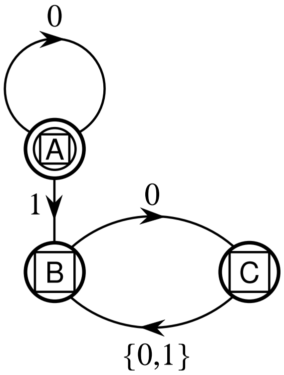


Fig. 5.5 The ϵ-machine constructed from fig. 5.4.

symbol σ leading to a tree node with index v, denoted by $u \underset{\sigma}{\rightarrow} v$, a corresponding transition is added to the machine from state u to state v with label σ. Having added all the transitions, the reconstruction procedure is completed. Fig. 5.5 shows the ϵ-machine constructed from the labeled tree of fig. 5.4.

Although the goal of the reconstruction procedure is to build a DFA, it sometimes happens that reconstruction results in an NFA. A nondeterministic transition in the reconstructed machine is due to the occurrence somewhere in the tree of two different branches $u \underset{\sigma}{\rightarrow} v$ and $u \underset{\sigma}{\rightarrow} v'$, with $v' \neq v$. This means that the reconstructed machine has two transitions with label σ leading from state u: it is an NFA. A number of factors that can cause nondeterministic transitions, two of which are described here.

The simplest reason for nondeterminism is insufficient statistics. That is, the tree depth D was chosen too large relative to the size N of the data sequence. In this case, empirical fluctuations in the observed length D subwords corrupt the shape of the overall tree, ultimately leading to subtrees with "missing" branches due to incomplete statistics. This artificially drives up the number of morphs observed, leading to an explosion in the number of machine states. The extra states are generally "dangling" states with no exiting transitions, and are generally reached by nondeterministic transitions. As more data is added to the tree, the missing branches in the subtrees are gradually filled in, reducing the number of morphs that occur, and the danglers and nondeterministic transitions disappear.

Another source of nondeterminism is nonstationarity in the data. If the length D subwords laid down on the tree were generated by a nonstationary source, or were collected in some nonstationary manner, branches at different levels of the tree may connect different pairs of morphs. For example, if the data is in the form of one or more spatial sequences taken during the evolution of some CA, the existence of spatial inhomogeneities ("walls" or "dislocations", see chapter 7 below) may very well lead to nondeterministic transitions.

A different problem occurs when L is very small, for example because D was small. In that case, the reconstructed machines can be inaccurate models of the underlying process that generated the data. This can be understood by observing that if the data source induces correlations on a length scale of L', morphs of depth $L < L'$ will fail to capture them. That is, two subtrees that are equivalent at depth L may be *distinct* at depth L'. Nodes are classed as future equivalent when in fact they are not. This leads to machines that are too "loose", in that they accept sequences that the data source cannot in fact generate.

In practice, the two extremes of (i) inaccuracy due to small morph depth and (ii) nondeterminism due to bad statistics are played off one another in a systematic fashion. For fixed, finite data size N, a series of reconstructions is performed for progressively increasing values of D and L. At first the machine is inaccurate because L is too small. Then as D and L increase the machine grows as the different classes of future equivalence are better resolved. If the data is stationary and if N is sufficiently large, there is a period during which several successively reconstructed machines are identical; at this point the reconstruction is said to have *stabilized.* Finally, as D and L grow too large, statistical fluctuations begin to dominate, the machine size blows up, and nondeterministic transitions appear.

Another effect, similar to nonstationarity, occurs when the data was not generated by a finite state process—it was, say, the output of some stack-based machine, which induces correlations across arbitrarily long strings. Then the reconstruction procedure never stabilizes. No matter how large N is, the machines continue to grow with D and L right up to the point where statistical fluctuations appear. Conversely, if a sequence of reconstructions from some data source of unknown type—i.e., an experimental system—never stabilizes, this is reason to suspect that the data source is in a more sophisticated computational class than finite state processes. In such a situation, the sequence of reconstructed machines is itself examined for regularities.[20] The reconstructed machines are successively better finite state approximations of the system. One example of this type of higher-level regularity in a nonlinear system, for which the limiting behavior is in the class of indexed context-free processes, can be found in ref. [29].

PART II

Spatial Patterns in One-Dimensional Cellular Automata

6 Dynamics in Language Space

In this chapter we draw the connection between the extended CA state space $\mathcal{A}^*$ of chapter 2 and regular language space $\mathcal{R}$ defined in chapter 4. The fundamental notion is to consider a CA state as a word in some regular language.[56,104] Obviously there is an infinite number of languages to which a given word may belong; our goal will be to find the language(s) that best describe the structural properties of typical states arising in the evolution of a CA.

It is at this point that the connection between automata theory and pattern bases becomes evident. A machine $\mathrm{M}(\mathcal{L})$ is not just a compact representation of a language $\mathcal{L}$ (i.e., of a set of strings). It is also a functional description of the structural properties shared by all of the words in $\mathcal{L}$; in other words, it is a specification of a particular *pattern*, of which each word $\omega \in \mathcal{L}$ is an example. We can turn this around to arrive at a precise, useful definition of pattern, one that is consistent with the discussion of chapter 1, amenable to quantitative description, and as we will see, particularly suited to CA. A pattern is the *structure* of a regular language $\mathcal{L}$—i.e., the set of features shared by the words in $\mathcal{L}$—and is described by the finite automaton $\mathrm{M}(\mathcal{L})$.* This definition of pattern also provides us with a definition of the *complexity* of a pattern that is at once objective, meaningful, and practical to use. To find the pattern bases of a particular CA, then, we will construct the machines $\mathrm{M}(\mathcal{L})$ for the languages $\mathcal{L}$ that best describe the fundamental patterns arising during the CA's evolution.

6.1 CA as Regular Language Processors

The connection between CA and regular languages is readily made by identifying the CA alphabet $\mathcal{A}$ with the finite state machine alphabet Σ. Then a CA state $\mathrm{s}_t \in \mathcal{A}^N$ is treated as a word in some language $\mathcal{L} \in \mathcal{R}$. It is important to note that this makes the analysis independent of lattice size, since $\mathcal{L}$ is a subset of the extended state space (because $\mathcal{A}^* \equiv \Sigma^*$) and hence may possibly contain strings of all lengths. When discussing CA with lattices of arbitrary but fixed size N, we will often use the regular language notation. It is understood that we mean only the strings in the language that are of length N. The general context, however, is to consider languages with words of all (or almost all) lengths, corresponding to different realizations of a particular pattern on CA lattices of different sizes. Thus $\mathcal{L}$ is usually taken to have infinite cardinality.

The operation of the CA update rule Φ is to recode a parent string s_{t-1} into a child string s_t. Stated in this way, it is clear that Φ can be implemented as a transducer, which we denote T_Φ. Note that representing the action of the CA in terms of a transducer—a *serial* device—is in seeming opposition to the fact that by definition CA perform *parallel* update of the lattice. In a single iteration of the CA, a finite number $(2r+1)$ of input symbols is read from s_{t-1} in order to specify each output

---

*It is clear that this notion of pattern can be immediately extended to other types of languages, such as context-free languages.

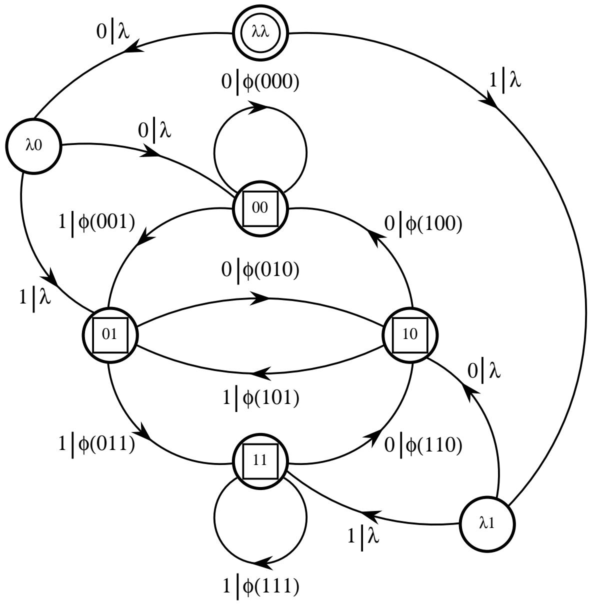


Fig. 6.1 The ECA update rule transducer T_Φ that reads a state s_{t-1} and emits its iterate $\mathrm{s}_t = \Phi(\mathrm{s}_{t-1})$. T_Φ is passed over the spatial array from left to right. The labels inside the states correspond to the previous two neighborhood symbols read. This is the machine for the infinite lattice case; finite lattices require additional constraints due to boundary conditions.

symbol in s_t. This means that T_Φ has a finite number of states, i.e., it is an FST (or more precisely, an FST with λ-outputs). Thus the representation of a CA update rule as a finite state transducer is essentially due to the spatially local coupling of the CA, as given by the finite radius r.

Since a transducer reads the symbols one by one—i.e., it *scans* the spatial configuration—the full definition of CA evolution must include a specification of the direction in which T_Φ is to be passed over the spatial array; here, we will always take this direction to be from left to right. In addition, on finite lattices the boundaries must be dealt with separately, a technical complication which we will ignore. The machine representation of T_Φ for an arbitrary elementary CA is shown in fig. 6.1. It operates as a finite-depth queue with two memory registers, in which a child symbol is emitted when the final member of its parent neighborhood is read. The output symbols of the transitions are given by the local update rule operating on the current parent neighborhood: $\sigma_{out}^{i-1} = \phi\big(\sigma_{in}^{i-2}, \sigma_{in}^{i-1}, \sigma_{in}^{i}\big)$. This is in fact just a description of the most memory-efficient way of programming one-dimensional CA dynamics for numerical experiments. The update transducers for CA of larger radius and alphabet size have more states, but are otherwise closely analogous. For a CA with radius r and alphabet of size k, the update transducer has a total of $2 \cdot k^{2r} - 1$ states. Of these, $k^{2r} - 1$ (including the start state) are transient synchronization states devoted to reading and storing the first $2r$ symbols $\sigma_{in}^0 \sigma_{in}^1 ... \sigma_{in}^{2r}$ in the input string, and whose exiting transitions have as outputs the null symbol λ. The remaining k^{2r} states are asymptotically accessible with exiting transitions labeled by $\sigma_{in}^i | \sigma_{out}^{i-r}$, where $\sigma_{out}^{i-r} = \phi\big(\sigma_{in}^{i-2r}, ..., \sigma_{in}^{i}\big)$.

6.2 Finite Machine Evolution

Representing the CA update rule as a transducer also makes it possible to trace the evolution of ensembles that are regular languages, by explicitly calculating the timeseries of machines $\mathrm{M}(\mathcal{L}_t)$ where $\mathcal{L}_0$ is a regular language of CA initial conditions and $\mathcal{L}_t = \mathbf{\Phi}(\mathcal{L}_{t-1})$. The method of doing this is the *FME algorithm*, as follows.

Definition. The **Finite Machine Evolution (FME) algorithm.** Given the CA update transducer T_Φ for a specific CA rule Φ and a finite automaton $\mathrm{M}_{t-1} = \mathrm{M}(\mathcal{L}_{t-1})$, we calculate $\mathrm{M}_t = \mathrm{M}(\mathcal{L}_t) = \mathrm{M}(\mathbf{\Phi}\mathcal{L}_{t-1})$ by using machine-composition and minimizing the resulting transducer with respect to the output symbols:

$$\mathrm{M}_t = [\mathrm{T}_\Phi \circ \mathrm{M}_{t-1}]_{\mathrm{out}} \tag{6.1}$$

This is a generalization of the method of ref. [104], mentioned in section 2.5, for constructing the iterates $\mathrm{M}(\Omega_t)$ of the global language $\Omega_0 = \mathcal{A}^*$.

Since both M_{t-1} and T_Φ have a finite number of states, their composition $\mathrm{T}_\Phi \circ \mathrm{M}_{t-1}$ is a finite state machine. Minimizing it with respect to the output symbols then gives a machine M_t that is necessarily a minimal DFA. This fact is essential to the connection between CA dynamics and regular language space, as the next section will show.

6.3 Pattern Dynamics

We are now in a position to view a CA as a mapping in regular language space, in which the evolution of an ensemble is given by the FME algorithm. In doing so, we necessarily discard the detailed information about *which* state in $\mathcal{L}_{t-1}$ gets mapped to which state in $\mathcal{L}_t$. On the other hand, we are now able to explicitly calculate the evolution of patterns *per se*. We can ask—and answer—the central question, *how do patterns evolve in cellular automata?* Succeeding chapters will investigate this issue in a number of different ways. First, we note a few important properties of $\mathcal{R}$.

The class of regular languages possesses many useful closure properties (see ref. [53]). For any two regular languages $\mathcal{L}_1$ and $\mathcal{L}_2$, we have

1. Set-theoretic difference: $(\mathcal{L}_1 - \mathcal{L}_2) \in \mathcal{R}$.
2. Complement: $\overline{\mathcal{L}_1} = (\mathcal{A}^* - \mathcal{L}_1) \in \mathcal{R}$.
3. Union: $(\mathcal{L}_1 \cup \mathcal{L}_2) \in \mathcal{R}$.
4. Intersection: $(\mathcal{L}_1 \cap \mathcal{L}_2) \in \mathcal{R}$.
5. Concatenation: $\{xy : x \in \mathcal{L}_1, y \in \mathcal{L}_2\} \in \mathcal{R}$.

In addition, there is the following property, crucial to the connection between regular languages and CA:

Proposition. Regular language space is closed under CA evolution.

Proof: By construction using the FME algorithm. For any $\mathcal{L} \in \mathcal{R}$ with corresponding machine $\mathrm{M}(\mathcal{L})$, the FME algorithm explicitly constructs the iterate $\mathrm{M}(\mathbf{\Phi}\mathcal{L})$. Since by construction $\mathrm{M}(\mathbf{\Phi}\mathcal{L})$ is a finite automaton, $\mathbf{\Phi}\mathcal{L} \in \mathcal{R}$. ▌

$\mathbf{\Phi}$ is an automorphism that maps regular language space into itself: $\mathbf{\Phi} : \mathcal{R} \rightarrow \mathcal{R}$. Thus the iterate of any regular language is regular: $\mathcal{L}_0 \in \mathcal{R} \Rightarrow \mathcal{L}_t \in \mathcal{R}, \forall t < \infty$. It has already been observed that when $\mathcal{L}_0 = \mathcal{A}^*$, the iterates are regular for all finite t.[104] We also note that, since any finite language is regular, the spatial language generated by any CA over a finite spatial lattice $(N < \infty)$ is also always regular. Thus, if either space or time is finite, the set of CA configurations is a regular language.[82] For infinite lattices, however, the asymptotic spatial language, found in the limit $t \rightarrow \infty$, need not be regular—or even recursively enumerable.[56] Somewhat surprisingly, this property need not hamper our computation-theoretic analysis, as will be seen in later chapters.

In drawing the connection between the evolution of regular languages and the qualitative theory of dynamical systems, it is important to observe that regular language space can to a certain extent be endowed with the topological properties necessary for an analysis using dynamical systems theory.

Remark. Regular language space induces a topology on the extended CA state space. We will call this topology the **regular language topology**.

Proof: The collection of regular languages $\mathcal{R} \equiv \{\mathcal{L}_i\}$ satisfies the two requirements of the basis of a topology:

(i) $\bigcup_i \mathcal{L}_i = \mathcal{A}^*$, since $\mathcal{A}^* \in \mathcal{R}$; and

(ii) $\mathcal{L}_i \cap \mathcal{L}_j \in \mathcal{R}$ for all $\mathcal{L}_i, \mathcal{L}_j \in \mathcal{R}$, by closure under intersection. ▌

Fig. 6.2 shows a schematic comparison of the operation of $\mathbf{\Phi}$ in configuration space $\mathcal{A}^N$, where it operates on sets of configurations, and in regular language space $\mathcal{R}$, where it operates on individual languages. The important advantage of the latter approach is that it allows us to investigate the dynamics of ensembles of structurally similar CA states without having to deal with the highly nontrivial complications introduced by tracing the evolution of individual states.

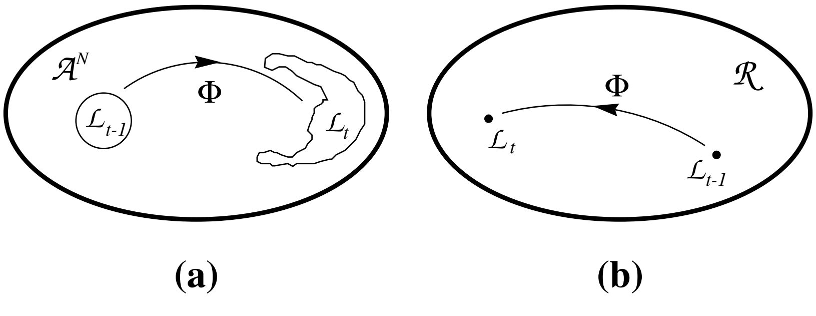


Fig. 6.2 Schematic of the operation of the language operator $\mathbf{\Phi}$ (a) in CA state space and (b) in regular language space. Note that case (b) subsumes all values of N simultaneously.

We can now identify different types of temporal evolution in language space.

Definition. An **invariant language** $\mathcal{L}$ is one that is mapped onto itself by the dynamic: $\mathbf{\Phi}(\mathcal{L}) = \mathcal{L}$. A **periodic language** is a language $\mathcal{L}$ such that $\mathbf{\Phi}^p(\mathcal{L}) = \mathcal{L}$ for some finite integer period p. A **language orbit** Γ is the set of iterates of a language: $\Gamma = \left\{\mathbf{\Phi}^t(\mathcal{L}) : \ t \geq 0\right\}$.

It is important to keep in mind the distinction between the evolution of languages and the evolution of individual states. For example, a temporally invariant or periodic language does not necessarily consist of states that are themselves temporally invariant or periodic, though the converse does hold.

Note that since any finite union of regular languages is itself a regular language, any periodic regular language orbit can be turned into a regular invariant language by taking the union over the orbit's elements. Aperiodic language orbits, which can occur only on infinite lattices, may be non-regular, however.

In succeeding chapters, we will outline some of the main qualitative features of regular language space, defining such fundamental structures as domains, vicinities, and attractors. We will then turn to specific systems and investigate their qualitative dynamics by means of these structures.

7 Regular Domains

This chapter outlines one of the central features in the qualitative analysis of CA space-time patterns: the *regular domain*. This is a precise formulation of a rather vaguely defined term in physics that is generally used to denote some spatio-temporal region in which “things are basically the same”. Well-known examples include ferromagnetic domains, in which all of the spins are oriented in the same direction, and roll patterns in the Rayleigh-Bénard system at the onset of convection, where a domain is a region in which all rolls have the same orientation. As we will see, one useful feature of the definition of domain given here is that it applies not only to uniform or periodic patterns, but also to “turbulent” systems, in which the domains themselves are disordered.

7.1 Definition of Regular Domain

Definition. A **regular domain** or “domain language” $\Lambda \in \mathcal{R}$ of a CA is a process language representing a set of spatial regions with the following two properties:
(i) *Temporal invariance:* Λ is mapped onto itself by the dynamic, i.e., $\mathbf{\Phi}\Lambda = \Lambda$; and
(ii) *Spatial homogeneity:* The process graph of Λ is strongly connected.[48] This is a form of spatial translation invariance and is similar to statistical stationarity of the spatial pattern.

Recall that if Λ is a regular language, then so is $\mathbf{\Phi}(\Lambda)$, and the FME algorithm gives a prescription for constructing $\mathbf{\Phi}(\Lambda)$ explicitly. This makes testing for invariance a practical task, even when Λ contains an infinite number of words—as is often the case.

It is straightforward to generalize this to more complicated dynamical behavior than single iteration invariance. A domain can be defined as a nonwandering orbit of languages $\{\mathcal{L}_1, \mathcal{L}_2, ...\}$. This includes those orbits with temporal behavior that is periodic or even chaotic. In each case, all languages in the orbit must be spatially homogeneous.

A domain describes a computationally-homogeneous *region* of the spatial array. In this way we take “domain” to mean the domain language Λ or, alternatively, any configuration $\mathrm{s} \in \Lambda$. For general CA there can be any number of different domains Λ^i, each with its own computational structure.

7.2 Types of Domains

The domain machine $\mathrm{M}(\Lambda)$ immediately provides information about the spatial properties of configurations in the domain language Λ. If the process graph of the $\mathrm{M}(\Lambda)$ consists of a single state with a single edge looping back on the state, then domain configurations are *spatially uniform.* If the process graph consists of V states arranged in a single periodic loop without any branching edges, then the domain configurations are *spatially periodic* with spatial period V. If the entropy rate of Λ (see section 4.5) is positive, then domain configurations must be spatially *disordered.* The complete description of the spatial properties of a domain is of course given by the domain machine $\mathrm{M}(\Lambda)$ itself.

To investigate the temporal behavior of individual domain configurations, we must look within the domain language to see how its members are mapped onto one another. As we noted in section 6.3, the temporal behavior of the *language* does not necessarily determine the temporal behavior of individual configurations in that language. Again, we can distinguish different types of behavior: domain configurations can be *temporally invariant*, *temporally periodic*, or *temporally disordered.* This last is characterized by a timeseries with a positive temporal entropy rate, defined as the growth rate in the number of temporal symbol sequences $...\sigma^i_{t-1}, \sigma^i_t, \sigma^i_{t+1}, ...$ as a function of sequence length; the temporal entropy can be estimated (for example) from explicit evolution of a Monte Carlo sample of domain configurations.

The spatial and temporal properties of a domain are linked by the dynamic. Because CA are deterministic systems, temporal disorder must necessarily be accompanied by spatial disorder. Similarly, a spatially uniform or periodic configuration is also temporally uniform or periodic. The converse does not hold, however. Configurations in a domain may be temporally uniform or periodic but spatially disordered. The simplest example of this is afforded by ECA 204, which implements the identity mapping $\Phi(\mathrm{s}) = \mathrm{s}$ for all configurations s. Under ECA 204, every set is invariant, so every spatially homogeneous language is a domain. The largest domain language is $\Lambda^0_{204} = \mathcal{A}^*$, the entire state space, which has an entropy rate of $h_0\left(\Lambda^0_{204}\right) = 1$. Typical configurations in Λ^0_{204} are spatially disordered, but temporally invariant.

Based on the spatial and temporal behaviors, then, we can distinguish a number of different types of domains, as follows:

1. Spatially and temporally uniform.
2. Spatially periodic and temporally uniform or periodic.
3. Spatially disordered and temporally uniform or periodic.
4. Spatially and temporally disordered. This type of domain will be called **chaotic**.

It is important to keep in mind that the different types of temporal behavior listed above refer to the characteristics of domain *configurations*, that is, to the characteristics of the CA states that make up a domain language—and not to the domain language itself, which is temporally invariant.

Some of the domain types are quite similar to the Wolfram classes of section 2.2. The list here has the following crucial difference, however: the behavior is defined over an *invariant set* (or perhaps a temporally periodic set)—i.e., a stationary language orbit—rather than over the usually nonstationary orbit of global languages Ω_t.

7.3 Dislocations, Walls and Particles

Definition. A **wall** is a boundary between adjacent domains. Two domains Λ^i and Λ^j separated by a wall will be written $\Lambda^{i,j}$. The wall itself will be denoted Γ^{ij}. Walls may consist of any number of cells, including zero. A **dislocation** is a wall of zero thickness separating two domains in the same language.

Definition. The **phase** $\phi^i(m)$ of site m in a domain Λ^i is the state q of the machine $\mathrm{M}(\Lambda^i)$ when symbol σ^m is read in a chosen direction.

Definition. The **relative phase** of two domains Λ^i and Λ^j separated by a wall at site m is given by

$$\delta\phi^{ij} = \phi^j(m) - \phi^i(m) \tag{7.1}$$

Strictly speaking, $\phi^i(m)$ in the definition of relative phase is not defined, since the domain Λ^i ends at cell $m-1$ because of the wall; therefore it is necessary to take $\phi^i(m)$ to be the phase that cell m *would have had* if the dislocation had not been present. This works for binary alphabets; for alphabets with more than two symbols, the definition of relative phase must be modified slightly—for example, by taking the relative phase to be difference $\widetilde{\delta\phi^{ij}} = \phi^j(m) - \phi^i(m-1)$. This modified definition is valid for all alphabets, but is slightly less intuitive in simple cases such as a dislocation caused by a phase shift in a domain pattern. Since we will be considering only binary alphabets in the examples in following chapters, the modification is unnecessary.

7.4 Domain Identification

The obvious question to ask at this point is, *for a given CA rule, what are the domains?* The method of answering this question consists of two parts: (i) discovering candidate languages; and (ii) verifying that the candidates satisfy the domain definition. It is likely that a fully analytic method of finding domains can be found, for example by a detailed examination of the first few iterates of the extended state space $\Omega_0 = \mathcal{A}^*$, but at this point such a method does not exist.

The candidate languages—that is, the languages that are under consideration as possible domains—are discovered experimentally using the machine reconstruction algorithm of chapter 5. For a given CA, the evolution from an arbitrary initial condition s_0 is explicitly calculated and the states s_t are used as the input data for machine reconstruction. Generally, the system is allowed to evolve for a time before data is collected, in order to allow short-lived transient behavior to die out. The usual goal is to find the languages that characterize the *typical* patterns that arise during the system's evolution. Therefore, a variety of different initial conditions and lattice sizes are chosen, and a single machine is reconstructed from all of the timeseries taken together. Alternative schemes include choosing a large lattice with a random initial condition and collecting data for a single state at time t, collecting data over some subregion of a large lattice, or choosing a different class of initial conditions. For example, in searching for multiple domains, typically one of the domains is discovered first and is designated Λ^0; then subsequent searches for other domains would choose initial conditions from the ensemble $\left(\mathcal{A}^* - \Lambda^0\right)$. In general, the search for domain candidates of a given CA is done iteratively, with each round of machine reconstruction being performed on an increasingly refined data set as indicated by the previous round.

Once a candidate language Λ has been found (in the form of a reconstructed machine $\mathrm{M}(\Lambda)$), the next step is to test it against the definition of regular domain. The iterate $\mathbf{\Phi}(\Lambda)$ is explicitly calculated using the FME algorithm of section 6.2, and $\mathrm{M}(\Lambda)$ and $\mathrm{M}(\mathbf{\Phi}(\Lambda))$ are compared to establish the first characteristic of a domain, i.e., temporal invariance. Automation of both the FME algorithm and machine comparison are straightforward. Spatial homogeneity is tested by direct examination of $\mathrm{M}(\Lambda)$ (or more precisely, its process graph), to see if it is strongly connected. For small machines, this is most simply done by visual inspection, but it is easily automated. In this way, testing the candidate Λ for invariance and spatial homogeneity constitutes a computer proof that Λ is or is not a domain of the CA.

7.5 Domain Transducers

From any finite set of domain languages $\left\{\Lambda^0, \Lambda^1, \ldots\Lambda^{m-1}\right\}$, a transducer can be constructed that maps each cell in domain Λ^i to symbol i, and maps each cell on wall Γ^{ij} to symbol $(m + mi + j)$. Thus for a set of m domains, there are $m(m + 2)$ symbols in the transducer's output alphabet. Alternative output codings can easily be devised, such as an output of m on *any* type of wall. Different output codings are minor variations on a theme, and will not be discussed here.

A space-time plot of the evolution of a given CA can be recoded into a plot of evolving domains and walls by separately applying this transducer to each spatial configuration s_t in turn. The recoded data explicitly shows the way in which the boundaries of domains evolve. In chapters 10 and 11 we will show the domains, domain transducers, and filtered space-time plots for a number of different CA.

This section describes a procedure for constructing a such a transducer, also called a **domain filter** because it "filters out" the details of the cells within a domain, mapping them all to the same value. A domain filter classifies an arbitrary CA state $\mathrm{s}_t \in \mathcal{A}^*$ into a given set of domains $\left\{\Lambda^i\right\}$ and the walls $\left\{\Gamma^{ij}\right\}$ between them. The filter can classify every cell in a CA state with at most two passes over the data. The construction procedure, outlined below, can be fully automated.

The context here is that the domain transducer is travelling over the CA state in a particular direction, reading and classifying symbols in sequence. In particular, if the transducer is reading from left to right, a wall Γ^{ij} is identified in the following way. Let $\omega = xy\sigma$ be a string such that $xy \in \Lambda^i$ for some unique i, but $\omega \notin \Lambda^k$ for any k. Further, choose y to be the shortest string such that $y\sigma \in \Lambda^j$ for some unique j (which may or may not be equal to i). Then σ is a wall of type Γ^{ij}.

Construction starts with the process graphs of the domains. There are three distinct steps in the construction procedure, related to three steps in the operation of the filter: synchronizing, encountering a wall, and re-synchronizing.

Before the transducer reads the first cell in the spatial array, it is in the start state representing total ignorance of the context. It has not yet read enough cells to identify which of the domains the cell is in, or if the cell is a wall. Therefore, the transducer must pass through a series of "synchronization" states, during which it reads symbols and branches on their values, but does not emit anything. These states are automatically generated from the set of domain process graphs by considering the entire set as a single nondeterministic finite automaton (NFA)—nondeterministic because all states are start states—and performing the standard NFA-to-DFA conversion on it. This results in a single machine with the various domain machines as recurrent components and a branching tree of transient synchronization states leading down into them. The transducer begins in the unique start state, parses its way down through the tree of synchronization states, and eventually passes into some state in one or another of the original domain process graphs, at which time it is synchronized and begins emitting symbols $\sigma_{out} = i$ for the domain Λ^i it is in. If a wall is encountered during synchronization, a "wall edge" is drawn as described below, except that no output symbol is emitted.

If the machine is in one of the recurrent states, and if a symbol is read for which there is no corresponding edge, this constitutes the observation of a domain wall. It is a symbol that is not consistent with the domain. To deal with all domain walls, it is necessary to draw an extra edge, to be labeled with a symbol identifying the two domains it separates, for every disallowed transition in the original machine. The domain transducer is thereby constructed to accept as inputs all words in $\mathcal{A}^*$, i.e., all possible spatial configurations.

In order to finish drawing the extra edge corresponding to a wall, it is necessary to specify the new edge's destination. The naive answer is to reset to the start state, since the domain structure of the input word has broken down. This is in fact not optimal, since there is more information at hand: specifically, the state the transducer was in when the wall was encountered, and the value of the wall cell. To take advantage of this information we apply the following scheme.

1. Let V_j be the state of the domain transducer T_Λ upon encountering the wall, and let a be the symbol read.
2. Consider the set $\mathcal{P}_j$ of words corresponding to paths in T_Λ that end in state V_j. Form the set $\mathcal{W}_j = \{\omega_j a\}$ by concatenating each word $\omega_j \in \mathcal{P}_j$ with symbol a.
3. Define the set $\mathrm{suff}(\mathcal{W}_j)$ of *legal* suffixes of words in $\mathcal{W}_j$, i.e., the suffixes that are in any domain language:
$$\mathrm{suff}(\mathcal{W}_j) = \left\{ y : \; xy \in \mathcal{W}_j \; , \; y \in \Lambda \text{ for some } \Lambda \in \left\{\Lambda^i\right\} \right\} \tag{7.2}$$
4. Determine the set $\{U_j\}$ of recurrent (domain) states reached by all words in $\mathrm{suff}(\mathcal{W}_j)$. If $\{U_j\}$ has a single member, the dislocation edge goes to that state.

If it has more than one member, state V_j is split into parallel states, one for each member of $\mathrm{suff}(\mathcal{W}_j)$. This in general does not introduce indeterminism into the transducer, since the size of $\mathrm{suff}(\mathcal{W}_j)$ is related to the number of distinct paths in the transducer converging on V_j. The splitting refines the equivalence class that V_j represents by separating paths leading into V_j that were previously identified.

5. Each newly drawn edge is classified according to the domains Λ^i and Λ^j of its source and destination states. For a set of m domains, the output label of an edge representing a wall Γ^{ij} between domains Λ^i and Λ^j is $\sigma_{out} = (m + mi + j)$.

7.6 Wall Trajectories

As we have noted, one of the applications of domain transducers is to trace the evolution of walls and dislocations at the boundaries between domains. Because of the local spatial coupling in CA, it is usually the case that if there is an isolated wall at cell m between two large domains at time t, there will be one (or more) walls nearby cell m at time $t + 1$. Other than this, however, there is in general no conservation of wall number. (The fact that the domain transducer scans the spatial pattern in a particular direction introduces some minor complications, but this is neither an essential nor an insurmountable problem.)

Given a machine that will encode the space-time data to recognize walls, it is possible to examine the encoded data for other, higher-order regularities. By designing the transducer properly, one can use this process to identify structures of arbitrary regular language complexity. The result is the original language “modulo” the transducer machine. In this all the encoded structures are excised, giving a clearer picture of long-range structures. This technique is applicable to a large class of systems, including those which possess regular domains. It also generalizes to structures of higher complexity, using recognizers based (say) on stack transducers that implement context-free or context-sensitive grammars for recognition.

Domain filtering is the first step in a hierarchical analysis of the system, in which the fundamental pattern bases in the data have been discovered (i.e., the domains) and filtered out, leaving a new space-time plot in which the features that were *not* identified (i.e., the walls) are the only features remaining. The next step in the hierarchy is to treat the filtered states as data in their own right, and to look for patterns in the walls. For example, in chapter 13 we will model wall evolution in a particular CA as a stochastic process. Another application is to apply machine reconstruction to the filtered data, to find finite automata describing the patterns formed by the walls. A simple example of this will be seen in chapter 10. For a general system, this hierarchy of models would be built as high as necessary, with each level exposing patterns and structural features that were obscured on the level previous. If, at some level, no new patterns are discovered, the hierarchy terminates. At that point, the analysis is essentially complete: the intrinsic computational capability of the CA has been identified.

8 Regular Vicinities and Attractors

This chapter continues our exploration of the connection between the dynamics of regular languages and the powerful methods of the theory of qualitative dynamics. Having identified the basic invariant sets in CA—regular domains—we next turn to the investigation of the stability and asymptotic temporal behavior of regular languages by defining regular vicinity and regular attractor.

8.1 Regular Vicinity: Definitions and Transducers

In discussing the stability of a domain or other invariant set, it is necessary to develop the notion of the *vicinity* of a language. In dynamical systems theory, an invariant set is stable if perturbations within an open neighborhood eventually fall back into the set. To adapt this to the present context, it is necessary to define an appropriate notion of "small" perturbation. In section 6.3, we noted that regular languages are a basis for a topology on the extended CA state space—and so regular languages are open sets in this topology. This allows us to define the regular vicinity of a language as an open set containing that language, and which is itself a regular language. The idea is to use the vicinity of a language in the way topological neighborhoods or ε-balls are used in qualitative dynamics.

Definition. A **vicinity** $V(\mathcal{L})$ of a language $\mathcal{L}$ is an open set in $\mathcal{A}^*$ such that $\mathcal{L} \subset V(\mathcal{L})$. A **regular vicinity** is a vicinity $V(\mathcal{L})$ that is a regular language: $V(\mathcal{L}) \in \mathcal{R}$.

The discreteness of the CA state space, coupled with absence of a preferred spatial origin—i.e., position independence—imposes a finite lower bound on the size of a perturbation. This precludes carrying over the standard notion of infinitesimal perturbation, which demands continuity of site values. The smallest position-independent perturbation to a CA state $\mathrm{s}_t = \ ...\sigma_t^{i-1}\sigma_t^i\sigma_t^{i+1}...$ is arguably a change in a single cell σ_t^i for some arbitrary i, resulting in a perturbed state $\tilde{\mathrm{s}}_t$ a Hamming distance of $|\tilde{\mathrm{s}}_t - \mathrm{s}_t|_H = 1$ away from s_t. To define "small perturbation" we therefore use Hamming distance, which leads to the following notion of Hamming vicinity.

Definition. The **Hamming vicinity** $V_H(\mathcal{L})$ of a regular language $\mathcal{L}$ is the vicinity generated by adding unit Hamming distance perturbation to each string in $\mathcal{L}$:

$$V_H(\mathcal{L}) = \left\{\tilde{\omega} : \ \tilde{\omega} = \omega \dot{+} \delta, \ \omega \in \mathcal{L}, \delta \in \{0^* + 0^*10^*\}\right\} \tag{8.1}$$

where $\dot{+}$ denotes the site-by-site sum modulo the alphabet size.

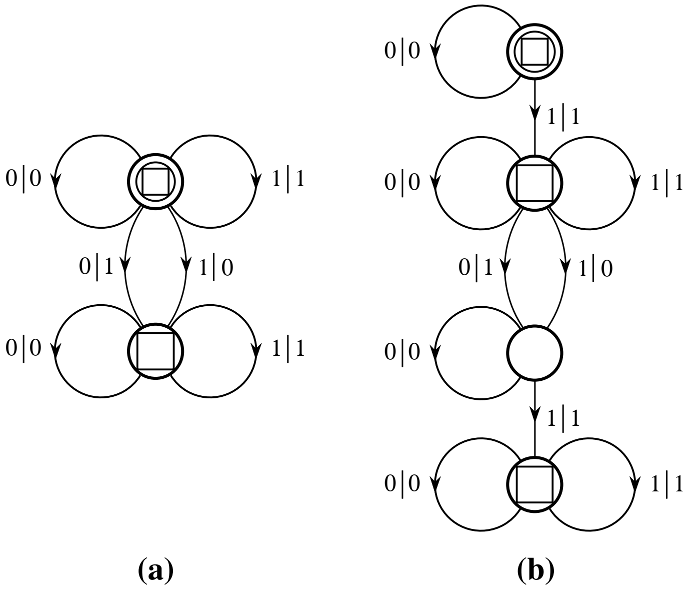
The Hamming vicinity $V_H(\mathcal{L})$ contains *all* possible single-site perturbations δ of *each* word in $\mathcal{L}$. Note also that by construction, $\mathcal{L} \subseteq V_H(\mathcal{L})$.

Some problems arise with perturbations of patterns on infinite lattices for certain CA. For rules with a "quiescent" symbol $\hat{\sigma}$, such that the pattern $(\hat{\sigma})^*$ is invariant, one can define "finite initial conditions"[63] as the set of states having a finite non-quiescent support—in other words, the cells with values other than $\hat{\sigma}$ are confined to a finitely bounded interval on the lattice. For example, in a CA with binary alphabet $\mathcal{A} = \{0, 1\}$ with designated quiescent symbol $\hat{\sigma} = 0$, the set of all finite initial conditions can be written $0^{\omega}(0 + 1)^{*}0^{\omega}$, where 0^{ω} denotes an infinite number of contiguous 0s. For these initial conditions, the effect of the perturbation may depend sensitively on whether it occurs within the non-quiescent region or outside it, for example by creating a domain wall in one case but not in the other. It is therefore convenient to restrict the vicinity to include only perturbations in the non-quiescent region. Note that the new vicinity type can be used with finite lattices with no additional consequences, since in that case the region of non-quiescent support is the whole finite lattice.

Definition. The **restricted Hamming vicinity** $V_r(\mathcal{L})$ of a regular language $\mathcal{L}$ is the vicinity generated by adding unit Hamming distance perturbation somewhere within the region of non-quiescent support of each string in $\mathcal{L}$:

$$V_H(\mathcal{L}) = \left\{\tilde{\omega} : \tilde{\omega} = \omega \dot{+} \delta_r, \; \omega \in \mathcal{L}\right\} \tag{8.2}$$

where the perturbation δ_r is in the set $\{0^* + 0^*10^*\}$ as above, but is further restricted to operate within the non-quiescent support of each string $\omega \in \mathcal{L}$.



(a) **(b)**

Fig. 8.1 (a) The Hamming vicinity transducer T_{V_H}. (b) The restricted version T_{V_r}. These transducers are irreducibly nondeterministic.

Figs. 8.1(a) and 8.1(b) show transducers used to generate $V_H(\mathcal{L})$ and $V_r(\mathcal{L})$ from any $\mathcal{L}$, respectively denoted by T_{V_H} and T_{V_r}. Since the perturbations required to generate these two vicinities can be accomplished by finite state transducers, $V_H(\mathcal{L})$ and $V_r(\mathcal{L})$ are necessarily regular languages, as the following proposition indicates.