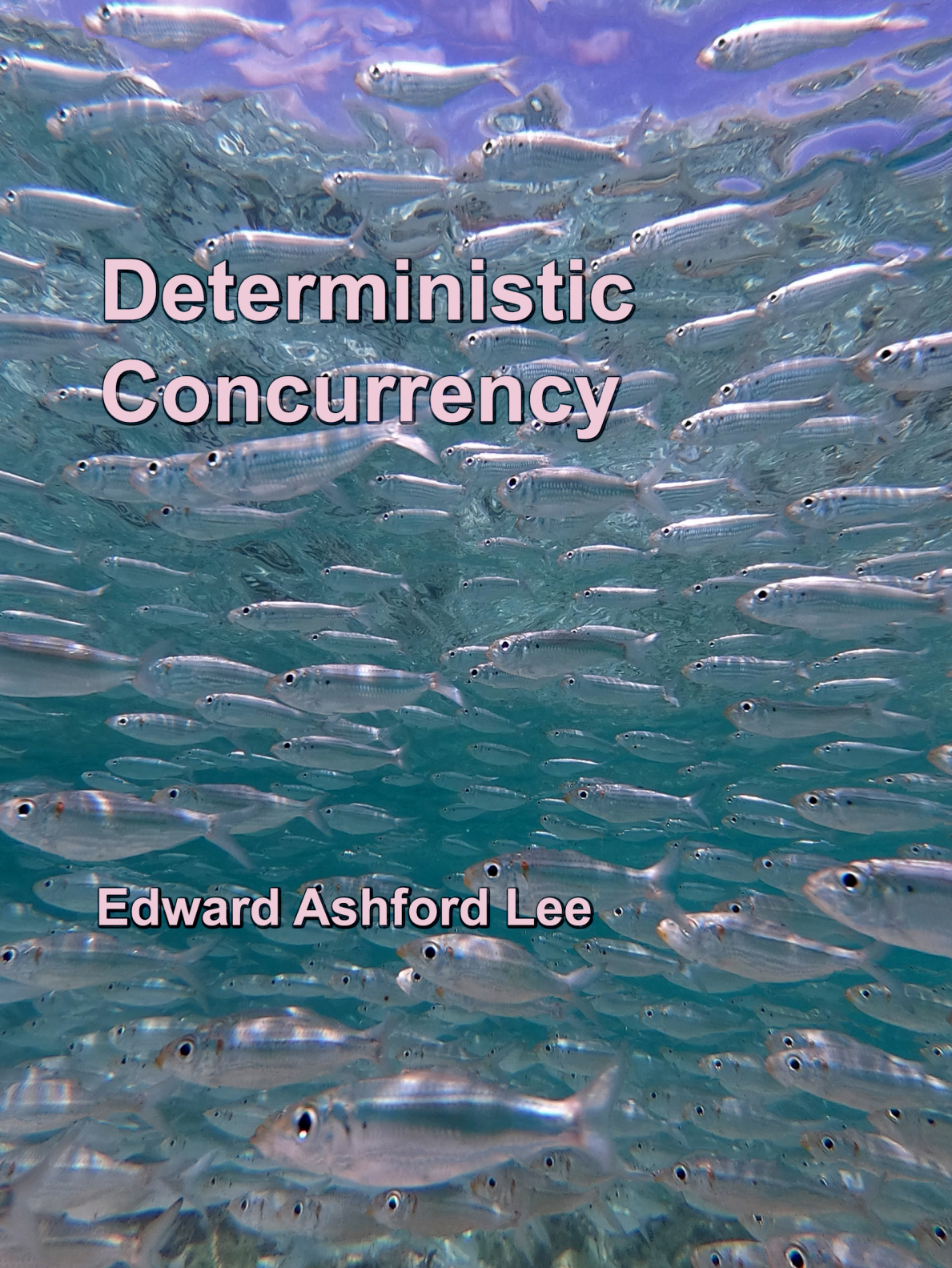

Deterministic
Concurrency
Edward Ashford Lee

Zeroth Edition, Version 0.01- DRAFT
January 2, 2026

Contents

Acknowledgements

(FIXME: TBD)

Preface

To many traditional computer science theorists, the title of this book may appear to be an oxymoron. It is not uncommon, particularly in the early literature, to equate concurrency with nondeterminism and to treat the two as a profound difficulty encountered when trying to understand programs. For example, Hennessy and Milner (1980) say, "what exactly is meant by the behaviour of non-deterministic or concurrent programs is far from clear." They go on:

> Any satisfactory comparison of the behaviour of concurrent programs must take into account their intermediate states as they progress through a computation, because differing intermediate states can be exploited in larger programming environments to produce different overall behaviour (e.g. deadlock).

Indeed, nearly all programming models at the time (and to this day, see Chapter 3) have the property that they can exhibit observably different behavior given the same inputs.

But we must be careful. What do we mean by "observably different behavior"? Whether a program deadlocks or not is typically quite important. Deadlocking is an observable behavior. But so is timing, and for many programs, the exact time at which an output is produced is observable but not important. When considering a

simple, single-threaded imperative program, we are likely to declare the program to be deterministic, meaning that given the same inputs, it always produces the same observable behavior, where what we mean by "observable behavior" is the printed outputs. We are unlikely to include in "observable behavior" the time at which each output is printed. A concurrent program may be similarly deterministic, as long as we restrict what mean by "observable behavior."

The first chapter of this book, which is largely reproduced from Lee (2021), confronts the meaning of the word "determinism." In this book, determinism is not a property of some physical system (a computer running a computer program, for example). Instead, it is a property of a model that system. And the structure of the model matters. The same physical system may have useful models that are both deterministic and nondeterministic.

(FIXME: To be continued)

Acronyms

APG	acyclic precedence graph	142
BDF	boolean dataflow	148
CCS	calculus of communicating systems	105
CRC	cyclic redundancy check	25
CSDF	cyclo-static dataflow	152
DDF	dynamic dataflow	151
FIFO	first in first out	63
FSM	finite-state machine	155
GLB	greatest lower bound	182
HDF	heterochronous dataflow	155
IDF	integer dataflow	148
KPN	Kahn process network	62
IP	internet protocol	8
LUB	least upper bound	181
PASS	periodic admissible sequential schedule	138
PN	process network	62
PSDF	parameterized SDF	158
SDF	synchronous dataflow	129
TCP	transmission control protocol	8

UDP	user datagram protocol	12

Notation

$\mathbb{B} = \{0,1\}$	binary digits	172
$\mathbb{T} = \{false, true\}$	truth values	172
$\mathbb{N} = \{0,1,2,\cdots\}$	natural numbers	172
$\mathbb{Z} = \{\cdots,-1,0,1,2,\cdots\}$	integers	172
$\mathbb{R}$	real numbers	172
$\mathbb{R}_+$	non-negative real numbers	172
$A \subseteq B$	subset	172
$\wp(A)$	powerset	172
2^A	powerset	172
$\emptyset$	empty set	172
$A \backslash B$	set subtraction	172
$A \times B$	cartesian product	173
$(a,b) \in A \times B$	tuple	173
A^0	singleton set	173
$f : A \to B$	function	173
$f : A \rightharpoonup B$	partial function	173
$g \circ f$	function composition	174
$f^n : A \to A$	function to a power	174
$(A \to B)$	set of all functions from A to B	174
B^A	set of all functions from A to B	174

$f^0(a)$	identity function	174
$\hat{f}\colon 2^A \to 2^B$	image function	174
$f\|_C$	restriction	176
π_I	projection	176
$\hat{\pi}_I$	lifted projection	177
$\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$	von Neumann numbers	178
A^ω	infinite sequences	178
A^*	finite sequences	177
$A^{\mathbb{N}}$	infinite sequences	177
A^{**}	finite and infinite sequences	177
λ	empty sequence	177
$\mathbf{\Lambda}_m^n$	empty function	112
ϕ	dataflow actor functional	113
$\leq$	partial order	179
$(A, \leq)$	poset	179
$\sqsubseteq$	prefix order	179
$(A, <)$	strict poset	181
$\mathbb{Q}$	rational numbers	182
$\bigvee B$	join, least upper bound, LUB	182
$\bigwedge B$	meet, greater lower bound, GLB	182
$\perp$	bottom element	183
∞	infinity	183
$[A \to B]$	set of all continuous functions	190
$a.b$	concatenation	64

Part I

Background

1 Determinism

This chapter is adapted from Lee (2021).

1.1 Determinism as a Property of Models

For most of my professional research career, I have sought more deterministic mechanisms for solving various engineering problems. My focus has always been on systems that combine the clean and neat world of computation with the messy and unpredictable physical world. Why the obsession with deterministic mechanisms? My wife, who is an expert in stochastic models, gives me a hard time about this obsession, observing, correctly, that deterministic models are just a special case. Why not, then, focus on the more general set of nondeterministic models?

Today, our society relies heavily on deterministic engineered systems. The balances in our bank accounts are a consequence of the inputs to the bank's computing systems. The email received is the email that was sent. The files on our computers contain the data that we put there. Our cars start when we push the start button. None of these is perfect, of course. Files and communication can get corrupted and machinery can fail. But for many systems, we easily see trillions of bits of information before encountering an error and years of reliable operation before encountering a failure.

None of this reliability comes for free. A great deal of engineering effort has gone into making the behavior of these systems repeatable. The underlying physics, a sea of electrons sloshing around, for example, offers no such repeatability. We should be in awe of these systems, but we are not because we have gotten used to their dependability. In the early days of computing, vacuum tubes would fail frequently enough that programs had to be kept short so that they would complete before a failure occurred. Computer memories were nowhere near reliable enough to replace paper ledgers. And cars frequently would not start. But today, computers operate continuously for months and years, performing billions of operations per second without error. When errors do occur, a second layer kicks in, using error-correcting codes or redundant systems to self correct. As a result, paper ledgers have vanished. And cars start.

There are many reasons for this dependability, but for the purposes of this book, I would like to focus on my obsession: the principle of determinism. I am not talking about the philosophical view that everything that happens is an inevitable consequence of pre-existing conditions, although that concept is related. I am instead talking about an engineering principle, where a model defines exactly one correct behavior for a system in response to its inputs. Like the philosophical concept of determinism, the engineering version has many fine points, subtleties, and consequences that are subject to debate. My purpose in this chapter is to address these fine points. I will start informally, but I will not shy away from subtleties.

The first issue is that I will treat determinism as a property of a model, not as a property of a physical artifact.

Example 1.1: Consider a logic gate, an AND gate for example. Is it deterministic? This is, of course, a trick question. If what we mean by "an AND gate" is a piece of silicon, then the answer to this question depends on whether the underlying physics of electrons sloshing through semiconductor atoms buffeted by thermal noise under the influence of electric fields is deterministic. I will consider later in the chapter the question of whether modern physics can answer that question, but suffice it to say, for now, that there is a different interpretation of the question that makes it much easier to answer.

If what we mean by "an AND gate" is a model, an abstraction, that given two boolean inputs, produces the output "true" if the two boolean inputs are "true" and otherwise produces the output "false," then this AND gate is deterministic. It defines exactly one correct behavior for each pattern of inputs. This model, however, leaves a great deal unsaid. When must the inputs be provided? When is the output provided? How are "true" and "false" to be represented in the physical world? Who or what is the observer of the output? What can that observer perceive about the AND gate? Determinism does not require that *all* aspects of a system be prescribed. It only requires that what we construe as "behavior" be prescribed.

Implicitly, I have assumed that what we mean by "inputs" is a pair of boolean values from the set {*true*, *false*} and what we mean by "behavior" is producing a boolean output chosen from the same set. This is a mathemat-

ical abstraction, not a physical artifact. If an observer is able to perceive, for example, the temperature of the AND gate, then this could be construed as part of its behavior, and the AND gate is no longer deterministic. Many possible temperatures are consistent with correct behavior of the AND gate. The abstraction also says nothing about the time at which these inputs are presented and the output is produced, so if the timing is observable, then again the AND gate is no longer deterministic.

Whether something is deterministic or not, therefore, depends on what can be observed about it and whether we consider these observations part of the "behavior."

1.1.1 Engineering Models vs. Scientific Models

A mathematical abstraction is a model. What makes this model useful? In my book, *Plato and the Nerd* (Lee, 2017), I observe that the purpose of a **scientific model** is to emulate a physical system. For an **engineering model**, the purpose of the physical system is to emulate the model. The mathematical abstraction of the AND gate that I have given is a rather poor model of electrons sloshing in silicon, so it is not likely to be used as a scientific model. The model is valuable because we are able to construct silicon that behaves like the model, not because it accurately describes what electrons do as they move through silicon atoms. Hence, it is more useful as an engineering model than as a scientific model.

In this book, I will focus primarily on deterministic engineering models, and hence, whether the physical world is in fact deterministic is not nearly as important as whether we can coerce the physical world to behave like a deterministic model. The reality today is that humanity has figured out how to get silicon electronics to behave like deterministic models with astonishing fidelity.

Of course, no model is perfect, and no physical realization is perfect. The famous quote from George Box, "all models are wrong, but some are useful" (Box and Draper, 1987), applies to scientific models, where it is incumbent on the model to match the physical world. The mirror image, "all physical realizations are wrong, but some are useful" (Lee and Sirjani, 2018), applies to engineering models, where it is incumbent on the physical world to match the model.

In this book, determinism is a property of models, not of physical realizations. I will analyze why this property is so valuable, and why it is worth a great deal of engineering effort to build physical systems that emulate deterministic models. I will argue that using deterministic models is not at odds with dealing with uncertainty nor with building fault-tolerant systems. On the contrary, it makes these things easier. I will discuss the relationship between deterministic models and determinism in the physical world. And I will review results that show that determinism is incomplete in that sets of deterministic models that are rich enough to model the physical world do not contain their own limit points. From a practical perspective, this means that nondeterminism is unavoidable in a broad class of models.

1.1.2 Determinism, Causation, and Randomness

Determinism in philosophy is the principle that every action is a consequence of its preconditions and that fixed rules uniquely determine these consequences. The closely related principle of **causation** (or **causality**) is that the preconditions and the rules *cause* the consequences. Nondeterminism, under this principle, can be thought of as . Many people find it difficult to accept that there can be uncaused action and will go to great lengths to find some cause when none is obvious, resorting eventually to God or some other supernatural cause when all else fails.

Causation is an explanation of *why* something happens. Instead of focusing on why, we can focus on *what* happens. In this view, it is the *uniqueness* of the consequences that is the core of the concept of determinism, not causation. There is only one possible consequence of each precondition. This uniqueness is the property I focus on in this book.

Nondeterminism is closely related to the concept of **randomness**, but there are important distinctions. In vernacular use, a random event is an unpredictable one. But even deterministic processes can be unpredictable, so under this view, randomness does not imply nondeterminism. Nondeterministic processes are always unpredictable, however, so nondeterminism does imply randomness in this vernacular understanding of the word.

There is a more technical interpretation to the word "random," however, which seeks to quantify the *likelihood* of various possible outcomes with **probabilities**.

Nondeterminism, in contrast, is about *possibilities*, not probabilities. It says nothing about likelihoods.

This more technical interpretation of randomness leads to a deeper split with nondeterminism, making these concepts almost orthogonal. Consider what we mean by "likelihood" and "probability." In the classical **frequentist** interpretation of these concepts, a probability is a prediction of the proportion of repeated experiments that will turn out some way. For example, what fraction of die tosses will produce snake eyes? But no repeat of any experiment begins with the same preconditions as the previous experiment, so this frequentist interpretation has no need for nondeterminsim.

Some experiments *cannot* be repeated, and yet we consider them random and assign them probabilities. For example, what is the probability of a major earthquake in San Francisco in the next 30 years? This probability has no valid frequentist interpretation. In the **Bayesian** interpretation, a probability is a measure of how much we *know* about the outcomes, not a measure of how frequently they occur (see Chapter 11 of Lee (2017) for a discussion of the frequentists vs. the Bayesians). Once again, randomness has no need for nondeterminism. If we know little about some deterministic system, we can consider its outcomes to be random and assign them probabilities. Whether the underlying system is nondeterministic becomes irrelevant.

1.1.3 The Practical Value of Determinism

Let me begin by pointing out that our engineering toolkits are full of extremely useful deterministic models. Differential equations are deterministic (with some important exceptions that I will discuss below). Most computer programming languages, if we exclude their mechanisms for concurrency, are deterministic. Synchronous digital logic, the most widely used electronic circuit design paradigm, is deterministic (and concurrent). Instruction set architectures are mostly deterministic. **TCP/IP** (**transmission control protocol/Internet protocol**), the central protocol in the Internet, is deterministic in the sense that a stream of bits in yields the same stream of bits out, even though packets may be dropped (even deliberately to shape traffic) or arrive out of order. Computer memory is deterministic, even when it is built on unreliable components, as demonstrated for example by the RAID project (redundant arrays of inexpensive disks) (Patterson et al., 1988).

Today, it is even possible to build deterministic cloud services on top of farms of unreliable servers.

All of the above are deterministic *models*. The underlying physical realization is almost certainly not deterministic, and considerable effort and cost has gone into building physical systems that are *sufficiently* faithful to these deterministic models. Perfect faithfulness is not achievable ("all physical realizations are wrong..."). But we would not be putting in all that effort if determinism were not quite valuable.

There are many reasons why determinism is valuable. Here are a few:

1. **Repeatability**. Every engineer tests a design while developing it. Such testing is valuable only under the assumption that if a system works once, it will work again in largely the same way. The behavior is repeatable given the same inputs. This principle is systematized in test-driven design, where collections of regression tests have well-defined correct behaviors. When a change is made to a design, the regression tests will reveal whether the change has resulted in some unexpected behavior.

2. **Consistency**. When two agents come to a conclusion, it is often valuable that the conclusions be the same. Your bank and you usually agree on your bank balance. You both keep track of that balance using deterministic computations. Given the same input information, both will produce the same results. Consistency is discussed in much more detail in Chapter 2.

3. **Predictability**. If behaviors in response to certain inputs are predictable, then they do not need to be discovered through testing or by surprise after deployment. As I will explain below, determinism does not assure predictability, but nondeterminism assures unpredictability. Some deterministic models are predictable.

4. **Fault detection**. A deterministic model gives an unambiguous definition of "correct behavior." This enables fault detection. Any behavior that deviates from the correct behavior is faulty behavior, meaning that some assumption has not been satisfied. An AND gate, for example, will not behave in conformance with the model if the temperature is too high. A CRC check on a value read from memory (something computed using a deterministic algorithm) reveals whether the memory has experienced a

fault that caused a bit to flip. Both of these faults would be harder to detect without a deterministic model defining correct behavior.

5. **Simplicity**. In a deterministic design, one input implies one behavior. In a nondeterministic design, it is easy to get an exponentially growing number of allowed behaviors. This makes comprehensive testing much more challenging. It can also compromise the ability to analyze a model using, for example, formal methods.

6. **Unsurprising behavior**. Often, we want engineered systems to be boring. They should not surprise us with unexpected behaviors. When a computer programmer gets unexpected behaviors from a multithreaded program, for example, it can be extremely disruptive, costly, and difficult to fix (Lee, 2006b).

7. **Composability**. When building large systems out of smaller components, having a clear understanding of the possible behaviors of those smaller components becomes essential. Deterministic models of those components makes this far easier.

When engineers are forced to move beyond these deterministic mechanisms, building correct designs becomes far more difficult. Unfortunately, many of today's most popular programming frameworks supporting parallel and distributed computing make it quite difficult to achieve deterministic behaviors (see Chapter 3).

1.1.4 The Practical Value of Nondeterminism

None of what I have said implies that nondeterminism has no value. Nondeterministic models can also be valuable. Here are a few reasons:

1. **Abstraction**. A nondeterministic model may provide a much simpler abstraction of a deterministic model (Cousot and Cousot, 1977). Note that this does not contradict Property 5 above. The nondeterministic *model* is simpler, but if it is a sound abstraction, the number of *behaviors* allowed by the model cannot possibly be smaller than those of the model it abstracts.

A smaller model may be easier to understand. It may also be easier to formally analyze, as long as the formal analysis does not require exhaustively exploring all possible behaviors.

2. **Uncertainty**. Nondeterministic models are useful as *scientific* models of systems where our knowledge of their behavior is incomplete. For example, modeling a human operator of a car is probably not a reasonable task for deterministic modeling. Such a model of a human operator, however, is almost certainly a scientific model, not an engineering model.

3. **Deferred design decisions**. For *engineering* models, nondeterminism can be a useful way to capture deferred design decisions. Deferred design decisions represent a different form of uncertainty. It is uncertainty about the model, not uncertainty about what is being modeled.

4. **Security**. Many techniques for securing software systems rely on good random number generators. Pseudo-random number generators are deterministic, a weakness that opens a vulnerability. Seeding a psuedo-random number generator with the result of a nondeterministic process can help.

5. **Don't care**. A model may have many acceptable behaviors in response to a given input. Serverless architectures in the Internet, for example, are useful because computations in response to inputs can be mapped to any available server. It is irrelevant to correctness in what order and where these computations are performed.

6. **Surprising behavior**. We don't always want engineered systems to be boring. An automated musical accompanist, for example, might be enriched by unpredictable behavior. Any artificial intelligence attempting to exhibit human-like behavior will also need to at least appear to be nondeterministic (see my book (Lee, 2020, Chapter 10) for a discussion of the connection between creativity and nondeterminism).

Notice that valid uses of determinism and nondeterminism are quite different from one another, which suggests that engineers should consciously choose between them. This is rarely easy. For example, when writing multithreaded or distributed software, almost all available languages, frameworks, and middleware are nondeterministic by default. Achieving determinism is left to the designer and is sometimes impossibly difficult (Lee, 2006b; Lohstroh and Lee, 2019).

Probabilistic, stochastic, or random models (I will treat these words as synonyms) are also useful, but as I pointed out in Section 1.1.2, the concept represented by these words is largely orthogonal to nondeterminism. In computer security, for example, random numbers are usually generated using a deterministic algorithm that ensures a desirable empirical probability distribution on observations. A nondeterministic source of random numbers may, in fact, be hopelessly inadequate for computer security without some characterization of its probability distribution. In the Bayesian interpretation of probability, it is irrelevant whether the thing being observed is deterministic. What is relevant is whether the observation can be anticipated given prior knowledge.

1.1.5 The Cost of Determinism

Building physical systems that behave like deterministic models usually comes at a price. A synchronous digital circuit, for example, has to be clocked slowly enough to provide comfortable margins to accommodate delay variability due to manufacturing tolerances and temperature. These margins are a cost in performance. Some power-users of computers have discovered that they can often "overclock" their CPU without introducing too many errors. This may be acceptable for gaming, for example, but it would probably not be wise for a bank to overclock their CPUs.

For distributed systems, determinism comes at the cost of latency (see Chapter 2). For applications where latency is a key performance metric, nondeterministic solutions may be a better choice. For example, some distributed applications choose to use **UDP** (**user datagram protocol**) rather than TCP for network communication. The UDP protocol is simpler and faster than TCP, but it sacrifices the guarantee of eventual in-order delivery of messages. If occasional packet losses are acceptable for a particular application, then this may be a reasonable choice.

As with all engineering, there are tradeoffs. To evaluate these tradeoffs, it is helpful to have a deeper understanding of what determinism really is. We look at that next.

1.2 What is Determinism?

John Earman, in his *Primer on Determinism*, states that "determinism is a doctrine about the nature of the world" and concludes that "a real understanding of determinism cannot be achieved without simultaneously constructing a comprehensive philosophy of science" (Earman, 1986, p. 21). If instead of a "doctrine about the nature of the world" we view determinism as a property of models, then no such philosophy is needed. We can focus instead on the usefulness of the concept of determinism.

As a property of models, determinism is easy to define:

Definition 1.1. *A model is **deterministic** if given all the **inputs** that are provided to the model, the model defines exactly one possible **behavior**.*

In other words, a model is deterministic if it is not possible for it to react in two or more ways to the same inputs.[1] Only one reaction is possible in the model. More precisely, only one reaction is *correct*; any other reaction is not one given by the model. In this definition, I have emphasized words that must be defined within the modeling paradigm to complete the definition, specifically, "inputs" and "behavior."

Example 1.2: If the behavior of a particle is its position $x(t)$ in a Euclidean space as a function of time t, where both time and space are continuums, and if the input $F(t)$ is a force applied to the particle with mass m at each instant t, then Newton's second law,

$$F(t) = m\frac{d^2}{dt^2}x(t) \tag{1.1}$$

is a deterministic model (mostly, see Section 1.4).

The *same particle*, however, may have an equally valid nondeterministic model. For the same definitions of behavior and input, the statement, "The particle accelerates in the direction of the net force," provides a nondeterministic model. It admits many more behaviors than the deterministic

[1] For a nice formalization of this concept, see Edwards (2018).

model. This latter model is a sound abstraction because the behavior of the deterministic model is among the behaviors of the nondeterministic one.

When considering a *physical* particle, the thing-in-itself rather than its model,[2] then Earman's hesitation comes to the foreground. A definitive answer to whether the actual particle is deterministic may never be possible. Any discussion of the determinism of a particle is necessarily a discussion of some model, not of the thing-in-itself, even if that fact goes unsaid. Some models of the particle are deterministic and some are not. I will address the physics of determinism in Section 1.4. But first, let us address some of the subtleties with models before we complicate the picture with the thing-in-itself.

1.2.1 Behavior, State, and Observation

When determining whether a model is deterministic, we need to define "behavior." Many models tie behavior to the notion of "state." Newtonian physics, for example, does this by defining a time continuum and modeling the state of a system as positions and momentums at a shared "instant" in time. "Behavior" can then be defined as the evolution of this state in the time continuum in response to the inputs, which are forces. Modern physics complicates this simple picture (see Section 1.4), but the Newtonian models nevertheless remain useful.

In their classic book on automata theory, Hopcroft and Ullman define "state" as follows:

Definition 1.2. *The **state** of the system summarizes the information concerning past inputs that is needed to determine the behavior of the system on subsequent inputs. (Hopcroft and Ullman, 1979, p. 13)*

The state is all the information about its past that can affect its future behavior. In other words, the state of a system is the information about the past such that any additional information tells us nothing about its future behavior. This definition requires defining "behavior," but it also requires a notion of "past" and "future" as well as the boundary between these, the "present." This notion

[2]The philosopher Immanuel Kant made the distinction between the world as-it-is, what he called the thing-in-itself (*das Ding an sich* in German), and the phenomenal world, or the world as it appears to us.

turns out to be problematic for very fundamental reasons. It is problematic in modern physics, but also in practical realizations of parallel, distributed, and cyber-physical systems (Lee, 2008). These systems have no well-defined "present" separating past and future. The notion of state, therefore, useful as it is, sits on shaky foundations.

In computing, automata theory is built around the notion of state (or equivalent notions working with sequences of symbols). Variants of automata theory use either a Newtonian time continuum or a discrete, countable model of time, both of which provide a "present" that separates past from future. Automata theories share with Newtonian physics the idea that time advances uniformly throughout the system and that there is a shared notion of an "instant" of time, a "present," at which the system is in some state.

In automata theory, unlike Newtonian physics, even if time is a continuum, the *state* evolves in discrete jumps. At an instant, the state of the system changes from some value s to some other value s'. Again, "behavior" can be defined as the evolution of this state in time. When the system is distributed or concurrent, where it consists of multiple interacting components, possibly spread out over space, nondeterminism proves to be a useful way to model the uncertainty about the order in which state changes occur in the distinct components.

1.2.2 Input-Output vs. State-Trajectory Behavior

An alternative way to define behavior is to introduce the notion of **outputs**, data or symbols that some observer interprets as the behavior of the system in response to some input. Instead of state-trajectory behavior, we have input-output behavior. The relationship between input-output behavior and state-trajectory behavior is fascinating, subtle, and complex. The essential question is what can an observer observe? The possible observations are the "output" of the system. But this is true even if we define "behavior" to be a state trajectory. In that case, we have implicitly defined an observer that can observe the state of the system at an instant in time. This definition will require us to define "an instant in time," which, as we will see, becomes difficult for a distributed system.

Consider automata theory, where a state transition system may be endowed an explicit notion of inputs and outputs. Inputs trigger state transitions and outputs result from state transitions. A transition system evolves as a *sequence* of state

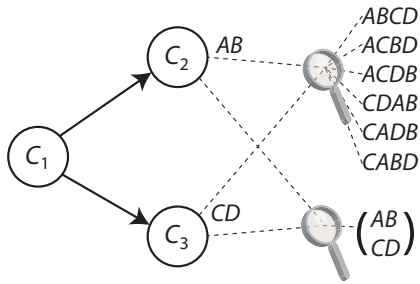


Figure 1.1: Three component system that is either deterministic or not depending on how you define the observer.

transitions, and hence, both the inputs and the outputs are sequences of symbols from some alphabet. A sequence of symbols forms a **sentence**, and the set of all sentences that are possible defines a **language**. In an input-output interpretation of behavior, a deterministic model is one for which, given any input sentence, there is only one possible output sentence.

The fact that inputs and outputs are defined in automata theory to be a *sequences* of symbols is important. An observer only ever sees sequences of symbols. These sequences cause no end of difficulties with modeling concurrent systems, where the order in which symbols occur may not be relevant or even well defined. Starting in the 1970s, Robin Milner and Tony Hoare pioneered methods for formally modeling such systems (see Chapter 14 of Winskel (1993) for a nice summary of these results). I will illustrate these difficulties with some key observations due to Milner.

In 1980, Milner published a series of lecture notes giving an elegant formalism that he called a Calculus of Communicating Systems (CCS) (Milner, 1980). This formalism made it abundantly clear that looking only at the sentences of inputs and outputs is insufficient.

Example 1.3: For a simple illustration of this, consider a distributed system consisting of three components C_1, C_2, and C_3 running on three computers, illustrated in Figure 1.1. C_1 sends a message to each of the other two to initiate the computation. In response, the other two perform some deterministic computation and output a sequence of results (a sentence).

Suppose C_2 always outputs AB and C_3 always outputs CD. Is the overall system deterministic?

As described, this system has no input, so, to be deterministic it should have exactly one "behavior." If by "behavior" we mean a single output sentence, as Milner does, then we are forced to combine the output sentences produced by C_2 and C_3. How should we combine them? If we define the "observer" to be some entity that simply observes the symbols A, B, C, and D as they are produced, then that observer will see some arbitrary interleaving of the two sentences AB and CD. There are six such interleavings, as shown at the upper right in the figure. Hence, the model is nondeterministic.

Arguably, the nondeterminism that arises here is a side effect of our insistence that an observer can only see a single output sequence. Indeed, Milner found this sort of nondeterminism unsatisfying and introduced a notion that he called **confluence**. In his formalism, as long C_2 and C_3 cannot interfere with each other's ability to produce their output, this system is deemed confluent, presented as a useful replacement for the concept of determinism.

There is another solution, however, which does not require replacing the notion of determinism. Instead, we can change what we mean by behavior by changing what an observer sees. If, instead of a single output sequence, the behavior is defined to be a *pair* of output sequences, then the model immediately becomes deterministic, as shown at the lower right in the figure. The one and only behavior is (AB,CD), a pair of sentences, each containing a sequence of two symbols.

Redefining behavior may seem like a sleight of hand, a trick. It is not. *Any* definition of behavior depends on a notion of an **observer**, and fundamentally, for any system, different observers see different things. Let me make this crystal clear with a trivial example.

Example 1.4: Consider the following C program:

```
1 int main(int argc, char* argv[]) {
2     printf("Hello World.\n");
3 }
```

Is this program deterministic? If the observer is a human sitting at a computer screen, then for this program to be deterministic, that observer should

be able to see only exactly one possible observation from running this program. Is that the case? What color will the characters "Hello World" be rendered in? How long will it take before the observer sees "Hello World"? Neither of these observable properties are specified by the program, so, if we include these properties in the notion of behavior, then the program is nondeterministic. Almost certainly, however, this is not what we intended. If we carefully define "observer" and restrict that observer to observing the sequence of symbols produced by the program, then the program becomes deterministic.

Returning to our example with C_1, C_2, and C_3, a different notion of deterministic programs was introduced by Gilles Kahn in 1974, a class of models that are now called Kahn process networks (KPN) (Kahn, 1974). These are closely related to dataflow models (Lee and Parks, 1995; Lee and Matsikoudis, 2009), which share with them a different notion of observer from Milner's that leads to different conclusions about determinism. A KPN is a network of processes that send messages to each other along defined channels, where each channel is assumed to preserve the order of the messages and deliver them reliably (like TCP). Kahn gave an elegant construction using the mathematics of partial orders to give conditions on the processes such that the overall behavior of the program is deterministic (see Chapter 4 and Kahn (1974)). Kahn and MacQueen later showed that that a simple constraint on the processes in the network, blocking reads, is sufficient to ensure determinism (Kahn and MacQueen, 1977).

The definition of behavior here, however, is not the sentences of Milner. Kahn defined behavior to be a collection of (possibly infinite) sequences of messages, each recording the messages traversed on one channel. "Behavior" in a KPN, therefore, is a tuple of possibly infinite sequences. Under the KPN model, C_1, C_2, and C_3 together form a deterministic system. No notion of confluence is needed.

Kahn networks also do not require any notion of state. Each process can be usefully modeled as a state machine, but there is no need to ever talk about a global state, some combination of the states of all the processes at some instant in time. The approach Kahn took towards defining the semantics of programs without appealing to a notion of state can be generalized to many other kinds of concurrent systems (Lee, 2006a), as done for example in the tagged signal model (Lee and Sangiovanni-Vincentelli, 1998). This is explained in Chapter 4.

An issue with Kahn networks is that coordinated decision making in a distributed system can become difficult (Lohstroh and Lee, 2019). There is no notion of the state of the system, so writing code like "if the state of the system is X do Y" becomes challenging.

A number of alternative concurrency models have emerged that preserve determinism and reintroduce a semantic notion of state without having to appeal to confluence. One such alternative concurrency model is embodied in synchronous-reactive languages (Benveniste and Berry, 1991), explained in Chapter 6. These languages introduce the notion of a global "clock" that ticks discretely, much like that found in synchronous digital logic design. At each tick of this now conceptual (rather than physical) clock, many computations are performed, possibly in parallel, until the entire system settles to a well-defined state. With certain constraints on the component computations and on their scheduling (Edwards and Lee, 2003), the resulting state is a unique function of the inputs. The resulting model can be conceptualized as lying somewhere between Milner's transition systems and Kahn's asynchronous process networks. During the computation at a tick, the behavior is more like a Kahn network, where data precedences constrain the order in which things happen, but no notion of a single global state trajectory is needed and computations can proceed in parallel and asynchronously. At the conclusion of a tick, things settle to a well-defined state, enabling a higher-level state-trajectory model that treats all the computation at a tick as a single atomic state transition. These models are deterministic as long as the observer is constrained to observe the state only at the conclusion of each tick. The execution of synchronous-reactive languages may be thought of as "punctuated chaos," where periods of chaotic, parallel, asynchronous computation are marked by isolated points of stable, well-defined state.

1.2.3 Time

Although synchronous-reactive languages have "clocks," they do not really have a notion of time. For cyber-physical systems (CPSs), which combine computation and networking with physical components (Lee, 2008), some notion of time becomes essential (Lee, 2009). It is not necessary (nor is it physically possible) to insist on the Newtonian notion, where time is a continuum with well-defined instants t that are shared by all components in the system. Instead, leveraging the punctuated chaos of synchronous-reactive languages, we can assign a semantic measure

of time elapsed between ticks (von Hanxleden et al., 2017; Schulz-Rosengarten et al., 2018). This makes it possible for models to combine Newtonian models of physical components with computational models, thereby offering a rigorous approach to CPS design that does not sacrifice determinism (Cremona et al., 2017).

We can make an even bigger commitment to a notion of time by explicitly time-stamping events, as done in discrete-event systems (see Chapter **??**). In such models, the concept of a global "tick" is replaced by timestamped messages with the constraint that every component processes messages in timestamp order. With an additional constraint that messages with identical timestamps be processed in a well-defined order, the model becomes deterministic. This is the principle behind the recently introduced reactor model (Lohstroh et al., 2019) as realized in the Lingua Franca language (Lohstroh et al., 2020). Semantically, such discrete-event models can be viewed as a generalization of synchronous-reactive models (Lee and Zheng, 2007) and have even been called "sparse synchronous" models by Edwards and Hui (2020). Reactors are described in Chapter 7.

1.2.4 Observers

Another major issue that emerges from Milner's calculus is the tension between state transitions and output sentences. Which are observed? It is easy to construct an automaton that makes nondeterministic state transitions and yet produces a deterministic output in response to inputs. Should this automaton be deemed deterministic or not? This tension has prompted some researchers to attempt to codify this distinction, sometimes using two distinct words, "determinate" and "deterministic"; see for example von Hanxleden et al. (2014). The word "determinate" is meant to capture the idea that observable outputs are uniquely defined by the inputs, whereas the word "deterministic" attempts to capture the idea that there is a unique way in which the outputs are determined. Using this distinction, our constructed automaton (the one that makes nondeterministic state transitions and yet produces a deterministic output) is determinate but not deterministic.[3]

[3]A similar distinction is given by Wisniewski et al. (2020), who define "strong" and "weak" determinism in terms of Petri nets augmented with inputs and outputs. They make a distinction between "stable markings" (ones where no transitions are enabled, given the inputs) and "unstable markings" (which can be viewed as transitory markings towards stable markings). A "weakly deterministic" Petri net is one where for each possible stable marking and input, there is exactly

However, this distinction is specious. An automaton is a model, not an implementation. An implementation might be realized by a computer program, in which case, the automaton is a model of the behavior of that program. The program itself is a model of the computations to be performed by one or more instruction set architectures (ISAs) communicating over some network fabric using some protocols. The ISA and the network protocols are themselves models of a physical system with electrons sloshing around. At which level should we determine whether the system is deterministic? How many words for determinism will we need to cover all these levels of models? If we are clear about what we mean by an observer, then no such distinction is needed and one word is sufficient. Determinism becomes a property of the combination of the model and the observer.

Another context in which this tension comes to the foreground is with Alonzo Church's lambda calculus (Church, 1932), a model of computation in which "lambda expressions" are subjected to syntactic rewriting following a set of rules. The Church-Rosser theorem shows if such an expression can be reduced to the point where none of the rewriting rules can be applied any more, then the same final expression results regardless of the order in which the rewriting rules are applied (Church and Rosser, 1936). This model of computation can be construed as nondeterministic if the observer can see the intermediate expressions or as deterministic if only the final expression is visible.

The notion of an observer, it turns out, has its own subtleties. Should an observer be passive and objective, or can the observer interact with the system? This distinction turns out to be important. In computer security, for example, any discussion of the security of the system requires a threat model that is explicit about the capabilities of an attacker. It is important whether the attacker can interact with the system or is restricted to passively observing it.

Milner's own definition of determinism exposes such subtleties in the concept of an observer. Milner's definition of determinism depends on a relation between automata that Milner called **bisimulation**. It is beyond the scope of this chapter to explain bisimulation, but suffice it to say that this concept depends on an observer that is not just passive and objective, but rather can interact with the system being

one successor stable marking, and a "strongly deterministic" Petri net is one where for any marking and input, there is exactly one successor marking. Another similar distinction is given by Khomenko et al. (2008), who define "output-determinacy" as a relaxation of determinacy.

observed.[4] In 1980, David Park found a gap in Milner's prior and simpler notion of **simulation**, in which a passive and objective observer automaton emulates the behavior of an observed automaton (Park, 1980). Park noticed that even if two automata simulate each other, they can exhibit significant differences in behavior. These differences are not observable by any passive, objective observer, but if the observer can interact with the observed automaton, providing inputs that depend on its observations, then the differences become visible. Park's observation led Milner to develop the notion of bisimulation (and the closely related notion of **observational equivalence**), an interactive form of simulation that ensures that two automata are indistinguishable even through interaction. He then based his notion of determinism on bisimulation (Milner, 1989).[5]

1.3 Determinism vs. Predictability

Determinism does not imply predictability. For a model to be predictable, we must be able to anticipate its behavior by examining the model rather than by just watching what it does. Once again, we must be careful to define behavior. No computer program is predictable, for example, if behavior includes generating heat. The computer program alone is not sufficient to anticipate how much heat will be generated by executing the program.

Turing machines are deterministic models. The "input" to a Turing machine is a binary bit sequence, which Alan Turing described as the initial sequence of marks on a tape. The "behavior" can be defined to be a final bit sequence, the final sequence of marks on the tape when the machine halts, or a special result, often called **bottom** and written with the symbol $\perp$, that indicates failure to halt. A Turing machine, therefore, is a model whose operation computes a function that maps an input bit sequence to either an output bit sequence or $\perp$.

A Turing machine is also deterministic to an observer that can observe the sequence of operations that lead to this final output, but this stronger form of determinism is less important to the notion of computation. In fact, one of the

[4] See my book (Lee, 2020, Chapter 12) for an in depth discussion of the distinction between observation and interaction. That chapter includes a gentle introduction to the concept of bisimulation.

[5] Sangiorgi (2009) gives a nice overview of the historical development of this idea. He notes that essentially the same concept of bisimulation had also been developed in the fields of philosophical logic and set theory.

most interesting things about the foundational theory of computation is that many different mechanisms, including some nondeterministic ones, can be used to compute the same set of functions. A human using pencil and paper and following well-defined rules, given enough paper and time, can compute the same set of functions. Your laptop, which uses low level operations that are quite different from those of a Turing machine and involve no tape, given enough time and memory, can also compute the same set of functions. Church's lambda calculus, which has expressiveness equivalent to Turing machines, is deterministic in the weaker sense, where "behavior" is the final irreducible expression, but its mechanisms for finding that expression need not be deterministic. Reduction rules may be applied in any order. The low-level mechanisms are not as important as the function that is computed, so we will stick with the definition of "behavior" that restricts the observer to observe the final binary result.

The question now becomes, is the deterministic behavior of a Turing machine predictable? Alan Turing showed that it is not. He showed that there is no mechanism, no systematic procedure that can predict, for all Turing machines, whether an execution will halt for a particular input (Turing, 1936). Similarly, there is no such procedure for determining whether an execution uses a finite amount of memory, which translates to a finite amount of tape in the classic Turing machine. This result caries over to all the similarly expressive mechanisms (so called **Turing complete** mechanisms), including Church's lambda calculus, a human with paper and pencil, and your laptop. This result decisively decouples determinism from predictability. Determinism does not imply predictability.

Turing machines and lambda calculus underly modern imperative and functional programming languages, respectively. Programmers, therefore, face the possibility that they will not be able to predict whether their programs will terminate. Usually, however, a programmer needs assurances that the program will terminate (e.g. a program that converts a text file into a PDF file) or will not terminate (e.g. a web server or an operating system). Fortunately, most programs are predictable in this sense, even if, in theory, it is impossible for all programs to be predictable in this sense.

Many computer programs are unpredictable in a more informal sense. In this more informal sense, the question is, by examining the program but not executing it, can we anticipate its behavior? Some programs are designed to be unpredictable in exactly this sense. Pseudo-random number generators, for example, fall in this category. You cannot tell from looking at the program what numbers it will

generate. You must execute the program instead. Machine-learning algorithms turn out also to be unpredictable in this sense. Cellular automata, a class of deterministic computational machines that are also unpredictable in this sense, are capable of surprising and complex behaviors, so complex as to prompt some thinkers to conclude that they underly all of physics (Wolfram, 2002).

Unpredictable deterministic models also arise in Newtonian physics. Nonlinear differential equation models, such as those modeling the thermodynamics of weather, exhibit behaviors that are so unpredictable that they are called "chaotic" (Lorenz, 1963). A chaotic model is one where arbitrarily small perturbations in the inputs or the initial conditions have arbitrarily large effects in the future.

Deterministic chaotic systems are, *not even in principle*, predictable. Lewis and MacGregor (2006) propose a thought experiment involving two spheres colliding with each other within a contained space. They calculate the precision with which the initial conditions must be known in order to predict the behavior after a certain number of collisions. They then assume that the initial positions of the spheres are to be determined optically and derive the wavelength of light that will achieve the required precision. They then show that a single photon of such light would have "more energy than is currently posited for the entire universe in order to resolve the initial state of the system with precision sufficient to predict its behavior after just 35 collisions" (Lewis and MacGregor, 2006, p. 10-11).

The "system" that Lewis and MacGregor analyze is, however, a model, not a physical system. The question we should be asking, therefore, is whether the *model* is predictable. The model, given by Newton's laws with discrete collisions, admits no closed-form solution, and therefore would have to be solved numerically to predict its behavior. A similar analysis could be done to determine the arithmetic precision and computational load required to accurately predict the behavior after 35 collisions. I have not done this analysis, but I suspect it would show that this is equally impossible.

1.3.1 Murphy's Law and Faults

An argument that I hear frequently against deterministic models goes something like this: In the real world, things will go wrong. Nothing is really predictable. Even deterministic models will be violated in practice, so why bother with de-

terministic models? Why not, instead, assume everything is random and design your system to tolerate this randomness?

This argument, if carried too far, suggests we should not bother with the reliable in-order packet delivery of TCP, the linchpin of the Internet. We should not bother with **CRC** (**cyclic redundancy check**) to ensure the integrity of computer memories. We should not bother with synchronous digital logic design, and instead use circuits that may or may not produce expected results. But we do bother with all these things. Why?

Today, computer memories have replaced ledgers in finance and law. Digital signatures have become an acceptable way to finalize legal contracts. Stock trades execute without human intervention or paper records. None of these would be possible without deterministic models and our ability to build physical systems that are highly faithful to these models. Electronic circuits perform billions of arithmetic operations per second and go for years without errors.

I repeat the core principles because they are so important. Determinism is a property of models, not of physical systems. An *engineering* model is a specification of how a physical system *should* behave, not a model of how the physical system *does* behave (the latter is a scientific model). When you have a model that defines how a system *should* behave, then you get, for free, the notion of a **fault**. A fault is a behavior that deviates from the specification.

The existence of faults does not undermine the value of deterministic models. In fact, the very notion of a fault is strengthened by deterministic models because they define more clearly what behavior a physical system *should* have. Detecting faults, therefore, is easier. A CRC, for example, detects at least some violations of a simple deterministic model of a computer memory. This enables fault-tolerant design, where the system reacts in predictable ways to faults.

Every realization of an engineering model can exhibit faults. When we successfully build a physical system that reliably behaves like a model, it does so only under certain assumptions. No computer will correctly execute a program if it overheats, is crushed, or is submerged in salt water. The model is faithfully emulated only under the assumption that none of these things has happened.

Making the assumptions clear also has value. The Ptides model (Zhao et al., 2007) for deterministic distributed execution of discrete-event programs, for example, which is realized in Google Spanner (Corbett et al., 2012) and Lingua

Franca (Lohstroh et al., 2020), assumes a bound on clock synchronization error and a bound on network latency to achieve extremely efficient distributed and fault-tolerant coordination of program components (see Chapter 7). Violations of these assumptions will occur in practice, but, in a well-designed system, they will be rare. Moreover, such violations are detectable because they manifest as software components seeing events out of timestamp order. A deterministic model, together with clearly stated and quantified assumptions under which a physical realization emulates the model, enable efficient designs that can react in predictable ways to faults. Hence, despite Murphy's Law, deterministic models are useful, even in the face of unpredictable failures.

1.4 Determinism in physics

The question of whether the physical world is deterministic has been controversial for a long time. In the early 1800s, Pierre-Simon Laplace argued that if someone (a "great intellect," later known as "Laplace's demon") were to know the precise location and velocity of every particle in the universe, then the past and future locations and velocities for each particle would be completely determined and could be calculated from the laws of classical mechanics (Laplace, 1901). Is this true?

In 2008, David Wolpert proved that Laplace's demon cannot exist (Wolpert, 2008). No such calculation is possible. Wolpert's proof relies on the observation that such a demon, were it to exist, would have to exist in the very physical world that it predicts. This results in a self-referentiality that yields contradictions, not unlike Turing's undecidability and Gödel's incompleteness theorems.

But Laplace's demon is about *prediction*, not just determinism. We already know that determinism does not imply predictability. Even if prediction is known to be impossible, we cannot conclude that the world is nondeterministic. We will look at the question of whether Newtonian physics, the state-of-the-art when Laplace lived, is a deterministic model. You may be surprised by the answer.

More recently than Laplace, Karl Popper, high priest of scientific positivism, also insists on a deterministic universe:

> One sometimes hears it said that the movements of the planets obey strict laws, whilst the fall of a die is fortuitous, or subject to chance. In my view the difference lies in the fact that we have so far been able to predict the movement of the planets successfully, but not the individual results of throwing dice. In order to deduce predictions one needs laws and initial conditions; if no suitable laws are available or if the initial conditions cannot be ascertained, the scientific way of predicting breaks down. In throwing dice, what we lack is, clearly, sufficient knowledge of initial conditions. With sufficiently precise measurements of initial conditions it would be possible to make predictions in this case also. (Popper, 1959, p. 198)

This quotation reflects a conventional wisdom, which dictates that Newton's laws provide a deterministic model of the universe. Also conventional wisdom is that quantum physics dashed that determinism. Neither of these is strictly true. A concise summary of the ways that determinism in these physics models have been interpreted is given by Hoefer (2016). A more in-depth study is given by Earman's *Primer on Determinism* (Earman, 1986). Here, I will relate some of these interpretations to the above discussion of determinism in engineering models and give my own perspective on the subject.

1.4.1 Nondeterminism in Newtonian Physics

A (rather controversial) example of nondeterminism in Newton's laws is due to the philosopher of science John Norton (Norton, 2007).[6] Norton considers a point mass precariously balanced on top of a smooth frictionless dome. Norton shows that, without violating any of Newton's laws, the mass can spontaneously begin sliding down the side of the dome in an arbitrary direction at an arbitrary time without anything causing it to start sliding. His argument is carefully constructed and surprised me when I first heard it. I was sure his argument was wrong, but I finally concluded that my certainty was based on circular reasoning.

Newton's second law, given in equation (1.1), states that at any time instant, the force imposed on an object equals its mass times its acceleration. If there is no force, the acceleration must be zero. If the acceleration is zero, then the velocity

[6] Another example with similar properties is given by Dhar (1993).

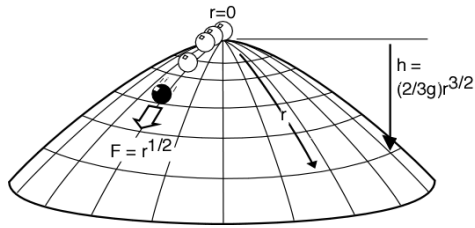


Figure 1.2: Norton's dome (from http://www.pitt.edu/~jdnorton/Goodies/Dome/index.html). From http://www.pitt.edu/~jdnorton/Goodies/Dome/index.html.

is not changing. Hence, it would seem that if the mass is not moving, balanced on the top of the dome, and no force is applied, then it should remain still, with velocity equal to zero. But Norton points out that it is possible for the mass to start sliding down the dome at any arbitrary time T without violating this law and without any force initiating the slide. At the instant T, the mass will have velocity zero and acceleration zero, so it is not moving. But at any time greater than T, say at $T+\epsilon$, no matter how small ϵ is, the mass may be no longer centered on the top of the dome. It will now be sitting on a slope, which means that gravity will exert a nonzero force in the downhill direction, and the mass will have a nonzero acceleration.

Specifically, Norton proposes a dome shape where the dome drops by a distance $h = (2/3g)r^{3/2}$, where r is the distance along the surface of the dome from the center of the dome and g is the force of gravity (see Figure 1.2).[7] There is nothing particularly special about this shape; it just makes the math work out simply. With this choice, the force on the mass tangent to the dome, as a function of the distance r from the center of the dome, is $F(r) = \sqrt{r}$. With this function, equation (1.1) admits many solutions. In particular, if we assume unit mass $m = 1$, then the following is a solution for any T:

$$r(t) = \begin{cases} (1/144)(t-T)^4 & t \geq T \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

[7] Gareth Davies points out in his blog that this is a rather odd specification of the dome shape and that the extent of the dome has to be limited to $r \leq g^2$. See https://blog.gruffdavies.com/2017/12/24/newtonian-physics-is-deterministic-sorry-norton/.

At the instant $t = T$, the mass is not moving, the net force on the mass is zero, and the mass is not accelerating. At any time larger than T, the mass is moving, the net force is not zero, and the mass is accelerating down the dome. T can have any value without violating Newton's second law. Equation (1.1) holds at every instant t.

There are many subtleties around Norton's example. First, it may be helpful to realize that there is no *first* instant at which the mass accelerates. Instead, the time $t = T$ is the *last* instant at which the mass is *not* accelerating. At all instants greater than T, there is a nonzero net downward force, gravity on a slope, so the mass accelerates. Zinkernagel (2010) claims that this property is the key flaw that would allow us to declare Norton's dome to not be a "Newtonian system." He says the force lacks a "first cause." But requiring a first cause would force us to reject many other innocuous systems that do not exhibit nondeterminism, including many reasonable models where force appears gradually (Fletcher, 2012).

Malament (2008) offers a fascinating analysis of the mathematics behind Norton's dome. First he addresses a preconception that was part of what made me initially skeptical about Norton's claim. I had always assumed that Newton's second law (1.1) would have a unique solution for any input force function F that could be generated by a reasonable Newtonian system. Norton's system seems reasonable in this sense because the force is just Newtonian gravity and the dome is a reasonably simple geometric shape. But Malament points out that uniqueness is not guaranteed if F is not continuously differentiable. Norton's function is not continuously differentiable at $r = 0$, and (1.1) admits many solutions. We could restrict "Newtonian Systems" to include only forces that are continuously differentiable.[8] This is not satisfactory to me, however. We would have to rule out a large number of shapes, including shapes that do not lead to nondeterminism, such as a table with an edge where a mass slides off the edge.

Another possible objection to Norton's dome is that such a shape could never be constructed perfectly. This argument is specious, however, because *every* shape that we can describe mathematically will have flaws when constructed in the physical world. This objection would effectively eviscerate Newtonian physics.

Yet another approach to rejecting Norton's claim is to impose constraints on the *solutions* to (1.1) rather than the force function. For example, Davies suggests that higher-order derivatives of the solution need to exist and be continuous for the

[8] A weaker but sufficient constraint would be to require the force to be locally Lipschitz.

solution to be a reasonable model of physical behavior.[9] The family of solutions given by (1.2) does not meet this requirement. However, this too is not satisfactory to me. Again, it would rule out a mass sliding off the edge of a table. Moreover, this amounts to saying something like, "the model is valid if we choose from among its behaviors the one that seems reasonable."

It seems that to regain determinism, we need to augment Newtonian physics with additional axioms that are not derivable from the core concepts. Newton could have given us a fourth law of motion going something like this: "a mass can only have one possible motion that conforms with the previous three laws." This would assume away nondeterminism in order to obtain a deterministic model of the physical world.

Fletcher (2012) points out, however, even this restriction has several possible versions, and any choice between these seems arbitrary. For example, we could declare Norton's dome to not be a "Newtonian system" altogether, or we could declare it to not be a "Newtonian system" only when the mass starts or ends at the peak of the dome. If the initial position of the mass is somewhere else on the dome, Newton's second law gives us a deterministic model. What if it starts somewhere else on the dome with some momentum towards the peak and crosses the peak? At the moment of crossing, does the system suddenly and instantaneously cease to be a Newtonian system?

Are Newton's other laws violated when the mass spontaneously starts sliding? Newton's first law states that an object will remain at rest or in uniform motion in a straight line unless acted upon by an external force. It may seem that having the mass spontaneously start to move violates this law, but actually it does not, at least under one reasonable interpretation of this law. Since an external force may vary in time, this first law needs also to be interpreted as a statement that holds at each instant of time. Under this interpretation, the first law becomes a special case of the second law where $F(t) = 0$.

There is another interpretation of Newton's first law, however, that restores determinism for this example, but this interpretation requires more than Newtonian physics and is ultimately based on circular reasoning. The first law can be interpreted to mean that there can be no *uncaused* changes in momentum, where the cause is some external effect. If the mass slides off the dome as I have described,

[9] See Gareth Davies' blog: https://blog.gruffdavies.com/2017/12/24/newtonian-physics-is-deterministic-sorry-norton/

then there must be some such external effect causing this. That effect must be something other than a Newtonian force, however, because any force would lead to a trajectory different from (1.2), which satisfies the second law. So what could that required external non-force effect be? The force of will of a conscious mind? The force of God? There is nothing in Newtonian physics that qualifies.

One could take this absence of a model for such an external non-force as an argument that the mass will not move. But the *absence* of a model cannot be construed as evidence. According to Norton, there is no *need* for such an external non-force for the mass to slide down the side of the dome, so there is no need for a model for this external non-force. Newton's laws are still satisfied. It is equally valid to demand a model for whatever keeps the mass perched on the dome. What non-force is that?

It will still disturb many readers that the mass can start moving with no provocation. Newton's laws are *also* satisfied by a mass that behaves itself and remains quietly perched at the top of the dome for all eternity. Isn't it reasonable to assume it will do that? An empirical approach would actually find this interpretation *unreasonable*, since any practical realization of Norton's dome will result in the mass sliding off the peak. The apparent reasonableness of this interpretation is due to a distaste for uncaused action, i.e., a distaste for nondeterminism. It is not due to empirical evidence and indeed flies in the face of empirical evidence. Hence, the argument that the mass will remain at the top of the dome is circular. It is not supported by the mathematics of Newton's laws alone and instead depends on the assumption that nothing happens without provocation. In other words, it concludes determinism based on an assumption of determinism.

The presupposition that every behavior has a cause is a difficult one to give up. In 1913, Bertrand Russell challenged the scientific world to give it up:

> All philosophers, of every school, imagine that causation is one of the fundamental axioms or postulates of science, yet, oddly enough, in advanced sciences such as gravitational astronomy, the word "cause" never occurs ... The law of causality, I believe, like much that passes muster among philosophers, is a relic of a bygone age, surviving, like the monarchy, only because it is erroneously supposed to do no harm. (Russell, 1913)

Objective discussion of causality is difficult because the notion of causality lurks in every aspect of natural language.[10] In my recent book, *The Coevolution* (Lee, 2020), I examine this presupposition of causality in much more depth, leveraging the arguments of Judea Pearl in (Pearl and Mackenzie, 2018) (who argues that reasoning about causality requires subjective involvement) and evolutionary biologists (who argue that the notion of causality may have arisen because of its evolutionary survival value rather than because it is a fact about the world). But for our purposes here, it is sufficient to simply observe that Newton's laws do not imply causality.

What about Newton's third law, which states that every action has an equal and opposite reaction? When gravity exerts a force on a mass, causing the mass to fall, the mass exerts an equal and opposite force on the earth, causing the earth to rise. Since the mass also gains momentum is some lateral direction, the earth must acquire an equal and opposite lateral momentum. The mass of the earth, however, is so much larger than the masses in Norton's example that, to the earth, the force exerted on it is negligible. At all times $t > T$, the nonzero net force downward and laterally on the mass will be balanced by an equal and opposite nonzero net force pulling the earth up and laterally. The effect of that force will not be measurable, but it is there, so the third law is also not violated.

Newtonian physics is often assumed to be time reversible, but reversing time on Norton's example has curious effects. Consider the scenario where the mass starts on the outskirts of the dome and we push it with just enough force that it reaches the top of the dome and stops. If we push too hard, it will go over the top of the dome. If we don't push hard enough, it will not reach the top of the dome and will fall back down. But if we push it with the Goldilocks force, just right, it will stop at the top, stay there for an arbitrary amount of time, and then spontaneously slide down the dome again sometime in the future. As soon as the mass perches at the top, it's history is lost. Nothing in its state reveals when the mass arrived at the top, thereby foiling Laplace's demon.

For this scenario to work, Norton points out that the shape of the dome is important. If the dome is a perfect hemisphere, then with the Goldilocks force, it will take infinite time for the mass to reach the top of the dome. It will keep slowing down as it approaches the top, but it will never actually reach the top.

[10] A nice collection of essays on the deep influences of the notion of causality on language is found in Copley and Martin (2014).

But there are many other dome shapes where the mass reaches the top in finite time. Norton's example, where the dome drops by a distance $h = (2/3g)r^{3/2}$, is one such dome shape.

In conclusion, either Newtonian physics admits nondeterminism, or Newtonian physics needs to be augmented with additional axioms that preclude nondeterminism. As I will show in Section 1.5, however, any set of additional axioms that preclude nondeterminism will also preclude many useful models of physical phenomena.

1.4.2 Metastable States

It turns out that many systems are vulnerable to similarly uncaused action under Newtonian physics. When the mass is perched on the top of the dome, it is in a **metastable state**. A metastable state is marginally stable, where infinitesimal disruptions throw the system out its precarious state. Norton's mass is vulnerable to falling out of its metastable state with no provocation.

Electronic circuits, particularly ones at the boundary between the continuous physical world and the discrete world of digital electronics, have long been known to be vulnerable to lingering for unbounded periods of time in a metastable state (Marino, 1981; Kinniment, 2007; Mendler et al., 2012). So the problem is not limited to cute examples of masses on domes. It is a fundamental problem at the boundary between the discrete, computational world of computers and the continuous physical world.

A particular kind of metastable system is a **bistable** system, one that has exactly two stable states and can persist for an indeterminate period of time in a metastable state between the two stable states. A digital circuit can be thought of as a piece of electronics that wants to be in one of two states, and in principle, situations where it can linger indefinitely between these two states are unavoidable.

Designers of such circuits go to great lengths to make sure that the probability of lingering gets extremely small as time advances. Consequently, it is rare for circuits to persist in a metastable state for very long. Such situations can occur, however, but are difficult to reproduce in the lab. They have occasionally been implicated in otherwise inexplicable crashes of computers. Is the basic operation

of such circuits deterministic? If not, then any electronic implementation of a Turing machine, a deterministic model, is actually nondeterministic.

Bistable behavior has also been observed by biologists in nerve axons. Under certain circumstances, these axons can linger for an indefinite period of time before settling into one of two resting potentials (Ditlevsen and Samson, 2013). It is likely that such metastability plays a role in brain function.

Building a physical realization of Norton's dome is impossible because no physical dome is perfectly smooth and frictionless and no mass is a point mass. Nevertheless, it is common to deliberately build systems that come as close as possible to such metastability. Sensitive instruments depend on metastable states. The instrument hovers in its (nearly) metastable state until the slightest nudge from the thing being measured pushes it off in one direction or the other. The circuit that reads the contents of a DRAM, a commonly used computer memory, for example, makes use of such metastability to read a tiny stored charge.

1.4.3 Relativity

The special and general theories of relativity are mostly deterministic in a similar sense that Newtonian physics is mostly deterministic. There are only a few corner cases, specifically singularities, that result in many possible futures given a specific past. The event horizon of black holes present such singularities, but these are unobservable in the rest of spacetime, and hence arguably pose no problem to the determinism of the theory. A conceivable class of singularities, called "naked singularities," however, have no event horizon and become observable. In 1969, Sir Roger Penrose posited that such naked singularities do not exist in the universe, a principle called "cosmic censorship." If this principle holds, then general relativity adds no sources of nondeterminism over and above any already present in classical physics.

Nevertheless, there is one aspect in which relativity complicates the notion of a deterministic model of physics. Specifically, it undermines the notion of "the state of the system," as used by Milner and Newton, and consequently, we can no longer talk about determinism in terms of the evolution of the state of the system in time. In relativity, there is no "instant in time." This Newtonian concept is replaced by a Cauchy surface, a hypersurface in four-dimensional space time. But

the Cauchy surface is different for each observer, so two distinct observers can disagree about the state of the system.

Example 1.5: The difficulty can be illustrated with the example in Figure 1.3. Consider a distributed system with two physically separated subsystems, one of which makes an instantaneous transition from state A to B, and the other of which makes a transition from C to D. Under a Newtonian model of time, one of the following transition sequences must be the true one, in the sense that it is actually what happens when the system transitions from (A,C) to (B,D):

$$\begin{array}{lllll} (A,C) & \rightarrow & (B,C) & \rightarrow & (B,D) \\ (A,C) & \rightarrow & (A,D) & \rightarrow & (B,D) \\ (A,C) & \rightarrow & (B,D) & & \end{array}$$

However, according to relativity, it is possible for none of these to be "true" for all observers. The order of physically separated events may depend on the observer. It is not that two different observers just *see* things delayed in different ways, it is that the true order for the different observers is different. There is no ground truth, in the sense that we cannot make a statement like, "at time t, the system was in state (B,C)." That statement can be true for one observer and false for another.

This problematic notion of time (and hence state) follows easily from the fact that the speed of light is the same for all observers.

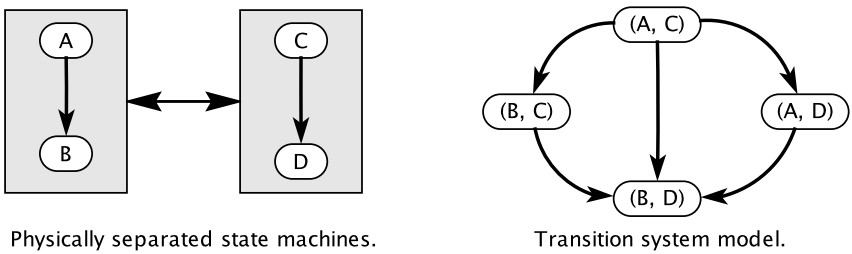


Figure 1.3: Model of a distributed system that relies on a notion of "system state."

Example 1.6: Consider a simple thought experiment, a variant of the famous Einstein train example. Suppose that Randy is seated at the center of a rapidly moving train car, and Jane is standing on a platform at a station where the train does not stop. Suppose that just at the instant that Randy passes Jane, he hits a button that emits a brief pulse of bright light. In Randy's frame of reference, the light will hit the front and back ends of the train car at the same time. However, in Jane's frame of reference, the light will hit the back of the train car before it hits the front. In Randy's frame of reference, the light has to travel the same distance in either direction, and since the speed of light is constant, it will strike both ends of the car simultaneously. But in Jane's frame of reference, the distance the light has to travel to the back of the train is less because the back of the train is moving toward where the light was emitted while the light is traveling toward the back of the train. Hence, the two events of light striking the ends of the train are simultaneous for Randy but not for Jane.

Note this example is not just a cute toy that is only realizable if you have trains that can travel close to the speed of light. The phenomenon is called the Sagnac effect, and it is used in ring laser gyroscopes, a key component of many inertial guidance systems, including many used in commercial aircraft. The Sagnac effect also has to be taken account in the design of GPS because of the rotation of the earth.

Instead of an evolving state in time, a relativistic system is a collection of events where, given a pair of events, one may precede the other. This is analogous to the contrast between Milner's transition systems (analogous to classical physics) and Kahn's precedence order constraints (analogous to relativity). Of course, if one event *causes* another, then relativity is careful to ensure that all observers see the one before the other, so nothing magical happens. To ensure this consistency, relativity posits that the effects of any event cannot propagate through space faster than the speed of light.

Interestingly, a similar dichotomy exists in the philosophical study of time, dubbed an "A-Series" (classical) model of time or a "B-Series" model by Gale (1966). An A-Series model of time is built on the tensed notions of past, present, and future, while a B-Series model of time is built on a partial order, a "precedes" relation between events. For example, the statement "I will complete this chapter today" is an A-Series statement and will have a different meaning if uttered tomorrow. In

contrast, "Completion of this chapter precedes the 2026 New Year" is a tenseless B-Series statement. It has the same meaning whenever it is uttered. Relativity requires a B-Series model of time.

The illusion of state is a powerful illusion, however, because no single observer can observe a system to be in two states at once. For example, no observer of the example of Figure 1.3 will see both (B, C) and (A, D). This becomes important when considering quantum models, which I do next.

1.4.4 Quantum Physics

It is common to assume that quantum physics immediately undermines any notion of the world being deterministic, but the story is more subtle. There are many interpretations, not all of which lead to nondeterminism.

First, under quantum mechanics, the evolution of the particle's wave function is deterministic, following the Schrödinger equation. If we redefine "behavior" to be the evolution of the wave function in time, rather than position and momentum, then the model is deterministic. The fact that the Schrödinger equation is deterministic prompted Stephen Hawking to colorfully proclaim that determinism can be salvaged:

> At first, it seemed that these hopes for a complete determinism would be dashed by the discovery early in the 20th century that events like the decay of radioactive atoms seemed to take place at random. It was as if God was playing dice, in Einstein's phrase. But science snatched victory from the jaws of defeat by moving the goal posts and redefining what is meant by a complete knowledge of the universe. ... In quantum theory, it turns out one doesn't need to know both the positions and the velocities [of the particles]. (Hawking, 2002)

It is enough to know how the wave function evolves in time.

But the wave function is not directly observable. It cannot be measured by instruments. If deterministic behavior has no observer, does it lose its value? If, on the other hand, we define "behavior" in terms of measurable quantities, such as position and momentum, then the quantum model becomes nondeterministic,

albeit in a subtle way. The wave function is commonly interpreted as giving the probabilities of the various possible inherently random measurement outcomes.

However, what could "probability" mean here? It turns out that it cannot be the usual notion of probability that you probably learned in an undergraduate class. Quantum probability is not an indicator of the relative frequency of various outcomes when performing repeated experiments. The so-called "no cloning theorem" in quantum mechanics says that such repeated experiments are impossible! The frequentist interpretation of probability has to be replaced by the Bayesian notion, which gives a different meaning to the word "probability" (see my book (Lee, 2017, Chapter 11) for a discussion of these two interpretations). In the Bayesian interpretation, a probability is a measure of what is unknown. Quantum probability is even stronger; it is a measure of what is *unknowable*.

It turns out that the mechanics of probability theory does not depend much on which interpretation you adopt. The math is the same. Interestingly, much of modern probability theory was developed by Laplace, who was convinced the world was completely deterministic. There is no contradiction here, however, because Laplace was a Bayesian. His probabilities measure uncertainty, or lack of knowledge, not intrinsic randomness.

Even if we take "behavior" to be the evolution of the wave function in time, despite our inability to measure it, subtleties remain. The Schrödinger equation gives the evolution of the wave function in classical Newtonian time, not relativistic time. Hence, even if we could measure the wave function, it would become possible for two distinct observers to see two different wave function evolutions for the same physical system! If the system includes metastable components, these distinct evolutions could be quite different indeed.

The role of an observer has always been important in quantum physics, albeit not in this relativistic sense. In what is now called the Copenhagen interpretation, originally proposed in the years 1925 to 1927 by Niels Bohr and Werner Heisenberg, the state of a system continues to be defined by probabilities until an external observer observes the state, and only at that point do the probabilities influence the outcome. Prior to being observed, all possible outcomes represented by the probabilities continue to remain possible. This requires an "observer" who is somehow separate from the system and measures the position of the particle. A "collapse" of the wave function occurs at the instant that a "measurement" is made, converting possibilities into certitude. This interpreta-

tion leaves unspecified what a "measurement" or "observation" is and typically puts the measurement apparatus outside the domain of quantum mechanics. This has led to sometimes bizarre interpretations, for example that conscious minds play a central role, which presumably means that the physical world didn't exist or played by different rules before conscious minds formed.

The Copenhagen interpretation, however, is firmly rooted in an insistence on imposing a notion of "state" on the system. The outcome of a measurement is taken as a definitive answer, a fact about the system. A more fully quantum interpretation would continue to model the system, even after measurement, using a wave function.

In the 1950s, the physicist Hugh Everett III dispensed with the distinct observer, instead bundling observer and observed under a single wave function that evolves deterministically under the Schrödinger equation. The measurement apparatus and measured system entangle and evolve together in a single wave function. This view is a straightforward, simple, and direct interpretation of quantum mechanics until one insists on the same sort of certitude that one gets from the collapse of the wave function posited by the Copenhagen interpretation. With this insistence, the theory gets rather extravagant.

Consider a particle, say, an electron, moving through space. In classical physics, at each instant t, it has a definite position and momentum. In quantum physics, it has a wave function. If we measure its position, say by putting a phosphorescent screen in its way, then the position of the electron will be revealed by a flash on the screen. Under the Copenhagen interpretation, at the time the electron hits the screen, its position is drawn randomly from a probability density function that puts weights equal to their likelihood on all regions of possible positions. Under Everett's interpretation, the photon, the screen, and the human observer become entangled and a single wave function covering all of them and continues to evolve deterministically.

But, under Everett's interpretation, where is the electron? Physicists seem to continue to insist that it must be somewhere, that it has "state," which leads to the most bizarre part of Everett's thesis. In his thesis, the electron is everywhere that its wave function permits it to be, but now in an uncountably infinite number of split-off universes. In each such universe, the electron is at one of the infinitely many possible positions. Each of these universes is somehow weighted by the

probability dictated by the wave function.[11] The insistence on state leads to uncountably many universes being spawned anytime there is an interaction between components of a system.

Because of the proliferation of universes, Everett's thesis is often called the "many worlds" interpretation of quantum mechanics. It means that every time there is an interaction between an observer and a subject, the universe splits. Depending on what is being measured, it could split in two or into an infinite number of possibilities. Since interactions are occurring all the time, a wildly extravagant proliferation of universes occurs at every instant in time. The model is deterministic, but the split universes cannot interact with one another, and hence, from the perspective of any entity in any one of those universes, the outcome of the experiment appears to be random.

Many physicists consider this consequence of Everett's thesis to be a *reductio ad absurdum* proof of the invalidity of the thesis. But others, including many highly respected physicists, accept the thesis as the best available explanation of quantum mechanics (see Becker (2018) for a very readable overview of the alternative explanations).

There is, however, a simpler interpretation that is consistent with Everett's core idea that there is a single wave function that continues to evolve deterministically. The simpler interpretation rejects the notion that there is a ground truth about the state of a system at an instant in time. The electron is never at a definitive location. Despite seeing the flash of light, even a human observer is mistaken to interpret this as definitive evidence of a true location. No experimental apparatus is perfect, and it takes a great deal of effort to build an experimental apparatus that delivers even reasonable confidence, much less certainty. Moreover, no human perceptual system, which sees the flash, is perfect, and no brain has perfect memory. It is simply wrong to conclude that the flash is an indicator of a definitive truth. It is no more than compelling evidence of a high likelihood. This simpler interpretation has no need for a proliferation of universes. In exchange, it sacrifices certainty. Specifically, it sacrifices the notion of "state."

[11] This is a rather mysterious part of the many worlds interpretation because the wave function defines a probability *density* for position, not a probability, so each of the uncountably many universes would have to have weight equal to zero. This stretches the notion of "existence" to the breaking point.

Einstein famously resisted aspects of quantum physics, a theory he helped to build, insisting that the theory was incomplete. He spent much of his later career searching for the "hidden variables," hitherto unknown aspects of the state of a physical system that would reconcile quantum theory with the principles of locality implied by relativity (that no event's consequences can propagate at faster than the speed of light). The conflict posed by quantum theory is between locality and a principle that some physicists call "reality." Here, **reality** is the principle that objects have real properties that determine the outcome of measurements, i.e., that they have what I have been calling a "state." **Locality** is the principle that reality at one location in space is not influenced instantaneously by measurements performed at a distant location (or more specifically, that influences propagate no faster than the speed of light).

In 1964, John Stewart Bell proved that quantum theory is incompatible with the principles of reality and locality (Bell, 1964). That is, if quantum theory is a faithful scientific model of the physical world, something that is well verified experimentally, then one of the two principles must be false. Most physicists, including Bell himself, seem to prefer to sacrifice locality over reality.

However, this makes quantum theory incompatible with relativity, which requires locality. Relativity, however, does not require *reality* in this sense. In fact, as I argued in the previous section, relativity is inconsistent with reality in this sense. Relativity shows that a physical system, at least one that is spread out over space, cannot have "state" that, at an "instant" (a boundary between the past and the future), determines all of its properties. Any notion of state is dependent on the observer. The so-called "relational interpretation" of quantum mechanics, first attributed to the theoretical physicist Carlo Rovelli, takes this perspective. The state of a quantum system is a relation between the observer and the system. There is no notion of state that is independent of an observer (see Rovelli (2017, 2018) for readable explanations of this approach).

In summary, the world is not predictable under either classical or modern physics. Is it deterministic? That depends on what model we use, and all deterministic models have logical problems. A deterministic Newtonian model requires a presupposition of determinism. A deterministic relativistic model requires giving up the notion of state. A deterministic quantum model requires either an extravagant proliferation of universes or a similar forsaking of the notion of state. I will argue next that these logical problems are inevitable. Any set of deterministic models

rich enough to say interesting things about the world is demonstrably incomplete, so these logical problems are unavoidable.

1.5 Incompleteness of Determinism

In 2016, I published a paper on the limits of modeling for cyber-physical systems in which I showed a sense in which determinism is incomplete (Lee, 2016) (see also my book (Lee, 2017, Chapter 10)). I review that result here. The short summary is that any set of deterministic models that is rich enough to encompass Newton's laws and also admits discrete behaviors does not contain its own limit points. Thus, any approach to modeling that relies on such a set of deterministic models has corner cases that exhibit nondeterminism. Nondeterminism, therefore, is inescapable in physical models.

1.5.1 Nondeterministic Collisions

Consider a model of the collisions of two billiard balls, as shown in Figure 1.4. Suppose that we model a collision as a discrete event, where the collision occurs

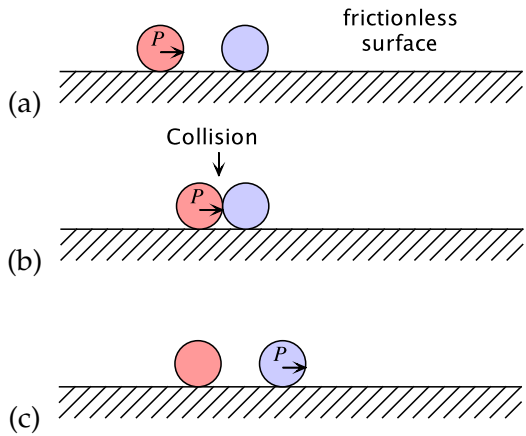


Figure 1.4: Collision of ideal billiard balls on a frictionless surface.

in an instant, having no duration in time. Assume that the balls are ideally elastic, meaning that no kinetic energy is lost when they collide. In this case, Newton's laws require that both energy and momentum be conserved; the total momentum and energy must be the same after the collision as before.

Let v_1 and v_1' be the velocity of the first ball before and after the collision, respectively. Let v_2 and v_2' similarly represent the velocity of the second ball before and after the collision. Conservation of momentum requires

$$m_1 v_1' + m_2 v_2' = m_1 v_1 + m_2 v_2, \tag{1.3}$$

where m_1 and m_2 are the masses of the two balls, respectively. Conservation of kinetic energy requires

$$\frac{m_1 (v_1')^2}{2} + \frac{m_2 (v_2')^2}{2} = \frac{m_1 (v_1)^2}{2} + \frac{m_2 (v_2)^2}{2}.$$

Assuming we know the starting speeds v_1 and v_2 and the masses, then we have two equations and two unknowns, v_1' and v_2'. This is a quadratic problem with two solutions.

Solution 1: Ignore the collision:

$$v_1' = v_1,\ v_2' = v_2.$$

Solution 2:

$$\begin{aligned} v_1' &= \frac{v_1(m_1 - m_2) + 2m_2 v_2}{m_1 + m_2} \\ v_2' &= \frac{v_2(m_2 - m_1) + 2m_1 v_1}{m_1 + m_2}. \end{aligned}$$

Note that solution 1 looks like tunneling, where the balls pass through one another without affecting each other, as a neutrino might. In solution 2, if $m_1 = m_2$, then the two masses simply exchange velocities, as suggested in Figure 1.4. If we rule out solution 1, then the model is deterministic.

Now consider a more elaborate scenario, shown in Figure 1.5. Two (ideal) billiard balls approach a third stationary ball from opposite sides on a frictionless surface and collide with the stationary ball simultaneously. How should they react? If we consider only conservation of momentum and energy, we still have two equations, but now there are now *three* unknowns.

At the time of the collision, there are *two* collisions. We could attempt to treat these as a pair of two-ball collisions, each of which has two possible outcomes. If we reject the tunneling solutions, then it would seem that only one outcome remains for each of the two collisions. But it is not obvious how to combine the two non-tunneling outcomes.

A first (naive) solution, using what is known as **Newton's hypothesis**, just superimposes the resulting momentums of the two non-tunneling outcomes. If the balls all have the same mass, then the left ball will transfer its momentum to the middle ball, the right ball will also transfer its momentum to the middle ball, and the equal and opposite momentums will cancel. All balls stop. Momentum is conserved, but not energy. This solution is shown at the top of Figure 1.6. Since there is no mechanism for energy dissipation in this model, this resolution is not satisfactory.

An alternative solution, using what is known in the literature as **Poisson's hypothesis**, introduces a form of superdense time (Lee, 2014). At the time of the collision, the kinetic energy of the balls is instantly converted to potential energy by compressing the middle ball. Then, without time elapsing, in a second microstep, the potential energy is reconverted to kinetic energy by the middle ball expanding. But how should this ball apportion the kinetic energy to the two outer balls? An intuitively appealing solution is shown at the bottom of Figure 1.6, which assumes the masses are equal and the two balls bounce off the center ball and end up with equal and opposite velocities. In general, however, then there are many solutions that conserve both momentum and energy! We seem to have no basis for picking one solution over another, so the model appears to become intrinsically nondeterministic.

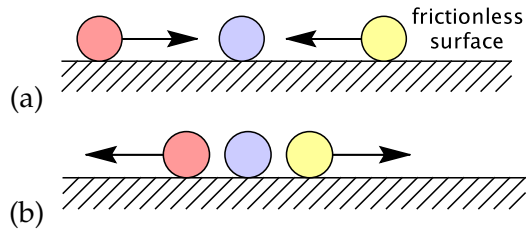


Figure 1.5: Collision of three billiard balls on a frictionless surface.

This thought experiment asked us to consider that the two collisions occur *simultaneously*. But, as we saw in the previous section, we have to choose an observer before simultaneity has any meaning. One observer may see the balls colliding simultaneously, another may see the left collision occurring first, and a third may see the right collision occurring first. It seems that any solution we come up with needs to have some sort of consistency across the experiences of these three observers.

Let's consider what happens when we treat the collisions as occurring in some order, but without any time elapsing between the collisions (this again relies on a superdense model of time and can be seen as a limiting case where an infinitesimal amount of time elapses between the collisions). As shown in Figure 1.7, when the collisions occur, we arbitrarily pick either the left collision or the right one, temporarily ignoring the other one. Rejecting the tunneling solution, we get a deterministic exchange of momentum. Without time elapsing, we find ourself in state (b) in the figure, at which point we must handle the second collision. Again we get a deterministic solution, leaving us in state (c). Again, without time elapsing, we handle a third collision, which leaves us in state (d). After time elapses, we find ourselves in state (e).

I first studied this problem while developing the hybrid system simulator in Ptolemy II (Cardoso et al., 2014). I experimented with different masses and with handling the collisions in different orders. I kept seeing plots like those shown in Figure 1.8. I was convinced that these plots should have been the same, that the order in which the software handled the collisions should not matter. Time did not elapse between collisions, so according to Newton's laws, the state of the system should not change. I spent weeks looking for the bug in the software before I finally realized that there was no bug in the software. More than one final state conserves both momentum and energy.

Inevitably, when I present this example, someone asks me how the "real world" behaves in this situation. This is a difficult question. One of the consequences of quantum mechanics is the Heisenberg uncertainty principle, which states that we cannot simultaneously know the position and momentum of an object to arbitrary precision. But discrete modeling of these collisions depends on modeling position and momentum precisely.

To many readers, it may seem odd to be invoking quantum mechanics on macroscale physics problems, where Newtonian mechanics usually works just fine. But

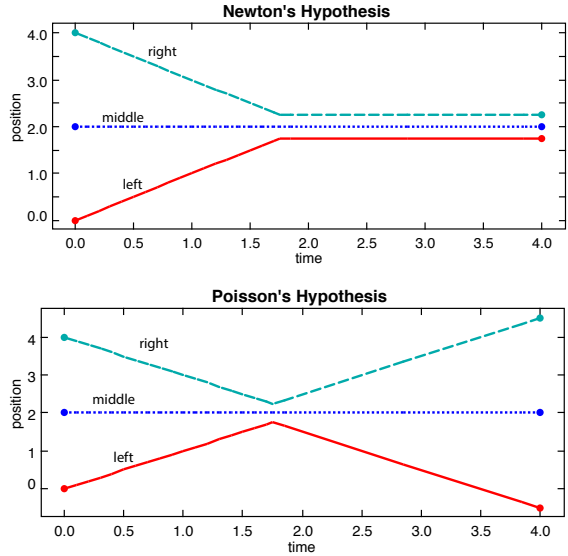


Figure 1.6: Newton's hypothesis vs. Poisson's hypothesis.

when the *order* of events affects the outcome, we find ourselves inevitably at quantum scales. The order of events can change with arbitrarily small differences in time or space.

Both relativity and quantum physics expose difficulties with this mix of time and space continuums with discrete events. Perhaps it is the mix that is problematic and we should instead reject either continuums or discreteness. Rejecting continuums amounts to accepting a hypothesis sometimes called "digital physics," a position that I challenge in my book *Plato and the Nerd* (Lee, 2017, Chapter 8).

Rejecting discreteness is a bit harder to debunk, but it has the same flavor as the assumptions that Malament and Fletcher point out are required to rule out Norton's dome as a valid Newtonian system (Malament, 2008; Fletcher, 2012). Discrete collisions are singularities, not unlike the singularity that makes Norton's dome result in forces that are not continuously differentiable. But ruling them out simply to preserve determinism seems rather arbitrary. More disturbingly, I have developed an example (a flyback diode circuit) (Lee, 2016) that shows that rejecting discreteness has the consequence of also having to reject causality.

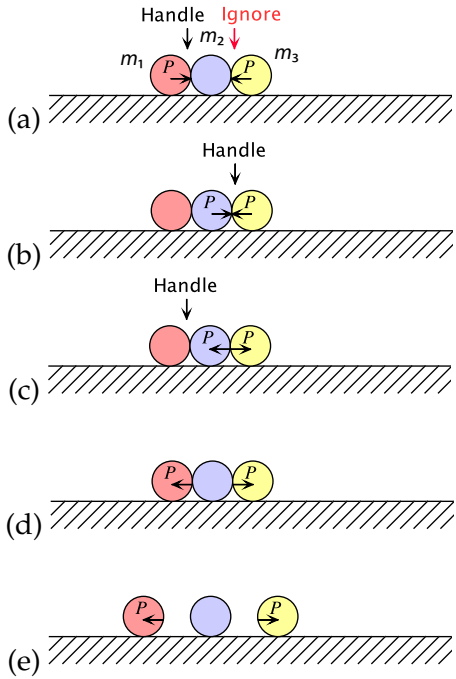


Figure 1.7: One of two orderings for handling collisions.

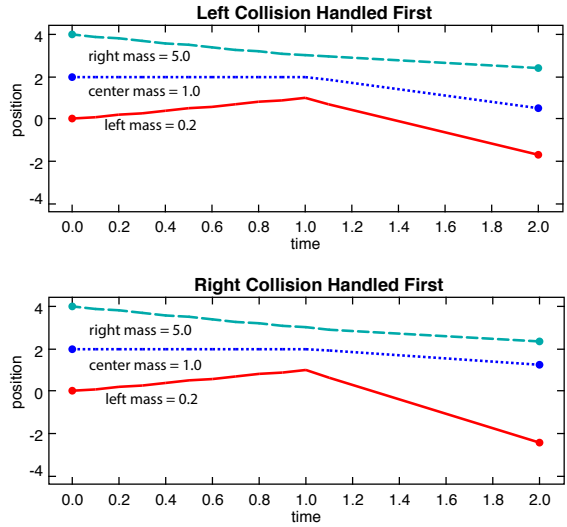


Figure 1.8: If the masses are different, the behavior depends on which collision is handled first.

Nevertheless, for the billiard balls example, rejecting discreteness is relatively easy. Using only classical mechanics, we can model the balls as stiff springs, yielding a model where there are no discontinuous changes in momentum. I have shown that this makes the model deterministic, but chaotic (Lee, 2016). Hence, arbitrarily small perturbations in initial conditions lead to arbitrarily large changes in final state. Combine this observation with quantum uncertainty, and the result is indistinguishable from nondeterminism in any practical sense.

1.5.2 Incompleteness

The three-billiard-ball example has the odd property that if the collisions are not simultaneous, then no matter how small the time difference between collisions is, the resulting behavior is deterministic. As the time between collisions approaches zero, as long as it remains nonzero, we have a deterministic model. But in the limit, the model becomes nondeterministic. This suggests that the set of deterministic models is incomplete, in that it does not contain its own limit points.

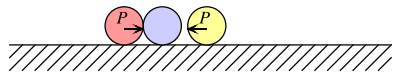


Figure 1.9: Non-simultaneous collisions.

I review here a construction from my previous paper (Lee, 2016) of such an incomplete set of deterministic models. Consider the set M of models describing one-dimensional motion of $N = 3$ ideal elastic balls subject to Newton's second law, where collisions are handled with impulsive forces, and all behaviors that conserve momentum and energy are allowed except for tunneling. Every model in M is closed, in that there are no external forces, so all behaviors are a consequence of the initial conditions. To keep things simple, the set M includes three balls with specific masses $m_1 = 0.2$, $m_2 = 1.0$, and $m_3 = 5$ (these are the masses that generate the behaviors shown in Figure 1.8). The units do not matter, as long as they are consistent. Assume initial positions $x_i(0) \in \mathbb{R}$, and initial velocities $v_i(0) \in \mathbb{R}$ given as follows:

$$\begin{aligned} x_1(0) &= -1 \\ v_1(0) &= 1 \\ x_2(0) &= 0 \\ v_2(0) &= 0 \\ x_3(0) &= 1 + \Delta \\ v_3(0) &= -1 \end{aligned}$$

where $\Delta \in \mathbb{R}$ is a real number. Δ is the only parameter that distinguishes models in M, but this one variable is sufficient to give us an uncountably infinite set of models. When $\Delta \neq 0$, the collisions will not be simultaneous, as illustrated in Figure 1.9.

Following Lee and Sangiovanni-Vincentelli (1998), we formally define a **model** as a set of behaviors. It is sufficient to consider the behavior of each model over the time interval $[0, 2]$ only. Let B be the set of all functions of the form $b\colon [0, 2] \to \mathbb{R}^3$. For a particular model $A \in M$ with parameter Δ_A, we say that $x \in B$ is a behavior of A if for all $t \in [0, 2]$, $x_i(t)$ is the position in our one-dimensional space of ball i at time t, for $i \in \{1, 2, 3\}$. These x_i are functions of time that satisfy the equations of

motion and conservation laws. In other words, a behavior of a model $A \in M$ is a function x giving the positions of the three balls as a function of time.

If a particular model $A \in M$ has exactly one behavior, then A is deterministic. As shown above, M contains only one nondeterministic model, let's call it $N \in M$, the one where $\Delta_N = 0$. We can now construct a sequence of deterministic models in D that should converge to N but does not. To do that, we need some notion of convergence.

Consider the subset $D \subset M$ of deterministic models. We can define a metric on D that allows us to talk about models being "close." Consider two models $A, A' \in D$. Because they are in D, these models are deterministic. Each admits one behavior, $x, x' \in B$, respectively. Because each has exactly one behavior, we can define a distance function d as follows,

$$d(A, A') = \frac{1}{2} \int_0^2 ||x(t) - x'(t)|| dt, \tag{1.4}$$

where $||x||$ is the L1 norm of a real vector x. This distance function measures the difference between two ball trajectories. It is easy to show that d is a metric, and hence (D, d) is a metric space.

Consider a sequence of models $A_i \in D$, $i \in \{1, 2, \cdots\}$ where

$$\Delta_{A_i} = 1/i^2.$$

As i gets larger, the time between collisions gets smaller, so in some sense, these models approximate the nondeterministic case N where $\Delta_N = 0$ ever more closely as i gets larger. It is easy to show that the sequence A_i is Cauchy, which means that for any $\epsilon > 0$, we can find a positive integer N such that for all positive integers $i, j > N$,

$$d(A_i, A_j) < \epsilon.$$

This means that as i and j get large, the trajectories of the balls in A_i get ever closer to the trajectories of the balls in A_j. With sufficiently large i and j, A_i becomes nearly indistinguishable from A_j. Nevertheless, this sequence has no limit in D.

The one nondeterministic model $N \in M$, where the collisions are exactly simultaneous, $\Delta_N = 0$, admits behaviors that are very distant from the behaviors of the models in the sequence A_i using the same metric. Some of the behaviors of N

will be close (in this metric space sense) to behaviors of a very different Cauchy sequence, $C_i \in D$, $i \in \{1, 2, \cdots\}$ where

$$\Delta_{C_i} = -1/i^2.$$

Here, the time between collisions is approaching zero from the other side.

A metric space that has Cauchy sequences that have no limit in the space is said to be "incomplete." The metric space (D, d) of deterministic models is incomplete. It does not contain all its limit points.

Every model in the sequence A_i is deterministic, and the models in the sequence get arbitrarily close to one another. Moreover, any set of deterministic models rich enough to encompass Newton's laws that allows discrete collisions must be rich enough to include this sequence of models, and therefore will be incomplete.

This has profound consequences that I explore in my book, *Plato and the Nerd* (Lee, 2017). Specifically, many people assume without proof that if some modeling technique can be shown to be able to arbitrarily closely approximate any model of interest, then this technique is "good enough" for all practical purposes. This example shows that this assumption is untrue unless the modeling technique defines a complete set of models. Our Cauchy sequence A_i arguably represents an arbitrarily close approximation to N, and yet it fails to capture a hugely important property of N, that it is nondeterministic.

1.6 Conclusion

I hope I have convinced you that determinism is a deep subject. In reality, I have only scratched the surface here. In my book, *The Coevolution*, I explore an even harder aspect, the connection between determinism and free will (Lee, 2020, Chapter 12). (Hint: observers become important again.) The focus there is to try to get a handle on the question of whether humans will ever build machines that have, in any sense, free will. More practically, the question is whether we humans will ever build machines that can and should be held accountable for their actions. That question is beyond the scope of this book.

This book focuses instead on the practical question of the usefulness and completeness of deterministic models. I have argued that they are extremely useful,

and that even though nondeterministic models also have their uses, those uses are disjoint. The choice of whether to use deterministic or nondeterministic models should be front-and-center in any engineering design effort, and yet it rarely is. Whether the models are deterministic or not typically depends on the choice of modeling and design frameworks, and many of these are nondeterministic by accident, not by intent.

I have shown that whether a model is deterministic or not depends on how one defines the inputs and behavior of the model. To define behavior, one has to define an observer. I compared and contrasted two classes of ways to do it, one based on the notion of "state" and another that more flexibly defines the observables. A state-based model further requires a restrictive model of time, one that is known to be an inaccurate model of physical time. Specifically, it requires an unambiguous simultaneity.

I examine determinism in models of the physical world. In what may surprise many readers, I show that Newtonian physics admits nondeterminism (unless one presupposes determinism) and that quantum physics may be interpreted as a deterministic model. Moreover, I show that both relativity and quantum mechanics undermine the notion of "state" and therefore require the more flexible ways of defining observables. Finally, I show that sufficiently rich sets of deterministic models are incomplete, Hence, no matter how much we value them, they will not solve all our problems.

In engineering practice, whether to use nondeterministic models becomes a central question. For scientific models, where it is incumbent on the model to match the thing being modeled, nondeterminism can be useful if it reflects inherent randomness or uncertainty about the thing being modeled. In this case, nondeterminism enables exploration of the range of possible behaviors. However, nondeterminism says nothing about the likelihood of any behavior, only about its possibility. Hence, a probabilistic model may be more useful than a nondeterministic one. A probabilistic model may usefully model a deterministic system (by interpreting probabilities as a measure of uncertainty) and may even, in some circumstances, be interpreted as a deterministic model (as we can do with quantum physics).

For engineering models, where it is incumbent on the thing being modeled to match the model, nondeterminism can be a useful abstraction mechanism, a way to get simpler models for analysis. It can also be useful for deferring design

decisions. However, it comes at a steep price. The model becomes less useful for testing, for evaluating the degree to which the thing being modeled matches the model. Nondeterminism that arises from sloppiness in the modeling framework or language, however, is rarely useful and should not be excused by the fact that the physical world is unpredictable in practice.

My own focus as an engineer is on cyber-physical systems, which combine the neat world of computation with the messy and unpredictable physical world. Deterministic models play an important role on both sides of this divide. For example, on the physical side, deterministic differential equation models can be useful descriptions of how a robot arm *should* behave, particularly if coupled with probabilistic models of how it may actually behave. On the cyber side, for distributed software, clear specifications of how the components *should* coordinate are useful, particularly if coupled with probabilistic models of network behavior that may compromise these specifications. In both cases, we are talking about a combination of engineering and scientific models.

2 Fundamental Tradeoffs

This chapter is adapted from **?**.

2.1 Concurrent, Parallel, and Distributed

2.2 The CAP Theorem

2.3 Consistency and Time

2.4 The CAL Theorem

2.5 Conclusion

3 Nondeterministic Concurrency

3.1 Threads

3.2 Actors

3.3 Publish and Subscribe

3.4 Remote Procedure Calls

3.5 Addressing the Challenges

Part II

Untimed Models

4 Process Networks

4.1 Kahn Process Networks

A **Kahn process network** (**KPN**) is a concurrent composition of sequential processes that communicate using a particular form of message passing. A key property of these networks is that, despite the concurrency, they are assured of being deterministic, in the sense that a KPN defines a unique sequence of messages on each communication channel between processes. That is, the messages that are communicated do not depend on the scheduling of the sequential processes. This chapter develops the theory behind such networks, explaining why they are deterministic, and discusses constraints that an execution engine must satisfy to execute them correctly.

A **process network** (**PN**) is a collection of components called **processes**, each representing a sequential, step-by-step procedure. A, B, and C in Figure 4.1 depict processes as rectangles. A process may have private state (variables that are invisible to other processes). The steps in its procedure may manipulate its state, send messages to processes (including possibly itself), or receive messages from processes (again including possibly itself).

For modularity, so that processes do not need to have references to each other, here we assume that a process sends and receives messages via named input or

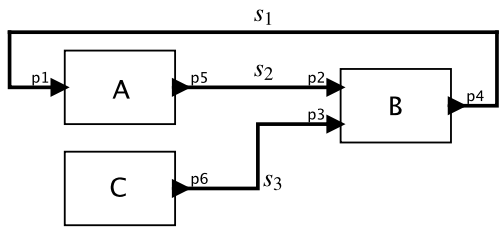


Figure 4.1: Example of a process network.

output **ports**.[1] In Figure 4.1, p1, p2, and p3 are input ports, and p4, p5, and p6 are output ports. An output port may be connected to an input port via a **channel** that carries a sequence of messages using a **first in first out** (**FIFO**) policy. Each message is called a **token** and may or may not carry a payload. A sequence of tokens is called a **stream**. In Figure 4.1, s_1, s_2, and s_3 represent streams. A stream may consist of a finite or infinite sequence of tokens.

A key assumption in process networks is that tokens are delivered to the destination input port reliably and in order. TCP sockets, for example, realize such semantics with reasonable confidence. The channel will **buffer** tokens for later delivery to the receiving process, and, in our model, will never lose tokens. A process can always send a token. It does not need to wait for the recipient to be ready to receive.

We begin with a special case of process networks that is easier to understand than the general case. This special case was introduced by Kahn and MacQueen (1977), and has been the subject of considerable study ever since. A **Kahn-MacQueen process** is a process with **blocking reads**. This means that the only mechanism that the process has to receive a token is to attempt to read a token from an input port; any such attempt blocks execution of the process if there is no available input token. The process will remain blocked until a token becomes available. In particular, a process cannot determine a priori whether a token is available. The only operation it has available is a blocking read.

A Kahn-MacQueen process is a program in an imperative language augmented with a blocking read statement and **nonblocking write** statement that reference the input and output ports of the process. Here, we describe Kahn-MacQueen processes using a C-like pseudo language that has structured control statements like `while` and `if-then-else`, plus statements like

```
t = read(p);
```

which performs a blocking read on input port p and returns a token `t`, and

```
write(p, t);
```

which writes the token `t` to port p. The `read` procedure call does not return until there is a token available at port p. The `write` statement returns immediately.

[1] The process networks implementation in Ptolemy II uses ports in this way (Ptolemaeus, 2014, Chapter 4).

Our language is a pseudo language in that we do not fully define it, and we omit important details like data types.

Example 4.1: Suppose that A in Figure 4.1 executes the following Kahn-MacQueen procedure,

```
1 write(p5, 0);
2 while(true) {
3   t = read(p1);
4   write(p5, t);
5 }
```

This procedure first produces an output token with payload value 0 and then enters an infinite loop where it reads an input token and sends it to the output port. If the input is a sequence tokens (1,2,3), for example, then the output will be (0,1,2,3).

A process with this behavior is called a **unit delay** because the input tokens appear at the output delayed by one step in the sequence. In practice, the value of the initial output (0 in this case) would typically be a **parameter** of the process rather than a built-in constant.

Let $a.b$ refer to the **concatenation** of sequences a and b. That is, $a.b$ is a sequence with prefix a followed by b. If a is infinite, then $a.b = a$. Using this notation, the process of Example 4.1 defines its output sequence as a function of its input sequence.

Example 4.2: If the input to the process of Example 4.1 is a sequence denoted by s_1, then the output is $s_2 = A(s_1) = (0).s_1$, where (0) is a sequence of length 1.

We can use the definition of process A from Example 4.1 to build a complete process network in Figure 4.1.

Example 4.3: Suppose that B executes the following procedure,

```
1 while(true) {
2   t2 = read(p2);
3   t3 = read(p3);
4   write(p4, t2+t3);
```

```
5 }
```

Suppose further that C executes the following procedure,

```
1 while(true) {
2   write(p6, 1);
3 }
```

An execution of the process network in Figure 4.1 yields the following sequences:

$$\begin{aligned} s_1 &= (1,2,3,4,\cdots) \\ s_2 &= (0,1,2,3,\cdots) \\ s_3 &= (1,1,1,1,\cdots) \end{aligned}$$

In the previous example, an execution of the process network yields infinite sequences. This is not always the case.

Example 4.4: Suppose that instead of the unit delay given in Example 4.1, A in Figure 4.1 executes the following Kahn-MacQueen procedure,

```
1 while(true) {
2   t = read(p1);
3   write(p5, t);
4 }
```

Such a process is called an **identity process**, because the output sequence is the same as the input sequence. An execution of the resulting process network yields

$$\begin{aligned} s_1 &= \perp \\ s_2 &= \perp \\ s_3 &= (1,1,1,1,\cdots) \end{aligned}$$

where $\perp$ is the empty sequence. Processes A and B both block immediately attempting to read a token provided by the other. Process C, on the other hand, is able to execute and produce an infinite sequence of outputs. Since the semantics of process networks requires that tokens on channels not be lost, the tokens produced by C must be stored until they are consumed by

B. In this case, however, they will never be consumed by B, and eventually, any execution platform will run out of memory to store the tokens.

The previous example illustrates two phenomena that can occur with process networks. The first is **deadlock**, where the processes on a directed cycle of the network are blocked waiting for tokens for each other. In this case, we have a **local deadlock**, because only part of the process network is deadlocked. In particular, C is not blocked.

The second phenomenon illustrated by this example is **unbounded memory**. This process network cannot correctly execute without an unbounded amount of memory for storing unconsumed tokens.

We will see below that, in general, whether a Kahn-MacQueen process network deadlocks is undecidable, even if we constrain the processes to a few very simple primitive ones. It is also undecidable whether such a process network can be executed with bounded memory. These two limitations are actually a consequence of the rich expressiveness of the model of computation. We will see that a very few primitive processes are sufficient to make the model of computation Turing complete, which means that it can describe every effectively computable function.

4.2 Semantics of Process Networks

Example 4.2 suggests that a Kahn-MacQueen process can be defined as a function that maps input sequences to output sequences. Consider a process with a single input port and single output port. The **data type** of a port p is a set T_p of token values that the port may consume or produce. The set T_p^{**} is the set of finite and infinite sequences of tokens of type T_p (see Appendix A). A process is therefore a function defined on such sequences.

Example 4.5: Suppose that in Example 4.2, the input and output data types are $T = \mathbb{N}$, the natural numbers. Then the unit delay is a function of form

$$A : \mathbb{N}^{**} \to \mathbb{N}^{**}$$

where for all $s_1 \in \mathbb{N}^{**}$,

$$A(s_1) = (0).s_1.$$

Historical Notes: Process Networks

Gilles Kahn (1946 – 2006), French computer scientist who developed Kahn process networks.

The notion of concurrent processes interacting by sending messages is rooted in Conway's **coroutines**. Conway (1963) described software modules that interacted with one another as if they were performing I/O operations. In Conway's words, "When coroutines *A* and *B* are connected so that *A* sends items to *B*, *B* runs for a while until it encounters a read command, which means it needs something from *A*. The control is then transferred to *A* until it wants to write whereupon control is returned to *B at the point where it left off*." The key idea is that both *A* and *B* maintain as part of their state their progress through their procedures, and transfer of control occurs to satisfy demands for data. The least fixed-point semantics is due to Kahn (1974), who developed the model of processes as continuous functions on a CPO. Kahn and MacQueen (1977) defined process interactions using nonblocking writes and blocking reads as a special case of continuous functions and developed a language for defining interacting processes. Their language included recursive constructs, an optional functional notation, and dynamic instantiation of processes. They gave a demand-driven execution semantics, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976). Berry (1976) generalized these processes with stable functions.

Sequences of tokens communicated by process networks are **unbounded lists**. The notion of unbounded lists as data structures first appeared in Landin (1965). This underlies the communication mechanism between processes in a process network. The UNIX operating system, due originally to Ritchie and Thompson (1974), includes the notion of **pipes**, which implement a limited form of process networks.

Kahn (1974) stated but did not prove that a maximal and fair execution of process network yields the least fixed point. This was later proved by Faustini (1982) and Stark (1995). It is known as the Kahn principle.

As we pointed out before, normally the initial output 0 would be a parameter of the process rather than a built-in constant.

In general, the semantics of a program is its meaning. In the case of process networks, the semantics is the streams defined by the network. Each stream is a finite or infinite sequence of tokens. In this section, we will show that if processes define functions that satisfy a particular constraint (they are continuous), then the semantics of the network is unique (i.e., the network defines exactly one sequence of tokens for each stream), and we can give a constructive procedure for building the streams defined by the network (i.e., a mechanism for executing the network). For this section, you will want to have mastered the material on the prefix order and complete partial orders, in Chapter A.

4.2.1 Least Fixed Point Semantics

Recall that a fixed point of a function $F: X \to X$ is an element $x \in X$ such that $F(x) = x$. Execution of every process network can be reduced to finding a fixed point of a function. The function is a composition of the functions defined by the individual processes.

Example 4.6: Consider the process network in Figure 4.1. This network is redrawn in Figure 4.2(a). In Figure 4.2(b), we reorganize the drawing and draw a box around the three processes. This box can itself be a considered a function with three input sequences and three output sequences, as illustrated in Figure 4.2(c). Figure 4.2(d) further abstracts this by aggregating the three streams into one.

Suppose that the three streams in Figure 4.2 have data type T. Then each stream s_i, $i = 1,2,3$, is a member of the set T^{**} of sequences of tokens of type T. In Figure 4.2(d), the function F therefore has the form

$$F: (T^{**})^3 \to (T^{**})^3.$$

Using the process definitions from Examples 4.1 and 4.3, we see that if the input is

$$s = ((a_1, a_2, \cdots), (b_1, b_2, \cdots), (c_1, c_2, \cdots)),$$

a three-tuple of sequences, then the output is

$$F(s) = ((b_1 + c_1, b_2 + c_2, \cdots), (0, a_1, a_2, \cdots), (1, 1, \cdots)),$$

which is also a three-tuple of sequences.

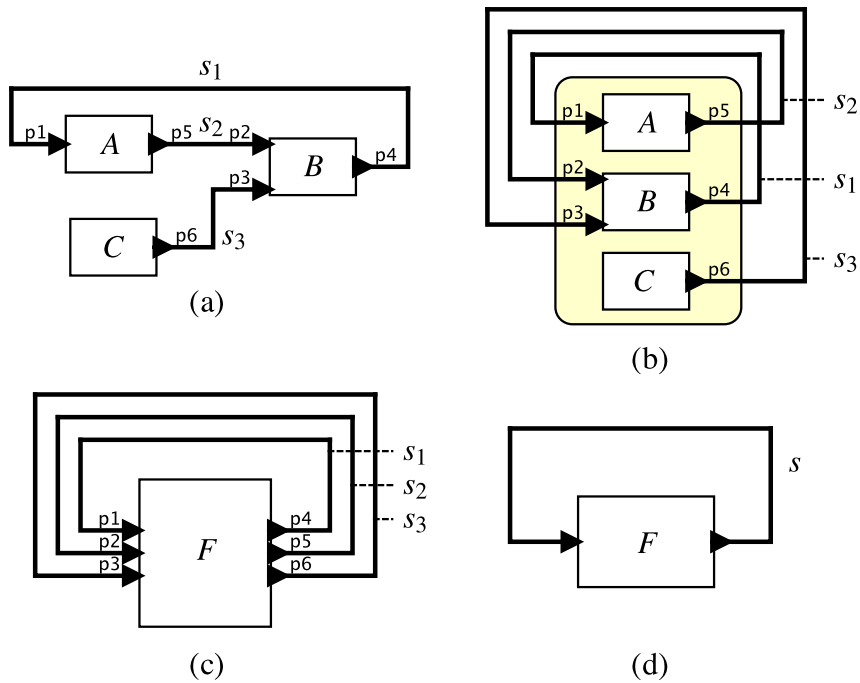


Figure 4.2: Execution of every process network can be reduced to finding a fixed point of a function, as illustrated by this sequence of transformations of a network.

Because of the feedback loop, we seek a sequence s that satisfies $F(s) = s$, so in this case, we require that

$$\begin{aligned} a_1 &= b_1 + c_1 \\ a_2 &= b_2 + c_2 \\ b_1 &= 0 \\ b_2 &= a_1 \\ c_1 &= 1 \\ c_2 &= 1 \\ &\cdots \end{aligned}$$

which implies that

$$\begin{aligned} a_1 &= 1 \\ a_2 &= 2 \\ b_1 &= 0 \\ b_2 &= 1 \\ c_1 &= 1 \\ c_2 &= 1 \\ &\cdots \end{aligned}$$

which is the same result we obtained before in Example 4.3.

Execution of a process network with n streams of type T can be reduced to finding a fixed point of a function of the form

$$F\colon (T^{**})^n \to (T^{**})^n.$$

If a process network has streams with different types, then the domain and codomain of the function will be a cartesian product of the sets of sequences of those types. The notation gets more complex, but not the concept. It may be surprising that *all* process networks can be reduced in the same way to finding a fixed point of a function.

Example 4.7: Consider the network in Figure 4.3(a). It may seem that this would be difficult to reduce to a fixed point problem because there is no feedback in the network. However, this can be redrawn as shown in Figure 4.3(b), and abstracted as shown in Figure 4.3(c). In this case, the function F is a rather trivial function. Regardless of the input stream, it always produces the same output stream. It is a **constant function**, where the output is independent of the input.

To be concrete, suppose that the data type is $T = \mathbb{N}$, the natural numbers, and that process A produces the sequence $(0,1,2,3,\cdots)$. Then $F\colon \mathbb{N}^{**} \to \mathbb{N}^{**}$ is a function such that for all $s \in \mathbb{N}^{**}$,

$$F(s) = (0,1,2,3,\cdots).$$

Hence, $s = (0,1,2,3,\cdots)$ is a fixed point of F (the only fixed point, in fact).

For the above examples, the function F has exactly one fixed point. This is not always the case.

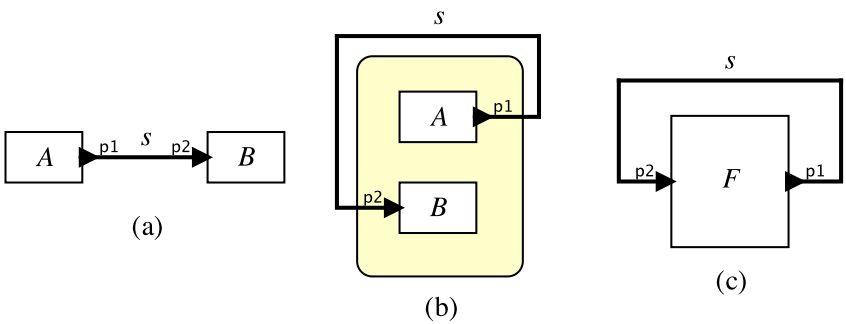


Figure 4.3: Even a process network with no feedback loops can be reduced to finding a fixed point of a function, as illustrated by this sequence of transformations of a network.

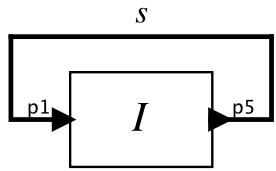


Figure 4.4: A process network with multiple fixed points, where I is the identity process.

Example 4.8: Consider an identity process I, described in Example 4.4, where for all $s \in T^{**}$,

$$I(s) = s.$$

Suppose that this process is used the process network shown in Figure 4.4. This process network has multiple fixed points. In fact, any stream $s \in T^{**}$ is a fixed point.

A prefix order is a partial-order relation $\sqsubseteq$ over sets of sequences T^{**} (see Appendix A). For any two $a, b \in T^{**}$, $a \sqsubseteq b$ if the sequence a forms the leading part of sequence b. If a is infinite, then $a \sqsubseteq b$ if and only if $a = b$.

When a process network has multiple fixed points, we will choose the **least fixed point**, where "least" is in the prefix order. The least fixed-point semantics is a **denotational semantics**; it gives the program a meaning without necessarily offering a procedure for finding that meaning. An **operational semantics**, in contrast, gives the meaning of a program in terms of a procedure for finding that meaning. Kahn-MacQueen processes, with their blocking reads, offer an operational semantics.

Example 4.9: For the process network in Figure 4.4, the least fixed point is $s = \perp$, the empty sequence. This choice is indeed the least fixed point because for all other fixed points s, $\perp \sqsubseteq s$. Moreover, it is exactly the fixed point that results from executing the Kahn-MacQueen process. The process deadlocks immediately, blocked on the first read. Hence, the result of execution is the empty sequence.

A match between an operational semantics and a denotational semantics is sometimes called **full abstraction**. The previous example suggests that with Kahn-MacQueen and the least fixed point semantics, we have full abstraction. However, when the semantics involves infinite structures, like infinite streams, what we mean by "a match" gets tricky. An operational semantics can never produce an infinite stream until it has run forever.

We will see next that with Kahn-MacQueen process networks, the least fixed point is always uniquely defined, and execution of the network always produces *a prefix* of that least fixed point. We will see that whether the operational semantics converges in the limit to the least fixed point depends on what we mean by convergence and on details of the execution policies that we have not yet discussed, such as management of buffer sizes. We will also show that some useful processes that cannot be defined as Kahn-MacQueen processes also yield a uniquely defined least fixed point, so the Kahn-MacQueen operational semantics is, in a sense, smaller than the least fixed-point denotational semantics.

4.2.2 Monotonic and Continuous Functions

A function $A: T^{**} \to T^{**}$ is monotonic in the prefix order if $a \sqsubseteq b$ implies that $A(a) \sqsubseteq A(b)$. A Kahn-MacQueen process defines a function from input sequences to output sequences as long as the imperative language used to define the language is deterministic. This means that given particular input streams at the input ports, the output streams are fully defined by the process. More interestingly, Kahn-MacQueen processes are monotonic.

Example 4.10: The unit delay function considered in Example 4.5 is monotonic. Consider two possible input streams $a, b \in \mathbb{N}^{**}$. The corresponding output sequences are $A(a) = (0).a$ and $A(b) = (0).b$. Clearly, if $a \sqsubseteq b$, then $A(a) \sqsubseteq A(b)$.

The fact that Kahn-MacQueen processes are monotonic is easy to see by considering their execution. Suppose $a, b \in \mathbb{N}^{**}$ are two possible input streams where $a \sqsubseteq b$. A Kahn-MacQueen process that is presented with input b will first read a sequence of tokens equal to a. Since the process is deterministic, once it has read this prefix of b, its output will be exactly the output sequence it would have produced had it been presented with input a. Upon continuing to read tokens

from b (which now differ from a), it will only extend the output stream. It has no mechanism for retracting previously produced tokens, and hence the output produced in response to b is an extension of the output produced in response to a.

Notice that the blocking reads of a Kahn-MacQueen process play an important role in ensuring monotonicity.

Example 4.11: Consider a process A with one input port and one output port that produces on its output the sequence (0) if the input is the empty sequence, and otherwise produces the output (1). This process clearly defines a function, in that the output is fully defined by the input. However, this function is not monotonic. Suppose that $a = \perp$, the empty sequence, and $b = (0)$. Then $a \sqsubseteq b$, but $A(a) = (0)$ is not a prefix of $A(b) = (1)$.

Notice that this process cannot be implemented as a Kahn-MacQueen process. In order to produce the output (0), the process needs to know that the input is the empty sequence. But the only operation it has available on the input port is to perform a blocking read. If the input is indeed empty, then it will remain blocked forever, and hence will not be able to produce the output (0).

The poset $(T^{**}, \sqsubseteq)$ is a complete partial order (CPO), as explained in Example A.16 of Appendix A. Moreover, by Proposition A.3, the poset $((T^{**})^n, \sqsubseteq)$ is also a CPO, where $\sqsubseteq$ is the pointwise prefix order. A function $F\colon (T^{**})^n \to (T^{**})^n$ is continuous if for all chains $C \subseteq (T^{**})^n$,

$$F(\bigsqcup C) = \bigsqcup \hat{F}(C) \,,$$

where $\hat{F}$ is the lifted version of F. By Proposition A.5, every continuous function is monotonic.

In practice, every Kahn-MacQueen process is not only monotonic, but also continuous. Why? Intuitively, continuity means that a function does not "wait forever" before producing output. Suppose that the chain $C = \{s_0, s_1, s_2, \cdots\}$ represents a sequence of partially constructed inputs to a Kahn-MacQueen process. Then $\bigsqcup C = s$ represents the eventual, complete input. The set

$$\hat{F}(C) = \{F(s_0), F(s_1), F(s_2), \cdots\}$$

represents the partially constructed outputs, given partially constructed inputs. By the definition of Kahn-MacQueen processes, these are exactly the outputs

that the process can produce. Then continuity requires that $\bigvee \hat{F}(C)$ be equal to the eventually complete output $F(s)$, and by the definition of Kahn-MacQueen processes, that is exactly the output.

Example 4.12: Suppose that A is a process with one input port and one output port. Suppose that its input is eventually going to be the infinite sequence

$$s = (0,1,2,3,\cdots)$$

Suppose that $C = \{s_0, s_1, s_2, \cdots\}$, where

$$\begin{aligned} s_0 &= (0) \\ s_1 &= (0,1) \\ s_2 &= (0,1,2) \\ &\cdots \end{aligned}$$

C is clearly a chain. Moreover,

$$\bigvee C = s.$$

Suppose further that A is the function

$$A(s) = \begin{cases} \bot & \text{if } s \text{ is finite} \\ (1) & \text{otherwise} \end{cases}$$

This function is clearly monotonic, but it is not continuous. In particular, when the function is applied to each partially constructed input in C, the output is $\bot$. Hence,

$$\hat{F}(C) = \{F(s_0), F(s_1), F(s_2), \cdots\} = \{\bot\}.$$

Hence,

$$\bigvee \hat{F}(C) = \bot\ .$$

However,

$$F(\bigvee C) = F(s) = (1).$$

These two are not equal. Intuitively, the function F has to wait forever to determine whether the input is finite or not.

Notice that this function cannot be implemented by a Kahn-MacQueen process. We might try to specify it as follows,

```
1 while(true) {
2    read(p1);
3 }
4 write(p2, 1);
```

That is, after reading an infinite number of inputs, we output a 1. If the input is finite, this process will block forever on one of the reads. In practice, it will never reach the point of producing the 1, so this procedure does not really implement the function.

The Kleene fixed-point theorem (Proposition A.7) then provides what we need. It states that if F is a continuous function, then it has a unique least fixed point. Moreover, it asserts that the least fixed point is the result of applying the function first to $\bot$, and then recursively to the result, etc., as follows:

$$\begin{aligned} s_0 &= \bot \\ s_1 &= F(\bot) \\ s_2 &= F(F(\bot)) \\ &\cdots \\ s_m &= F^m(\bot) \\ &\cdots \end{aligned}$$

Thus, instead of solving a system of equations, as we did in Example 4.6, we can construct the solution by just repeatedly applying the function F.

Example 4.13: Consider the same process network of Example 4.6. Recall that the function $F\colon (T^{**})^3 \to (T^{**})^3$ is given by

$$F(s) = ((b_1 + c_1, b_2 + c_2, \cdots), (0, a_1, a_2, \cdots), (1, 1, \cdots)),$$

where the input is

$$s = ((a_1, a_2, \cdots), (b_1, b_2, \cdots), (c_1, c_2, \cdots)),$$

This function is easily shown to be continuous.

In this case, the bottom of the CPO is the three-tuple of empty sequences, so the constructive procedure given by the Kleene fixed-point theorem proceeds as follows,

$$\begin{aligned} F((\bot,\bot,\bot)) &= (\bot,(0),(1,1,\cdots)) \\ F(F((\bot,\bot,\bot))) &= ((1),(0,1),(1,1,\cdots)) \\ F(F(F((\bot,\bot,\bot)))) &= ((1,2),(0,1,2),(1,1,\cdots)) \\ &\cdots \end{aligned}$$

This eventually converges to the same sequences found in Example 4.3.

Notice that this procedure immediately creates a practical problem. The very first invocation of the function F yields as the third element of the output tuple an infinite sequence. If we were to literally implement this procedure in a computer, then we would never get past the first step because we would exhaust available memory in any attempt to generate this infinite sequence. The following section will consider practical execution policies.

Example 4.14: For the variant given by Example 4.4, which has a local deadlock, the Kleene procedure immediately converges to the final answer,

$$F((\bot,\bot,\bot)) = (\bot,\bot,(1,1,\cdots)).$$

Example 4.15: For the process network in Figure 4.4, recall that the least fixed point is $s = \bot$. The Kleene procedure immediately converges to this solution.

Example 4.16: Recall that the unit delay process of Example 4.1 is a function of form

$$A : \mathbb{N}^{**} \to \mathbb{N}^{**}$$

where for all $b \in \mathbb{N}^{**}$,

$$A(b) = (0).b.$$

This function is continuous. Suppose we put it in a feedback loop, as we did with the identify function in Figure 4.4. Then the Kleene procedure would

yield

$$\begin{aligned} A(\perp) &= (0) \\ A(A(\perp)) &= (0,0) \\ A(A(A(\perp))) &= (0,0,0) \\ &\cdots \end{aligned}$$

The least upper bound of this chain is the infinite sequence $(0,0,0,\cdots)$.

4.3 Execution of Process Networks

The least fixed point of a process network often includes infinite sequences. In practice, no computer program can construct an infinite sequence. Our interpretation is that process networks whose semantics include infinite sequences do not terminate. They are able to continue to execute for as long as we would like them to execute. At any point during the execution, the sequences that they have produced are mere approximations of the denotational semantics of the network. When there are multiple infinite sequences defined by a network, then there are many possible approximations. In this section, we consider execution policies that provide useful approximations.

We need some definitions.

Definition 4.1. *A **correct execution** of a process network is one which, at any time during execution, has constructed a prefix of each stream defined by the denotational semantics of the network.*

These prefixes will always be finite, even if the semantics of the network includes infinite sequences.

Definition 4.2. *A **maximal execution** is a correct execution that either does not halt, or if it halts, has produced exactly every sequence defined by the denotational semantics of the network.*

What we mean by "halt" in this case is that the execution no longer appends tokens to streams. A maximal execution will only halt if the denotational semantics of

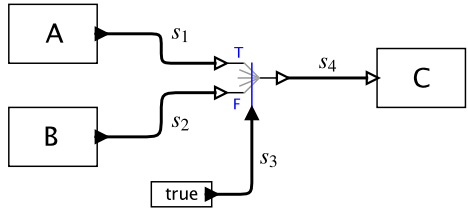


Figure 4.5: Example of a process network where unfair execution is desirable because fair execution will exhaust available memory.

the network defines only finite streams. An **infinite execution** is an execution that does not halt. All infinite executions are maximal.

Definition 4.3. *A **fair execution** is one that ensures that if any process is able to produce an output token or read an input token, then it will eventually be allowed to do so.*

The following proposition, known as the **Kahn principle**, stated by Kahn and MacQueen (1977) and later proved by Faustini (1982) and Stark (1995), provides key guidance for execution policies.

Proposition 4.1. *Any two fair and maximal executions of a process network produce the same sequences of tokens, equalling the least fixed point that is the denotational semantics.*

The Kahn principle seems to suggest that practical executions of process networks should always be fair and maximal. This is not in fact obvious. The following example shows that fair and maximal executions are not always desirable.

Example 4.17: Consider the process network in Figure 4.5. Suppose that processes A and B produce infinite sequences, and C is able to consume an infinite sequence. Suppose that the process labeled true produces an output of type $\mathbb{T} = \{true, false\}$ that is a constant infinite sequence

$$s_3 = \{true, true, true, \cdots\}.$$

The remaining process in the center, which has no label, is a **Select** process. It accepts a sequence of control tokens on the bottom port, and uses these to merge the sequences at its other two input ports. The merged sequence is the output. Specifically, the Select process has three input ports, T, F, and control, where the control port is shown on the bottom of its icon. It has a single output port p. The process is defined by the following Kahn-MacQueen procedure,

```
1 while{true} {
2   c = read(control);
3   if (c) {
4     t = read(T);
5   } else {
6     t = read(F);
7   }
8   write(p, t);
9 }
```

Any fair execution of this process network has to permit B to produce its infinite sequence of tokens, even though those tokens will never be consumed. This will require unbounded memory to store unconsumed tokens.

There is, however, a maximal execution of this network that runs in bounded memory. Such an execution would only permit process B to produce a finite number of tokens. Such an execution is maximal and correct, but not fair.

Definition 4.4. *A **bounded execution** is an execution of a process network where there is a natural number $M \in \mathbb{N}$ such that at all times during the execution there are no more than M unconsumed tokens.*

A fair and maximal execution of the network in Figure 4.5 is not bounded.

Definition 4.5. *An **effective** process network is one where every token that can be produced by a process will eventually be read by the destination process(s) (Geilen and Basten, 2003).*

The process network in the previous example is not effective. For networks that are not effective, we do not necessarily want fair and maximal execution, because such execution will eventually fail due to running out of memory.

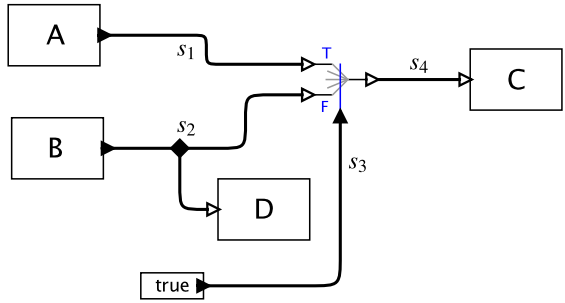


Figure 4.6: Example of a process network where demand-driven execution fails to deliver a bounded execution.

Even for networks that are effective, executing them with bounded memory is not always easy. Suppose that we modify the network in Figure 4.5 so that the process labeled true produces a random sequence of *true* and *false* values. How can we ensure that A and B so not overflow available memory on their outputs? Even for much simpler networks there is risk of memory overflow.

Example 4.18: Suppose that in the network in Figure 4.3(a), process A produces tokens faster than process B. Then eventually, we will run out of memory.

We seek, therefore, an execution policy that will deliver maximal executions for all process networks, bounded executions for networks for which bounded execution is possible, and fair executions for effective process networks. We call these **effective executions**. Effective executions are not required to be fair for networks that are not effective. They are not required to be bounded for networks that have no bounded execution. They are required to be maximal for all networks.

There is a long history of failed attempts to achieve this goal. One popular proposed technique is **demand-driven execution**, related to the lazy evaluators of Lisp (Friedman and Wise, 1976; Morris and Henderson, 1976), where no token is produced unless there is a downstream process that is ready to consume it. Unfortunately, this does not solve the problem, as illustrated by the following example.

Example 4.19: A variant of Figure 4.5 is shown in Figure 4.6. The only difference is that is a new process D has been added that consumes tokens from process B. Each token produced by B needs to be routed to both the Select process and to D. This can be accomplished by a **fork** process, indicated in the figure by a small black diamond. The fork can be implemented, for example, by the Kahn-MacQueen procedure

```
1 while (true) {
2    t = read(in);
3    write(out1, t);
4    write(out2, t);
5 }
```

Thus, there will be two buffers, one storing tokens destined for the Select, and one storing tokens destined for D.

Assume that D is able to read an infinite input sequence. As a consequence, it will demand tokens. In demand-driven execution, the fork will respond by demanding inputs. Once the fork receives an input, the above Kahn-MacQueen implementation will send the token to both outputs, and hence the buffer at the F input of the Select will eventually overflow. If the fork is implemented differently so that it only sends outputs in response to demands, then it will be obligated to store the token locally in case the token is later demanded. This too will overflow available memory.

In the previous example, demand-driven execution can deliver bounded execution only if we somehow regulate the demands issued by the two sink processes, C and D. But how would we do that? Moreover, suppose that network contains cycles, or more interestingly, a multiplicity of cycles. How would demands be generated?

Data-driven execution is the complement of demand-driven execution. Instead of **sink** processes driving the execution (processes with no output ports), **source** processes drive the execution (processes with no input ports). A process is permitted to execute only when it has available input data, except source processes, which are always permitted to execute. This obviously does not achieve the goal, since even a network as simple as that in Figure 4.3(a) cannot be assured of remaining bounded.

The problem of achieving effective executions is harder than it first appears. Fair execution, demand-driven execution, and data-driven execution all fail. It is not surprising that the problem is hard, however. It turns out that whether a bounded execution exists is undecidable, as is whether an infinite execution exists. We defend this claim in the next subsection. After that (Section 4.3.2), we show how to solve the undecidable problem by giving a policy that delivers effective execution.

4.3.1 Turing Completeness of Process Networks

In his PhD thesis, Buck (1993) proved that Kahn process networks could realize a universal Turing machine using only a small set of very simple processes. This means that the KPN model of computation is as expressive as any other programming language capable of realizing all effectively computable functions. It also means that some questions about process networks become undecidable, meaning that there is no algorithm that can answer the question for all process networks. One such undecidable question is whether a given process network can be executed with bounded memory. Another undecidable question is whether a process network will halt (through deadlock or starvation).

One way to show the Turing completeness of KPN is to construct a universal Turing machine using only a small set of primitive processes that operate on only a binary data type. This can be quite tedious, but a major step in that direction is to show how to construct an unbounded stack. Two such stacks can be used to represent two sides of the tape in a Turing machine.

One possible implementation of a stack as a KPN is given in Figure 4.7. This example is one of the demos included in the Ptolemy II distribution (Ptolemaeus, 2014). The top-level network uses a pair of Select and Switch processes in a feedback loop with a UnitDelay. This loop stores the contents of the stack in the KPN buffers between the processes. The StackCounter controls these to either accept a new input from the Source when a data value is being pushed onto the stack or pop a value off the stack and route it to the Display process. The TestSequence process produces a `true` to push a new value onto the stack and a `false` to pop a value off the stack.

The Select process is the same as that used in example 4.17. It chooses one of two input ports on which to perform a blocking read, depending on a boolean control

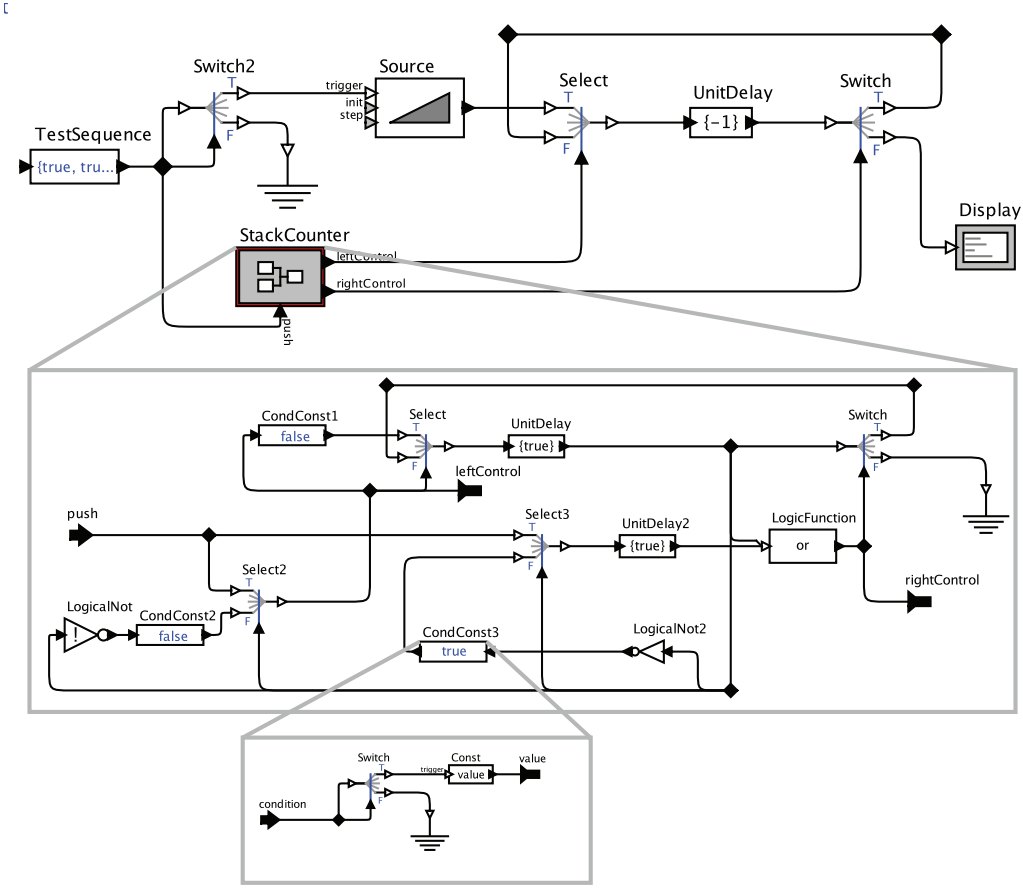


Figure 4.7: A KPN network implementing an unbounded stack onto which tokens can be pushed and popped.

input. The **Switch** process does the converse. It routes input tokens to one of two output ports depending on a boolean control input. The **UnitDelay** process is a parameterized version of the unit delay that we saw in Example 4.1. It outputs a constant and then just forwards input tokens to the output.

The hard work is done by the StackCounter subsystem. In addition to Switch, Select, and UnitDelay, this subsystem uses several instances of CondConst, the implementation of which is shown at the bottom. This subsystem discards false-valued inputs and, upon receiving a true-valued input, sends a constant-valued (given by a parameter) to the output port. The StackCounter also uses processes that perform logic operations on boolean-valued inputs and a process that reads and discards inputs (which has an icon that looks like an electrical ground).

How this network implements a stack is complicated. But the key point is that the memory of the stack is provided by buffers that store tokens that have been produced but not yet consumed. None of the individual processes provide any memory that could could function as the Turing machine tape. The amount of memory used by these buffers is not bounded, like a Turing machine tape. It depends on the test sequence that is provided as input.

Using two of these stacks, it is possible to emulate a tape, where moving the tape to the left or right pops a token off one stack and pushes it onto the other. With some additional logic to control these pushes and pops, it is possible to build a universal Turing machine.

4.3.2 Effective Execution

Because of the Turing completeness of KPN, whether a process network has a bounded execution is undecidable. This means that there is no terminating algorithm that can, for all process networks, determine whether it has a bounded execution. Nevertheless, we seek a coordination mechanism that will deliver a bounded execution for every process network that has a bounded execution. In other words, we want to solve an undecidable problem.

Fortunately, we do not need to deliver an answer in bounded time. Only a process network that runs forever can possibly require unbounded memory. It is OK for our coordination mechanism to take forever to give a definitive answer to the

undecidable problem as long as it delivers a bounded execution when that is possible and delivers an effective execution when it is not.

A partial solution was given by Parks (1995). Parks' solution delivers bounded and maximal execution for every process network that has a bounded execution. But it does not guarantee a fair execution and hence falls short of delivering an effective execution.

Parks' algorithm is simple. An execution starts with bounded buffers on all connections between processes. It does not matter what the bound is. When a process attempts to write to a buffer that is full, the write blocks. That is, the process stalls, waiting for room to become available. Now, one of three things can happen. First, the network might execute forever, continuing to produce and consume tokens, even though some processes are blocked on a write. This achieves an infinite (hence maximal) bounded execution. Second, the network might deadlock where all processes have either terminated or are blocked on a read. In this case, the process network specifies a finite execution, and the finite execution has been completed. The execution is again maximal and bounded. Third, the network might deadlock where at least one process is blocked on a write. In this third case, Parks' strategy finds the smallest buffer on which a process is write blocked and increases its capacity so as to unblock the write-blocked process. Execution continues until again one these three scenarios occurs.

For many process networks, Parks' strategy achieves an effective execution. But not for all, as illustrated by the following example.

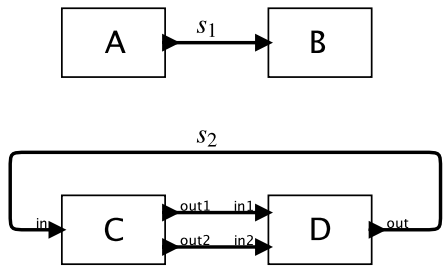


Figure 4.8: Example of a process network where Parks' algorithm does not yield an effective execution.

Example 4.20: Consider the process network in Figure 4.8. Suppose that all four processes can produce and consume infinite streams. Moreover, suppose that C is defined by the following Kahn-MacQueen procedure,

```
1 while (true) {
2   t = read(in);
3   write(out1, t);
4   write(out1, t);
5   write(out2, t);
6 }
```

Suppose further that D is defined by the following Kahn-MacQueen procedure,

```
1 write{out, 0);
2 while (true) {
3   t = read(in2);
4   t = read(in1);
5   t = read(in1);
6   write(out, t);
7 }
```

It is easy to see that the least fixed point yields the infinite stream $s_2 = (0,0,0,\cdots)$.

Now suppose that we execute this network using Parks' algorithm, starting with initial buffer sizes of 1. D successfully produces its first output, enabling C to complete the read on line 2. C will then perform the write on line 3, but the write on line 4 will block because the buffer capacity has been reached. As a consequence, D will block on the read on line 3. A local deadlock occurs.

Under Parks' algorithm, no corrective action is taken because processes A and B are able to continue running indefinitely. As a consequence, this is not a fair execution. To be fair, every process that is able to produce outputs must be allowed to do so. C is able to produce outputs, but we have blocked it artificially as part of our execution policy (for this reason, the state of the execution is called a **local artificial deadlock**).

Geilen and Basten (2003) identified this flaw in Parks' algorithm, and proposed an alternative execution strategy. Their strategy is similar to that of Parks in that

execution starts with arbitrarily bounded buffers. But while Parks' algorithm looks only for a **global artificial deadlock** (all processes are blocked, at least one on a write), Geilen and Basten's strategy looks for local artificial deadlock.

Specifically, consider a cycle of processes $(A_1, A_2, \cdots, A_n)$, $n \geq 1$, where A_i is connected to A_{i+1} for $i < n$, and A_n is connected to A_1. When we say "A_i is connected to A_{i+1}," we mean that either A_i send tokens to A_{i+1} or vice versa. So $(A_1, A_2, \cdots, A_n)$ is an **undirected cycle**. If at any time during the execution there is an undirected cycle where all of these processes are blocked, at least one on a write, then we have a local artificial deadlock. When such an artificial deadlock is detected, the Geilen and Basten strategy will increase the size of at least one buffer on which a process in the cycle is write blocked.

Geilen and Basten (2003) prove that this strategy delivers an effective execution for effective process networks. But they make no guarantee about networks that are not effective.

One of the challenges in implementing both Parks' algorithm and Geilen and Basten's is performing a distributed deadlock detection. Interesting implementations are described by Olson and Evans (2005), Allen et al. (2007), and Dulloo and Marquet (2004).

For all of the examples we have considered so far, it is relatively easy to determine whether a bounded execution exists. This is not always the case, as illustrated by the following example.

Example 4.21: Consider for example the network shown in Figure 4.9. The output is an ordered sequence of integers of the form $2^n 3^m 5^k$, where n, m and k are non-negative integers. These are known as the **Hamming numbers**, and this program for computing them was studied by Dijkstra (1976) and Kahn and MacQueen (1977).

To understand how this network works, we need to understand what each of the processes does. The ones labeled ScaleN simply read an integer-valued input token, multiply it by N, and send the product to the output port. They repeat this sequence forever, as long as there are input tokens. The UnitDelay processes are parameterized versions of the unit delay that we saw in Example 4.1. They first produce an output token with value N, then repeat forever the following sequence: read the input token and send it to

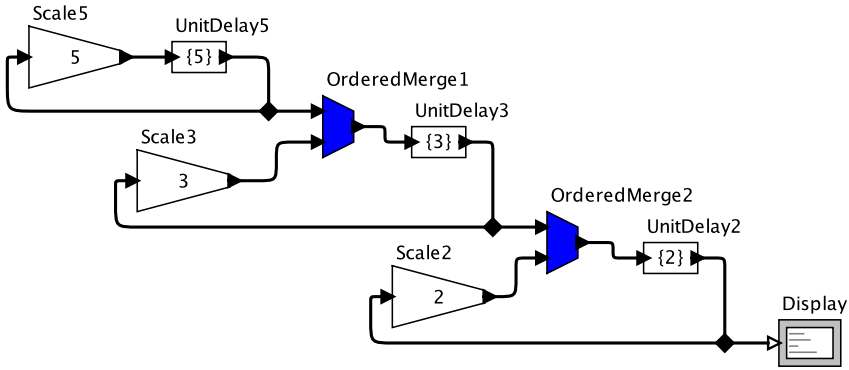


Figure 4.9: A process network for which it is difficult to determine appropriate buffer sizes.

the output. As before, the black diamonds are fork processes. They direct a single token sequence to multiple destinations (over separate, unbounded buffers).

The processes labeled **OrderedMerge***M* are the only nontrivial processes in this program. Given a sequence of numerically increasing tokens on each input port, they produce a numerically increasing merge of the two input sequences on their output ports. A Kahn-MacQueen implementation could look like this:

```
1  t1 = read(in1);
2  t2 = read(in2);
3  while(true) {
4    if (t1 < t2) {
5       write(out, t1);
6       t1 = read(in1);
7    } else {
8       write(out, t2);
9       t2 = read(in2);
10   }
11 }
```

You can now understand how this program generates the Hamming numbers. The loop at the upper left in the figure produces the sequence

$$(5, 25, 125, \cdots, 5^n, \cdots).$$

That sequence is merged into the the second loop, which as a result produces the sequence

$$(3, 5, 9, 15, 25, \cdots, 5^n 3^m, \cdots.$$

That sequence is merged into the final loop, which produces the Hamming numbers, in numerically increasing order, without repetitions.

Although it is far from easy to see at a glance, it is possible to prove that any finite bound on buffer sizes will cause the network to fail to produce all the Hamming numbers. This network intrinsically requires unbounded buffers. It is effective, and both Parks' and Geilen and Basten's strategies deliver an effective execution.

4.4 Convergence of Execution to the Denotational Semantics

The Kahn principle states that maximal and fair executions produce sequences that match that the least fixed point. But what do we mean by "match?" Every execution is finite at all times, and can only have produced a prefix of the least fixed point if that least fixed point includes infinite sequences. Intuitively, we would like such executions to converge in the limit to the least fixed point. But to make this intuition precise, we have to define a notion of convergence for sequences.

We use the notion of a topology.

Definition 4.6. *Let X be any set. A collection τ of subsets of X is called a **topology** if three conditions are satisfied:*

1. *X and $\emptyset$ are members of τ.*
2. *The intersection of any two members of τ is in τ.*
3. *The union of any collection of members of τ is in τ.*

For any topology τ, the members of τ are called the **open sets** of the topology. See the sidebar on page 91 for an explanation of how this relates to the usual notion of open sets of real numbers. In our case, we are not concerned with real numbers, but rather with sequences of tokens of arbitrary type.

There is an important but subtle distinction between conditions (2) and (3) above. A "collection of members of τ" may be infinite. Thus, in a topology, the *union* of an infinite number of open sets is an open set. But the *intersection* of an an infinite number of open sets may not be an open set. The intersection of any *finite* number

Sidebar: Limit of a Sequence of Real Numbers

An **open neighborhood** around a in $\mathbb{R}$ is the set

$$N_{a,\varepsilon} = \{x \in \mathbb{R} \mid a - \varepsilon < x < a + \varepsilon\}$$

for some positive real number ε. An **open set** A in the reals is a subset of $\mathbb{R}$ such that for all $a \in A$, there is an open neighborhood around a that is a subset of A. The collection of such open sets in the reals is called the **standard topology** for the reals. It is a topology because it includes both $\mathbb{R}$ itself and $\emptyset$, it includes the intersection of any two such open sets, and it includes an arbitrary union of such open sets. Notice that it *does not* include any arbitrary intersection of open sets. Consider for example the intersection of the sets

$$N_{0,\varepsilon}, \quad \text{where } \varepsilon = 1/n, \quad n \in \{1, 2, 3, \cdots\}.$$

The intersection of these sets is the singleton set $\{0\}$, which is not an open set.

Consider a sequence of real numbers

$$s : \mathbb{N} \to \mathbb{R}.$$

This sequence is said to converge to a real number a if for all open sets A containing a there exists an integer $n \in \mathbb{N}$ such that for all integers $m > n$ the following holds:

$$s(m) \in A.$$

This notion of convergence captures the intuition that the sequence eventually gets arbitrarily close to a.

Sidebar: Scott Topologies

Let X be a poset. A subset $O \subseteq X$ is **Scott open** if it is an upper set and all directed sets $D \subseteq X$ where $\bigvee D \in O$ have a non-empty intersection with O; i.e., $D \cap O \neq \emptyset$.

The **Scott topology** on X is the set of Scott open subsets of X. The Scott topology is named after Dana Scott (1932–present), a renowned computer scientist and mathematician. It has a number of interesting properties.

First, for the poset $(T^{**}, \sqsubseteq)$, every open set of the Scott topology is an upper set. To see this, observe that an open neighborhood in this topology, by definition, is an upper set. Next, observe that the union of any family of upper sets is an upper set. Finally, since the open sets in the topology are unions of open neighborhoods, every open set must be an upper set.

A **Hausdorff space** is a topology in which distinct points have disjoint neighborhoods. Specifically, if X is a set and τ is a topology over X, then for any two $x_1, x_2 \in X$ where $x_1 \neq x_2$, there are two open sets $X_1, X_2 \in \tau$ where $x_1 \in X_1$, $x_2 \in X_2$, and $X_1 \cap X_2 = \emptyset$. As a mnemonic, in a Hausdorff space, two distinct points can be "housed off" by open sets.

The Scott topology over T^{**} is *not* a Hausdorff space. This property of the Scott topology accounts for much of the difficulty in reasoning about this topology because the standard topology over the reals *is* Hausdorff.

To see that the Scott topology is not Hausdorff, let $x_1, x_2 \in T^{**}$ be two sequences where $x_1 \neq x_2$ and $x_1 \sqsubseteq x_2$. These two distinct points cannot be "housed off" by open sets. Any open set that contains x_1 also contains x_2 because an open set is an upper set.

A **Kolmogorov space** is a topology in which, given two distinct points, at least one of them has an open neighborhood not containing the other. Specifically, if X is a set and τ is a topology over X, then for any two $x_1, x_2 \in X$ where $x_1 \neq x_2$, there is an open set $A \subset X$ that contains only one of the two points x_1 and x_2. This condition intuitively means that the points are topologically distinguishable.

The Scott topology over T^{**} *is* a Kolmogorov space. If $x_1 \sqsubseteq x_2$, then the open neighborhood N_{x_2} contains x_2 and not x_1. If $x_2 \sqsubseteq x_1$, then the open neighborhood N_{x_1} contains x_1 and not x_2. If neither is a prefix of the other, then N_{x_2} contains x_2 and not x_1 and N_{x_1} contains x_1 and not x_2.

of open sets is required to be an open set, but not of an *infinite* number of open sets.

Consider a set T and the set T^{**} of finite and infinite sequences of elements of T.

Definition 4.7. *Given a* finite *sequence* $s \in T^{**}$, *an* ***open neighborhood*** N_s *around* s *is the set*

$$N_s = \{s' \in T^{**} \mid s \sqsubseteq s'\}.$$

N_s is the set of sequences with prefix s.

N_s is an **upper set**,[2] meaning that N_s is **upward complete**. That is, it contains every element $a \in T^{**}$ that is greater than any $s' \in N_s$, $s' \sqsubseteq a$.

Let τ be the collection of all sets that are arbitrary unions of such open neighborhoods. Then τ is a topology. This topology is a particular kind of Scott topology (see sidebar on page 92).

Notice that an open neighborhood is defined by a *finite* prefix. Were we to allow an infinite prefix, then a singleton set consisting of a single infinite sequence would be an open neighborhood. So infinite prefixes are not allowed.

Note further that an arbitrary intersection of open sets is not necessarily an open set. Consider any infinite chain $C \subset T^{**}$ containing only finite sequences. Each $c_i \in C$ defines an open neighborhood (the set of all sequences with c_i as a prefix). The intersection of all of these open neighborhoods can only contain sequences that have every $c_i \in C$ as a prefix. There is only one such sequence, $\bigvee C$, so this intersection of open sets is a singleton set containing exactly one infinite sequence. This singleton set is not an open set.

Using our Scott topology, we can now define a notion of convergence of a sequence of sequences. Consider a sequence of sequences

$$S : \mathbb{N} \to T^{**}.$$

This sequence is said to converge to a sequence $a \in T^{**}$ if for all open sets A containing a, there exists an integer $n \in \mathbb{N}$ such that for all integers $m > n$, the

[2] For a poset $(X, \leq)$ and subset $A \subseteq X$, an **upper set** $\uparrow A$ of A is defined to be

$$\uparrow A = \{x \in X \mid a \leq x \text{ for some } a \in A\}.$$

That is, an upper set of A is the set of all upper bounds of elements of A.

Sidebar: Three Views of Convergence

There are several ways to think of limits of sequences of sequences. Here, we compare a least upper bound as a limit, a metric space limit, and a topological limit. Consider set $D = \{0,1\}$ and the CPO $A = D^{**}$, with $\sqsubseteq$, the prefix order. Consider the following sequence of sequences:

$$C = \{c_0, c_1, c_2, ...\} = \{(0,1), (0,0,1), (0,0,0,1), ...\}. \tag{4.1}$$

Note that no two elements of C are joinable, so the set itself does not have an upper bound, much less a least upper bound. However, in both a metric-space formulation and a topological formulation, this sequence converges to $c = (0,0,0,\cdots)$, an infinite sequence of zeros.

For a metric-space formulation, we can use the Baire distance d of Definition B.2. The sequence (4.1) converges in the metric space (A,d) to c. That is, for any real number $\epsilon > 0$, there is a $K \in \mathbb{N}$ such that for all $i > K$, $d(c_i, c) < \epsilon$. For any $i \in \mathbb{N}$,

$$d(c_i, c) = 1/(i+2).$$

Moreover, for all $j > i$, $d(c_j, c) < 1/(i+2)$. For any $\epsilon > 0$, there is an i such that $1/(i+2) < \epsilon$. Hence, the sequence converges to c in this metric space, even though it has no LUB in the prefix order.

The sequence (4.1) also converges in the Scott topology on A to $c = (0,0,0,\cdots)$. An open neighborhood anchored by a finite sequence $b \in A$ is the set of all sequences having prefix b. In a Scott topology, any union of such open neighborhoods is called an open set. The sequence $(c_0, c_1, c_2, \cdots)$ converges topologically to the infinite sequence $c = (0,0,0,\cdots)$ if for all open sets N containing c, there is a $K \in \mathbb{N}$ such that for all $i > K$, $c_i \in N$.

Every open set containing c includes at least one open neighborhood anchored by a finite prefix of c. Every finite prefix of c is a finite sequence of zeros. Specifically, let $b_0 = \bot, b_1 = (0), b_2 = (0,0)$, etc. Then an open neighborhood containing c is N_{b_j} for some $j \in \mathbb{N}$. For any such open neighborhood, let $K = j$, and note that $c_K \in N_{b_j}$ (because c_K has K leading zeros). Moreover, for all $i > K$, $c_i \in N_{b_j}$. Hence, the sequence converges topologically to c, even though it has no least upper bound.

following holds:

$$S(m) \in A.$$

Intuitively, this means that for any finite prefix $p \sqsubseteq a$, the sequences in S eventually all have prefix p.

In summary, in the Scott topology, for a sequence of sequences to converge to an infinite sequence, it is necessary for the sequences to grow without bound. They need to eventually produce every finite prefix of the infinite sequence.

This notion of convergence creates an interesting conundrum.

Definition 4.8. *A **convergent execution** of a process network is one that eventually produces every finite prefix of the least fixed point.*

A maximal execution, by contrast, is only required to produce some prefix of the least fixed point. A maximal execution does not necessarily converge to the least fixed-point.

Example 4.22: For the example in Figure 4.5, a maximal execution is permitted to avoid executing process B. In particular, the output s_3 of process true is an infinite sequence of boolean `true` tokens, so the output s_2 of process B is never needed and will never be consumed by the Select. But if B is capable of producing an infinite sequence, then the least fixed point solution for s_2 is an infinite sequence. Any execution that converges to this least fixed point solution will exhaust available memory because tokens produced by B will not be consumed.

Maximal and fair executions, however, do converge. For effective process networks, Geilen and Basten's execution strategy, described in Section 4.3.2, will deliver an effective execution, and that effective execution will converge to the least fixed point. For networks that are not effective, however, the strategy is less satisfactory because it may exhaust available memory. For such networks, Parks' algorithm may be preferable, even though it does not converge to the least fixed point.

4.5 Beyond Kahn-MacQueen

So far, we have defined Kahn-MacQueen processes in terms of their implementation using an imperative language with blocking read and nonblocking write operations. All functions defined this way are continuous, but there are continuous functions that cannot be defined this way. In this section, we focus on the properties of the *functions* realized by a component rather than on their implementation as a Kahn-MacQueen process. This opens up the possibility of a richer set of implementations, including those that internally are concurrent.

4.5.1 Sequential Functions

Prior to Kahn and MacQueen's work, Vuillemin (1973) formally defined a class of functions that exactly match those implementable as a Kahn-MacQueen process. He called a function in this class a **sequential function**.

Let X^m be the set of m-tuples of sequences in $X = T^{**}$, for some data type T. For an m-tuple $x \in X^m$, let x_i, $i \in \{1, \cdots, m\}$, be the i^{th} sequence in the tuple. Consider a function $F\colon X^m \to Y^n$.

Definition 4.9. *$F\colon X^m \to Y^n$ is **sequential** if it is continuous and for any $x \in X^m$, there exists an $i \in \{1, \cdots, m\}$, such that for all $x' \in X^m$ where $x \sqsubseteq x'$,*

$$x_i = x'_i \implies F(x) = F(x'). \tag{4.2}$$

Intuitively, at all times during an execution, there is an input channel (the i^{th} one) that blocks further output. This is just like the Kahn-MacQueen blocking read, but it does not require the process to be implemented in as a single imperative procedure.

Many useful functions, and even some trivial ones, however, are not sequential. An identity function with two or more input streams, for example, is not sequential.

Example 4.23: Consider a two-input, two-output identity function, $I_2\colon X^2 \to X^2$, where for all $x \in X^2$, $I_2(x) = x$. This is a rather trivial function. It is continuous, but not sequential. This identity function is not sequential because

there is no i satisfying (4.2). Extending either input will extend the output, by the definition of the function.

Consider the example shown in Figure 4.10, where A is an arbitrary source producing the sequence x_2. In the figure, $(x_1, x_2) = (y_1, y_2)$. Using this example, it is easy to see that there is no Kahn-MacQueen implementation of this identity function. Any blocking read on the upper port of I_2 will prevent the process from handling tokens on the lower port. And any implementation that does no reads at all of the upper port will not implement the identity function if the input x_1 is not empty.

This identity function, however, is continuous (and, hence, monotonic), so the program in Figure 4.10 is deterministic. The least fixed point for Figure 4.10 yields an empty sequence for x_1 and y_1 and $x_2 = y_2$ is whatever sequence A produces.

The sequential property is not compositional. The one-input, one-output identity function I_1 is sequential (trivially). However, a parallel composition of two such functions, shown in Figure 4.11 realizes the two-input, two-output identity function I_2, which is not sequential. On the other hand, any such composition of continuous functions is continuous, so continuity *is* a compositional property.

4.5.2 Stable Functions

Berry (1976) defined a larger class of functions than sequential functions that he called **stable functions**.

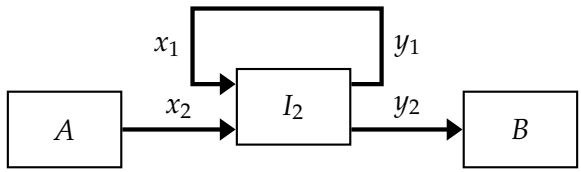


Figure 4.10: An example illustrating that the two-input, two-output identity function is not sequential.

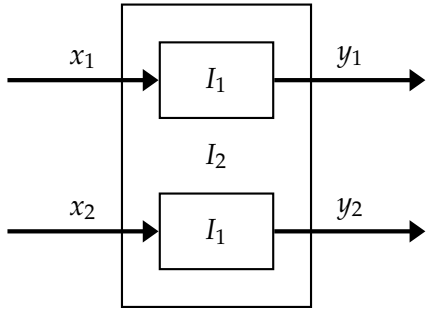


Figure 4.11: An example illustrating that sequential functions are not compositional.

Definition 4.10. *A function $F: X \to Y$ on a complete lower semilattice is **stable** if it is continuous and for all joinable sets $C \subseteq X$, $\hat{F}(C)$ is joinable and*

$$\bigwedge \hat{F}(C) = F(\bigwedge C). \tag{4.3}$$

Intuitively, if two possible inputs do not contain contradictory information (they can later be joined, i.e., they have a least upper bound), then neither will the two corresponding outputs. Sequential functions are stable, but not all stable functions are sequential. The two-input, two-output identify function on streams is (trivially) stable. But so are more interesting functions.

Example 4.24: Let $X = \{\bot, 0, 1\}$ with the **flat partial order**, where $\bot < 0$ and $\bot < 1$. The **Gustave function** (Berry, 1976) is the least function $G: X^3 \to X$ s.t.

$$\begin{aligned} G(0,1,\bot) &= 0 \\ G(1,\bot,0) &= 0 \\ G(\bot,0,1) &= 0 \end{aligned}$$

This example is easily extended to a function that operates on sequences in $\{0,1\}^{**}$ and hence can become a component in a Kahn network. Informally, such a function takes three sequences in. It can output a length-one sequence (0) if the head of the first two input sequences is 0 and 1, respectively, or the head of the

first and third sequence is 1 and 0, respectively, or the head of the second and third sequence is 0 and 1, respectively. For all other inputs, the output is an empty sequence. In each of these three cases, whether input tokens are available on the third stream is irrelevant. Hence, a blocking read will not work because it is impossible to tell which input to block on.

Note that although equation (4.3) looks quite like the definition of continuous, it uses the join rather than the meet. Not all functions that satisfy (4.3) are continuous. Consider a function $F : X \to Y$, where $X = Y = \mathbb{N} \cup \{\infty\}$, with the usual numerical ordering, given by

$$F(x) = \begin{cases} \bot_X = 0 & \text{if } x \text{ is finite} \\ x & \text{otherwise} \end{cases}$$

We can show that this function is not continuous. Consider the chain $C = \mathbb{N}$. This chain is joinable, where $\bigvee C = \infty$. In this case, $\hat{F}(C) = \{\bot_X\} = \{0\}$, which is obviously joinable, where $\bigvee \hat{F}(C) = \bot_X = 0$. But $F(\bigvee C) = F(\infty) = \infty$, which is not equal to $\bot_X$, so the function is not continuous.

However, we can show that for all joinable sets $C \subseteq X$, $\hat{F}(C)$ is joinable, and the function F does satisfy (4.3). Given a joinable set $C \subseteq X$, $\hat{F}(C)$ can only take on three possible values, $\{0\}$, $\{0, \infty\}$, or $\{\infty\}$, all three of which are joinable, so $\hat{F}(C)$ is joinable. The second constraint, that $\bigwedge \hat{F}(C) = F(\bigwedge C)$, is easy to check. Note that $\bigwedge C$ will be finite unless $C = \{\infty\}$. Hence,

$$F(\bigwedge C) = \begin{cases} 0 & \text{if } C \neq \{\infty\} \\ \infty & \text{otherwise} \end{cases}$$

For the right hand side, $\bigwedge \hat{F}(C)$, it is equal to the same expression,

$$\bigwedge \hat{F}(C) = \begin{cases} 0 & \text{if } C \neq \{\infty\} \\ \infty & \text{otherwise} \end{cases}$$

Stable functions like these represent a useful class of Kahn processes that react to patterns across their inputs in well-defined ways. They can be more difficult to implement, however, because an imperative process with blocking reads will not provide an implementation.

4.6 Dynamic Kahn Networks

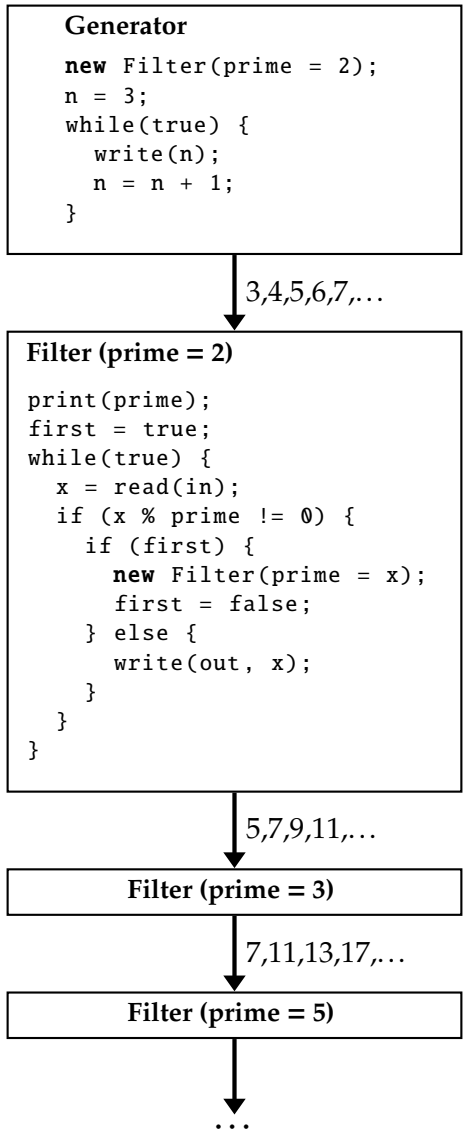


Figure 4.12: Sieve of Eratosthenes as a dynamically growing process network.

An interesting feature of Kahn networks is that it is possible to grow them dynamically. Kahn and MacQueen (1977) give an example implementing the **sieve of Eratosthenes** in a manner similar to that shown in Figure 4.12. The program starts with just the Generator process, which, on its first line, spawns a new Filter process and connects its output to the input of that process. The new Filter process accepts integer inputs and discards any that are multiples of of its `prime` parameter. The Generator process then starts feeding the Filter process increasing integers starting with 3.

When the Filter first encounters an integer that is not a multiple of its `prime` parameter, it has discovered a new prime number, so it spawns another downstream Filter to filter out multiples of that prime number. After that first encounter, it just forwards numbers that are not multiples of its `prime`. The chain of processes grows indefinitely as new prime numbers are discovered.

Semantically, this process network can be modeled as an infinite chain of processes. Each process realizes a continuous function, so the least fixed-point semantics still works. Using the strategy illustrated in Figure 4.3, we can construct a composite function with an infinite number of input and output streams. The composite function is continuous, so the program has a single, well-defined, infinite behavior given by the least fixed point of the composite function.

4.7 Nondeterministic Extensions of Kahn Networks

Determinism is the most notable feature of Kahn networks. But some applications benefit from nondeterministic mechanisms, for example, situations where components of a system appear and disappear for reasons unrelated to the computation, such as faults. It is possible to extend Kahn networks with nondeterministic mechanisms. The result is systems that are deterministic by default, with nondeterministic mechanisms judiciously introduced only exactly where needed. This contrasts with the techniques of Chapter 3, which are nondeterministic by default and leave it to the programmer to figure out how to achieve determinism when needed. It also contrasts with some stream processing formalisms that embrace nondeterminism from the start, as described for example by Stephens (1997) and Broy and Stefanescu (2001). A basic building block to judiciously introduce nondeterminism is the nondeterministic merge.

4.7.1 Nondeterministic Merge

A common extension to Kahn networks in a **nondeterministic merge**, a component with some number of input streams that nondeterministically merges tokens from those streams onto a single output stream. A Hewitt-Agha actor, as described in Section 3.2, can be modeled as single-input Kahn processes preceded by a nondeterministic merge.

A simple realization of a nondeterministic merge is given in Ptolemy II (Ptolemaeus, 2014, Chapter 4). There, a nondeterministic merge with n input ports spawns a thread for each port with an infinite loop that performs a blocking read on that port. When an input token arrives, the thread consumes the token, acquires a mutex lock, and writes the token to the output stream. When inputs arrive nearly simultaneously, the threads compete to acquire the mutex, and the resulting race determines the order in which output tokens are produced.

A nondeterministic merge is useful when upstream components may or may not be present. For example, their network connectivity may be sporadic, in which case, a deterministic Kahn node that reads their token would be blocked until connectivity is restored. The nondeterminstic merge only blocks one of its threads, leaving open the possibility of forwarding token from other input ports.

The introduction of such nondeterminism, however, undermines fundamental features of the semantics of the network. The nondeterministic merge can no longer be modeled as a function from a tuple of input streams to an output stream. Plotkin (1976) suggests instead a **powerdomain** construction, in which the nondeterministic merge is modeled as a function from a tuples of input streams into a set of allowable output streams:

$$F\colon X^n \to 2^X, \tag{4.4}$$

where n is the number of input streams, $X = T^{**}$ is the set of streams of tokens of type T, and 2^X is the set of all subsets of X. For a particular n-tuple $x \in X^n$ of input streams, $F(x)$ will be the set of all possible interleavings of the tokens in those streams.

Unfortunately, the powerdomain construction fails to distinguish some components that are different. Two components may be modeled by exactly the same function of the form of (4.4) and yet can exhibit significantly different behavior, as we will see next.

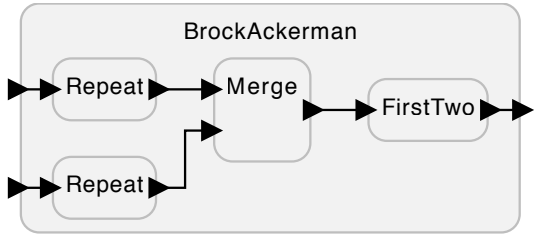


Figure 4.13: A system with two distinguishable implementations that realize the same powerdomain function.

4.7.2 The Brock-Ackerman Anomaly†

Brock and Ackerman (1981) give an example now known as the **Brock-Ackerman anomaly** of two nondeterministic stream operators that realize identical powerdomain functions of the form of (4.4) and yet can exhibit significantly different behavior.

Consider the system shown figure 4.13. This system has two inputs at the left that can accept streams of integer-valued tokens and one output at the right that produces a stream of integers. The subsystems labeled "Repeat" take each input integer and repeat it twice on their outputs. For example, if the input to the top Repeat is the sequence (1,2), then the output will be the sequence (1,1,2,2). The subsystem labeled "Merge" is a nondeterministic merge. For example, given the two input sequences (1,2) and (3,4), it can produce any of (1,2,3,4), (1,3,2,4), (1,3,4,2), (3,4,1,2), (3,1,4,2), or (3,1,2,4). If Merge receives nothing on one of its inputs, then it will simply produce whatever it receives on the other input. The subsystem labeled "FirstTwo" simply outputs the first two inputs it receives. For example, given (1,2,3,4), it will produce (1,2).

Brock and Ackerman (1981) then gave two subtly different realizations of the FirstTwo subsystem:

1. The first realization produces outputs as it receives inputs. That is, as soon as it sees a 1 on its input, it will produce 1 on its output.

† This section is adapted from Lee (2022).

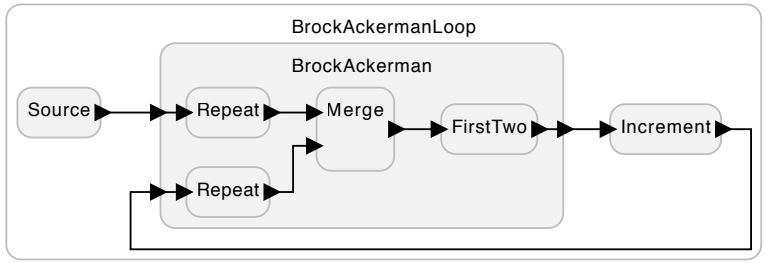


Figure 4.14: A use of the system in figure 4.13 where the two variants of the FirstTwo subsystem yield different behaviors.

2. The second realization waits until there are two inputs available before producing any output. That is, it will not produce any output until both 1 and 2 are available, at which point it will produce the sequence (1,2).

If you model this "BrockAckerman" as a function of the form (4.4), you will get exactly the same function for these two different realizations. For example, if the system is presented with inputs (5) and (6), i.e., two sequences of length one, the possible outputs for either realization are (5,5), (5,6), (6,5), and (6,6). The choice of realization has no effect on these possibilities.

Nevertheless, the two realizations yield different behaviors in some circumstances. Consider the system in figure 4.14. The subsystem labeled "Increment" simply adds one to each input. For example, given the input sequence (1,2), it will produce (2,3). In this usage, it makes a difference which of the two realizations of FirstTwo is used. Suppose that the subsystem labeled "Source" provides on its output the length-one sequence (5). Under realization (1) of FirstTwo, there are two possible outputs from the BrockAckerman subsystem, (5,5) and (5,6). But under realization (2), there is only one possible output, (5,5). The powerdomain construction fails to model this difference.

One way to understand the system in figure 4.14 is to think of the context of the "BrockAckerman" component as **interacting** with the component. It provides inputs that are based on observations of previous outputs via a **feedback** loop. This example illustrates that such interaction is fundamentally different from passive **observation**, where the inputs provided to a system do not depend on observations (see Wegner (1998) for an in-depth discussion of the power of interaction).

Hence, with this example, Brock and Ackerman (1981) proved that two systems that are indistinguishable by a passive observer cannot be substituted one for the other without possibly changing the behavior. They are not equivalent.

Note that the Brock-Ackerman system is not a Turing-Church computation because of the nondeterministic merge component. The Turing-Church theory admits no such nondeterminism. In Chapter 12 of Lee (2020), I show that a passive observer, one that can only see the inputs and outputs of a system, cannot tell the difference between such a nondeterministic system and a deterministic one (a deterministic one *would* be a Turing-Church computation). Only through interaction with the system is it possible to tell the difference. The feedback shown in Figure 4.14 is one form of such interaction. That argument depends on another celebrated result in computer science, Milner's concept of bisimulation, but this is beyond the scope of this book.

4.8 Rendezvous

In Section 4.3, we considered the use of blocking writes to keep buffers bounded. An extreme form of such blocking writes uses no buffer at all, requiring instead that the sending process **rendezvous** with the receiving process. A write to a stream blocks until a corresponding read from the stream is performed, and vice versa, a read from a stream blocks until a corresponding write is performed. Such rendezvous semantics underlie Hoare's **communicating sequential processes** (**CSP**) (Hoare, 1978) and Milner's **calculus of communicating systems** (**CCS**) (Milner, 1980).

By itself, a rendezvous-based framework for distributed computation is not Turing complete. Its memory use is bounded (at zero) by construction, and hence it cannot possibly implement an arbitrary Turing machine. Such frameworks rely instead on the programs that constitute the communicating processes for expressiveness.

It is possible to restrict rendezvous-based frameworks to be deterministic. To do so, each component in the system can be constrained to be a sequential imperative program that can rendezvous (read or write) with only one other process at a time. However, this restriction makes it quite difficult to construct useful programs. As a consequence, most such frameworks are nondeterministic. They

include primitives where a process can declare a willingness to rendezvous with any of several other processes at one step. With this, the order in which multiple rendezvous occur is nondeterministic. Most processes, therefore, cannot be modeled as functions except by using the powerdomain construction. But that construction fails to distinguish processes that may behave differently.

4.9 Conclusion

This chapter has developed a rigorous foundation for understanding process networks as a model of deterministic concurrency. Beginning with Kahn–MacQueen process networks, we showed how a concurrent system composed of sequential processes communicating through unbounded FIFO channels can nevertheless define a unique, deterministic semantics. The key insight is that such networks correspond to continuous functions on complete partial orders of streams, and their meaning is given by the least fixed point of a global functional representation of the network.

By framing process networks in terms of monotonicity, continuity, and least fixed points, we separated concerns of what a network means from how it is executed. This separation allows executions to be analyzed as approximations of a semantic object that may itself be infinite. The Kahn principle provides a powerful guarantee: any fair and maximal execution converges to the least fixed point, regardless of scheduling. This result explains why determinism can be preserved even in highly concurrent settings.

At the same time, the chapter has emphasized that execution is not merely a technical detail. Practical execution strategies must confront fundamental limitations arising from Turing completeness, including the undecidability of deadlock, termination, and bounded memory execution. Through a series of examples, we saw that fairness, maximality, and boundedness can be mutually incompatible, and that naïve demand-driven or data-driven policies are insufficient. More sophisticated strategies, such as those of Parks and of Geilen and Basten, demonstrate how effective execution can be achieved in many cases, while also clarifying the inherent tradeoffs.

We also broadened the model beyond imperative Kahn-MacQueen processes by characterizing components directly in terms of the functions they define. Sequen-

tial and stable functions generalize the original process model while preserving determinacy, revealing continuity as the essential compositional property. This functional perspective accommodates richer internal implementations, including concurrent ones, without sacrificing the semantic guarantees of process networks.

Finally, by introducing a notion of convergence based on the Scott topology, we made precise what it means for an execution to approach its semantic ideal. This notion explains why some maximal executions fail to converge and why convergence itself may be undesirable for networks that intrinsically require unbounded resources.

Taken together, these results establish process networks as a mathematically robust and practically relevant model of deterministic concurrency. They also expose the deep connections between concurrency, computation, and domain theory that will recur throughout the remainder of this book. Subsequent chapters build on these ideas to explore alternative models, richer forms of interaction, and different tradeoffs between expressiveness, analyzability, and implementability.

5 Dataflow

Broadly, the term **dataflow** (or **data flow**) is used in the literature for any model where each chunk of computation is (at least conceptually) performed when its input data are available. This contrasts with imperative models of computation, where the sequence of computational steps is explicitly specified. The Kahn-MacQueen process networks of the previous chapter combine imperative and dataflow models, in that each process is assumed to be specified imperatively, but processes stall to await availability of data.

In this chapter, we consider a purer form of dataflow originally attributed to Dennis (1974), who envisioned it as an alternative to the standard Von Neumann computer architecture. In Dennis dataflow, primitive operations, such as addition or multiplication, are enabled when their input data are available rather than when a program counter reaches an instruction specifying the operation. A program is given as a graph or network connecting such operations. Dennis called the primitive operations **actors**, but they are quite different from Hewitt-Agha actors, so we will use the more verbose term "dataflow actor".

Dennis envisioned dataflow actors to be primitive operations performed in hardware, like the instructions in a Von Neumann **instruction set architecture** (**ISA**). His work led to considerable experimentation with dataflow computer architectures (Arvind et al., 1991; Srini, 1986). Although such machines were capable, in principle, of exploiting parallel hardware more effectively, none of them were commercially successful. This may have been partly due to the lack of competitive programming models. The programming languages of the day were not easily compiled for execution on a dataflow machine, and languages developed specifically for the machines (Johnston et al., 2004) were slow to catch on.

The dataflow concept need not be restricted to the microarchitecture level. Dataflow actors can equally well be complex operations, even those specified using imperative code in conventional languages. As consequence, the connection with the Kahn networks of the previous chapter is quite strong. Dennis dataflow can be viewed as a special case of Kahn process networks (Lee and Parks, 1995).

The concepts behind dataflow have had practical realizations in stream languages (Stephens, 1997) and operating systems (such as Unix pipes). Interest in these models of computation was rekindled by the re-emergence of parallel computing

in the form of multicore architectures (Creeger, 2005). Dataflow models of computation have been explored for programming such parallel machines (Gordon et al., 2002), distributed systems (Lázaro Cuadrado et al., 2007; Olson and Evans, 2005; Parks and Roberts, 2003), and embedded systems (Lin et al., 2006; Jantsch and Sander, 2005). Considerable effort has gone into improved execution policies (Thies et al., 2005; Geilen and Basten, 2003; Turjan et al., 2003; Lee and Parks, 1995) and standardization (Object Management Group (OMG), 2007; Hsu et al., 2004).

In this chapter, we consider programs given as networks of dataflow actors and use the theory developed in the previous chapter to find conditions under which such a network is deterministic. We then consider special cases that yield to useful analysis and scheduling optimizations.

5.1 Firing and Firing Rules†

Let $\mathcal{S} = T^{**}$ be the set of finite or infinite sequences of tokens with some data type T.

Definition 5.1. *A **dataflow actor** with m inputs and n outputs is a pair (R, f), where*

1. *$R \subseteq \mathcal{S}^m$ is a non-empty set of m-tuples of* finite *sequences;*
2. *$f : \mathcal{S}^m \to \mathcal{S}^n$ is a (possibly partial) function defined at least on R;*
3. *The sequences in the tuple $f(\boldsymbol{r})$ are finite for every $\boldsymbol{r} \in R$;*
4. *for all $\boldsymbol{r}, \boldsymbol{r}' \in R$, if $\boldsymbol{r} \neq \boldsymbol{r}'$, then $\{\boldsymbol{r}, \boldsymbol{r}'\}$ does not have an upper bound in $(\mathcal{S}^m, \sqsubseteq)$.*

*We call each $\boldsymbol{r} \in R$ a **firing rule**, and f the **firing function** of the dataflow actor.*

The last condition is equivalent to the following statement: for any given m-tuple $\boldsymbol{s} \in \mathcal{S}$, there is at most one firing rule $\boldsymbol{r}$ in R such that $\boldsymbol{r}$ is a prefix of $\boldsymbol{s}$, $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$. If there were two such firing rules in R, then $\boldsymbol{s}$ would be an upper bound for these two, contradicting condition 4.

If $m = 0$, then R is the singleton set $\{\emptyset\}$, so condition 4 is trivially satisfied. If $n = 0$, then condition 3 is trivially satisfied.

†This section is based on Lee and Matsikoudis (2009).

5.1.1 Dataflow Processes

Let (R, f) be a dataflow actor with m inputs and n outputs. We can define a Kahn process $F : \mathcal{S}^m \to \mathcal{S}^n$ based on this actor, and a reasonable condition to impose is that for any $\boldsymbol{s} \in \mathcal{S}^m$,

$$F(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}).F(\boldsymbol{s}') & \text{if there exists } \boldsymbol{r} \in R \text{ such that } \boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'; \\ \lambda^n & \text{otherwise.} \end{cases} \tag{5.1}$$

Here, λ^n is an n-tuple of empty sequences. This is not a definition of F because F appears on both sides of the equality. The following, however, *is* a definition:

Definition 5.2. *A **dataflow process** corresponding to a dataflow actor (R, f) is the least function F: $\mathcal{S}^m \to \mathcal{S}^n$ that satisfies (5.1).*

This requires some elaboration. It is not obvious that such an F exists, nor, if it does exist, that it is unique, nor that any such F is continuous, and therefore defines a Kahn process. Although not obvious, all three of these statements are true.

First, for there to be a "least function," we need an order relation on *functions* from tuples of sequences to tuples of sequences rather than merely on tuples of sequences. Let $\mathcal{S}^m \to \mathcal{S}^n$ denote the set of all functions from $\mathcal{S}^m$ into $\mathcal{S}^n$. We can define a **prefix order** on this set of functions that is analogous to the pointwise order on tuples of sequences.

Definition 5.3. *The function $F : \mathcal{S}^m \to \mathcal{S}^n$ is a **prefix** of the function $G : \mathcal{S}^m \to \mathcal{S}^n$, written $F \sqsubseteq G$, if and only if $F(\boldsymbol{s}) \sqsubseteq G(\boldsymbol{s})$ for any m-tuple $\boldsymbol{s}$.*

With this definition $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$ is a poset. Moreover, it is a pointed poset. Its bottom element is the **empty function**

$$\boldsymbol{\Lambda}^n_m : \mathcal{S}^m \to \mathcal{S}^n \quad \text{such that for all } \boldsymbol{s} \in \mathcal{S}^m, \boldsymbol{\Lambda}^n_m(\boldsymbol{s}) = \lambda^n. \tag{5.2}$$

The bottom element is simply a function that yields an n-tuple of empty sequences for all arguments.

Proposition 5.1. *The ordered set $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$ is a CPO.*

Proof. This follows immediately from the fact that $(\mathcal{S}^n, \sqsubseteq)$ is a CPO. □

Now consider a function that operates on functions,

$$\phi : (\mathcal{S}^m \to \mathcal{S}^n) \to (\mathcal{S}^m \to \mathcal{S}^n).$$

Functions that operate on functions are sometimes called **functionals**.

Definition 5.4. *For a dataflow actor (R, f) with m inputs and n outputs, we can define a **dataflow actor functional** ϕ such that for any $F \in \mathcal{S}^m \to \mathcal{S}^n$ and any m-tuple $\boldsymbol{s}$,*

$$\phi(F)(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}).F(\boldsymbol{s}') & \text{if there exists } \boldsymbol{r} \in R \text{ such that } \boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'; \\ \lambda^n & \text{otherwise.} \end{cases} \tag{5.3}$$

Unlike (5.1), this *is* a definition because ϕ does not appear to the right of the equality.

Theorem 5.1. *ϕ is monotonic.*

Proof. Let F_1 and F_2 be arbitrary functions of type $\mathcal{S}^m \to \mathcal{S}^n$, and suppose that $F_1 \sqsubseteq F_2$. We need to show that $\phi(F_1) \sqsubseteq \phi(F_2)$.

For any $\boldsymbol{s} \in \mathcal{S}^m$, if there is a firing rule $\boldsymbol{r} \in R$ such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, then by condition 4 in definition 5.1, $\boldsymbol{r}$ is the only firing rule that is a prefix of $\boldsymbol{s}$, and hence $\phi(F_1)(\boldsymbol{s}) = f(\boldsymbol{r}).F_1(\boldsymbol{s}')$ and $\phi(F_2)(\boldsymbol{s}) = f(\boldsymbol{r}).F_2(\boldsymbol{s}')$, where $\boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'$. However, by assumption, $F_1(\boldsymbol{s}') \sqsubseteq F_2(\boldsymbol{s}')$ for any m-tuple $\boldsymbol{s}'$, and hence $\phi(F_1)(\boldsymbol{s}) \sqsubseteq \phi(F_2)(\boldsymbol{s})$.

If there is no firing rule that is a prefix of $\boldsymbol{s}$, then $\phi(F_1)(\boldsymbol{s}) = \lambda^n = \phi(F_2)(\boldsymbol{s})$.

In either case, $\phi(F_1)(\boldsymbol{s}) \sqsubseteq \phi(F_2)(\boldsymbol{s})$, and hence ϕ is monotonic. □

Theorem 5.2. *ϕ is continuous.*

Proof. Let $D \subseteq \mathcal{S}^m \to \mathcal{S}^n$ be directed in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$, and $\boldsymbol{s} \in \mathcal{S}^m$ an arbitrary m-tuple. We need to show that $\bigvee \hat{\phi}(D) = \phi(\bigvee D)$, where $\hat{(\phi)}$ is the lifted version of ϕ.

If there is a firing rule $\boldsymbol{r} \in R$ such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, then by condition 4 in definition 5.1, $\boldsymbol{r}$ is unique, and hence for every $F \in \mathcal{S}^m \to \mathcal{S}^n$, $\phi(F)(\boldsymbol{s}) = f(\boldsymbol{r}).F(\boldsymbol{s}')$, where $\boldsymbol{s} = \boldsymbol{r}.\boldsymbol{s}'$. Thus,

$$\begin{aligned}\bigvee \{\phi(F)(\boldsymbol{s}) \mid F \in D\} &= \bigvee \{f(\boldsymbol{r}).F(\boldsymbol{s}') \mid F \in D\} \\ &= f(\boldsymbol{r}).\bigvee \{F(\boldsymbol{s}') \mid F \in D\} \\ &= f(\boldsymbol{r}).(\bigvee D)(\boldsymbol{s}') \\ &= \phi(\bigvee D)(\boldsymbol{s}).\end{aligned}$$

Notice that since D is directed in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$, it has a least upper bound therein, $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$ being a CPO.

If there is no firing rule $\boldsymbol{r} \in R$ such that $\boldsymbol{r} \sqsubseteq \boldsymbol{s}$, for every $F \in \mathcal{S}^m \to \mathcal{S}^n$, $\phi(F)(\boldsymbol{s}) = \lambda^n$, and hence

$$\bigvee \{\phi(F)(\boldsymbol{s}) \mid F \in D\} = \lambda^n = \phi(\bigvee D)(\boldsymbol{s}).$$

In both cases,

$$\bigvee \{\phi(F)(\boldsymbol{s}) \mid F \in D\} = \phi(\bigvee D)(\boldsymbol{s}),$$

and hence ϕ is continuous.

□

Since ϕ is a continuous function over the CPO $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$, by the Kleene fixed-point theorem (proposition A.7), it has a least fixed point F in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$ which must satisfy (5.1).

Moreover, we can give a constructive procedure for finding that least fixed point. We can start with the least element in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$, the bottom function $\boldsymbol{\Lambda}$ given

in (5.2), and iterate ϕ to obtain the following sequence of functions:

$$\begin{aligned} F_0 &= \mathbf{\Lambda}_m^n, \\ F_1 &= \phi(F_0), \\ F_2 &= \phi(F_1), \\ &\ldots. \end{aligned} \tag{5.4}$$

Since ϕ is monotonic, and $\mathbf{\Lambda}_m^n$ is a prefix of every other function in $\mathcal{S}^m \to \mathcal{S}^n$, the set $\{F_0, F_1, F_2, \ldots\}$ is a chain and hence a directed in $(\mathcal{S}^m \to \mathcal{S}^n, \sqsubseteq)$. Thus, it has a least upper bound therein, and that least upper bound is is the least fixed point of ϕ, by the Kleene fixed-point theorem.

Let us examine this chain more closely for some fixed m-tuple $\boldsymbol{s}$. Suppose that there is some sequence of firing rules $(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots)$ such that $\boldsymbol{s} = \boldsymbol{r}_1.\boldsymbol{r}_2.\ldots$. Then, for this particular m-tuple, we can rewrite (5.4) in the following form:

$$\begin{aligned} \boldsymbol{s}_0 &= F_0(\boldsymbol{s}) = \lambda^n \\ \boldsymbol{s}_1 &= F_1(\boldsymbol{s}) = f(\boldsymbol{r}_1) \\ \boldsymbol{s}_2 &= F_2(\boldsymbol{s}) = f(\boldsymbol{r}_1).f(\boldsymbol{r}_2) \\ &\ldots \end{aligned} \tag{5.5}$$

This is an exact description of the operational semantics in Dennis dataflow for a single actor. Start with the dataflow actor producing only the empty sequence. Then find the prefix of the input that matches a firing rule, and invoke the firing function on that prefix, producing a partial output. Notice here that because of condition 4 in definition 5.1, no more than one firing rule can match a prefix of the input at any time. Then find the prefix of the remaining input that matches another firing rule, invoke the firing function on that prefix, and concatenate the result with the output.

In general, even when $\boldsymbol{s}$ is infinite, it is possible that there is only a finite sequence of firing rules $(\boldsymbol{r}_0, \ldots, \boldsymbol{r}_p)$ such that $\boldsymbol{s} = \boldsymbol{r}_0.\ldots.\boldsymbol{r}_p.\boldsymbol{s}'$, with $\boldsymbol{s}'$ having no prefix in R. In both the operational semantics of Dennis dataflow and a least fixed-point semantics, the firings simply stop, and the output is finite.

When $m = 0$ (there are no inputs), then $\mathcal{S}^m$ is a singleton set $\{\emptyset\}$. In this case, the least fixed point of ϕ is a source process, and if $\emptyset \in R$, then it produces the sequence $f(\emptyset).f(\emptyset).\cdots$. If $f(\emptyset)$ is non-empty, then the dataflow process produces a tuple of output sequences that is infinite and periodic. This might seem limiting

for dataflow processes that act as sources, but in fact it is not; a source with a more complicated output sequence can be constructed using a feedback composition, as we will see.

When $n = 0$ (there are no outputs), then the least fixed point of ϕ is a sink process, producing the sequence $\emptyset.\emptyset.\cdots$. We can define the concatenation of empty sets to be the empty set, $\emptyset.\emptyset.\cdots = \emptyset$. Note that this not the same as an empty sequence nor a tuple of empty sequences. It is no sequences at all.

In view of this perfect coincidence with the operational semantics, we are tempted to define a Kahn process based on the dataflow actor (R, f) as this least fixed point of ϕ. But in order to do this, we still need to prove that, in the general case, this least fixed point of ϕ is itself a continuous function, and thus a Kahn process. It suffices to prove the following theorem:

Theorem 5.3. *For any $F : \mathcal{S}^m \to \mathcal{S}^n$, if F is continuous, then $\phi(F)$ is also continuous for any dataflow actor functional ϕ.*

Proof. Let $F : \mathcal{S}^m \to \mathcal{S}^n$ be a continuous function, and $D \subseteq \mathcal{S}^m$ directed in $(\mathcal{S}^m, \sqsubseteq)$.

Suppose, toward contradiction, that there are $\boldsymbol{r}_1, \boldsymbol{r}_2 \in R$ and $\boldsymbol{s}_1, \boldsymbol{s}_2 \in D$ such that $\boldsymbol{r}_1 \neq \boldsymbol{r}_2$, but $\boldsymbol{r}_1 \sqsubseteq \boldsymbol{s}_1$ and $\boldsymbol{r}_2 \sqsubseteq \boldsymbol{s}_2$. Then since D is directed in $(\mathcal{S}^m, \sqsubseteq)$, $\{\boldsymbol{s}_1, \boldsymbol{s}_2\}$ has an upper bound in D, which is also an upper bound of $\{\boldsymbol{r}_1, \boldsymbol{r}_2\}$, in contradiction to condition 4 in definition 5.1.

Therefore, there is at most one $\boldsymbol{r} \in R$ that is a prefix any given tuple in D.

If there is such an $\boldsymbol{r} \in R$, then

$$\begin{aligned}
\bigvee \{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} &= \bigvee \{f(\boldsymbol{r}).F(\boldsymbol{s}') \mid \boldsymbol{r}.\boldsymbol{s}' \in D\} \\
&= f(\boldsymbol{r}).\bigvee \{F(\boldsymbol{s}') \mid \boldsymbol{r}.\boldsymbol{s}' \in D\} \\
&= f(\boldsymbol{r}).F(\bigvee \{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\}) \\
&= \phi(F)(\bigvee D).
\end{aligned}$$

Notice that since D is directed in $(\mathcal{S}^m, \sqsubseteq)$, $\{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\}$ is also directed in $(\mathcal{S}^m, \sqsubseteq)$, and in particular, $\boldsymbol{r}.\bigvee \{\boldsymbol{s}' \mid \boldsymbol{r}.\boldsymbol{s}' \in D\} = \bigvee D$.

If there is no such an $\boldsymbol{r} \in R$, there is no firing rule in R that is a prefix of some tuple in D, and hence

$$\bigvee \{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} = \lambda^n = \phi(F)(\bigvee D).$$

In both cases,

$$\bigvee \{\phi(F)(\boldsymbol{s}) \mid \boldsymbol{s} \in D\} = \phi(F)(\bigvee D),$$

and hence $\phi(F)$ is continuous.

□

Since $\mathbf{\Lambda}_m^n$ is trivially continuous, an easy induction suffices to see that the least fixed point of ϕ is a continuous function.

Note here that the firing function f need not be continuous. In fact, it does not even need to be monotonic. The continuity of the least fixed point of ϕ is guaranteed if (R, f) is a valid dataflow actor satisfying the conditions in definition 5.1.

Example 5.1: Consider a system where the set of token values (the data type) is the binary digits, $\mathbb{B} = \{0, 1\}$. Let us examine some possible sets $R \subset \mathcal{S}$ of firing rules for unary firing functions $f : \mathcal{S} \to \mathcal{S}$.

The following sets of firing rules all satisfy condition 4 in definition 5.1:

$$\begin{gathered} R = \{\lambda\}; \\ R = \{(0)\}; \\ R = \{(0), (1)\}; \\ R = \{(0,0), (0,1), (1,0), (1,1)\}. \end{gathered} \tag{5.6}$$

The first of these corresponds to a firing function that can fire infinitely regardless of the length of the input sequence and consumes no tokens. The second consumes only the leading zeros from the input sequence and then stops firing. The third consumes one token from the input on every firing, regardless of its value. The fourth consumes two tokens on the input on every firing, again regardless of the values.

The following example shows the firing rules that would be needed for two-input nondeterministic merge.

Example 5.2: Consider a two-input, one output actor operating on streams of type $\mathbb{B}$. A set of firing rules that does not satisfy condition 4 is:

$$R = \{(\lambda, (0)), (\lambda, (1)), ((0), \lambda), ((1), \lambda)\}. \tag{5.7}$$

Such firing rules would allow an actor to nondeterministically merge two input sequences.

Writing firing rules like those in the previous example is tedious and, for larger datatypes, impractical. We therefore rewrite using a **wildcard** $*$ as follows:

$$R = \{(\lambda, (*)), ((*), \lambda)\}. \tag{5.8}$$

The wildcard matches any token in the datatype of the actor inputs.

The firing rules in example 5.1 can all be realized in Kahn-MacQueen process wtih blocking reads. But dataflow actors are more general and need not be implementable using blocking reads.

Example 5.3: Consider the following firing rules:

$$R = \{((1), (0), \lambda), ((0), \lambda, (1)), (\lambda, (0), (1))\}. \tag{5.9}$$

These firing rules satisfy condition 4 in definition 5.1 and correspond to the Gustave function, a function defining a process which is stable but not sequential.

Given any dataflow actor conforming with definition 5.1, we now have a continuous function describing its input-output behavior over all firings. Any network of such actors can be rearranged as shown in Figure 4.2 yielding a composite function of form $F\colon \mathcal{S}^N \to \mathcal{S}^N$, where N is the total number of streams in the network. This function is a parallel composition of continuous functions and is therefore itself continuous. Consequently, F has a unique least fixed point, as explained in the previous chapter. We take that fixed point to be the semantics of the network. We have therefore demonstrated the following proposition:

Proposition 5.2. *Every network of dataflow actors is deterministic in the sense that it unique **denotational semantics** given by the least fixed point of a monotonic function on streams.*

We do not yet quite have an operational semantics because we have only talked about the execution of a single actor using the procedure (5.5). We have a bit more work to do.

5.1.2 Feedback and Stateful actors

The definition of a dataflow actor, so far, looks extremely limiting. The firing function f and firing rules are the same for each of a sequence of firings. This means that a dataflow actor's behavior cannot change over time. Many useful actors seem to be excluded by this model.

Example 5.4: A unit delay actor with type $\mathbb{B}$ produces an initial output and then subsequently forwards each input token to the output. This seems to require firing rules that do not satisfy condition 4 in definition 5.1:

$$R = \{\lambda, (*)\}. \tag{5.10}$$

The first firing would use the first rule, consuming no input and producing the initial output. Subsequent firings would use the second rule, consuming and forwarding an input token.

The unit delay actor in the previous example is deterministic. But such firing rules would also correspond to an actor that could nondeterministically consume or not consume an input token upon firing. Hence, allowing the firing rules of (5.10) opens the door to nondeterministic networks.

Fortunately, a small addition to the model vastly improves its expressiveness. All we need is to permit the connections between dataflow actors to have **initial tokens**.

Example 5.5: The unit delay can be implemented as a dataflow actor conforming to definition 5.1 using a feedback loop with an initial token, as shown in Figure 5.1. The type of the feedback path, in this case, can be

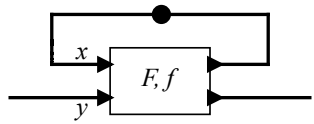


Figure 5.1: Initial token on a feedback loop enables stateful behavior.

boolean, $\mathbb{B}$. The firing rules are

$$R = \{((1), \lambda), ((0), (*))\}.$$

The firing function, which has form $f : \mathcal{S}^2 \to \mathcal{S}^2$, is given by

$$f(x, y) = \begin{cases} ((0), (i)) & \text{if } x_0 = 1 \\ ((0), (y_0)) & \text{otherwise,} \end{cases}$$

where x_0 and y_0 denote the first token in the upper and lower input streams, respectively and i denotes the initial output of the unit delay.

This example is easily generalized so that any dataflow actor can have state. The initial state is the value of the initial token on a feedback connection.

Example 5.6: For the unit delay actor, the feedback connection in Figure 5.1 represents a state variable that keeps track of whether a firing is the first firing. The initial token has value $i = 1$, and all subsequent values on this feedback path are 0.

Using such a feedback path with an initial token, the firing function can be completely different for each value of the feedback token as long as the conditions in definition 5.1 are satisfied. The feedback path can have an arbitrary data type and therefore can represent the state of the actor, with the initial token giving the initial state.

5.1.3 Fixed Point Semantics with Initial Tokens

The initial tokens that we have added are not taken into account in the demonstration of determinism in proposition 5.2. In this section, we adapt the fixed point semantics to allow initial tokens.

Consider an arbitrary network of dataflow actors with zero or more initial token on any connection from the output of one actor to the input of another. Rearrange the actors as shown in Figure 5.2 to form a composite function of form $F: \mathcal{S}^N \to \mathcal{S}^N$, where N is the total number of streams in the network. This is nearly identical to what we did with Kahn networks in Figure 4.2, with the exception that there is now an initial token on the connection from B to A. As before, the F function is a parallel composition of continuous functions and hence is itself continuous. But

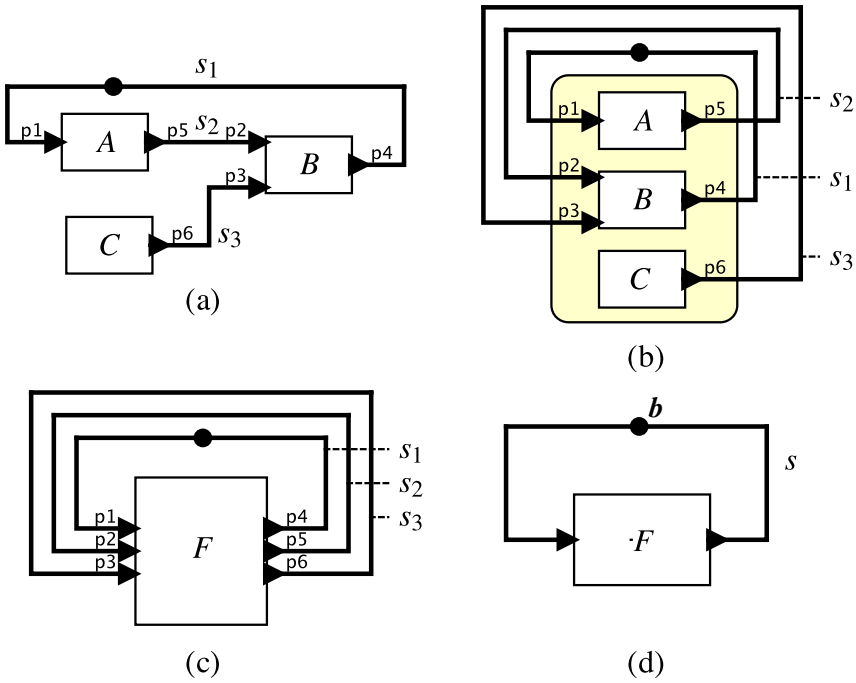


Figure 5.2: Fixed point semantics adapted to allow initial tokens.

now, assume that the N feedback paths are initialized with an N-tuple $\boldsymbol{d}$ of finite sequences, each containing zero or more initial tokens.

Let the semantics of the network be defined to be the N-tuple $\boldsymbol{s}$ of sequences coming out of F. Unlike Figure 4.2, the input to F is no longer the same as its output $\boldsymbol{s}$ because of the initial tokens. The network, therefore, defines $\boldsymbol{s}$ as an N-tuple satisfying

$$\boldsymbol{s} = F(\boldsymbol{d}.\boldsymbol{s}),$$

where $\boldsymbol{d}.\boldsymbol{s}$ is the concatenation of the sequences in $\boldsymbol{d}$ and $\boldsymbol{s}$. With this model, we are no longer looking for a fixed point of F.

Define a new function $F_{\boldsymbol{d}}: \mathcal{S}^N \to \mathcal{S}^N$ as follows:

$$F_{\boldsymbol{d}}(\boldsymbol{s}) = F(\boldsymbol{d}.\boldsymbol{s}). \tag{5.11}$$

With this definition, we seek a fixed point $\boldsymbol{s}$ of $F_{\boldsymbol{d}}$, where

$$\boldsymbol{s} = F_{\boldsymbol{d}}(\boldsymbol{s}).$$

Proposition 5.3. *If F is continuous, then so is $F_{\boldsymbol{d}}$ for any N-tuple of finite sequences $\boldsymbol{d}$.*

Proof. Let C be a chain in $\mathcal{S}^N$. Note that

$$\bigvee \hat{F}_{\boldsymbol{d}}(C) = \bigvee \hat{F}(C_{\boldsymbol{d}}),$$

where $\hat{F}_{\boldsymbol{d}}$ is the lifted version of $F_{\boldsymbol{d}}$, and

$$C_{\boldsymbol{d}} = \{\boldsymbol{d}.\boldsymbol{c} \mid \boldsymbol{c} \in C\}.$$

Because C is a chain, so is $C_{\boldsymbol{d}}$. Because F is continuous,

$$\bigvee \hat{F}(C_{\boldsymbol{d}}) = F(\bigvee C_{\boldsymbol{d}}).$$

But now,

$$\bigvee C_{\boldsymbol{d}} = \boldsymbol{d}.\bigvee C,$$

so

$$\bigvee \hat{F}_{\boldsymbol{d}}(C) = F(\boldsymbol{d}.\bigvee C) = F_{\boldsymbol{d}}(\bigvee C),$$

which means that $F_{\boldsymbol{d}}$ is continuous. □

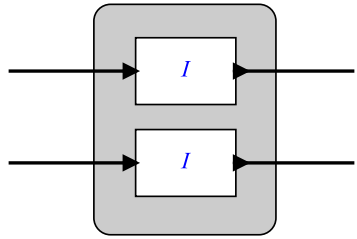


Figure 5.3: A two-input two-output identity process described as an aggregation of two one-input one-output identity processes.

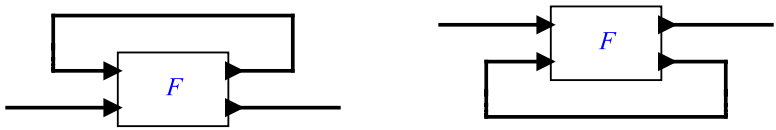


Figure 5.4: If F is an identity process, the appropriate firing rules are (5.7).

Because $F_{\boldsymbol{d}}$ is continuous, it has a unique least fixed point, and hence we have established the following:

Proposition 5.4. *A network of dataflow actors with initial tokens is deterministic in same sense as in proposition 5.2.*

If we assume here is some sequence of firing rules $(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots)$ such that $\boldsymbol{d}.\boldsymbol{s} = \boldsymbol{r}_1.\boldsymbol{r}_2\ldots$, then exactly the same constructive procedure as in (5.5) will work.

5.1.4 Commutative Firings

While dataflow actors conforming to definition 5.1 yield continuous Kahn processes, condition 4 is more restrictive than what is really necessary. The firing rules in (5.7), for example, are not only the firing rules for the dangerous nondeterminate merge, but also the firing rules for a perfectly harmless two-input two-output identity process.

Example 5.7: Let $I\colon \mathcal{S} \to \mathcal{S}$ be a simple identify function. We can define a dataflow actor with firing rule $R = \{(*)\}$ and firing function $f\colon \mathcal{S} \to \mathcal{S}$ given

by,

$$f(\boldsymbol{s}) = (s_0),$$

where s_0 is the first token in $\boldsymbol{s}$. Suppose now that we want to define a dataflow actor that is the parallel composition of two such identity actors, as shown in Figure 5.3. At first glance, it might seem that this sort of identity process could be implemented using the following firing rules:

$$R = \{((*),(*))\}.$$

With these firing rules, however, the two-input, two-output identity function would only produce outputs when it can consume one token from each input. The two examples in Figure 5.4 show why this will not work. In the first example, the first (top) input and output should be the empty sequence under the least-fixed-point semantics, so there will never be a token to trigger any firing rule. In the second example, the second (bottom) input and output have the same problem. The firing rules of (5.7), however, have no difficulty with these cases.

To overcome this limitation, we can replace condition 4 with a more general rule.

Definition 5.5. *A **commutative dataflow actor** is a dataflow actor according to definition 5.1, but with rule 4 replaced by the following more elaborate condition:*

4 *for all* $\boldsymbol{r}, \boldsymbol{r}' \in R$, *if* $\boldsymbol{r} \neq \boldsymbol{r}'$ *and* $\{\boldsymbol{r}, \boldsymbol{r}'\}$ *has an upper bound in* $(\mathcal{S}^m, \sqsubseteq)$, *then* $f(\boldsymbol{r}).f(\boldsymbol{r}') = f(\boldsymbol{r}').f(\boldsymbol{r})$ *and* $\boldsymbol{r} \bigwedge \boldsymbol{r}' = \lambda^m$.

This revised condition states that if any two firing rules are consistent, namely they have a common upper bound, and therefore can possibly be enabled at the same time, then it makes no difference in what order we use these firing rules; the values of the firing function at these consistent rules commute with respect to the concatenation operator. Furthermore, any two consistent firing rules have no common prefix other than the m-tuple of empty sequences

Definition 5.5 makes our dataflow model more **compositional** (Talcott, 1996), in that an aggregation of dataflow actors is itself a dataflow actor The two-input, two-output identify is perhaps the simplest example illustrating the compositionality problem. One-input one-output identity processes are trivially described as

dataflow actors that satisfy definition 5.1, but a two-input two-output identity process cannot be so described.

It is easy to see that when condition 4 in definition 5.5 is satisfied,

$$\boldsymbol{r} \vee \boldsymbol{r}' = \boldsymbol{r}.\boldsymbol{r}' = \boldsymbol{r}'.\boldsymbol{r}; \tag{5.12}$$

that is, the least common extension (least upper bound) of any two consistent firing rules is their concatenation, in either order.

To prove determinism with this generalization, we need to reconstruct the functional that we used to define the Kahn process. Let $P_R(\boldsymbol{s})$ denote the set $\{\boldsymbol{r} \in R \mid \boldsymbol{r} \sqsubseteq \boldsymbol{s}\}$. This is a possibly empty finite set. The functional ϕ' is defined such that for any function $F : \mathcal{S}^m \to \mathcal{S}^n$ and any m-tuple $\boldsymbol{s}$,

$$\phi'(F)(\boldsymbol{s}) = \begin{cases} f(\boldsymbol{r}_1).\cdots.f(\boldsymbol{r}_p).F(\boldsymbol{s}') & \text{if } P_R(\boldsymbol{s}) \neq \emptyset \text{ and } \{\boldsymbol{r}_1,\ldots,\boldsymbol{r}_p\} = P_R(\boldsymbol{s}); \\ \lambda^n & \text{otherwise.} \end{cases}$$

Here, we assume, as before, that $\boldsymbol{s}'$ is defined so that $\boldsymbol{s} = \boldsymbol{r}_1.....\boldsymbol{r}_p.\boldsymbol{s}'$. Notice that because of (5.12), for any permutation π on $\{1,\ldots,p\}$,

$$\boldsymbol{r}_1.....\boldsymbol{r}_p = \boldsymbol{r}_{\pi(1)}.....\boldsymbol{r}_{\pi(p)},$$

and similarly, because of condition 4 in definition 5.5,

$$f(\boldsymbol{r}_1).\cdots.f(\boldsymbol{r}_p) = f(\boldsymbol{r}_{\pi(1)}).\cdots.f(\boldsymbol{r}_{\pi(p)}).$$

Therefore, it makes no difference in what order we invoke the enabled firing rules.

As before, we define the Kahn process F for the commutative dataflow actor (R, f) to be the least fixed point of the functional ϕ'. It is straightforward to extend the results on ϕ to conclude that both the functional ϕ' and its least fixed point F are continuous; the proofs are practically identical, albeit notationally more complicated.

Example 5.8: For the two-input, two-output identify function of Figure 5.3, we see that we can use the firing rules of (5.7), and a firing function $f : \mathcal{S}^2 \to \mathcal{S}^2$ such that for any firing rule $\boldsymbol{r} \in R$, $f(\boldsymbol{r}) = \boldsymbol{r}$. This gives us a commutative dataflow actor for the two-input two-output identity.

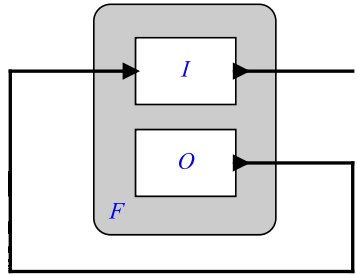


Figure 5.5: A composition that does not form a dataflow actor but forms a commutative dataflow actor.

Example 5.9: More interestingly, we can use the same firing rules (5.7) to implement a commutative dataflow actor with two inputs and one output of type $\mathbb{B}$ and firing function $f : \mathcal{S}^2 \to \mathcal{S}$ such that for each firing rule $\boldsymbol{r} \in R$,

$$f(\boldsymbol{r}) = \begin{cases} (1) & \text{if } \boldsymbol{r} = ((1), \lambda) \text{ or } \boldsymbol{r} = (\lambda, (1)); \\ \lambda & \text{otherwise.} \end{cases}$$

This actor outputs a token with value 1 for each input token with value 1 regardless of which port receives the input. Such a process is a version of what called a **parallel or** because it realizes a logical disjunction that is tolerant of unknown inputs. As soon as either input is known to be *true*, it outputs *true*. Contrast this with the logical or operator in many programming languages, often written `x || y`, which evaluates `x` and, only if `x` is *false*, evaluates `y`. No output is produced until `x` can be evaluated.

The resulting Kahn process is interesting because it is neither sequential nor stable, and yet it is continuous.

Example 5.10: As a final example, consider the composition of Figure 5.5. The top process is an identity process, and the bottom one a source of the infinite sequence $(0, 0, \ldots)$. How can we define the firing rules R and firing function f for the composition?

A first, naive attempt would be to let $R = \{(*)\}$. However, with the feedback arc in Figure 5.5, this results in no firing rule ever becoming enabled. Instead, we need $R = \{(*), \lambda\}$, which violates condition 4 in definition 5.1. However, if we define the firing function such that

$$f((0)) = ((0), \lambda),$$
$$f((1)) = ((1), \lambda), \text{ and}$$
$$f(\lambda) = (\lambda, (0)),$$

then condition 4 of definition 5.5 is satisfied and the composition forms a commutative dataflow actor.

The previous example is easily generalized to show that for any parallel composition of dataflow actors, like that shown in Figure 5.2(b), we can define firing rules and a firing function, based on those of the original dataflow actors, that satisfy definition 5.5.

Any network of dataflow actors can be restructured into a parallel composition with feedback, as illustrated in Figure 5.2. Any such network can be modeled as a single commutative dataflow actor with feedback, possibly containing initial tokens. We now have (finally!) a candidate operational semantics. The function F in Figure 5.2(d) can be realized by a simple firing function that invokes any of the firing functions of the component actors that is enabled. This firing function is (trivially) commutative. Repeated execution of this firing function provides a constructive procedure for finding the least fixed point.

In plain language, a network of dataflow actors can be executed using a very simple policy. Invoke the firing function of any **enabled actor** (one that has sufficient inputs). Repeat. Stop only if and when there are no more enabled actors. This procedure will always yield a prefix of the denotational semantics, but whether it converges in any sense to the denotational semantics depends on the choices made when more than one actor is enabled. This is the scheduling problem.

5.1.5 Scheduling

The constructive procedure given by (5.5) ensures that repeated firings always yield a prefix of the least fixed point semantics. Whether the procedure *converges* to the least fixed point semantics depends on the same notions of convergence considered in Section 4.4 of the previous chapter and on scheduling decisions.

In the previous chapter, we assumed that a Kahn process had its own thread of execution that would forge ahead until blocked on a read (or write, if blocking writes are used). For dataflow actors, however, the execution is divided into firings. Rather than devoting a thread to each actor, we could have some number of **worker threads** invoke firing functions for various actors according to some scheduling policy. Finding reasonable scheduling policies turns out to be a sophisticated problem.

As we saw in the previous chapter, seemingly reasonable techniques like data-driven execution and demand-driven execution can result in undesirable executions. It may even be preferable to choose schedules that fail to converge to the least fixed point semantics or executions that fail to be fair.

The same effective execution methods considered in Section 4.3.2 of the previous chapter work for dataflow actors. Instead of blocking writes, however, a scheduler just chooses to not invoke the firing function of an actor whose output buffers are full. However, for dataflow actors, it is often possible to provide much more sophisticated schedulers. The rest of this chapter is devoted to considering special cases of dataflow networks that yield to such sophisticated scheduling techniques.

5.2 Synchronous Dataflow

Synchronous dataflow (**SDF**), also called **static dataflow**,[1] was introduced by Lee and Messerschmitt (1987b) and is implemented, among other places, in Ptolemy II (Lee et al., 2014). SDF is a simple special case of dataflow where each actor firing consumes and produces a fixed number of tokens. The questions of whether buffers are bounded and whether executions are finite become decidable, and optimized static scheduling becomes possible, albeit sometimes intractable.

Example 5.11: A simple SDF actor consumes N input tokens with numerical values and outputs one token containing their average. A scheduler simply needs to ensure that the upstream actor that provides the tokens executes enough times that the required N input tokens are available when this averaging actor fires.

A further special case is the **homogeneous SDF** model, where a dataflow actor fires consumes exactly one token on each of its input ports and produces exactly one token on each output port. In this case, the scheduler simply has to ensure that each actor fires after the actors that supply it with data. An iteration of the model consists of exactly one firing of each actor in some data-precedence-constrained order.

As with more general dataflow models, SDF allows initial tokens on connections between actors.

Definition 5.6. *A **complete iteration** of a network of SDF actors, if it exists, is a finite non-zero number of firings that begins and ends with each connection having the same number of tokens as its initial number of tokens.*

When an iteration exists, the model can be execute infinitely with bounded buffers and the scheduling problem becomes a bounded **job-shop scheduling** problem,

[1] The term "synchronous dataflow" can cause confusion because it is not synchronous in the sense of SR, considered in Chapter 6. There is no global clock in SDF models, and actors are fired asynchronously. For this reason, some authors prefer the term "static dataflow." This does not avoid all confusion, however, because Dennis (1974) had previously coined the term "static dataflow" to refer to dataflow graphs where buffers could hold at most one token. Since there is no way to avoid a collision of terminology, we stick with the original "synchronous dataflow" terminology used in the literature. The term SDF arose from a signal processing concept, where two signals with sample rates that are related by a rational multiple are deemed to be synchronous.

which is well studied in the literature. To find an iteration, we write and solve balance equations, as we explore next.

5.2.1 Balance Equations

Consider a single connection between two actors, A and B, as shown in Figure 5.6. The notation here means that when A fires, it produces M tokens on its output port, and when B fires, it consumes N tokens on its input port. M and N are non-negative integers. Suppose that A fires q_A times and B fires q_B times. All tokens that A produces are consumed by B if and only if the following **balance equation** is satisfied,

$$q_A M = q_B N. \tag{5.13}$$

Given values q_A and q_B satisfying (5.13), the system remains in balance; A produces exactly as many tokens as B consumes.

Suppose we wish to process an arbitrarily large number of tokens, a situation that is typical of streaming applications. A naive strategy is to fire actor A an arbitrarily large number q_A times, and then fire actor B q_B times, where q_A and q_B satisfy (5.13). This strategy is naive, however, because it requires storing an arbitrarily large number of unconsumed tokens in a buffer. A better strategy is to find the smallest non-negative q_A and q_B that satisfy (5.13) and are not both zero. Then we can construct a schedule that fires actor A q_A times and actor B q_B times, and we can repeat this schedule as many times as we like without requiring any more memory to store unconsumed tokens. That is, we can achieve an **unbounded execution** (an execution processes an arbitrarily large number of tokens) with bounded buffers (buffers with a bound on the number of unconsumed tokens). In each complete iteration, actor B consumes exactly as many tokens as actor A produces.

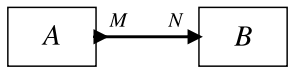


Figure 5.6: SDF actor A produces M tokens when it fires, and actor B consumes N tokens when it fires.

Example 5.12: Suppose that in Figure 5.6, $M = 2$ and $N = 3$. There are many possible solutions to the corresponding balance equation, one of which is $q_A = 3$ and $q_B = 2$. With these values, the following schedule can be repeated forever:

$$A, A, A, B, B.$$

An alternative schedule could also be used:

$$A, A, B, A, B.$$

The latter schedule has an advantage in that it requires less memory for storing intermediate tokens; B fires as soon as there are enough tokens, rather than waiting for A to complete its entire cycle.

Another solution to (5.13) is $q_A = 6$ and $q_B = 4$. This solution includes more firings in the schedule than are strictly needed to keep the system in balance.

The equation is also satisfied by $q_A = 0$ and $q_B = 0$, but if the number of firings of actors is zero, then no useful work is done. Clearly, this is not a solution we want. Negative solutions are also not meaningful.

A reasonable strategy finds the least integer solution to the balance equations that is not all zero and constructs a schedule that fires the actors in the model the requisite number of times, given by this solution.

In a more complicated SDF model, every connection between actors results in a balance equation. Hence, the model defines a system of equations, and finding the least integer solution is not entirely trivial.

Example 5.13: Figure 5.7 shows a network with three SDF actors. The connections result in the following system of balance equations:

$$\begin{aligned} q_A &= q_B \\ 2q_B &= q_C \\ 2q_A &= q_C. \end{aligned}$$

The least positive integer solution to these equations is $q_A = q_B = 1$, and $q_C = 2$, so the following schedule can be repeated forever to get an unbounded

execution with bounded buffers,

$$A, B, C, C.$$

The balance equations do not always have a non-trivial solution, as illustrated in the following example.

Example 5.14: Figure 5.8 shows a network with three SDF actors where the only finite solution to the balance equations is the trivial one, $q_A = q_B = q_C = 0$. A consequence is that there is no unbounded execution with bounded buffers for this model. It cannot be kept in balance.

Definition 5.7. *An SDF model that has a non-zero finite solution to the balance equations is said to be* ***consistent****. If the only solution is zero, then it is* ***inconsistent****.*

An inconsistent model has no unbounded execution with bounded buffers.

We can compactly represent the balance equations in matrix form. Define a **connectivity matrix** Γ that has one row for each connection and one column for each actor. The entry $\Gamma_{i,j}$ at row i and column j has a non-negative number p if actor j produces p tokens on connection i and $-c$ if actor j consumes c tokens from connection i. Then define the vector $\boldsymbol{q}$ to represent the number of firings of each actor and the vector $\boldsymbol{o}$ to be a zero vector, with one zero for each connection. The balance equations can then be written in compact form,

$$\Gamma \boldsymbol{q} = \boldsymbol{o} \,. \tag{5.14}$$

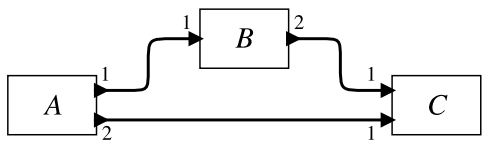


Figure 5.7: A consistent SDF model.

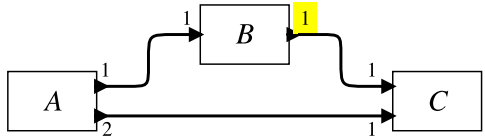


Figure 5.8: An inconsistent SDF model.

Example 5.15: For the dataflow network in Figure 5.7, the balance equations can be written

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o} \tag{5.15}$$

A solution to the balance equations is

$$\begin{bmatrix} q_A \\ q_B \\ q_C \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

Example 5.16: For the dataflow network in Figure 5.8, the balance equations differ from the previous equations in only one number, $\Gamma_{2,2}$:

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o} \tag{5.16}$$

This equation has no nontrivial solution.

Note that in general the Γ matrix need not be square because the number of connections need not equal the number of actors. Moreover, any permutation of the rows of Γ yields an identical system of equations because the numbering of the connections is arbitrary. You can also permute the columns, but then you must also permute the elements of the $\boldsymbol{q}$ vector.

5.2.2 Solving the Balance Equations

Each balance equation has the form

$$q_A p = q_B c,$$

where p and c are non-negative integers. For a connected graph, a straightforward way to solve these equations is to pick one actor, say A, and set $q_A = 1$. Then

$$q_B = q_A p/c = p/c.$$

This will be a positive rational number. Then, for all other equations involving B, solve for the other similarly. Continue until either you have a rational solution for all q variables or you reach a contradiction, where the other side already has an incompatible solution. If you reach a contradiction, then there is no nontrivial solution and the dataflow graph is inconsistent. If you do not reach a contradiction, and graph is connected, then you have a rational solution for all q variables. Now find the least common multiple (LCM) of the denominators of the rational solutions and multiply all q variables by this LCM. You will now have the smallest positive integer solution to the balance equations. If the graph is disconnected, then repeat the procedure for each connected component. Note that if any actor produces or consumes zero tokens on a connection, this is equivalent to not having the connection in the graph.

Example 5.17: Consider the dataflow graph shown in Figure 5.9 (we will consider the initial tokens later). The production and consumption parameters are shown next to each port. The balance equations can be

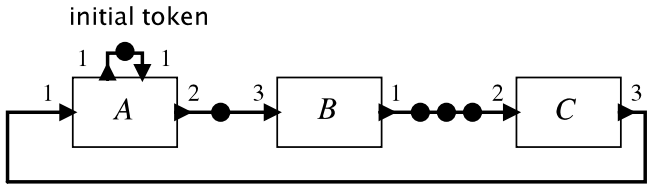


Figure 5.9: A cyclic SDF model.

written,

$$\begin{bmatrix} -1 & 0 & 3 \\ 0 & 0 & 0 \\ 2 & -3 & 0 \\ 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Notice that the second row, which represents the self loop on actor A represents a connection in which A produces one token and consumes one token, for a net production or consumption of zero tokens.

We can solve the balance equations using the procedure outlined above. Start with $q_A = 1$. The connection from A to B requires the following:

$$q_A = 1 \quad \Rightarrow \quad q_B = 2/3$$

The connection from B to C requires the following:

$$q_B = 2/3 \quad \Rightarrow \quad q_C = 2/6 = 1/3$$

The connection from C to A requires the following:

$$q_C = 1/3 \quad \Rightarrow \quad q_A = 1,$$

so the graph is consistent. The LCM of the denominators is 3, so multiplying through by 3 we get

$$q_A = 3, \quad q_B = 2, \quad q_C = 1.$$

5.2.3 Properties of the Connectivity Matrix

In linear algebra, a collection of vectors is **linearly independent** if no vector in the set is a linear combination of the other vectors in the set. Equivalently, vectors $v_1, v_2, \cdots, v_N$ are **linearly dependent** if there exist non-zero numbers $q_1, q_2, \cdots, q_N$ such that

$$q_1 v_1 + q_2 v_2 + \cdots + q_N v_N = \boldsymbol{o}, \tag{5.17}$$

where $\boldsymbol{o}$ is the zero vector.

Example 5.18: The following set X of vectors is linearly independent, whereas Y is linearly dependent:

$$X = \left\{ \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix} \right\} \quad Y = \left\{ \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix} \right\}.$$

To see that Y is linearly dependent, note that the negative of the sum of the second two vectors is equal to the first. To see that X is linearly independent, we can use the above procedure to attempt to find a non-zero solution to (5.17) for these vectors. We will reach a contradiction.

The balance equations are identical to equation (5.17); the vectors $v_1, v_2, \cdots, v_N$ are the columns of Γ. Hence, a non-trivial solution to the balance equations exists if and only if the columns of the connectivity matrix Γ are linearly dependent.

In linear algebra, the number of linearly independent columns of a matrix is called the **rank** of the matrix. This is also equal to the number of linearly independent rows, so the rank of the matrix cannot exceed the lesser of the number of columns and the number of rows. Hence, the rank of the connectivity matrix Γ is less than or equal to the smaller of the number of actors a and the number of connections c. If the graph is **connected**, then $c \geq a - 1$.

Proposition 5.5. *The rank of the connectivity matrix Γ of a connected dataflow graph is either $a-1$ or a, where a is the number of actors.*

Proof. For $a = 2$, the Γ_a matrix will have two columns, so the rank cannot be bigger than 2. If there is only one connection, it will have one row, be consistent, and the rank will be $a - 1 = 1$. If there is more than one connection, then the rank will still be one if the graph is consistent and two if it is not.

For arbitrary $a > 2$, we can use induction. Assume the rank of Γ_a is a or $a - 1$. If we add one more actor and a single connection to it from any preexisting actor, Γ_{a+1} will have one more column, so the rank cannot get larger than larger than $a + 1$. The new connectivity matrix will have the form,

$$\Gamma_{a+1} = \left[\begin{array}{c|c} \Gamma_a & \boldsymbol{o} \\ \hline \boldsymbol{\rho}^T & \end{array} \right],$$

where $\boldsymbol{o}$ is a zero column vector and $\boldsymbol{\rho}^T$ is a row vector representing the added connection. This row vector will have a non-zero entry in the last column, and hence this new row vector must be linearly independent of all rows above it, all of which have a zero in the last column. Hence, the addition of the new actor and single connection increases the rank of the matrix by at least one. Addition of more connections to the new actor may further increase the rank, but not to more than $a+1$. Hence, the rank of Γ_{a+1} must be either a or $a+1$. □

Proposition 5.6. *A connected dataflow graph with a actors is consistent if and only if the rank of its connectivity matrix is equal to $a-1$.*

Proof. If the rank of Γ is a, then all a columns are linearly independent and hence there cannot be any nontrivial solution to the balance equations. If it is not a, then by proposition 5.5 it must be $a-1$. □

Example 5.19: The X vectors in Example 5.18 form the columns of a connectivity matrix for an inconsistent dataflow graph with three actors. The Y vectors form the columns of a connectivity matrix for an consistent dataflow graph. We leave it as an exercise to construct these graphs.

5.2.4 Dynamics of Execution

Given a dataflow graph with a actors and c connections, suppose we wish to construct a sequential schedule. I.e., we want to fire one actor at a time, possibly forever. Suppose $\boldsymbol{v}$ is a **one-hot** column vector of dimension a that is all zero except for a one in position n, where n is the index of an actor. Then a sequential schedule can be represented as a list of such vectors,

$$\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots .$$

Define another column vector $\boldsymbol{b}$ of dimension c that contains nonnegative integers giving the number of token on the buffers for each connection. Let $\boldsymbol{b}_0$ represent the

number of initial tokens on each connection. Then the execution of the sequential schedule has the following effect on the buffers:

$$\boldsymbol{b}_{n+1} = \boldsymbol{b}_n + \Gamma \boldsymbol{v}_n \tag{5.18}$$

for any $n \geq 0$. The iteration in (5.18) can be thought of as a **symbolic execution** of the dataflow program. It doesn't actually invoke any firings, but rather emulates the effect of such firings on the buffers.

Definition 5.8. *A **periodic admissible sequential schedule** (**PASS**) of length $K > 0$ is a sequence*

$$\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_{K-1}$$

such that $\boldsymbol{v}_n \geq \boldsymbol{o}$ for each $n \in \{0, \cdots, K-1\}$, and

$$\boldsymbol{b}_K = \boldsymbol{b}_0 + \Gamma(\boldsymbol{v}_0 + \cdots + \boldsymbol{v}_{K-1}) = \boldsymbol{b}_0.$$

A PASS is a finite schedule that brings buffers back to their initial sizes, and hence it can be repeated indefinitely with bounded memory. Let $\boldsymbol{q} = \boldsymbol{v}_0 + \cdots + \boldsymbol{v}_{K-1}$, and note that a PASS requires that $\Gamma \boldsymbol{q} = \boldsymbol{o}$. Hence, a necessary condition for the existence of a PASS is that the balance equations have a non-zero solution. Hence, a PASS can only exist for a consistent SDF graph.

It is easy now to see how to systematically construct a PASS. First, solve the balance equations to find $\boldsymbol{q}$. Then begin a series of steps where you select an **enabled actor** and append it to the schedule. An enabled actor i is one that has not yet been scheduled $\boldsymbol{q}_i$ times and has sufficient tokens on its input buffers to satisfy at its firing rules. If there is no enabled actor, stop. Otherwise, for the selected actor, construct the one-hot vector and update the buffer sizes using (5.18). The algorithm terminates when every actor i has been appended to the schedule $\boldsymbol{q}_i$ times, in which case a PASS has been found, or no actor is enabled, in which case the program deadlocks.

Example 5.20: Figure 5.10 shows an SDF model with initial tokens on a feedback loop. The balance equations are

$$\begin{aligned} 3q_A &= 2q_B \\ 2q_B &= 3q_A. \end{aligned}$$

The least positive integer solution exists and is $q_A = 2$, and $q_B = 3$, so the model is consistent. With four initial tokens on the feedback connection, as shown, the following schedule is a PASS that can be repeated forever,

$$A, B, A, B, B.$$

This schedule starts with actor A because at the start of execution, only actor A can fire. Actor B does not have sufficient tokens. When A fires, it consumes three tokens from the four initial tokens, leaving one behind. It sends three tokens to B. At this point, only B can fire, consuming two of the three tokens sent by A, and producing two more tokens on its output. At this point, actor A can fire again because there are exactly three tokens on its input. It will consume all of these and produce three tokens. At this point, B has four tokens on its input, enabling two firings. After those two firings, both actors have been fired the requisite number of times, and the buffer on the feedback arc again has four tokens. The schedule has therefore returned the dataflow graph to its initial buffer sizes.

Were there any fewer than four initial tokens on the feedback path, the model would deadlock. If there were only three tokens, for example, then A could fire, followed by B, but neither would have enough input tokens to fire again.

At each step of this scheduling procedure, unlike the previous rather trivial example, there is usually more than one enabled actor. This offers opportunities for optimization. For example, you might pick actors in an attempt to minimize buffer memory or to give the most compact schedule (see the sidebar on page 142). These optimization problems can be quite complex.

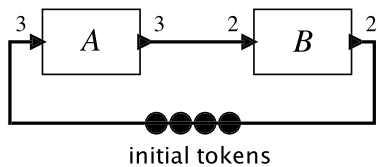


Figure 5.10: An SDF model with initial tokens on a feedback loop.

Example 5.21: Compact disks (CDs) and many digital encodings of audio sample audio signals at 44.1 kilosamples per second. In 1987, a new standard was introduced for digital audio tapes (DATs) that deliberately used a different sample rate, 48 kilosamples per second, in a desperate and ultimately futile attempt to discourage unauthorized copying of digital audio data. Digitally converting a CD to DAT requires sophisticated signal processing that is easy today, but was expensive at the time. The conversion can be realized by a series of four digital filters, each of which convert their input to a new sample rate.

A realization of such a series is shown in Figure 5.11. The first filter, labeled *A*, doubles the sample rate. The second downsamples its input to 2/3 the sample rate. The third upsamples by 8/7. And the final one downsamples by 5/7. If the input sample rate is 44,100 samples per second, the output sample rate is 48,000 samples per second.

Solving the balance equations for this SDF graph yields a smallest integer solution,

$$\boldsymbol{q}^T = [147, 147, 98, 28, 32, 160].$$

A schedule that minimizes buffer memory is this one (Bhattacharyya et al., 1996b):

```
ABABCABCABABCABCDEAFFFFFBABCABCABABCDEAFFFFFBCABABCABCABABCD
EAFFFFFBCABABCABCDEAFFFFFBABCABCABABCABCDEAFFFFFBABCABCABABC
DEAFFFFFBCABABCABCABABCDEAFFFFFEBCAFFFFFBABCABCDEAFFFFFBABCAB
CABABCABCDEAFFFFFBABCABCABABCDEAFFFFFBCABABCABCABABCDEAFFFFF
BCABABCABCDEAFFFFFBABCABCABABCABCDEAFFFFFBABCABCABABCDEAFFFF
FEBCAFFFFFBABCABCABABCDEAFFFFFBCABABCABCDEAFFFFFBABCABCABABC
ABCDEAFFFFFBABCABCABABCDEAFFFFFBCABABCABCABABCDEAFFFFFBCABAB
```

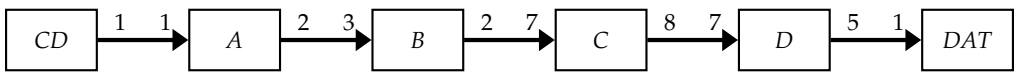


Figure 5.11: An SDF model that converts a CD sample rate signal to a DAT sample rate signal.

```
CABCDEAFFFFFBABCABCABABCABCDEAFFFFFEBAFFFFFBCABCABABCDEAFFFFF
BCABABCABCABABCDEAFFFFFBCABABCABCDEAFFFFFBABCABCABABCABCDEAF
FFFFBABCABCABABCDEAFFFFFBCABABCABCABABCDEAFFFFFBCABABCABCDEF
FFFFEFFFFF
```

This schedule is complex to find an quite chaotic.

For SDF graphs, as a consequence of the methods above, it is **decidable** whether the program can be executed with bounded memory, and it is decidable whether the program halts (deadlocks). Thus, SDF, as a model of computation, is less expressive than dataflow in general and less expressive than Kahn process networks. It is not Turing complete.

5.2.5 Parallel Scheduling

The procedure outlined in Section 5.2.4 can be adapted to formulate a parallel scheduling problem, where dataflow actors are mapped onto parallel computing resources. One strategy is to use the **symbolic execution** of (5.2.4) to construct an **acyclic precedence graph** (**APG**) rather than directly constructing a schedule. You can then use standard job-shop scheduling techniques on that graph.

To construct an APG, when performing the symbolic execution of (5.18), for each token recorded in the $\boldsymbol{b}$ vector, keep track of whether it is an initial token and, if not, which firing produced the token. When encountering an enabled actor, instead of adding it to a list schedule, add it to a graph and add directed arc from each firing that produced at least one of the tokens it consumes. When the procedure terminates, you will have a graph that represents a finite number of firing and the dependencies between them.

Example 5.22: Consider the dataflow graph shown in Figure 5.9. As before, we solve the balance equations to obtain

$$q_A = 3, \quad q_B = 2, \quad q_C = 1.$$

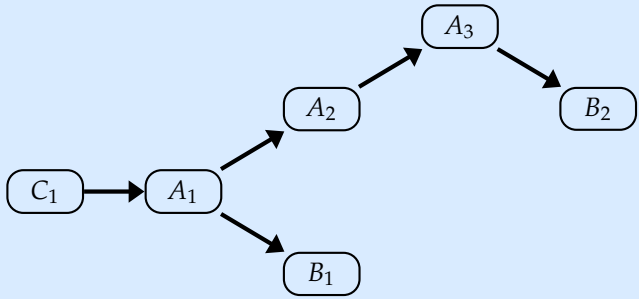
Initially, the only enabled actor is C and the tokens it consumes are all initial tokens. So we add C to the APG with no incoming arcs. Upon firing C, A becomes enabled and will consume tokens that were produced by the firing

Sidebar: SDF Schedulers

An advantage of using SDF is that there may be many possible schedules for a given model, including some that execute actors in parallel. In this case, actors in the dataflow graph can be mapped onto different processors in a multicore or distributed architecture for improved performance. Lee and Messerschmitt (1987a) adapt classical job-shop scheduling algorithms (Coffman, 1976), particularly those introduced by Hu (1961), to SDF by converting the SDF graph into an acyclic precedence graph (**APG**). Lee and Ha (1989) classify scheduling strategies into **fully dynamic scheduling** (all scheduling decisions are made at run time), **static assignment scheduling** (all decisions except the assignment to processors are made at run time), **self-timed scheduling** (only the timing of an actor firing is determined at run time), and **fully-static scheduling** (every aspect of the schedule is determined before run time). Sih and Lee (1993a) extend the job-shop scheduling techniques to account for interprocessor communication costs (see also Sih and Lee (1993b)). Pino et al. (1994) show how to construct schedules for heterogeneous multiprocessors. Falk et al. (2008) give a parallel scheduling strategy based on clustering and demonstrate significant performance gains for multimedia applications.

In addition to parallel scheduling, other scheduling optimizations are useful (see Bhattacharyya et al. (1996b) for a collection of these). Ha and Lee (1991) relax the constraints of SDF to allow data-dependent iterative firing of actors (a technique called quasi-static scheduling). Bhattacharyya and Lee (1993) develop optimized schedules for iterated invocations of actors (see also Bhattacharyya et al. (1996a)). Bhattacharyya et al. (1993) optimize schedules to minimize memory usage and later apply these optimizations to code generation for embedded processors (Bhattacharyya et al., 1995). Murthy and Bhattacharyya (2006) collect algorithms that minimize the use of memory through scheduling and buffer sharing. Geilen et al. (2005) show that model checking techniques can be used to optimize memory. Stuijk et al. (2008) explore the tradeoff between throughput and buffering (see also Moreira et al. (2010)). Sriram and Bhattacharyya (2009) develop scheduling optimizations that minimize the number of synchronization operations in parallel SDF.

of C. Hence, we add A to the graph with an arc from C to A. Continuing in this manner, we obtain the following APG:



The previous example has an interesting subtlety. Two of the firings of A depend on the previous firing of A because of the self loop on A. Such a self loop represents state. B has no such self loop, and hence the second firing of B does not depend on the first firing. These two firings can, in principle, execute in parallel. This would not be safe if actor B were to have any internal state.

As a consequence, even though self loops representing state have no effect on the balance equations, it is important to include them in a model to represent dependencies introduced by state variables.

Once an APG has been constructed, building a parallel schedule becomes a job-shop scheduling problem. This is a well studied problem with a vast literature giving many algorithms for approximating optimal solutions depending on what you wish to optimize and what you know about firings and communication overhead (see sidebar on page 142).

A common optimization criterion that is used is to minimize the **makespan**, defined to be the time at which all executions in the APG have completed. Constructing such an optimal schedule requires knowing the execution times of the firings. With bounded processing resources, this is a well-known NP-complete problem, meaning that the best known optimal algorithms have complexity that is an exponential function of the number of nodes in the graph. For small and medium sized graphs, brute-force solutions that find the optimal schedule may be tractable, but it is relatively rare to know precise execution times, so the value of such schedules may be questionable.

Example 5.23: For the APG constructed in the previous example, assume that the execution times are all one time unit. We can easily construct the minimum makespan schedule. A two-processor schedule with the first processor executing $C, A1, A2, A3, B2$ and the second executing $B1$ (starting at time 2) achieves the minimum makespan of five.

This example is simple enough that we can consider all possible schedules. It also becomes clear that there is no value in having more than two processors and that the minimum achievable makespan is five (it is the total execution time of the longest path in the graph). But most problems are not so forgiving, so heuristics may be necessary.

One classic heuristic for job-shop scheduling is the **Hu level scheduling** strategy (Hu, 1961). In this strategy, each node in the APG is assigned a **level**, which is the length of the longest upstream chain. The schedule then consists of executing all nodes with level 0 first in any order. When these are complete, then execute all nodes with level 1. Etc. This strategy ensures that, regardless of execution times, all precedences will be satisfied. The strategy rarely yields the minimum makespan, although in the previous example, with everything having unit execution time, it will.

Example 5.24: In the APG of the previous example, C_1 has level 0, A_1 has level 1, A_2 and B_1 have level 2, A_3 has level 3, and B_2 has level four. The same schedule as before results from the Hu level scheduling algorithm.

Many more sophisticated algorithms have developed over time. For example, Sih and Lee (1993b) give a heuristic that takes into account the time that communication between processors takes and contention for limited communication resources.

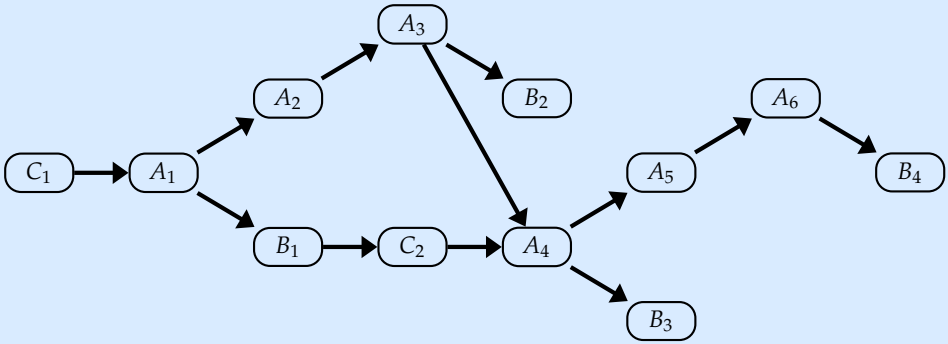
One limitation of the scheduling methods described so far is that they consider only a single complete iteration of an SDF graph. Overlapping of these iterations could yield overall faster execution. One way to do this is to consider bigger complete iterations. In other words, instead of choosing the smallest positive integer solution $\boldsymbol{q}$ to the balance equations, choose a larger solution $K\boldsymbol{q}$, where $K > 1$ is an integer **vectorization factor**.

Example 5.25: Define the **throughput** to be the number of minimal complete iterations per unit time. With the minimum makespan schedule for Figure 5.9 that we constructed in the previous examples, the throughput is one complete iteration for every five time units. Can we do better?

An alternative solution to the balance equations is

$$q_A = 6, \quad q_B = 4, \quad q_C = 2.$$

An APG for this vectorization factor of two is given below:



The minimum makespan for this APG is 8, yielding a throughput of 1/4, which is better than the previous schedule, which yielded a throughput of 1/5.

In principle, for the example in Figure 5.9, the limit on throughput is is imposed by the self loop on A. In the limit, a throughput of 1/3 should be achievable. However, no finite vectorization factor will yield this throughput.

5.3 Boolean Controlled Dataflow

The limited expressiveness of SDF can be quite limiting at times. Harmless programs that are easy to schedule often do not conform with the SDF model.

Example 5.26: Consider a decision-making program, shown in Figure 5.12, that uses a stream of boolean conditions c generated by actor C to choose whether to forward the output of actors A or B to D. This is not an SDF program, though it is not hard to see how to schedule it.

Sidebar: StreamIt

Thies et al. (2002) give a textual programming language, **StreamIt**, based on SDF and intended for use with streaming data applications such as multimedia. Software components (called **filters** rather than actors) produce and consume fixed amounts of data. The language provides compact structured constructs for common patterns of actor composition, such as chains of filters, parallel chains of filters, or feedback loops.

A key innovation in StreamIt is the notion of a **teleport message** (Thies et al., 2005). Teleport messages improve the expressiveness of SDF by allowing one actor to sporadically send a message to another; that is, rather than sending a message on every firing, only some firings send messages. The teleport message mechanism nonetheless ensures determinism by ensuring that the message is received by the receiving actor in exactly the same firing that it would have if the sending actor had sent messages on every firing. But it avoids the overhead of sending messages on every firing. This approach models a communication channel where tokens are sometimes, but not always, produced and consumed. But it preserves the determinism of SDF models, where the results of execution are the same for any valid schedule.

Sidebar: Multidimensional Synchronous Dataflow

Murthy and Lee (2002) describe **multidimensional SDF** (**MDSDF**). Whereas a channel in SDF carries a sequence of tokens, a channel in MDSDF carries a multidimensional array of tokens. That is, the history of tokens can grow along multiple dimensions. This model is effective for expressing certain kinds of signal processing applications, particularly image processing, video processing, radar and sonar.

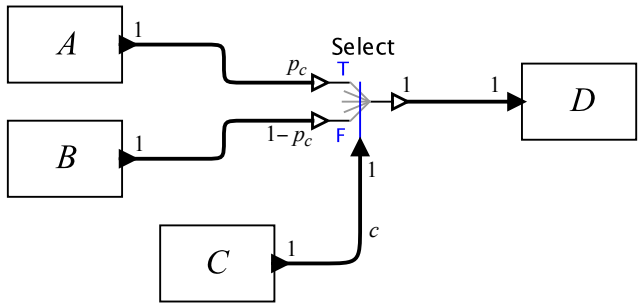


Figure 5.12: A decision-making program that is easily handled with BDF.

This example uses a **Select** actor similar to the Select process used in the previous chapter. Its firing rules are

$$R = \{((*), \lambda, (true)), (\lambda, (*), (false)).$$

If the third input (the bottom one in the figure) has a *true*-valued token, then it consumes one token from the first input (the top one, labeled 'T') and nothing from the second input. If the third input has a *false*, then it consumes from the 'F' input.

I have previously shown (Lee, 1991) that the balance equations can be generalized to handle actors like Select. Let c represent the boolean token consumed by the Select. Let p_c represent the probability that this token is *true*. Then the system will be balanced on average if the following balance equations have a non-trivial solution:

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & 0 & 0 & -p_c & 0 \\ 0 & 1 & 0 & p_c - 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \\ q_S \\ q_D \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o} \tag{5.19}$$

Here, the consumption of the Select on its T and F ports is represented symbolically by $-p_c$ and $-(1-p_c)$. It is easy to see that these balance equations are satisfied by

$$\boldsymbol{q}^T = (p_c, 1-p_c, 1, 1, 1). \tag{5.20}$$

This is a positive solution, but not an integer solution, in general.

Dataflow programs where production and consumption numbers are given in terms of probabilities for boolean variables are called **boolean dataflow** (**BDF**) programs. Buck (1994) has further generalized this idea to allow for the production and consumption numbers to be given in terms of probability mass functions for integer-valued variables. He called the resulting graphs **integer dataflow** (**IDF**) programs.

How to construct a schedule for BDF programs is addressed in the PhD theses of Ha (1992) and Buck (1993). For this example, if we use the solution (5.20) to the balance equations, then constraining p_c to be either zero or one gives an integer repetition vector $\boldsymbol{q}$. From (5.18), it is easy to see that a schedule constructed using either zero or one for p_c will yield a complete iteration. Thus, we only need to construct two schedules, one for the outcome where c is *true* ($p_c = 1$) and one for the outcome where c is *false* ($p_c = 0$):

$$S_0 = (C, B, S, D), \quad S_1 = (C, A, S, D).$$

For situations where exactly one of each of N controlling booleans is produced in each iteration, and these booleans are not stored on buffers for use in future iterations, we would need, in the worst case, 2^N schedules.

In many cases, the schedules can be compactly represented as a **quasi-static schedule**, where some firings are annotated with a boolean expression indicating whether they will occur in any particular iteration. For this example, the following will work:

$$S = (C, c?A, !c?B, S, D),$$

where $c?A$ means that A fires if c is *true* and $!c?B$ means that B fires if c is *false*. However, not all programs yield to such a simple treatment.

Example 5.27: Consider the more problematic program shown in Figure 5.13. The balance equations can be written:

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & 0 & 0 & -p_c \\ 0 & 1 & 0 & p_c - 1 \\ 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \\ q_S \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o} \tag{5.21}$$

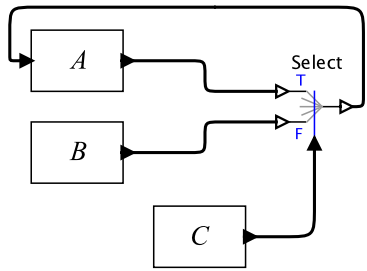


Figure 5.13: A problematic decision-making program.

This Γ matrix has full rank (rank four) for all values of p_c except $p_c = 1$. Hence, the system is in balance only if C only produces *true*-valued tokens. But in that case, the program immediately deadlocks. In fact, it is assured of deadlocking unless the very first output of C is *false*.

Much more interesting programs sometimes yield to such a quasi-static scheduling strategy.

Example 5.28: Consider the program of Example 4.21 and Figure 4.9, which produces the Hamming numbers, an ordered sequence of integers of the form $2^n3^m5^k$, where n, m and k are non-negative integers. These numbers can be produced by a BDF program.

The only nontrivial actor in Figure 4.9 is the OrderedMerge, which merges two streams of increasing numbers into one stream of increasing numbers. This can be implemented as shown in Figure 5.14. Actors A and B provide increasing sequences of numbers and C consumes the merged increasing sequence. The **MinMax** and **Xor** actors here are ordinary SDF actors. MinMax consumes one integer-valued token from each input port, outputs the larger of the two on the max output port and the smaller on the min output port. It does not matter what it does if they are equal. On the swapped output port, it produces a *true* if the lower input went to the max port and a *false* otherwise. The Xor actor takes one boolean-valued token from each input and outputs the exclusive or of the two. The initial token at the output of

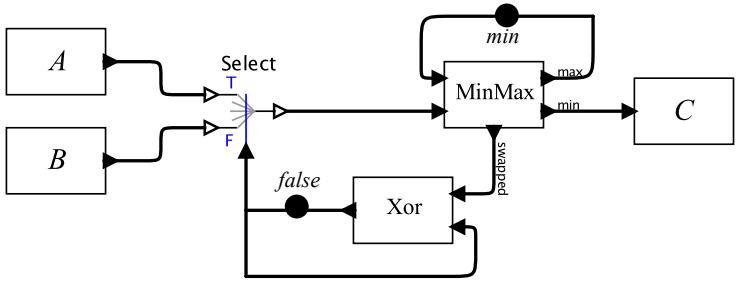


Figure 5.14: A BDF implementation of an ordered merge towards a program generating the Hamming numbers.

Xor has value *false*. The initial token on the upper feedback path has value *min*, the smallest integer value allowed, or $-\infty$ if that is allowed by the data type.

This program can be kept in balance by a quasi static schedule. Let x represent the value of the boolean token at the output of Xor at the beginning of each iteration. Then the following schedule provides a complete iteration:

$$S = (x?A, !x?B, \text{Select}, \text{MinMax}, C, \text{Xor}).$$

However, the full program in Figure 4.9 that generates the Hamming numbers does not have the same structure as Figure 5.14. The output of the ordered merge is fed back to B, thereby giving it structure more like Figure 5.13. The program in Figure 4.9 has no bounded quasi-static schedule.

Analyzing programs like that in Figure 5.14 can get arbitrarily complicated. Here, the boolean token that is used in the schedule starts with an initial token. In general, the notion of a complete iteration may need to be augmented to not only restore buffers to their initial sizes, but also to restore boolean tokens stored across iterations to their initial values, though this is not needed in this example. Logic operations on the boolean tokens, such as the Xor in the figure, can be taken into account using symbolic algebra. When Switch and Select are used to conditionally route streams of boolean tokens, they function as logic operations themselves that

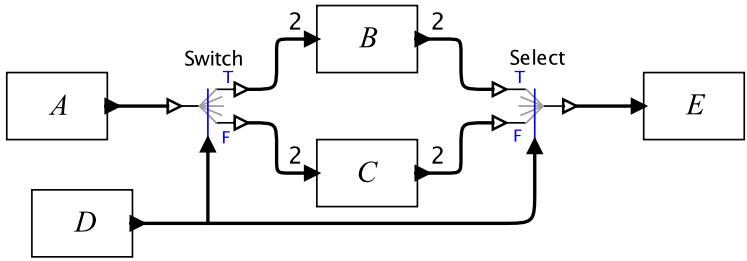


Figure 5.15: A BDF program that has no finite complete cycle.

can be analyzed. See Buck and Lee (1992, 1993) for more illustrations of such analysis.

Unlike SDF, for BDF, having a non-trivial solution to the balance equations does not provide assurance that, given enough initial tokens, there exists a finite complete iteration. Consider the example in Figure 5.15. Except for the Switch and Select, all are SDF actors that, unless otherwise indicated, produce or consume a single token on each port. But actors B and C each consume and produce two tokens per firing. The symbolic balance equations have a solution where all actors fire twice while B fires p times and C $(1-p)$ times, where p is the probability of a *true* token from D. Suppose that the boolean sequence produced by D is (*true*, *false*, *false*, *false*, $\cdots$). There is no finite number of firings that form a complete iteration, and hence there is no quasi-static schedule.

Whether a BDF program can be kept in balance without deadlocking is, in general, undecidable (Buck, 1993). This is easy to see by recognizing that the stack program given in Figure 4.7 of the previous chapter is, in fact, a BDF program. The fact that implements a stack means that its buffers cannot be bounded. In fact, there are many programs that will not yield to quasi-static scheduling and will not remain balanced.

An alternative to constructing a quasi-static schedule is to use a run-time scheduler, which makes all scheduling decisions at runtime based on buffer status and firing rules. An example implementation is given in Ptolemy II (Lee et al., 2014), where it is called **dynamic dataflow** (**DDF**). As with Kahn process networks, we can apply Parks' algorithm to ensure that such a scheduler delivers a bounded memory schedule when that is possible. Such a strategy would work well for the

examples in Figures 5.14 and 5.15. All the same convergence questions apply to dynamic dataflow in a manner very similar to process networks, as discussed in Section 4.4.

In summary, the symbolic balance equations can be used to analyze, optimize, and schedule some programs, but not all. We next consider another way to improve the expressiveness of SDF without admiting undecidable problems.

5.4 Modal Dataflow

A disadvantage of SDF is that every actor must produce and consume a fixed amount of data; the production and consumption rates cannot depend on the data. BDF relaxes this constraint, but its quasi-static schedules can be difficult to find. DDF further relaxes the constraint, abandons static scheduling and static analysis for deadlock or boundedness. A number of variants of dataflow, however, are more expressive than SDF but still amenable to some forms of static analysis.

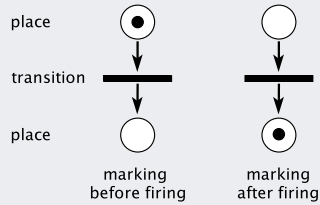
In this section, we outline several models, all of which conform to a simple pattern. Actors in these models are **modal**, meaning that they have more than one mode of operation. In each mode, the behavior of an actor is SDF; it produces and consumes a fixed number of tokens. But these numbers can differ across modes. The differences between the modal dataflow models we present here concern primarily how the switching between modes is governed.

5.4.1 Cyclo-Static Dataflow

One of the first interesting modal dataflow models is **Cyclo-static dataflow** (**CSDF**), introduced by Bilsen et al. (1996). Whereas an SDF actor is given by single pair (R, f) of firing rules and firing function, a CSDF actor provides a sequence $((R_1, f_1) \cdots, (R_P, f_P))$ for some period $P \geq 1$. The first firing is governed by (R_1, f_1), the second by (R_2, f_2), until we reach (R_P, f_P). Then we begin again with (R_1, f_1). Each pair (R_i, f_i) is required to correspond with an SDF actor (the number of tokens consumed and produced must be fixed).

Sidebar: Petri Nets

Petri nets, named after Carl Adam Petri, are a popular modeling formalism related to dataflow (Murata, 1989). They have two types of elements, **places** and **transitions**, depicted as white circles and rectangles, respectively. A place can contain any number of tokens, depicted as black circles. A transition is **enabled** if all places connected to it as inputs contain at least one token.



Once a transition is enabled, it can **fire**, consuming one token from each input place and depositing one token on each output place. The state of a network, called its **marking**, is the number of tokens on each place in the network. The figure above shows the marking of a simple network before and after a firing. If a place provides inputs to more than one transition, then a token on that place may trigger a firing of either destination transition (one or the other fires, nondeterministically).

Petri net transitions are like dataflow actors; they fire when sufficient inputs are available. In basic Petri nets, tokens have no value. A firing just moves tokens from one place to another. Also, places do not preserve token ordering, unlike dataflow buffers. Like homogeneous SDF, transitions are enabled by a single token on each input place. Unlike SDF, a place may feed more than one transition, yielding nondeterminism.

There are many variants of Petri nets, at least one of which is equivalent to SDF. In particular, **colored tokens** have values, where the color represents the value. Transitions can manipulate colors (analogous to the firing function of a dataflow actors). Arcs connecting places to transitions can be weighted to indicate that more than one token is required to fire a transition, or that a transition produces more than one token, like SDF production and consumption rates. And finally, the Petri net can be constrained so that for each place, there is exactly one source transition and exactly one destination transition. With order-preserving places, such Petri nets are SDF graphs.

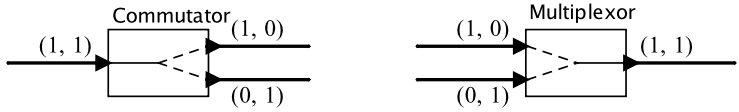


Figure 5.16: Two simple examples of CSDF actors.

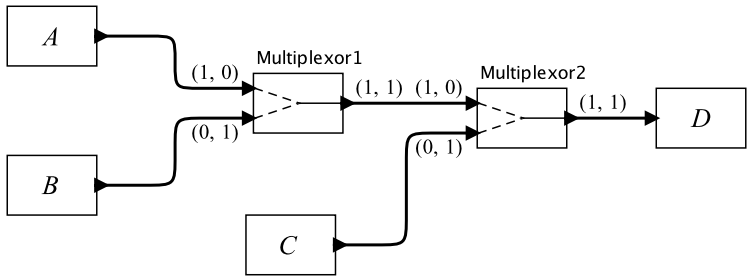


Figure 5.17: Two simple examples of CSDF actors.

Example 5.29: A CSDF version of the Switch actor is a **Commutator**, illustrated in Figure 5.16, which splits an input stream into two output streams. It consumes one input token in each firing and routes the token to the top output in odd firings and to the bottom output in even firings. The **Multiplexor**, also illustrated in the figure, merges two input streams by alternating their tokens. In the figure, the number of tokens produced and consumed for each port is shown as a sequence of length $P = 2$. The corresponding firing functions just read and write the appropriate ports.

It is straightforward to generalize the balance equations to handle CSDF. Simply replace the production and consumption numbers with their average. When finding an integer solution, find the least integer solution so that the number of firings of each actor is an integer multiple nP of its period P.

Example 5.30: Consider the program in Figure 5.17, which multiplexes outputs from A, B, and C in a pattern that looks like $(a_1, c_1, b_1, c_2, a_2, c_3, b_2, c_4, \cdots)$.

We can write down the balance equations in matrix form as follows:

$$\Gamma \boldsymbol{q} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1/2 & 0 \\ 0 & 1 & 0 & 0 & -1/2 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1/2 \\ 0 & 0 & 0 & 0 & 1 & -1/2 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{o}$$

The smallest integer solution is

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 4 \\ 2 \\ 4 \end{bmatrix} \tag{5.22}$$

In this solution, q_{M_1} and q_{M_2} are multiples of the periods ($P = 2$) of their CSDF. specifications, so this solution can be used directly to construct a schedule that will form a complete iteration.

Once we have a suitable solution to the balance equations, all of the same scheduling techniques can be used as for SDF. For example, we could perform symbolic execution and construct an acyclic precedence graph and then construct a schedule using job-shop scheduling techniques.

5.4.2 Heterochronous Dataflow

A related technique called **heterochronous dataflow** (**HDF**) is described by Girault et al. (1999). In their model, each dataflow actor has associated with it **finite-state machine** (**FSM**), which is a finite set of states and a transition function that chooses a next state as a function of the current state and the inputs. Each state s is associated with an SDF actor (R_s, f_s). And each state machine has an initial state.

With each actor in its initial state, we have an SDF graph which, if it is consistent, will have a uniquely defined minimal complete iteration for which we can con-

struct a static schedule. To execute this graph, we first execute this static schedule for the initial states. We then evaluate the transition functions, which possibly causes each of the actors to change state. We now have a new SDF graph which, again if it is consistent, will have a uniquely defined minimal complete iteration. We execute a schedule for this second complete iteration. Upon completion of this schedule, we again evaluate the transition functions for each actor that executed at least once, ending up with a new SDF graph. We can repeat this process indefinitely with bounded memory as long as each of the resulting SDF graph is consistent. In the worst case, the total number of SDF graphs that must be analyzed and scheduled is equal to the product of the number of states of all the SDF graphs. This number is finite, so, in principle, this is doable.

Example 5.31: The program in Figure 5.17 can be realized as an HDF graph. The Multiplexor actors have two states. The initial state consumes one token from the top input port and zero from the bottom port and produces one output token. The other state consumes one token from the *bottom* input port and zero from the top port and produces one output token. The transition function is such that whenever it is evaluated, it returns the state that is not the current state.

With each of the two multiplexors in their initial states, we have an SDF graph where the least integer solution is

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Notice the zeros in this solution for actors B and C. The corresponding schedule is (A, M_1, M_2, D).

In the new state, we again have an SDF graph. This time, the least solution to the balance equations is

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}. \tag{5.23}$$

The corresponding schedule is (C, M_2, D).

We now evaluate the transition function for M_2 but not M_1 because M_1 did not execute in this schedule. Solving the balance equations for the new SDF graph, we get

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

and a schedule (B, M_1, M_2, D).

The next iteration gives us (5.23) again. Evaluating the transition functions now takes us back to the initial state, so we can start all over.

Notice that if we sum the solutions to the balance equations over the four iterations before returning to the initial state, we get the same result as the CSDF solution (5.22),

$$\begin{bmatrix} q_A \\ q_B \\ q_C \\ q_D \\ q_{M_1} \\ q_{M_2} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 4 \\ 2 \\ 4 \end{bmatrix}.$$

Notice that HDF requires special care in handling consumption or production rates of zero. Any subgraph that is connected on the far side of such a production or consumption should be given zero repetitions. It also requires care when

choosing when to evaluate the transition functions. They should be evaluated only after a complete iteration has completed, but only for actors with one or more firings in that iteration. Because of these subtleties, HDF can considerably more difficult to use than CSDF.

5.4.3 SDF Scenarios

Geilen and Stuijk (2010) introduce **synchronous dataflow scenarios**, which are like a flattened version of HDF. A program has some set of **senarios**, each of which is an SDF graph. An SDF graph is executed through a complete iteration before the next scenario takes over. The structure of the SDF graph can be completely different in each scenario. A practical realization must provide a mechanism for tokens to be carried from one scenario to the next.

5.4.4 Parameterized SDF

A similar variant of SDF that is called **parameterized SDF** (**PSDF**), introduced by Bhattacharya and Bhattacharyya (2000). PSDF allows the production and consumption rates of ports to be given by a parameter rather than being a constant, but the overall structure is held constant. The value of the parameter is permitted to change, but only between complete iterations. When the value of such a parameter changes, a new schedule must be used for the next complete iteration.

5.5 Conclusion

This chapter has examined dataflow models of computation as a foundation for deterministic concurrent systems. Starting from a precise definition of dataflow actors in terms of firing rules and firing functions, we showed how individual actors induce continuous Kahn processes and how networks of such actors admit a unique denotational semantics given by the least fixed point of a continuous functional. This establishes determinism at the semantic level, independent of execution order or scheduling decisions.

A key theme has been the separation between semantics and execution. Although the denotational semantics of a dataflow network is well defined and

deterministic, operational behavior depends critically on scheduling policies. By introducing commutative firings, we broadened the class of actors that can be composed while preserving determinism, enabling networks to be executed by repeatedly firing any enabled actor. This yields a simple and flexible operational model, but one whose convergence properties depend on scheduling choices and resource constraints.

The chapter then focused on synchronous dataflow (SDF) as an important and practically useful specialization. By restricting actors to fixed token production and consumption rates, SDF admits static analysis techniques that are not available in general dataflow networks. Balance equations and the associated connectivity matrix characterize whether a model is consistent and can execute indefinitely with bounded memory. For consistent graphs, periodic admissible sequential schedules can be constructed, and both deadlock freedom and boundedness become decidable properties. These results make SDF particularly attractive for signal processing and embedded systems, where predictability, resource bounds, and compile-time scheduling are essential.

SDF also admits parallel scheduling through well-studied job-shop scheduling techniques. The chapter shows how to construct an acyclic precedence graph for an SDF graph. This APG can then be turned over to a variety of parallel scheduling methods.

More expressive variants, including boolean-controlled, cyclo-static, and modal dataflow, relax the rigidity of SDF while retaining many of its analyzable properties. These models occupy a useful middle ground between the full generality of Kahn process networks and the strict regularity of SDF, enabling systems whose structure or rates vary over time but remain amenable to disciplined execution strategies.

Overall, dataflow models illustrate how strong semantic guarantees—specifically determinism—can coexist with concurrency and parallelism. By carefully constraining the interaction between computation and communication, dataflow provides a framework in which nondeterminism is avoided not by limiting parallelism, but by structuring it. The next chapter will explore a contrasting approach, synchronous reactive models, which impose global coordination rather than local firing rules to achieve deterministic behavior.

Part III

Timed Models

6 Synchronous Reactive Models

6.1 Clocks and Ticks

6.2 Fixed Point Semantics

6.3 Constructive Semantics

6.4 Scheduling Strategies

6.5 Symbolic Execution

7 Reactors

7.1 TBD

8 Distributed Reactors

8.1 TBD

This chapter borrows from **?**.

Part IV

Appendices

A Partially Ordered Sets

This chapter provides mathematical preliminaries used in subsequent chapters to develop the theory of concurrent systems. It reviews basic ideas and notation in logic, with particular emphasis on sets, functions, partial orders, and fixed-point theorems. The applications in subsequent chapters are essential to develop a full understanding of the role that these mathematical models play in concurrent systems.

A.1 Sets

In this section, we review the notation for sets. A **set** is a collection of objects. When object a is in set A, we write $a \in A$. We define the following sets:

- $\mathbb{B} = \{0, 1\}$, the set of **binary digits**.
- $\mathbb{T} = \{\mathit{false}, \mathit{true}\}$, the set of **truth values**.
- $\mathbb{N} = \{0, 1, 2, \cdots\}$, the set of **natural numbers**.
- $\mathbb{Z} = \{\cdots, -1, 0, 1, 2, \cdots\}$, the set of **integers**.
- $\mathbb{R}$, the set of **real numbers**.
- $\mathbb{R}_+$, the set of **non-negative real numbers**.

When set A is entirely contained by set B, we say that A is a **subset** of B and write $A \subseteq B$. For example, $\mathbb{B} \subseteq \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}$. The sets may be equal, so the statement $\mathbb{N} \subseteq \mathbb{N}$ is true, for example. The **powerset** of a set A is defined to be the set of all subsets. It is written $\wp(A)$ or 2^A (for a justification of the latter notation, see the sidebar on page 178). The **empty set**, written $\emptyset$, is always a member of the powerset, $\emptyset \in \wp(A)$.

We define **set subtraction** as follows,

$$A \backslash B = \{a \in A \ : \ a \notin B\}$$

for all sets A and B. This notation is read "the set of elements a from A such that a is not in B."

A **cartesian product** of sets A and B is a set written $A \times B$ and defined as follows,

$$A \times B = \{(a, b) \ : \ a \in A, b \in B\}.$$

A member of this set (a, b) is called a **tuple**. This notation is read "the set of tuples (a, b) such that a is in A and b is in B." A cartesian product can be formed with three or more sets, in which case the tuples have three or more elements. For example, we might write $(a, b, c) \in A \times B \times C$. A cartesian product of a set A with itself is written $A^2 = A \times A$. A cartesian product of a set A with itself n times, where $n \in \mathbb{N}$ is written A^n. A member of the set A^n is called an ***n*-tuple**. By convention, A^0 is a **singleton set**, or a set with exactly one element, regardless of the size of A. Specifically, we define $A^0 = \{\emptyset\}$. Note that A^0 is not itself the empty set. It is a singleton set containing the empty set (for insight into the rationale for this definition, see the box on page 178).

A.2 Relations and Functions

A **relation** from set A to set B is a subset of $A \times B$. A **partial function** f from set A to set B is a relation where $(a, b) \in f$ and $(a, b') \in f$ imply that $b = b'$. Such a partial function is written $f : A \rightharpoonup B$. A **total function** or just **function** f from A to B is a partial function where for all $a \in A$, there is a $b \in B$ such that $(a, b) \in f$. Such a function is written $f : A \to B$, and the set A is called its **domain** and the set B its **codomain**. Rather than writing $(a, b) \in f$, we can equivalently write $f(a) = b$.

Example A.1: An example of a partial function is $f : \mathbb{R} \rightharpoonup \mathbb{R}$ defined by $f(x) = \sqrt{x}$ for all $x \in \mathbb{R}_+$. It is undefined for any $x < 0$ in its domain $\mathbb{R}$.

A partial function $f : A \rightharpoonup B$ may be defined by an **assignment rule**, as done in the above example, where an assignment rule simply explains how to obtain the value of $f(a)$ given $a \in A$. Alternatively, the function may be defined by its **graph**, which is a subset of $A \times B$.

Example A.2: The same partial function from the previous example has the graph $f \subseteq \mathbb{R}^2$ given by

$$f = \{(x, y) \in \mathbb{R}^2 \ : \ x \geq 0 \text{ and } y = \sqrt{x}\} \ .$$

Note that we use the same notation f for the function and its graph when it is clear from context which we are talking about.

The set of all functions $f: A \to B$ is written $(A \to B)$ or B^A. The former notation is used when the exponential notation proves awkward. For a justification of the notation B^A, see the box on page 178.

The **function composition** of $f: A \to B$ and $g: B \to C$ is written $(g \circ f): A \to C$ and defined by

$$(g \circ f)(a) = g(f(a))$$

for any $a \in A$. Note that in the notation $(g \circ f)$, the function f is applied first.

A function whose domain and codomain are the same set is called an **endofunction**. For an endofunction $f: A \to A$, the composition with itself can be written $(f \circ f) = f^2$, or more generally

$$\underbrace{(f \circ f \circ \cdots \circ f)}_{n \text{ times}} = f^n$$

for any $n \in \mathbb{N}$. In case $n = 1$, $f^1 = f$. For the special case $n = 0$, the function f^0 is by convention the **identity function**, so $f^0(a) = a$ for all $a \in A$. Note that $f^n \in A^A$ for all $n \in \mathbb{N}$.

For every function $f: A \to B$, there is an associated function $\hat{f}: \wp(A) \to \wp(B)$ defined on the powerset of A as follows,

$$\forall\, A' \subseteq A, \quad \hat{f}(A') = \{b \in B \,:\, \exists\, a \in A',\ f(a) = b\}.$$

We call $\hat{f}$ the **lifted** version of f. When there is no ambiguity, we may write the lifted version of f simply as f rather than $\hat{f}$.

For any $A' \subseteq A$, $\hat{f}(A')$ is called the **image** of A' for the function f. The image $\hat{f}(A)$ of the domain A is called the **range** of the function f.

Example A.3: The image $\hat{f}(\mathbb{R})$ of the function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is $\mathbb{R}_+$.

A function $f: A \to B$ is **onto** (or **surjective**) if $\hat{f}(A) = B$. A function $f: A \to B$ is **one-to-one** (or **injective**) if for all $a, a' \in A$,

$$a \neq a' \Rightarrow f(a) \neq f(a'). \tag{A.1}$$

That is, no two distinct values in the domain yield the same values in the codomain. A function that is both one-to-one and onto is called a **bijection**.

Example A.4: The function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = 2x$ is a bijection. The function $f: \mathbb{Z} \to \mathbb{Z}$ defined by $f(x) = 2x$ is one-to-one, but not onto. The function $f: \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x, y) = xy$ is onto but not one-to-one.

The previous example underscores the fact that an essential part of the definition of a function is its domain and codomain.

Proposition A.1. *If $f: A \to B$ is onto, then there is a one-to-one function $h: B \to A$.*

Proof. Let h be defined by $h(b) = a$ where a is any element in A such that $f(a) = b$. There must always be at least one such element because f is onto. We can now show that h is one-to-one. To do this, consider any two elements $b, b' \in B$ where $b \neq b'$. We need to show that $h(b) \neq h(b')$. Assume to the contrary that $h(b) = h(b') = a$ for some $a \in A$. But then by the definition of h, $f(a) = b$ and $f(a) = b'$, which implies $b = b'$, a contradiction. □

The converse of this proposition is also easy to prove.

Proposition A.2. *If $h: B \to A$ is one-to-one, then there is an onto function $f: A \to B$.*

Any bijection $f: A \to B$ has an **inverse** $f^{-1}: B \to A$ defined as follows,

$$f^{-1}(b) = a \in A \text{ such that } f(a) = b\,, \tag{A.2}$$

for all $b \in B$. This function is defined for all $b \in B$ because f is onto. And for each $b \in B$ there is a single unique $a \in A$ satisfying (A.2) because f is one-to-one. For any bijection f, its inverse is also a bijection.

A.2.1 Restriction and Projection

Given a function $f: A \to B$ and a subset $C \subseteq A$, we can define a new function $f|_C$ that is the **restriction** of f to C. It is defined so that for all $x \in C$, $f|_C(x) = f(x)$.

> **Example A.5:** The function $f: \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$ is not one-to-one. But the function $f|_{\mathbb{R}_+}$ is.

Consider an n-tuple $a = (a_0, a_1, \cdots, a_{n-1}) \in A_0 \times A_1 \times \cdots \times A_{n-1}$. A **projection** of this n-tuple extracts elements of the tuple to create a new tuple. Specifically, let

$$I = (i_0, i_1, \cdots, i_m) \in \{0, 1, \cdots, n-1\}^m$$

for some $m \in \mathbb{N} \backslash \{0\}$. That is, I is an m-tuple of indexes. Then we define the projection of a onto I by

$$\pi_I(a) = (a_{i_0}, a_{i_1}, \cdots, a_{i_m}) \in A_{i_0} \times A_{i_1} \times \cdots \times A_{i_m} .$$

The projection may be used to permute elements of a tuple, to discard elements, or to repeat elements.

Projection of a tuple and restriction of a function are related. An n-tuple $a \in A^n$ where $a = (a_0, a_1, \cdots, a_{n-1})$ may be considered a function of the form $a: \{0, 1, \cdots, n-1\} \to A$, in which case $a(0) = a_0$, $a(1) = a_1$, etc. Projection is similar to restriction of this function, differing in that restriction, by itself, does not provide the ability to permute, repeat, or renumber elements. But conceptually, the operations are similar, as illustrated by the following example.

> **Example A.6:** Consider a 3-tuple $a = (a_0, a_1, a_2) \in A^3$. This is represented by the function $a: \{0, 1, 2\} \to A$. Let $I = \{1, 2\}$. The projection $b = \pi_I(a) = (a_1, a_2)$, which itself can be represented by a function $b: \{0, 1\} \to A$, where $b(0) = a_1$ and $b(1) = a_2$.
>
> The restriction $a|_I$ is not exactly the same function as b, however. The domain of the first function is $\{1, 2\}$, whereas the domain of the second is $\{0, 1\}$. In particular, $a|_I(1) = b(0) = a_1$ and $a|_I(2) = b(1) = a_2$.

A projection may lifted just like ordinary functions. Given a set of n-tuples $B \subseteq A_0 \times A_1 \times \cdots \times A_{n-1}$ and an m-tuple of indexes $I \in \{0, 1, \cdots, n-1\}^m$, the **lifted**

projection is

$$\hat{\pi}_I(B) = \{\pi_I(b) \ : \ b \in B\} \ .$$

A.3 Sequences

A tuple $(a_0, a_1) \in A^2$ can be interpreted as a sequence of length 2. The order of elements in the sequence matters, and is in fact captured by the natural ordering of the natural numbers. The number 0 comes before the number 1. We can generalize this and recognize that a **sequence** of elements from set A of length n is an n-tuple in the set A^n. A^0 represents the set of empty sequences, a singleton set (there is only one empty sequence). We denote the **empty sequence** λ.

The set of all **finite sequences** of elements from the set A is written A^*, where we interpret $*$ as a wildcard that can take on any value in $\mathbb{N}$. Since $0 \in \mathbb{N}$, $\lambda \in A^*$. A member of this set with length $n > 0$ is an n-tuple. The $*$ operator here is known as the **Kleene star** (or **Kleene operator** or **Kleene closure**), after American mathematician Stephen Cole Kleene (1909-1994).

The set of **infinite sequences** of elements from A is written $A^{\mathbb{N}}$ or A^{ω}. The set of **finite and infinite sequences** is written

$$A^{**} = A^* \cup A^{\omega} \ .$$

Finite and infinite sequences play an important role in the semantics of concurrent programs. They can be used, for example, to represent streams of messages sent from one part of the program to another. Or they can represent successive assignments of values to a variable. For programs that terminate, finite sequences will be sufficient. For programs that do not terminate, we need infinite sequences.

A.4 Partial Orders

The notion of **order** is central to much of the structure in concurrent systems. Examples of ordering relationships include the usual intuitive ordering of numbers (0 is less than 1, etc.), but also many other concepts. For example, event a causes event b, 3.14159 is a better approximation of π than 3.14, and John is Sarah's father are all elements of ordering relations. For many such ordering relations, not all elements of the sets of interest are ordered. For example, John and Jane may not

Insight: Exponential Notation for Sets of Functions

The exponential notation B^A for the set of functions of form $f: A \to B$ is worth explaining. Recall that A^2 is the cartesian product of set A with itself, and that $\wp(A)$ is the powerset of A. These two notations are naturally thought of as sets of functions. A construction attributed to John von Neumann defines the natural numbers as follows,

$$\begin{aligned} \mathbf{0} &= \emptyset \\ \mathbf{1} &= \{\mathbf{0}\} = \{\emptyset\} \\ \mathbf{2} &= \{\mathbf{0}, \mathbf{1}\} = \{\emptyset, \{\emptyset\}\} \\ \mathbf{3} &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ &\cdots \end{aligned}$$

With this definition, the powerset $\mathbf{2}^A$ is the set of functions mapping the set A into the set $\mathbf{2}$. Consider one such function, $f \in \mathbf{2}^A$. For each $a \in A$, either $f(a) = \mathbf{0}$ or $f(a) = \mathbf{1}$. If we interpret "$\mathbf{0}$" to mean "nonmember" and "$\mathbf{1}$" to mean "member," then indeed the set of functions $\mathbf{2}^A$ represents the set of all subsets of A.

Similarly, the cartesian product $A^\mathbf{2}$ can be interpreted as the set of functions of form $f: \mathbf{2} \to A$, or using von Neumann's numbers, $f: \{\mathbf{0}, \mathbf{1}\} \to A$. Consider a tuple $a = (a_0, a_1) \in A^\mathbf{2}$. It is natural to associate with this tuple a function $a: \{\mathbf{0}, \mathbf{1}\} \to A$ where $a(\mathbf{0}) = a_0$ and $a(\mathbf{1}) = a_1$. The argument to the function is the index into the tuple. We can now interpret the set of functions B^A of form $f: A \to B$ as a set of tuples indexed by the set A instead of by the natural numbers.

Let $\omega = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \cdots\}$ represent the set of **von Neumann numbers**. This set is closely related to the set $\mathbb{N}$. Given a set A, it is now natural to interpret A^ω as the set of all infinite sequences of elements from A, equal to $A^\mathbb{N}$.

The singleton set $A^\mathbf{0}$ can now be interpreted as the set of all functions whose domain is the empty set and codomain is A. There is exactly one such function (no two such functions are distinguishable), and that function has an empty graph. Before, we defined $A^\mathbf{0} = \{\emptyset\}$. Using von Neumann numbers, $A^\mathbf{0} = \mathbf{1}$, corresponding nicely with the definition of a zero exponent on ordinary numbers. You can think of $A^\mathbf{0} = \{\emptyset\}$ as the set of all functions with an empty graph.

It is customary in the literature to omit the bold face font for $A^\mathbf{0}$, $\mathbf{2}^A$, and $A^\mathbf{2}$, writing instead simply A^0, 2^A, and A^2.

be related at all, unlike John and Sarah. Such partial orders play a central role in mathematical models of concurrent systems.

Definition A.1. *A **partial order** on a set A is a relation from A to A satisfying the following properties. For all $a, b, c \in A$, the relation is*

1. ***reflexive**: $a \leq a$*
2. ***antisymmetric**: $a \leq b$ and $b \leq a$ implies that $a = b$.*
3. ***transitive**: $a \leq b$ and $b \leq c$ implies that $a \leq c$.*

In this definition, we have written the relation using the symbol $\leq$. Specifically, the relation is $\leq$, where $\leq \subseteq A \times A$, and $(a_0, a_1) \in \leq$ is equivalently written $a_0 \leq a_1$. The latter notation is referred to as **infix notation**.

A **partially ordered set** or **poset** is a set A and a partial order relation $\leq$ on that set. We can write a poset as a tuple $(A, \leq)$, or $(A, \leq_A)$ if we need to make a distinction between a partial order $\leq_A$ on set A and another partial order $\leq_B$ on another set B.

Example A.7: A set A of sets and the subset relation $\subseteq$ form a poset $(A, \subseteq)$. The relation $\subseteq$ is referred to as the **subset order**.

For sets of sequences, there is a natural partial order relation based on the notion of a **prefix**.

Definition A.2. *For a set A and the set of finite and infinite sequences A^{**} of elements of A, the **prefix order** is a relation $\sqsubseteq$ from A^{**} to A^{**} such that for any $s, s' \in A^{**}$, $s \sqsubseteq s'$ if either s is the empty sequence, or for all $n \in \mathbb{N}$ where $s(n)$ is defined, $s'(n)$ is defined and is equal to $s(n)$.*

It is easy to show that $(A, \sqsubseteq)$ is a poset by showing that the prefix order conforms with Definition A.1.

Given a poset $(A, \leq)$ and two elements $a, a' \in A$, if either $a \leq a'$ or $a' \leq a$, then a and a' are said to be **comparable**. Otherwise, they are **incomparable**.

Definition A.3. *A **chain** $C \subseteq A$ is a subset of a poset $(A, \leq)$ where any two members of the subset are comparable.*

A **total order** is a poset $(A, \leq)$ where A itself is a chain.

Example A.8: The posets $(\mathbb{N}, \leq)$, $(\mathbb{Z}, \leq)$, and $(\mathbb{R}, \leq)$, where $\leq$ is the ordinary **numeric order** are total orders.

Example A.9: The powerset $2^{\mathbb{N}}$ is the set of all sets of natural numbers. $(2^{\mathbb{N}}, \subseteq)$, where $\subseteq$ is the subset order, is a poset but not a total order.

A finite poset may be represented by a **Hasse diagram**, where elements of the poset are placed above one another with line segments indicating the order relation. If two elements a and b are joined by a line segment, and a is below b in the diagram, then $a < b$.

Example A.10: Figure A.1 gives the Hasse diagram of a poset

$$A = \{\bot, a, b, c, d, e, f, g, h\},$$

where $\bot$ is the **bottom** element (below everything else in the poset), e, f, g, and h are just above $\bot$, and a, b, c, and d are each above two other elements, as shown in the diagram. In this poset, $\{\bot, e, b\}$, for example, is a chain. The elements a and g are incomparable.

A **pointed** poset $(A, \leq)$ is one that has a **bottom element**, often written using a special upside-down 'T' character, as in $\bot_A$. The bottom element is less than or equal to every element in A. When the set is understood, the bottom element may be written simply $\bot$, as it is in the previous example.

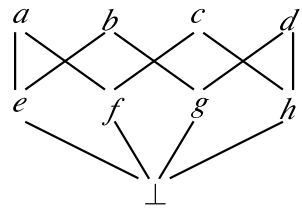


Figure A.1: A Hasse diagram of a simple partial order.

Given a poset $(A, \leq)$, we can induce another relation $<$ called the **strict partial order** relation defined as follows.

$$\forall\, a, a' \in A, \quad a < a' \Leftrightarrow a \leq a' \text{ and } a \neq a' \,.$$

$(A, <)$ is called a **strict poset**.

A.4.1 Orders on Tuples

Given a poset $(A, \leq_A)$, the set A^2 has a **pointwise order** defined by

$$(a_0, a_1) \leq (a'_0, a'_1) \Leftrightarrow a_0 \leq_A a'_0 \text{ and } a_1 \leq_A a'_1,$$

for all $(a_0, a_1), (a'_0, a'_1) \in A^2$. With this order relation, $(A^2, \leq)$ is clearly a poset. An alternative order for A^2 is the **lexicographic order**, defined by

$$(a_0, a_1) \leq (a'_0, a'_1) \Leftrightarrow a_0 <_A a'_0 \text{ or } (a_0 = a'_0 \text{ and } a_1 \leq_A a'_1),$$

for all $(a_0, a_1), (a'_0, a'_1) \in A^2$.

Both pointwise order and lexicographic order generalize trivially to A^m for $m > 2$. They also generalize to A^*, A^ω, and A^{**}.

> **Example A.11:** Let $A = \{a, b, c, \cdots, z\}$ be the letters of the alphabet and $\leq$ be the usual **alphabetical order** for those letters. Then A^* is the set of all finite sequences of letters, which includes the set of all words. The lexicographic order for A^* is called the **dictionary order** because it gives the order in which words appear in a dictionary. The dictionary order is a total order, whereas the pointwise order for this poset would not be. In particular, under the pointwise order, the sequence (a, b) would be incomparable to the sequence (b, a), whereas under the dictionary order, the first is less than the second.

A.4.2 Upper and Lower Bounds

Given a poset $(A, \leq)$ and a subset $B \subseteq A$, an **upper bound** of B, if it exists, is an element $a \in A$ such that for all $b \in B$, $b \leq a$. A **least upper bound** or **LUB**, if it exists, is an upper bound a such that for all other upper bounds a' we have $a \leq a'$. A set may have an upper bound and no LUB.

Example A.12: Consider the poset $(\mathbb{Q}, \leq)$ of **rational numbers**, where $\leq$ is the natural numeric order. Let $B \subset \mathbb{Q}$ be the subset of rational numbers whose value is less than the irrational number π (this relation "less than" is not the $\leq$ relation in this poset, since π is not a member of the poset, but we understand the definition of this subset anyway). In this poset, B has many upper bounds, but no least upper bound. Its LUB would have to be the greatest rational number less than π, and there is no such number.

If a set $B \subseteq A$ has a least upper bound in the poset $(A, \leq)$, then it is said to be **joinable** in $(A, \leq)$, and the LUB is called the **join** of B and written $\bigvee B$.

Example A.13: Consider the poset $(2^{\mathbb{N}}, \subseteq)$ of sets of natural numbers under the subset order. Every subset of $B \subseteq 2^{\mathbb{N}}$ is joinable, and the join is the union of the sets in B,

$$\bigvee B = \bigcup_{b \in B} b \,.$$

It is easy to see that this is a LUB. Any other bound must contain at least this union. It is not accidental that the notation $\bigvee$ is similar to $\cup$. This similarity can be very helpful in getting used to the notation.

Correspondingly, a subset $B \in A$ may have a **lower bound** in the poset $(A, \leq)$. This will be an element $a \in A$ such that for all $b \in B$, $a \leq b$. The set B may also have a **greatest lower bound** or **GLB**, defined similarly to the LUB. The GLB, if it exists, is called the **meet** of B in $(A, \leq)$ and is written $\bigwedge B$.

Example A.14: Any subset B as in the previous example has a meet given by

$$\bigwedge B = \bigcap_{b \in B} b \,.$$

Again, it is not accidental that the notation $\bigwedge$ reminds us of $\cap$.

A.4.3 Complete Partial Orders

Recall that a pointed poset $(A, \leq)$ is one that has a **bottom element**. The bottom element is the meet of the entire set,

$$\bot_A = \bigwedge A \in A \, .$$

Definition A.4. *A nonempty subset $D \subseteq A$ of poset $(A, \leq)$ is a **directed set** if every pair of elements in D has an upper bound in D. Equivalently, D is directed if every non-empty finite subset of D is joinable in D.*

Every chain is a directed set. In fact, directed sets can be viewed as generalizations of the idea of chains.

Definition A.5. *A **complete partial order** or **CPO** $(A, \leq)$ is a pointed poset where every directed subset is joinable in A.*

CPOs have extremely useful properties and are widely used in computer science to study semantics. To understand why they are so useful, note first that any *finite* directed subset is trivially joinable, by the definition of a directed set. To be a CPO, this property has to extend to *infinite* directed subsets. This may seem like a small step, but it is not. Directed subsets in a CPO may be thought of as being "directed" towards some goal. That goal is the least upper bound, whose existence is guaranteed by the fact that the set is a CPO. This LUB is in a sense the "**limit**" of the directed set. This notion of a limit turns out to be very powerful and surprisingly widely applicable.

The intuition behind the LUB of a directed set is easiest to understand for a chain $C \subseteq A$, which is a particularly simple kind of directed set. In this case, interpreting the LUB as a limit of the chain is completely natural. Being able to rely on the existence of this limit is valuable. Any ordered sequence in a CPO, therefore, has a limit. This is not a trivial property.

Example A.15: None of the sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ with numeric order is a CPO. Neither is the set of von Neumann numbers ω under the subset order. Each of these sets is itself a chain with no least upper bound, so clearly it cannot be that every chain has a least upper bound. If we augment $\mathbb{N}$ with an infinite element ∞, bigger than all elements in $\mathbb{N}$, then the resulting set $\mathbb{N} \cup \{\infty\}$

with the (extended) numeric order is a CPO. Similarly, the set $\omega \cup \{\omega\}$ is a CPO under the subset order, because every element of ω is a subset of ω.

The sets considered in the previous example are all totally ordered. For such sets, every directed subset is a chain. In fact, every subset is a chain. So to determine whether the set is a CPO, we only need to consider whether every chain has a least upper bound. Some sets that are not totally ordered also have the property that every directed set is chain.

Example A.16: Given any set A, the set of finite sequences A^* of elements from A with the prefix order $\sqsubseteq$ is not a CPO. It is easy to construct a chain of growing finite sequences that has no upper bound that is itself a finite sequence. However, the set of finite and infinite sequences A^{**} is a CPO under the prefix order. To show this, first note that every directed subset of A^{**} is a chain (given two sequences that are both a prefix of a third, one of the two must be a prefix of the other). Thus, we simply note that given any chain, if the chain is finite, its least upper bound is its maximal element; if the chain is infinite, its least upper bound is the infinite sequence defined by the union of the elements of the chain. The notion of limits of such chains plays a major role in the semantics of concurrent systems.

It turns out that to decide whether any poset is a CPO, it is enough to consider only whether every chain has a least upper bound. It is not necessary to consider directed sets that are not chains. This follows from the following alternative definition of a CPO.

Definition A.6. *A pointed poset $(A, \leq)$ is a **complete partial order (CPO)** if every chain in A is joinable in A.*

This definition captures the intuition every chain has a limit, in the sense of a least upper bound. To proof that this definition is equivalent to Definition A.5 appears to be quite difficult. It seems to require the machinery of ordinals and the axiom of choice and is beyond the scope of this text. For a discussion, see Davey and Priestly (2002). In this text, we will use both definitions, though most of the time the second one will prove easier to use. In the case of the prefix order, every directed set is a chain, so the equivalence of the two definitions is trivial.

The following proposition enables us to construct more complicated CPOs from simpler ones.

Proposition A.3. *Given a CPO $(A, \leq_A)$, $(A^n, \leq)$ is a CPO for any $n \in \mathbb{N}$, where $\leq$ is the pointwise order.*

Proof. First note that if $n = 0$, the proposition is trivial since the set has only one element, and every finite pointed poset is a CPO. It is also trivial for $n = 1$. For $n > 1$, we use definition A.6 and consider only chains in A^n. Denote such a chain by

$$C = \{(a_{1,1}, a_{1,2}, \cdots, a_{1,n}), (a_{2,1}, a_{2,2}, \cdots, a_{2,n}), \cdots (a_{i,1}, a_{i,2}, \cdots, a_{i,n}), \cdots\}$$

where $i \leq j$ implies that $a_{i,m} \leq_A a_{j,m}$ for all $m \in \{1, \cdots, n\}$. Let $A_j = \{a_{1,j}, a_{2,j}, \cdots, a_{i,j}, \cdots\}$ for $j = 1, 2, \cdots$. It is easy to see then that

$$\bigvee C = (\bigvee A_1, \bigvee A_2, \cdots, \bigvee A_n) .$$

Hence, the chain has a LUB. Since every such chain has a LUB, the poset is a CPO.

□

The following proposition follows trivially using the same proof technique, and proves quite useful.

Proposition A.4. *Given a CPO $(A, \leq_A)$ and any set B, $(A^B, \leq)$ is a CPO, where $\leq$ is the pointwise order.*

The following example illustrates how we can pull together several of these ideas to begin addressing questions of program semantics.

Example A.17: Let B be a set of variable names in a computer program and let A be the set of values that those variables can take on. During the (possibly nonterminating) execution of the program, a variable $b \in B$ takes on a (possibly infinite) sequence of values. The sequence of values is a member of the set A^{**}. We can define the **semantics** of the program to be a function $f : B \to A^{**}$ that for every $b \in B$ yields a sequence $f(b) \in A^{**}$. Since

A^{**} is a CPO under the prefix order, by proposition A.4 $(A^{**})^B$ is also a CPO under the **pointwise prefix order**.

Let a **partial execution** of the program be a function $f_i : B \to A^*$ for $i \in \mathbb{N}$. A partial execution always yields a finite sequence. If the execution of the program can be described as a chain of such partial executions (in the pointwise prefix order), then the limit of this chain in A^{**} can be taken to be the semantics f.

Describing an execution of program in this way is natural for many programs. Consider a determinate sequential program that updates values for variables as it executes. Each such update appends a new value to the end of the sequence consisting of the previous values of that variable. The sequence of growing sequences of values of variables is clearly a chain in the prefix order.

A.4.4 Flat Partial Orders

Any set can be turned into a CPO by choosing a **flat order relation**. Given an arbitrary set A, augment the set with one additional element that serves as the bottom element. Specifically, let B be the augmented set and the one additional element be $\perp_B$, so $B = A \cup \{\perp_B\}$. Define a poset $(B, \leq)$, where the order relation is such that $\perp_B \leq a$ for all $a \in A$ and $a \leq a' \Rightarrow a = a'$ for all $a, a' \in A$. This is called a **flat partial order**. Any two distinct elements a and a' in A are incomparable. Moreover, every directed subset of B is a chain with either one or two elements. Trivially, each such chain has an upper bound. Hence, $(B, \leq)$ is a CPO. In this CPO, all chains are finite, so the notion of a limit of a chain is rather trivial.

Example A.18: An example of a flat partial order is illustrated in by the Hasse diagram in Figure A.2(a), where $A = \{a_1, a_2\}$ and $B = \{\perp_B, a_1, a_2\}$.

Example A.19: Suppose that $(B, \leq_B)$ is the flat partial order of the previous example. Then $(B^2, \leq)$, where $\leq$ is the pointwise order, is illustrated by the Hasse diagram in figure A.2(b). By Proposition A.3, $(B^2, \leq)$ is also a CPO.

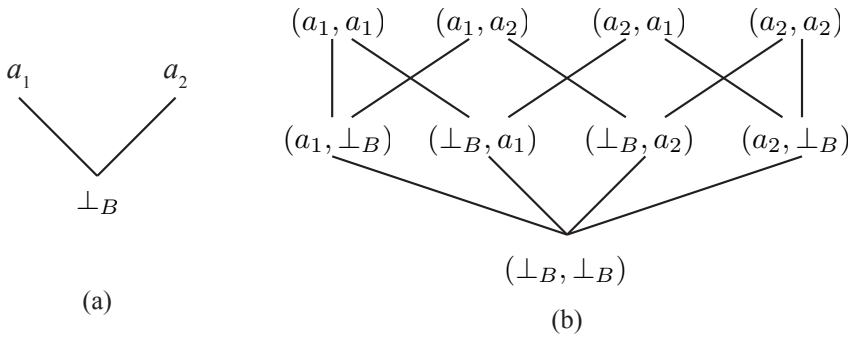


Figure A.2: (a) Flat partial order for $B = \{\perp_B, a_1, a_2\}$. (b) Pointwise order on B^2.

A.4.5 Lattices

Some partial orders have more structure than we have so far assumed.

Definition A.7. *A **lattice** is a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique greatest lower bound $a \bigwedge a' \in A$ and a unique least upper bound $a \bigvee a' \in A$.*

Example A.20: Given a set of sets A, the poset $(A, \subseteq)$ formed with the subset order is a lattice. Given two sets $a, a' \in A$, their GLB is their intersection $a \cap a'$, and their LUB is the union $a \cup a'$. If A is a powerset, then this lattice is known as the **powerset lattice**.

Example A.21: The posets in figure A.2 are not lattices. For example, in figure A.2(a), the subset a_1, a_2 has no upper bound, much less a least upper bound. Adding an artificial **top** element (commonly denoted $\top$), would change these posets so that they both become lattices.

Some posets have some of the structure of lattices, but not all of it. A **lower semilattice** or **meet semilattice** is a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique greatest lower bound $a \bigwedge a' \in A$.

Example A.22: The posets in figure A.2 are lower semilattices. You can exhaustively verify that for any pair of elements, there is a single unique GLB.

Example A.23: The poset $(A^{**}, \sqsubseteq)$ of finite and infinite sequences from the set A is a lower semilattice under the prefix order. Any two sequences in A^{**} have a unique common prefix (which may be the empty sequence).

An **upper semilattice** or **join semilattice** is dually defined as a poset $(A, \leq)$ where any two elements $a, a' \in A$ have a unique least upper bound $a \bigvee a' \in A$.

A **complete lattice** is a poset $(A, \leq)$ where every subset (not just every pair elements) has a LUB and GLB (and similarly for **complete lower semilattice** or **complete upper semilattice**). A complete lattice is also a CPO.

Example A.24: The powerset $2^{\mathbb{R}}$ of the reals with the subset order, $(2^{\mathbb{R}}, \subseteq)$ is a complete lattice. Given any set of sets of reals, the GLB is the intersection of the sets and the LUB is the union.

A.5 Functions on Posets

In this section, we define classes of functions on posets that have particularly useful properties.

A.5.1 Monotonic Functions

A function from a poset to a poset may preserve order.

Definition A.8. *A function $f : A \to B$ from poset $(A, \leq_A)$ to poset $(B, \leq_B)$ is **monotonic** or **order preserving** if $a \leq_A a' \implies f(a) \leq_B f(a')$.*

Example A.25: Consider a function $f : A^{**} \to A^{**}$ that is monotonic in the prefix order. This means that if a sequence s is a prefix of another s', i.e. that $s \sqsubseteq s'$, then $f(s) \sqsubseteq f(s')$. Functions with this property have many special

qualities. The property implies that it is "safe" to evaluate the function with partial information about its input. Given only a prefix of what the input will eventually be, we can nonetheless evaluate the function using only the prefix, and we will get partial information (a prefix) about what the output will eventually be when the input is complete.

A function $f: A \to B$ is an **order embedding** if

$$a \leq_A a' \iff f(a) \leq_B f(a') \,. \tag{A.3}$$

Note that an order embedding is monotonic, but a monotonic function is not necessarily an order embedding. An order embedding is also necessarily one-to-one (see exercise **??**). An order embedding $f: A \to B$ that is onto is called an **order isomorphism** from A to B. Since by exercise **??** it is also one-to-one, an order isomorphism is necessarily a bijection. Two posets are **order isomorphic** if there is an order isomorphism from one to the other. Order isomorphism is a rather strong relationship between posets. The posets are essentially the same, in that one can be obtained from the other by just renaming the elements.

Example A.26: Let $A = \{0, 1, 2, \cdots, 255\} \subset \mathbb{N}$ and $B = \mathbb{B}^8$. Here, B is the set of all sequences of 8 binary digits. Let $f: B \to A$ be a function that for each $b \in B$ yields $f(b) \in A$ that is the number represented by b when b is interpreted as a binary unsigned number with the high-order bit first. Assume $\mathbb{B}$ is ordered so that $0 < 1$ and that B is endowed with a lexicographic order. Assume A is endowed with the usual numeric order. Then f is an order isomorphism.

A.5.2 Continuous Functions

If we are considering functions on CPOs rather than arbitrary posets, then an even more useful property than monotonicity is continuity.

Definition A.9. *A function $f: A \to B$ from CPO $(A, \leq_A)$ to CPO $(B, \leq_B)$ is **continuous** if for all chains $C \subseteq A$,*

$$f(\bigvee C) = \bigvee \hat{f}(C) \,,$$

where $\hat{f}$ is the lifted version of f.

The **set of all continuous functions** from A to B is denoted $[A \to B]$, and is obviously a subset of the set B^A of all functions from A to B.

Proposition A.5. *Every continuous function $f: A \to B$, where $(A, \leq_A)$ and $(B, \leq_B)$ are CPOs, is monotonic.*

Proof. Consider any $a, a' \in A$ where $a \leq a'$. Let $C = \{a, a'\}$, a chain. Note that $\bigvee C = a'$. Since f is continuous, we know that

$$\bigvee \hat{f}(C) = \bigvee \{f(a), f(a')\} = f(\bigvee C) = f(a') .$$

Hence, $f(a) \leq f(a')$, so the function is monotonic. □

Not every monotonic function is continuous however.

Example A.27: Consider the CPO $(A, \leq)$, where $A = \mathbb{N} \cup \{\infty\}$ and $\leq$ is the numeric order. Let $f: A \to A$ be given by

$$f(a) = \begin{cases} 1 & \text{if } a \neq \infty \\ 2 & \text{otherwise} \end{cases}$$

This function is obviously monotonic. But it is not continuous. To see that, let $C = \mathbb{N}$ and note that $\bigvee C = \infty$. Hence, $f(\bigvee C) = 2$. However, $\hat{f}(C) = \{1\}$, because every element of C is finite. Hence, $\bigvee \hat{f}(C) = 1 \neq 2$. So the function is not continuous.

The following proposition specializes proposition A.4 to continuous functions.

Proposition A.6. *Given two CPOs $(A, \leq_A)$ and $(B, \leq_B)$, let $[B \to A] \subset A^B$ be the set of all continuous functions from B to A. Then $([B \to A], \leq)$ is a CPO under the pointwise order $\leq$.*

Proof. First we need to show that $[B \to A]$ has a bottom element. This is easy. The bottom element is a function $g \in [B \to A]$ where for all $b \in B$, $g(b) = \bot$. This function is obviously continuous and total, and hence is in $[B \to A]$.

Second, we need to show that any chain of functions in $[B \to A]$ has a LUB and that the LUB is continuous. Consider a chain of functions

$$C = \{f_1, f_2, ...\} \subset [B \to A].$$

Since each of these functions is continuous (and hence monotonic), then for any $b \in B$, the following set is also a chain,

$$C'_b = \{f_1(b), f_2(b), ...\} \subset A.$$

Since A is a CPO, this set has a LUB. Define the function $g: B \to A$ such that for all $b \in B$,

$$g(b) = \bigvee C'_b.$$

Then in the pointwise order, it must be that

$$g = \bigvee C = \bigvee \{f_1, f_2, ...\}.$$

It remains to show that g is in $[B \to A]$. To show this, we must show that it is continuous. We must show that for all chains $D \subset B$,

$$g(\bigvee D) = \bigvee \hat{g}(D).$$

Writing the elements of $D = \{d_1, d_2, ...\}$, observe that

$$\begin{aligned}
\bigvee \hat{g}(D) &= \bigvee \{g(d_1), g(d_2), ...\} \\
&= \bigvee \{\bigvee \{f_1(d_1), f_2(d_1), ...\}, \bigvee \{f_1(d_2), f_2(d_2), ...\} ...\} \\
&= \bigvee \{\bigvee \{f_1(d_1), f_1(d_2), ...\}, \bigvee \{f_2(d_1), f_2(d_2), ...\} ...\} \\
&= \bigvee \{\bigvee \hat{f_1}(D), \bigvee \hat{f_2}(D), ...\} \\
&= \bigvee \{f_1(\bigvee D), f_2(\bigvee D), ...\} \\
&= g(\bigvee D).
\end{aligned}$$

Note that the above implicitly uses the axiom of choice, which states that given a set of sets, one can construct a new set by collecting one element from each of the sets in the set of sets.

□

A.6 Fixed Points

Given an endofunction $f: A \to A$, if there is value $a \in A$ such that $f(a) = a$, that value is called a **fixed point**. Existence and uniqueness of fixed point points play a central role in the semantics of programs. Existence of a fixed point will be interpreted as "the program has a meaning." Uniqueness will be interpreted as "the program has no more than one meaning." For functions that have multiple fixed points, we are often interested in the **least fixed point**, which is a fixed point $a \in A$ such that for any other fixed point $a' \in A$, $a \leq a'$. The following proposition assures the existence and uniqueness of such a least fixed point for continuous functions, and moreover gives a constructive procedure to determine that least fixed point.

Proposition A.7. ***Kleene fixed-point theorem.*** *For any monotonic function $f: A \to A$ on a CPO $(A, \leq)$, let*

$$C = \{f^n(\bot) \ : \ n \in \mathbb{N}\} \ .$$

Then if $\bigvee C = f(\bigvee C)$, $\bigvee C$ is the least fixed point of f. Moreover, if f is also continuous, then $\bigvee C = f(\bigvee C)$.

Proof. The first part of this theorem does not require that f be continuous, but only that it be monotonic. Suppose $\bigvee C = f(\bigvee C)$. This is a fixed point. Let a be any other fixed point, i.e. $f(a) = a$. We can show that $\bigvee C \leq a$, and hence $\bigvee C$ is the least fixed point. First, observe that $\bot \leq a$. Since f is monotonic, this implies that $f(\bot) \leq f(a) = a$. Again, since f is monotonic, this implies that $f(f(\bot)) \leq f(f(a)) = f(a) = a$. Continuing in this fashion, for any $n \in \mathbb{N}$,

$$f^n(\bot) \leq f^n(a) = a \ .$$

Hence, a is an upper bound of C. Since $\bigvee C$ is the least upper bound of C, it follows that $\bigvee C \leq a$, and hence $\bigvee C$ is the least fixed point of f.

For the second part of this theorem, we require that f be continuous, and not just monotonic. First, we observe that C is a chain in the CPO $(A, \leq)$. To see that, note that $\bot \leq f(\bot)$. Since f is monotonic, this implies that $f(\bot) \leq f(f(\bot))$. Continuing, we see that for all $n \in \mathbb{N}$, $f^n(\bot) \leq f^{n+1}(\bot)$, so C is a chain. Since C is a chain, it has a LUB $\bigvee C$.

Next note that $\hat{f}(C) \cup \{\bot\} = C$. Moreover, $\bigvee(\hat{f}(C) \cup \{\bot\}) = \bigvee \hat{f}(C)$ (an additional $\bot$ in a chain will not change its least upper bound). Combining these two facts, we conclude that $\bigvee \hat{f}(C) = \bigvee C$. But since f continuous, we also know that $\bigvee \hat{f}(C) = f(\bigvee C)$. Hence, $f(\bigvee C) = \bigvee C$, and hence $\bigvee C$ is a fixed point. □

This theorem has profound consequences. It states that for a continuous function f, the least fixed point of this function can be found by first evaluating $f(\bot)$, then $f(f(\bot))$, then $f^3(\bot)$, etc. The result of this sequence of evaluations is a chain in A, and since A is a CPO, this chain has a limit. That limit is the least fixed point of the function. Hence, this theorem offers a **constructive procedure** for finding the least fixed point of a continuous function. The ascending chain $C = \{f^n(\bot) : n \in \mathbb{N}\}$ is known as the **Kleene chain** of the function f. Applications of this theorem are scattered throughout the text.

A.7 Cardinality

In mathematics, the **cardinality** of a set is a measure of the number of elements of the set. For example, set $A = \{1, 2, 3, 4\}$ has four elements, and therefore A has a cardinality of 4. More interestingly, the set $\mathbb{N}$ has an infinite number of elements, and so does the set $\mathbb{R}$. And yet, in a rather strong sense, $\mathbb{N}$ has fewer elements than $\mathbb{R}$. Even more interestingly, although we can show that the set $\mathbb{N}$ and $\mathbb{Z}$ have the same number of elements, the elements of $\mathbb{N}$ can be listed in numeric order, but the elements of $\mathbb{Z}$ cannot.

The study of the semantics benefits considerably from a solid understanding of the cardinality of sets. For example, we compare the number of programs expressible in a particular programming language to the number of functions of form $f : \mathbb{N} \to \mathbb{N}$ and show that the former is distinctly smaller than the latter. It then follows immediately that not all such functions can be expressed by computer programs in that language.

More particular to concurrent systems, it is useful to understand how many behaviors of programs are enabled by concurrency. It is also useful to understand how to distinguish finite behaviors (ones that either terminate or deadlock) from infinite ones.

A.7.1 Countable and Uncountable Sets

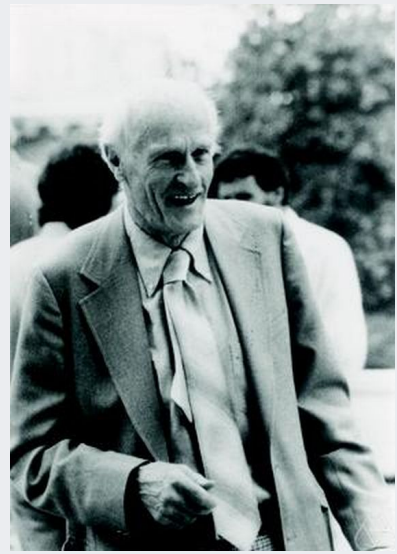
A set A is **countable** if there exists a one-to-one function $f: A \to \mathbb{N}$. Such a function associates with each element $a \in A$ a unique number $n \in \mathbb{N}$.

If there is a function that is one-to-one and also onto (and hence a bijection), then the set A is **countably infinite**. To determine whether a set is countable or countably infinite, we simply need to find such functions or show that none exists.

Sidebar: Fixed Point Theorems

The Kleene fixed-point theorem, Proposition A.7, is named after American mathematician Stephen Cole Kleene (1909–1994). This theorem is often attributed to Alfred Tarski, (1901–1983), a Polish-American logician and mathematician and a professor of mathematics at the University of California, Berkeley. **Tarski's fixed-point theorem** is similar to the Kleene fixed-point theorem, but in its original statement, it is about monotone functions on complete lattices. Because of this attribution, techniques used in computer science that are based on fixed-point theorems are often called **Tarskian**.

Another well-known theorem in this family is the **Knaster-Tarski fixed-point theorem**, developed earlier by Tarski and Bronislaw Knaster. This theorem is a special case of Tarski's fixed-point theorem that applies to the powerset lattice.



Stephen Cole Kleene (1909-1994). Photo by Konrad Jacobs, Erlangen, copyright (1978) MFO, Mathematisches Forschungsinstitut Oberwolfach, licensed under the Creative Commons Attribution-Share Alike 2.0 Germany license.

Example A.28: The set of integers $\mathbb{Z}$ is countable. To show this, let $f: \mathbb{Z} \to \mathbb{N}$ be defined by

$$\forall\, n \in \mathbb{Z}, \quad f(n) = \begin{cases} 2n-1 & \text{if } n > 0 \\ -2n & \text{otherwise} \end{cases}$$

This function is easily shown to be a bijection, and hence $\mathbb{Z}$ is countably infinite.

Given two functions $f: A \to B$ and $g: B \to \mathbb{N}$ that are known to be one-to-one (or onto), the composition $f \circ g$ is also one-to-one (or onto). Hence, if a set B is known to be countable (countably infinite), then if we find a function $f: A \to B$ that is one-to-one (or onto), then we can conclude that A is countable (or countably infinite).

Example A.29: The set of finite sequences $\mathbb{B}^*$ of binary digits $\mathbb{B}$ is countable. To show this, first let $\mathbb{N}_+ = \mathbb{N} \backslash \{0\}$ and note that it is easy to show that $\mathbb{N}_+$ is countably infinite (see exercise **??**). Next, let $g: \mathbb{B}^* \to \mathbb{N}_+$ be the function that yields the number represented by the sequence of binary digits as a digital encoding as follows

$$\forall\, b = (b_0, b_1, \cdots, b_{n-1}) \in \mathbb{B}^*, \quad f(b) = 2^n + \sum_{i=0}^{n-1} b_i 2^{(n-i-1)} .$$

Specifically,

$$\begin{aligned} f(\lambda) &= 2^0 = 1 \\ f((0)) &= 2^1 + 0 \cdot 2^0 = 2 \\ f((1)) &= 2^1 + 1 \cdot 2^0 = 3 \\ f((0,0)) &= 2^2 + 0 \cdot 2^0 + 0 \cdot 2^1 = 4 \\ &\text{etc.} \end{aligned}$$

It is easy to see that this function is one-to-one (any two distinct bit patterns yield a different positive natural number). It is also easy to see that it is onto

(any positive natural number has such a binary representation). Hence, this function is a bijection. Since $\mathbb{N}_+$ is countably infinite, so is $\mathbb{B}^*$.

A bijection $f: A \to N$ can be viewed as a list of the elements of the set A, since it assigns a unique natural number to each. It is said to **enumerate** the set. The enumeration is a list, possibly infinite, that includes all elements of A. Moreover, if the function is only one-to-one and not onto, it can still enumerate A by considering the numeric order of each $f(a)$ for $a \in A$ as giving the position of a is the list. This list may be finite if the set is countable but not countably infinite. Thus we see that to show that a set A is countable, we need simply give a way to enumerate the elements in A.

Proposition A.8. *The set $\mathbb{N}^*$ of finite sequences of natural numbers is countable.*

Proof. To enumerate $\mathbb{N}^*$, let $f: \mathbb{N}^* \to \mathbb{N}$ be a function defined by

$$f(s) = \sum_{i=0}^{n-1} (|s_i| + 1)$$

for any sequence $s = (s_0, s_1, \cdots, s_{n-1}) \in \mathbb{N}^*$ of length n. Note that for each $j \in \mathbb{N}$, there are a finite number of sequences s in $\mathbb{N}^*$ where $f(s) = j$. For $j = 0$, only the empty sequence $s = \lambda$ yields $f(s) = 0$. For $j = 1$, only the sequence $s = (0)$ results in $f(s) = 1$. For $j = 2$, the sequences (1), (−1), and (0,0) yield $f(s) = 2$. For $j = 3$, we get (2), (1,0), (−1,0), (0,1), (0 − 1), and (0,0,0). Continuing, it is easy to see that we can enumerate the set $\mathbb{N}^*$ by listing first the sequences s where $f(s) = 0$, then the ones where $f(s) = 1$, then the ones where $f(s) = 2$, etc. Since every sequence s yields a finite $f(s)$, every sequence s is included in this list. Hence, $\mathbb{N}^*$ is countable.

□

If a set A is countable, then any subset $B \subseteq A$ is also countable. More generally, if a set A is countable, then any set B for which there exists a one-to-one function $f: B \to A$ is also countable (see problem **??**(a)). This is evident because the existence of such a function shows that A is at least as large as B, since every $b \in B$ must yield a unique $a = f(b) \in A$. These facts make it easy to prove the following rather useful proposition.

Proposition A.9. *The set B^* of finite sequences of elements from any countable set B is countable.*

Proof. Since B is countable, there is a one-to-one function $g: B \to \mathbb{N}$. From this function, we can easily construct a one-to-one function $f: B^* \to \mathbb{N}^*$ as follows,

$$f(x) = (g(b_0), g(b_1), \cdots, g(b_m))$$

for all $x = (b_0, b_1, \cdots, b_m) \in B^*$. Since this function is one-to-one, and by proposition A.8 $\mathbb{N}^*$ is countable, it follows that B^* is countable. □

A set that is not countable is **uncountable**. Let $\mathbb{B}^\omega$ be the set of all infinite sequences of bits.

Proposition A.10. *$\mathbb{B}^\omega$ is uncountable.*

Proof. We give a variant of **Cantor's diagonal argument**, which is a proof by contradiction. Assume that $\mathbb{B}^\omega$ is countable. That is, it can be enumerated. Let the enumeration be written $b_1, b_2, b_3, \cdots$. For any $n \in \mathbb{N}$, let $b_i(n)$ denote the n-th bit of $b_i \in \mathbb{B}^\omega$. Let $c \in \mathbb{B}^\omega$ be a bit sequence where

$$c(n) = \overline{b_n(n)}$$

where $\bar{b}$ is 0 if $b = 1$ and 1 if $b = 0$. That is, the n-th bit of c is the complement of the n-th bit of the n-th element of the enumeration. Note therefore that c is not in the enumeration, so the enumeration does not include all the elements of $\mathbb{B}^\omega$. Hence, $\mathbb{B}^\omega$ is not countable. □

An uncountable set B is strictly larger than any countable set A in that there is no one-to-one function $f: B \to A$ (see problem **??**(a)). In other words, any function from B to A must yield the same element in A for some distinct elements in B.

If B is uncountable, then any superset $C \supseteq B$ is also uncountable. Thus, it follows from proposition A.10 that $\mathbb{N}^\omega$, $\mathbb{Z}^\omega$, and $\mathbb{R}^\omega$ are all uncountable. More generally, if a set B is uncountable, then any set C for which there exists an onto function

$f: C \to B$ is also uncountable (see problem **??**(b)). This is evident because the existence of such a function shows that C is at least as large as B. These facts make it easy to prove the following rather useful proposition (see problem **??**(c)):

Proposition A.11. *A^ω is uncountable for any set A with two or more elements.*

An immediate consequence is that the powerset $2^{\mathbb{N}}$ of $\mathbb{N}$ is uncountable (because in the notation $2^{\mathbb{N}}$, 2 represents a set with two elements, as discussed in the box on page 178). Moreover, given any set A that it as least as large as $\mathbb{N}$ (i.e. any infinite set, countable or not, or equivalently, any set for which there is an onto function $f: A \to \mathbb{N}$), the powerset 2^A is uncountable.

A.8 Discrete Sets

Recall that a countable set is one for which there exists a one-to-one function $f: A \to \mathbb{N}$. A **discrete** set is a poset $(A, \leq_A)$ for which there exists such a function that it is also an order embedding to $(\mathbb{N}, \leq)$, where $\leq$ is the numeric order. A countable set can be enumerated. A discrete set has an order relation and can be enumerated *in order*. Not all countable sets are discrete.

Proposition A.12. *The set of integers $\mathbb{Z}$ with the numeric order is countable but not discrete.*

Proof. To show this, we must show that there is no order embedding $f: \mathbb{Z} \to \mathbb{N}$. Suppose to the contrary that there is such an order embedding. The image $\hat{f}(\mathbb{Z})$ is a subset of $\mathbb{N}$, and hence must have a least element. Call that least element $m \in \mathbb{N}$ and let $i \in \mathbb{Z}$ be such that $f(i) = m$. Note that $i - 1 \in \mathbb{Z}$, and by (A.3), since $i - 1 \leq i$, it must also be that $f(i-1) \leq f(i)$. Since $f(i) = m$ is the least element, it must be true that $f(i-1) = f(i)$. But since f is necessarily one-to-one (exercise **??**), this implies that $i - 1 = i$, a contradiction. □

The same proof method suffices to show the following:

Proposition A.13. *Every discrete poset has a least element.*

As a consequence, the set $\mathbb{Q}$ of rational numbers is not discrete. But it is not sufficient to have a least element to be discrete. The set of non-negative rational numbers, for example, has a least element, but it too is not discrete. To show that, we consider the subset containing only the *positive* rational numbers, which does not have a least element, and leverage the following proposition.

Proposition A.14. *If a poset $(A, \leq)$ is discrete and $B \subseteq A$, then $(B, \leq)$ is discrete.*

Proof. Given an order embedding $f : A \to \mathbb{N}$, it is easy to see that the restriction $f|_B$ is also an order embedding. □

To show that the set of non-negative rationals is not discrete, knowing that the set of positive rationals is not discrete, we use the contrapositive of this proposition:

Proposition A.15. *if a poset $(B, \leq_B)$ is not discrete, then neither is any larger set $A \supseteq B$ with an order $\leq_A$ satisfying $a \leq_A a' \Leftrightarrow a \leq_B a'$ for all $a, a' \in B$.*

Specifically, let B be the set of positive rationals, which by proposition A.13 is not discrete. Then let A be the non-negative rationals, a superset. By proposition A.15, that set is not discrete.

A.9 Computability

In the classical theory of computing, a **computation** is a mapping from inputs to outputs. Given inputs, the computation produces outputs. Suppose that an input x and output y are represented as a sequence of binary digits, so $x, y \in \mathbb{B}^*$. Then it would seem that the notation of computation as the same as the mathematical notion of a function $f : \mathbb{B}^* \to \mathbb{B}^*$. But it turns out that this is not quite the case. It turns out that most such functions are not effectively computable.

We can use cardinality results to study the problem of **computability**, which is about what functions can be implemented by computer programs (or more fundamentally, by algorithms that terminate with an answer).

Consider the set of all **decision problems** of the form $f : \mathbb{N} \to \mathbb{B}$. For each natural number $n \in \mathbb{N}$, the function f yields a 0 or 1.

> **Example A.30:** Consider a function f that produces $f(n) = 0$ if n is even and $f(n) = 1$ if n is odd. We can write a computer program in any modern programming language that will implement this function. Note that the this is not entirely trivial, because the input is *any* natural number, with no bound on its size. If the input is given as a bit sequence (a member of $\mathbb{B}^*$), a binary representation of the number where the first bit is the low-order bit, then the program *is* trivial. It outputs the first bit.

Since every $f \in \mathbb{B}^{\mathbb{N}}$ is a decision problem, by Proposition A.10, there are an uncountable number of such decision problems.

Consider the set C of all computer programs written in some programming language. Every such program can be represented by a finite number of bits, so $C \subseteq \mathbb{B}^*$. By Proposition A.9, $\mathbb{B}^*$ is countable, and hence C is countable. Hence, C is strictly smaller than the set of decision problems.

As a consequence, for any programming language, there are decision problems that cannot be realized by a program in that language. Such problems are said to be **undecidable**.

A.10 Conclusion

This chapter has introduced some of the mathematical tools that we will need to study concurrency. The most essential of these is the Kleene fixed-point theorem, which gives a constructive procedure (the Kleene chain) for finding the least fixed point of a continuous function. This result proves to have an astonishing variety of applications, many of which are explored in the main chapters. In particular, it gives a deterministic semantics to certain concurrent models of computation. It also suggests an execution policy for such models of computation.

B Metric Spaces

This chapter provides an overview of metric spaces.

(FIXME: Very incomplete)

B.1 Metrics and Ultrametrics

Definition B.1. *An **ultrametric** on a set A is a function $d\colon A\times A\to\mathbb{R}_+$ such that for all $a,b,c\in A$:*

1. $d(a,b)=d(b,a)$,
2. $d(a,b)=0 \iff a=b$ *and*
3. $\max(d(a,b),d(b,c))\geq d(a,c)$.

B.2 The Cantor Metric

Cantor metric

B.3 The Baire Distance

An untimed version of the Cantor metric for sequences is known as the **Baire distance**.

Definition B.2. *With $A = D^{**}$, the **Baire distance** is a function $d\colon A \times A \to \mathbb{R}_+$ be such that for all $a, b \in A$,*

$$d(a,b) = 1/(n+1)$$

where n is the index of the first position at which a and b differ and is zero if $a = b$. The first element of a sequence $a \in A$ has index 0, the second has index 1, etc.

Example B.1: For example, $d((0),(1)) = 1$ and $d((0),(0,1)) = 1/2$.

Proposition B.1. *The Baire distance is an ultrametric. Conditions (1) and (2) in Definition B.1 are trivially satisfied. We can show that condition (3) is satisfied, and therefore d is an ultrametric.*

Proof. Let n_1 be such that $\max(d(a,b),d(b,c)) = 1/(n_1+1)$. This means that a and b are identical at least up to index $n_1 - 1$, and b and c are also identical at least up to index $n_1 - 1$. Hence it must be true that a and c are identical at least up to index $n_1 - 1$.

Let n_2 be such that $d(a,c) = 1/(n_2+1)$. Hence n_2 is the earliest index at which a and c differ. Since a and c are identical at least up to index $n_1 - 1$, it must be true that

$$n_2 \geq n_1.$$

Hence

$$1/(n_1+1) \geq 1/(n_2+1),$$

which is what we needed to show.

□

Bibliography

Allen, G. E., P. E. Zucknick, and B. L. Evans, 2007: A distributed deadlock detection and resolution algorithm for process networks. In *Int. Conf. on Acoustics, Speech, and Signal Proc. (ICASSP)*, IEEE, vol. II, pp. 33–36.

Arvind, L. Bic, and T. Ungerer, 1991: *Evolution of Data-Flow Computers*, Prentice-Hall.

Becker, A., 2018: *What Is Real? The Unfinished Quest for the Meaning of Quantum Physics*. Basic Books.

Bell, J. S., 1964: On the Einstein Podolsky Rosen paradox. *Physica*, **1(3)**, 195–200.

Benveniste, A. and G. Berry, 1991: The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, **79(9)**, 1270–1282.

Berry, G., 1976: Bottom-up computation of recursive programs. *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, **10(3)**, 47–82. Available from: http://eudml.org/doc/92028.

Bhattacharya, B. and S. S. Bhattacharyya, 2000: Parameterized dataflow modeling of dsp systems. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 1948–1951.

Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. Lee, 1995: Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, **42(3)**, 138–150. doi:10.1109/81.376876.

Bhattacharyya, S. S., J. T. Buck, S. Ha, and E. A. Lee, 1993: A scheduling framework for minimizing memory requirements of multirate dsp systems represented as dataflow graphs. In *VLSI Signal Processing VI*, IEEE, pp. 188–196. doi:10.1109/VLSISP.1993.404488.

Bhattacharyya, S. S. and E. A. Lee, 1993: Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI Signal Processing Systems*, **6(3)**, 271–288. doi:10.1007/BF01608539.

Bhattacharyya, S. S., P. Murthy, and E. A. Lee, 1996a: APGAN and RPMC: Complementary heuristics for translating dsp block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, **2(1)**, 33–60. doi:10.1023/A:1008806425898.

Bhattacharyya, S. S., P. K. Murthy, and E. A. Lee, 1996b: *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, Mass.

Bilsen, G., M. Engels, R. Lauwereins, and J. A. Peperstraete, 1996: Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, **44(2)**, 397–408. doi:10.1109/78.485935.

Box, G. E. P. and N. R. Draper, 1987: *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics, Wiley, ISBN 0471-81033-9.

Brock, J. D. and W. B. Ackerman, 1981: Scenarios, a model of non-determinate computation. In *Conference on Formal Definition of Programming Concepts*, Springer-Verlag, vol. LNCS 107, pp. 252–259. doi:10.1007/3-540-10699-5_102.

Broy, M. and G. Stefanescu, 2001: The algebra of stream processing functions. *Theoretical Computer Science*, **258**, 99–129.

Buck, J. T., 1993: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. Thesis Tech. Report UCB/ERL 93/69, University of California, Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/93/jbuckThesis/.

—, 1994: Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *IEEE Asilomar Conf. on Signals, Systems, and Computers*.

Buck, J. T. and E. A. Lee, 1992: The token flow model. In *Data Flow Workshop*. Available from: https://ptolemy.berkeley.edu/publications/papers/92/tokenFlow/tokenFlowAus.pdf.

—, 1993: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 1, pp. 429–432.

Cardoso, J., E. A. Lee, J. Liu, and H. Zheng, 2014: *Continuous-Time Models*, Ptolemy.org, Berkeley, CA. ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Church, A., 1932: A set of postulates for the foundation of logic. *Annals of Mathematics*, **32(2)**, 346–366. Available from: https://www.jstor.org/stable/1968337.

Church, A. and J. B. Rosser, 1936: Some properties of conversion. *Transactions of the American Mathematical Society*, **39(3)**, 472–482. doi:10.2307/1989762.

Coffman, E. G., Jr. (Ed), 1976: *Computer and Job Scheduling Theory*. Wiley.

Conway, M. E., 1963: Design of a separable transition-diagram compiler. *Communications of the ACM*, **6(7)**, 396–408.

Copley, B. and F. Martin, 2014: *Causation in Grammatical Structures*. Oxford University Press, Oxford, England.

Corbett, J. C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, 2012: Spanner: Google's globally-distributed database. In *OSDI*. doi:10.1145/2491245.

Cousot, P. and R. Cousot, 1977: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, ACM Press, pp. 238–252.

Creeger, M., 2005: Multicore CPUs for the masses. *ACM Queue*, **3(7)**, 63–64.

Cremona, F., M. Lohstroh, D. Broman, E. A. Lee, M. Masin, and S. Tripakis, 2017: Hybrid co-simulation: it's about time. *Software and Systems Modeling*, **18**, 1655–1679. doi:10.1007/s10270-017-0633-6.

Davey, B. A. and H. A. Priestly, 2002: *Introduction to Lattices and Order*. Cambridge University Press, second edition ed.

Dennis, J. B., 1974: First version data flow procedure language. Report MAC TM61, MIT Laboratory for Computer Science.

Dhar, A., 1993: Nonuniqueness in the solutions of Newton's equation of motion. *American Journal of Physics*, **61(1)**, 58–61. doi:10.1119/1.17411.

Dijkstra, E. W., 1976: *A Discipline of Programming*. Prentice Hall.

Ditlevsen, S. and A. Samson, 2013: Introduction to stochastic models in biology. In Bachar, M., J. Batzel, and S. Ditlevsen, eds., *Stochastic Biomathematical Models: with Applications to Neuronal Modeling*, Springer, pp. 3–35. doi:10.1007/978-3-642-32157-3_1.

Dulloo, J. and P. Marquet, 2004: Design of a real-time scheduler for kahn process networks on multiprocessor systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*.

Earman, J., 1986: *A Primer on Determinism*, vol. 32 of *The University of Ontario Series in Philosophy of Science*. D. Reidel Publishing Company.

Edwards, S., 2018: *On Determinism*, Springer, Cham, Switzerland, vol. LNCS 10760. pp. 240–253. doi:10.1007/978-3-319-95246-8_14.

Edwards, S. and J. Hui, 2020: The sparse synchronous model. In *2020 Forum for Specification and Design Languages, FDL 2020, Kiel, Germany, September 15-17, 2020*, IEEE, pp. 1–8.

Edwards, S. A. and E. A. Lee, 2003: The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, **48(1)**, 21–42. doi:10.1016/S0167-6423(02)00096-5.

Falk, J., J. Keiner, C. Haubelt, J. Teich, and S. S. Bhattacharyya, 2008: A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *Embedded Software (EMSOFT)*, ACM.

Faustini, A. A., 1982: An operational semantics for pure dataflow. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming (ICALP)*, Springer-Verlag, vol. Lecture Notes in Computer Science (LNCS) Vol. 140, pp. 212–224.

Fletcher, S. C., 2012: What counts as a newtonian system? the view from norton's dome. *European Journal for Philosophy of Science*, **2**, 275–297. doi:10.1007/s13194-011-0040-8.

Friedman, D. P. and D. S. Wise, 1976: CONS should not evaluate its arguments. In *Third Int. Colloquium on Automata, Languages, and Programming*, Edinburg University Press.

Gale, R., 1966: Mctaggart's analysis of time. *American Philosophical Quarterly*, **3(2)**, 145–152. Available from: https://www.jstor.org/stable/20009201.

Geilen, M. and T. Basten, 2003: Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, Springer, LNCS, pp. 319–334. Available from: http://www.ics.ele.tue.nl/~tbasten/papers/esop03.pdf.

Geilen, M., T. Basten, and S. Stuijk, 2005: Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Design Automation Conference (DAC)*, ACM, pp. 819–824. doi:10.1145/1065579.1065796.

Geilen, M. and S. Stuijk, 2010: Worst-case performance analysis of synchronous dataflow scenarios. In *CODES+ISSS*, ACM, pp. 125–134.

Girault, A., B. Lee, and E. A. Lee, 1999: Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, **18(6)**, 742–760.

Gordon, M. I., W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, 2002: A stream compiler for communication-exposed architectures. In *10th international conference on Architectural support for programming languages and operating systems*, ACM Press, 605428, ISBN 1-58113-574-2, pp. 291–303.

Ha, S., 1992: Compile-time scheduling of dataflow program graphs with dynamic constructs. Ph.D. Thesis Tech. Report UCB/ERL 92/43. Available from: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2024.html.

Ha, S. and E. A. Lee, 1991: Compile-time scheduling and assignment of dataflow program graphs with data-dependent iteration. *IEEE Transactions on Computers*, **40(11)**, 1225–1238. doi:10.1109/12.102826.

Hawking, S., 2002: Godel and the end of the universe. *Stephen Hawking Public Lectures*. Available from: http://www.hawking.org.uk/godel-and-the-end-of-physics.html.

Hennessy, M. and R. Milner, 1980: On observing nondeterminism and concurrency. In de Bakker, J. and J. van Leeuwen, eds., *Automata, Languages and Programming (ICALP)*, Springer, vol. Lecture Notes in Computer Science, LNCS 85, pp. 299–309. doi:10.1007/3-540-10003-2_79.

Hoare, C. A. R., 1978: Communicating sequential processes. *Communications of the ACM*, **21(8)**, 666–677.

Hoefer, C., 2016: Causal determinism. *The Stanford Encyclopedia of Philosophy*, **(Spring 2016 Edition)**. Available from: http://plato.stanford.edu/archives/spr2016/entries/determinism-causal/.

Hopcroft, J. and J. Ullman, 1979: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, ISBN ISBN 0-201-02988-X.

Hsu, C.-J., F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, 2004: DIF: An interchange format for dataflow-based design tools. In *International Workshop on Systems, Architectures, Modeling, and Simulation*.

Hu, T. C., 1961: Parallel sequencing and assembly line problems. *Operations Research*, **9(6)**, 841–848.

Jantsch, A. and I. Sander, 2005: Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, **152(2)**, 114–129.

Johnston, W. M., J. R. P. Hanna, and R. J. Millar, 2004: Advances in dataflow programming languages. *ACM Computing Surveys*, **36(1)**, 1–34.

Kahn, G., 1974: The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., pp. 471–475.

Kahn, G. and D. B. MacQueen, 1977: Coroutines and networks of parallel processes. In Gilchrist, B., ed., *Information Processing*, North-Holland Publishing Co., Amsterdam, pp. 993–998.

Khomenko, V., M. Schaefer, and W. Vogler, 2008: Output-determinacy and asynchronous circuit synthesis. *Fundamenta Informaticae*, **88(4)**, 541–579.

Kinniment, D. J., 2007: *Synchronization and arbitration in digital systems*. John Wiley & Sons, Ltd., New York.

Landin, P. J., 1965: A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, **8(2)**, 89–101.

Laplace, P.-S., 1901: *A Philosophical Essay on Probabilities*. John Wiley and Sons, Hoboken, NJ, translated from the sixth French edition by F. W. Truscott and F. L. Emory.

Lee, E. A., 1991: Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distributed Systems*, **2(2)**, 223–235.

—, 2006a: Concurrent semantics without the notions of state or state transitions. In Asarin, E. and P. Bouyer, eds., *International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, Springer-Verlag, vol. LNCS 4202. doi:10.1007/11867340_2.

—, 2006b: The problem with threads. *Computer*, **39(5)**, 33–42. doi:10.1109/MC.2006.180.

—, 2008: Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, IEEE, pp. 363 – 369. doi:10.1109/ISORC.2008.25.

—, 2009: Computing needs time. *Communications of the ACM*, **52(5)**, 70–79. doi:10.1145/1506409.1506426.

—, 2014: Constructive models of discrete and continuous physical phenomena. *IEEE Access*, **2(1)**, 1–25. doi:10.1109/ACCESS.2014.2345759.

—, 2016: Fundamental limits of cyber-physical systems modeling. *ACM Transactions on Cyber-Physical Systems*, **1(1)**. doi:10.1145/2912149.

—, 2017: *Plato and the Nerd — The Creative Partnership of Humans and Technology*. MIT Press. Available from: https://ptolemy.berkeley.edu/~eal/pnerd//.

—, 2020: *The Coevolution: The Entwined Futures of Humans and Machines*. MIT Press, Cambridge, MA. Available from: https://ptolemy.berkeley.edu/~eal/books/TheCoevolution.pdf.

—, 2021: Determinism. *ACM Transactions on Embedded Computing Systems (TECS)*, **20(5)**, 1–34. doi:10.1145/3453652.

—, 2022: What can deep neural networks teach us about embodied bounded rationality. *Frontiers in Psychology*, **25**. doi:10.3389/fpsyg.2022.761808.

Lee, E. A. and S. Ha, 1989: Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference (GLOBECOM)*, vol. 2, pp. 1279–1283. doi:10.1109/GLOCOM.1989.64160.

Lee, E. A. and E. Matsikoudis, 2009: *The Semantics of Dataflow with Firing*, Cambridge University Press. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/.

Lee, E. A. and D. G. Messerschmitt, 1987a: Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, **C-36(1)**, 24–35. doi:10.1109/TC.1987.5009446.

—, 1987b: Synchronous data flow. *Proceedings of the IEEE*, **75(9)**, 1235–1245. doi:10.1109/PROC.1987.13876.

Lee, E. A., S. Neuendorffer, and G. Zhou, 2014: *Dataflow*, Ptolemy.org, Berkeley, CA. ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Lee, E. A. and T. M. Parks, 1995: Dataflow process networks. *Proceedings of the IEEE*, **83(5)**, 773–801. doi:10.1109/5.381846.

Lee, E. A. and A. Sangiovanni-Vincentelli, 1998: A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, **17(12)**, 1217–1229. Available from: http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/.

Lee, E. A. and M. Sirjani, 2018: What good are models? In *Formal Aspects of Component Software (FACS)*, Springer, vol. LNCS 11222.

Lee, E. A. and H. Zheng, 2007: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, ACM, pp. 114 – 123. doi:10.1145/1289927.1289949.

Lewis, E. R. and R. J. MacGregor, 2006: On indeterminism, chaos, and small number particle systems in the brain. *Journal of Integrative Neuroscience*, **5(2)**, 223–247. Available from: https://people.eecs.berkeley.edu/~lewis/LewisMacGregor.pdf.

Lin, Y., R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner, 2006: SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*. Available from: http://www.eecs.umich.edu/~sdrg/publications.php.

Lohstroh, M., Í. Íncer Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, 2019: Reactors: A deterministic model for composable reactive systems. In *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*, Springer-Verlag, vol. LNCS 11971. doi:10.1007/978-3-030-41131-2_4.

Lohstroh, M. and E. A. Lee, 2019: Deterministic actors. In *Forum on Specification and Design Languages (FDL),*. Available from: https://ptolemy.berkeley.edu/publications/papers/19/Lohstroh_Lee_DeterministicActors_FDL_2019.pdf.

Lohstroh, M., C. Menard, A. Schulz-Rosengarten, M. Weber, J. Castrillon, and E. A. Lee, 2020: A language for deterministic coordination across multiple timelines. In *Forum for Specification and Design Languages (FDL)*, IEEE. doi:10.1109/FDL50818.2020.9232939.

Lorenz, E. N., 1963: Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, **20**, 130–141. doi:10.1175/1520-0469(1963)020.

Lázaro Cuadrado, D., A. P. Ravn, and P. Koch, 2007: Automated distributed simulation in Ptolemy II. In *Parallel and Distributed Computing and Networks (PDCN)*, Acta Press.

Malament, D. B., 2008: Norton's slippery slope. *Philosophy of Science*, **75**, 799–816. doi:10.1086/594525.

Marino, L. R., 1981: General theory of metastable operation. *IEEE Transactions on Computers*, **C-30(2)**, 107–115.

Mendler, M., T. R. Shiple, and G. Berry, 2012: Constructive Boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design*, **40(3)**, 283–329. doi:10.1007/s10703-012-0144-6.

Milner, R., 1980: *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer, Berlin Heidelburg.

—, 1989: *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ, USA.

Moreira, O., T. Basten, M. Geilen, and S. Stuijk, 2010: Buffer sizing for rate-optimal single-rate dataflow scheduling revisited. *IEEE Transactions on Computers*, **59(2)**, 188–201. doi:10.1109/TC.2009.155.

Morris, J. H. and P. Henderson, 1976: A lazy evaluator. In *Conference on the Principles of Programming Languages (POPL)*, ACM.

Murata, T., 1989: Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, **77(4)**, 541–580. doi:10.1109/5.24143.

Murthy, P. K. and S. S. Bhattacharyya, 2006: *Memory Management for Synthesis of DSP Software*. CRC Press.

Murthy, P. K. and E. A. Lee, 2002: Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, **50(8)**, 2064–2079. doi:10.1109/TSP.2002.800830.

Norton, J. D., 2007: *Causation as Folk Science*, Clarendon Press, Oxford, book section 2. pp. 11–44.

Object Management Group (OMG), 2007: A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG. Available from: http://www.omg.org/omgmarte/.

Olson, A. G. and B. L. Evans, 2005: Deadlock detection for distributed process networks. In *ICASSP*.

Park, D., 1980: *Concurrency and Automata on Infinite Sequences*, Springer, Berlin, Heidelberg, vol. LNCS 104. doi:10.1007/BFb0017309.

Parks, T. M., 1995: Bounded scheduling of process networks. Ph.D. Thesis Tech. Report UCB/ERL M95/105, UC Berkeley. Available from: http://ptolemy.eecs.berkeley.edu/papers/95/parksThesis.

Parks, T. M. and D. Roberts, 2003: Distributed process networks in Java. In *International Parallel and Distributed Processing Symposium*, pp. 138–146.

Patterson, D. A., G. Gibson, and R. Katz, 1988: A case for redundant arrays of inexpensive disks (RAID). In *International Conference on Management of Data (SIGMOD)*. doi:10.1145/50202.50214.

Pearl, J. and D. Mackenzie, 2018: *The Book of Why: The New Science of Cause and Effect*. Basic Books, New York.

Pino, J. L., T. M. Parks, and E. A. Lee, 1994: Automatic code generation for heterogeneous multiprocessors. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 445–448. doi:10.1109/ICASSP.1994.389626.

Plotkin, G., 1976: A powerdomain construction. *SIAM Journal on Computing*, **5(3)**, 452–487.

Popper, K., 1959: *The Logic of Scientific Discovery*. Hutchinson & Co., London and New York.

Ptolemaeus, C., 2014: *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, Berkeley, CA, ISBN 978-1-304-42106-7. Available from: http://ptolemy.org/books/Systems.

Ritchie, D. M. and K. L. Thompson, 1974: The UNIX time-sharing system. *Communications of the ACM*, **17(7)**, 365 – 375.

Rovelli, C., 2017: *Reality is Not What It Seems: The Journey to Quantum Gravity*. Riverhead Books, New York.

—, 2018: *The Order of Time*. Riverhead Books, New York.

Russell, B., 1913: On the notion of cause. *Proceedings of the Aristotelian Society*, **13**, 1–26.

Sangiorgi, D., 2009: On the origins of bisimulation and coinduction. *ACM Transactions on Programming Languages and Systems, Vol. 31, No. 4, Article 15, Pub. date: May 2009.*, **31(4)**, 15:1–15:41. doi:10.1145/1516507.1516510.

Schulz-Rosengarten, A., R. von Hanxleden, F. Mallet, R. De Simone, and J. Deantoni, 2018: Time in SCCharts. In *2018 Forum on Specification Design Languages (FDL)*, pp. 5–16.

Sih, G. C. and E. A. Lee, 1993a: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, **4(2)**, 175–187. doi:10.1109/71.207593.

—, 1993b: Declustering : A new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, **4(6)**, 625–637. doi:10.1109/71.242160.

Srini, V., 1986: An architectural comparison of dataflow systems. *Computer*, **19(3)**.

Sriram, S. and S. S. Bhattacharyya, 2009: *Embedded Multiprocessors: Scheduling and Synchronization*. CRC press, 2nd ed.

Stark, E. W., 1995: An algebra of dataflow networks. *Fundamenta Informaticae*, **22(1-2)**, 167–185.

Stephens, R., 1997: A survey of stream processing. *Acta Informatica*, **34(7)**.

Stuijk, S., M. C. Geilen, and T. Basten, 2008: Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, **57(10)**, 1331–1345. doi:10.1109/TC.2008.58.

Talcott, C. L., 1996: Interaction semantics for components of distributed systems. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pp. 154–169. doi:10.1007/978-0-387-35082-0_11.

Thies, W., M. Karczmarek, and S. Amarasinghe, 2002: StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, Springer-Verlag, vol. LNCS 2304. doi:10.1007/3-540-45937-5_14.

Thies, W., M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, 2005: Teleport messaging for distributed stream programs. In *Principles and Practice of Parallel Programming (PPoPP)*, ACM. doi:10.1145/1065944.1065975.

Turing, A. M., 1936: On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, **42**, 230–265.

Turjan, A., B. Kienhuis, and E. Deprettere, 2003: Solving out-of-order communication in Kahn process networks. *Journal on VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, **40(1)**, 7–18. doi:10.1007/s11265-005-4935-5.

von Hanxleden, R., T. Bourke, and A. Girault, 2017: Real-time ticks for synchronous programming. In *2017 Forum on Specification and Design Languages (FDL)*, IEEE, pp. 1–8.

von Hanxleden, R. et al., 2014: SCCharts: Sequentially constructive Statecharts for safety-critical applications. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ACM, New York, NY, USA, PLDI '14, ISBN 978-1-4503-2784-8, pp. 372–383. doi:10.1145/2594291.2594310.

Vuillemin, J., 1973: *Proof Techniques for Recursive Programs*. Ph. d. thesis.

Wegner, P., 1998: Interactive foundations of computing. *Theoretical Computer Science*, **192(2)**, 315–351.

Winskel, G., 1993: *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, MA, USA.

Wisniewski, R., I. Grobelna, and A. Karatkevich, 2020: Determinism in cyber-physical systems specified by interpreted petri nets. *Sensors*, **20(5565)**, 22. doi:10.3390/s20195565.

Wolfram, S., 2002: *A New Kind of Science*. Wolfram Media, Inc.

Wolpert, D. H., 2008: Physical limits of inference. *Physica*, **237(9)**, 1257–1281. doi:10.1016/j.physd.2008.03.040.

Zhao, Y., E. A. Lee, and J. Liu, 2007: A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, pp. 259 – 268. doi:10.1109/RTAS.2007.5.

Zinkernagel, H., 2010: *Causal Fundamentalism in Physics*, Springer, Dordrecht.

Index