This is a chapter from the book

System Design, Modeling, and Simulation using Ptolemy II

First Edition, Version 1.0

Please cite this book as:

Claudius Ptolemaeus, Editor,
System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014.
http://ptolemy.org/books/Systems.

3 Dataflow

Edward A. Lee, Stephen Neuendorffer, Gang Zhou

Contents

Ptolemy II was created to enable heterogeneous models to be developed and simulated together as part of an overall system model. As discussed in previous chapters, a key innovation in Ptolemy II is that, unlike other design and modeling environments, Ptolemy II supports multiple models of computation that are tailored to specific types of modeling problems. These models of computation define how the model will behave and are determined by the director that is used within that model. In Ptolemy II terminology, the director realizes a domain, which is an implementation of a model of computation. Thus, the director, domain, and model of computation are all tied together; when you construct a model that contains an SDFDirector (a synchronous dataflow director), for example, you have constructed a model "in the SDF domain," using the SDF model of computation.

This chapter describes the **dataflow domains** that are currently available in Ptolemy II, which include synchronous (static) and dynamic dataflow models. Dataflow domains are appropriate for applications that involve processing streams of data values. These streams can flow through sequences of actors that transform them in some way. Such models are often called **pipe and filter** models, because the connections between actor are analogous to pipes that carry flows, and the actors are analogous to filters the change the flows in some way. Dataflow domains mostly ignore time, although SDF is capable of modeling streams with uniformly spaced time between iterations. Subsequent chapters discuss other domains and their selection and use.

3.1 Synchronous Dataflow

The **Synchronous dataflow** (**SDF**) domain, also called **static dataflow**,[1] was introduced by Lee and Messerschmitt (1987b), and is one of the first domains (or models of computation) developed for Ptolemy II. It is a specific type of **dataflow** model. In dataflow models, actors begin execution (they are fired) when their required data inputs become available. SDF is a relatively simple case of dataflow; the order in which actors are exe-

[1]The term "synchronous dataflow" can cause confusion because it is not synchronous in the sense of SR, considered in Chapter 5. 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.

cuted is static, and does not depend on the data that is processed (the values of the tokens that are passed between actors).

In a **homogeneous SDF** model, an actor fires when there is a token on each of its input ports and produces a token on each output port. In this case, the director simply has to ensure that each actor fires after the actors that supply it with data, and an iteration of the model consists of one firing of each actor. Most of the examples in Chapter 2 were homogeneous SDF models.

Not all actors produce and consume a single token each time they are fired, however; some require multiple input tokens before they can be fired and produce multiple output tokens. The SDF scheduler, which is responsible for determining the order in which actors are executed, supports more complex models than homogeneous SDF. It is capable of scheduling the execution of actors with arbitrary data rates, as long as these rates are given by specifying the number of tokens consumed and produced by the firing of each actor on each port.

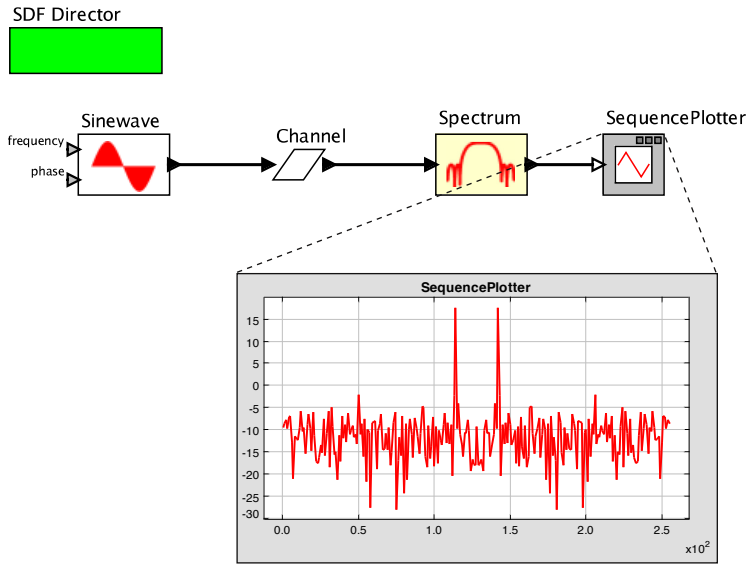


Figure 3.1: A multirate SDF model. The Spectrum actor requires 256 tokens to fire, so one iteration of this model requires 256 firings of Sinewave, Channel, and SequencePlotter, and one firing of Spectrum. [online]

Example 3.1: One example of an actor that requires multiple input tokens to fire is the Spectrum actor (see box on page 101). Figure 3.1 shows a system that computes the spectrum of the same noisy sine wave that we constructed in Figure 2.20. The Spectrum actor has a single parameter that specifies the order of the fast Fourier transform (**FFT**) used to calculate the spectrum. Figure 3.1 shows the output of the model with *order* set to 8 and the number of iterations set to 1. (See Chapter 17, Section 17.2 to improve the labeling of the plot.)

When the *order* parameter is set to 8, the Spectrum actor requires 2^8 (256) input samples to fire, and produces 2^8 output samples. In order for the Spectrum actor to fire once, the actors that supply its input data, Sinewave and Channel, must each fire 256 times. The SDF director extracts this relationship and defines one iteration of the model to consist of 256 firings of Sinewave, Channel, and SequencePlotter, and one firing of Spectrum.

This example implements a **multirate** model; that is, the firing rates of the actors are not identical. In particular, the Spectrum actor executes at a different rate than the other actors. It is common for the execution of a multirate model to consist of exactly one iteration. The director determines how many times to fire each actor in an iteration using balance equations, as described in the next section.

3.1.1 Balance Equations

Consider a single connection between two actors, A and B, as shown in Figure 3.2. 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 nonnegative integers. Suppose that A fires q_A times and B fires q_B times. All tokens that A produces are

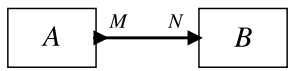


Figure 3.2: SDF actor A produces M tokens when it fires, and actor B consumes N tokens when it fires.

consumed by B if and only if the following **balance equation** is satisfied,

$$q_A M = q_B N. \tag{3.1}$$

Given values q_A and q_B satisfying (3.1), 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 (3.1). 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 positive q_A and q_B that satisfy (3.1). 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 round of the schedule, called an iteration, actor B consumes exactly as many tokens as actor A produces.

Example 3.2: Suppose that in Figure 3.2, $M = 2$ 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:

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

In fact, 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 (3.1) 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.

The SDF director, by default, finds the least positive integer solution to the balance equations, and constructs a schedule that fires the actors in the model the requisite number of times, given by this solution. An execution sequence that fires the actors exactly as many times as specified by this solution is called a **complete iteration**.

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 positive integer solution is not entirely trivial.

Example 3.3: Figure 3.3 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 3.4: Figure 3.4 shows a network with three SDF actors where the only solution to the balance equations is the trivial one, $q_A = q_B = q_C = 0$. A conse-

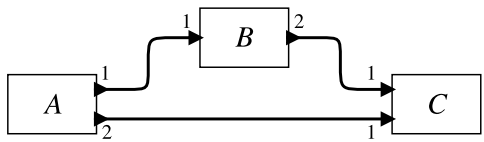


Figure 3.3: A consistent SDF model.

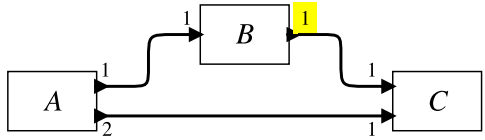


Figure 3.4: An inconsistent SDF model.

quence is that there is no unbounded execution with bounded buffers for this model. It cannot be kept in balance.

An SDF model that has a non-zero 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.

Lee and Messerschmitt (1987b) showed that if the balance equations have a non-zero solution, then they also have a solution where q_i is a nonnegative integer for all actors i. Moreover, for connected models (where there is a communication path between any two actors), they give a procedure for finding the least positive integer solution. Such a procedure forms the foundation for a scheduler for SDF models.

Example 3.5: Figure 3.5 shows an SDF model that makes extensive use of the multirate capabilities of SDF. The **AudioCapture** actor captures sound from the microphone on the machine on which the models run, producing a sequence of samples at a default rate of 8,000 samples per second. The Chop actor extracts chunks of 128 samples from each input block of 500 samples (see box on page 106). The Spectrum actor computes the power spectrum, which measures the power as a function of frequency (see box on page 101). The two SequenceToArray actors (box on page 106) construct arrays that are then plotted using ArrayPlotter actors (see Chapter 17). The particular plots that are shown are the response to a whistle. Notice the peaks in the spectrum at about 1,700 Hz and -1,700 Hz.

The SDF director in this model figures out that the AudioCapture actor needs to fire 500 times for each firing of Chop, Spectrum, and SequenceToArray, and the plotters.

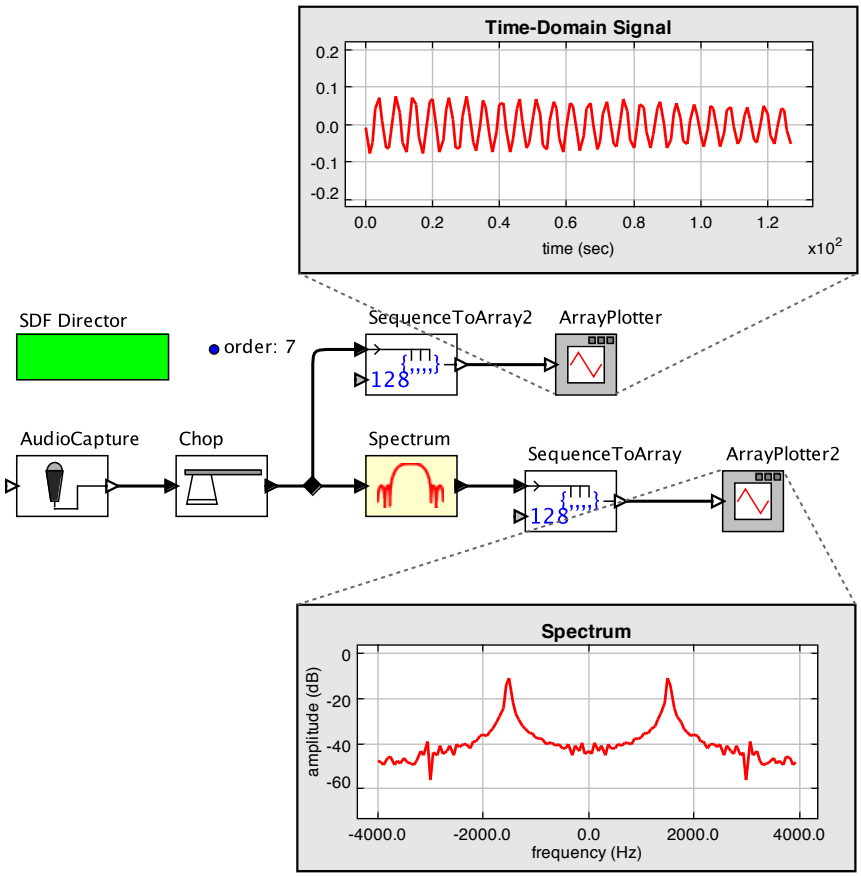


Figure 3.5: Model that computes the power spectrum of the audio signal captured from the microphone. The plots here show a whistle at about 1,700 Hz. [online]

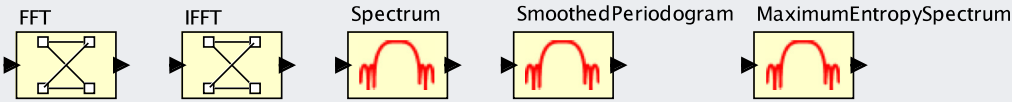
Sidebar: SDF Schedulers

A key 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 strategies, other scheduling optimizations are also 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. Synchronization ensures that an actor does not fire before it receives the data it needs to fire. However, synchronization is not required if a previous synchronization already provides assurance that the data are present. By manipulating the schedule, one can minimize the number of required synchronization points.

Sidebar: Frequency Analysis

The SDF domain is particularly useful for signal processing. One of the basic operations in signal processing is to convert a time domain signal into a frequency domain signal and vice versa (see Lee and Varaiya (2011)). Actors that support this operation are found in the `Actors→SignalProcessing→Spectrum` library, and shown below:



- **FFT** and **IFFT** calculate the discrete Fourier transform (DFT) and its inverse, respectively, of a signal using the fast Fourier transform algorithm. The *order* parameter specifies the number of input tokens that are used for each FFT calculation. It is a "radix two" algorithm, which implies that the number of tokens is required to be a power of two, and the *order* parameter specifies the exponent. For example, if *order*=10, then the number of input tokens used for each firing is $2^{10} = 1024$. The remaining actors implement various spectral estimation algorithms, and are all composite actors that use FFT as a component. These algorithms output signal power in decibels (dB) as a function of frequency. The output frequency ranges from $-f_N$ to f_N, where f_N is the Nyquist frequency (half the sampling frequency). That is, the first half of the output represents negative frequencies and the second half represents positive frequencies.
- **Spectrum** is the simplest of the spectral estimators. It calculates the FFT of the input signal and converts the result to a power measurement in dB.
- **SmoothedPeriodogram** calculates a power spectrum by first estimating the autocorrelation of the input. This approach averages the inputs and is less sensitive to noise.
- **MaximumEntropySpectrum** is a parametric spectral estimator; it uses the Levinson-Durbin algorithm to construct the parameters of autoregressive (AR) models that could plausibly have generated the input signal. It then selects the model that maximizes the entropy (see Kay (1988)). It is the most sophisticated of the spectral estimators and typically produces the smoothest estimates.

Outputs from the three spectral estimators are compared in Figure 3.6, where the input consists of three sinusoids in noise. Choosing the right spectral estimator for an application is a sophisticated topic, beyond the scope of this book.

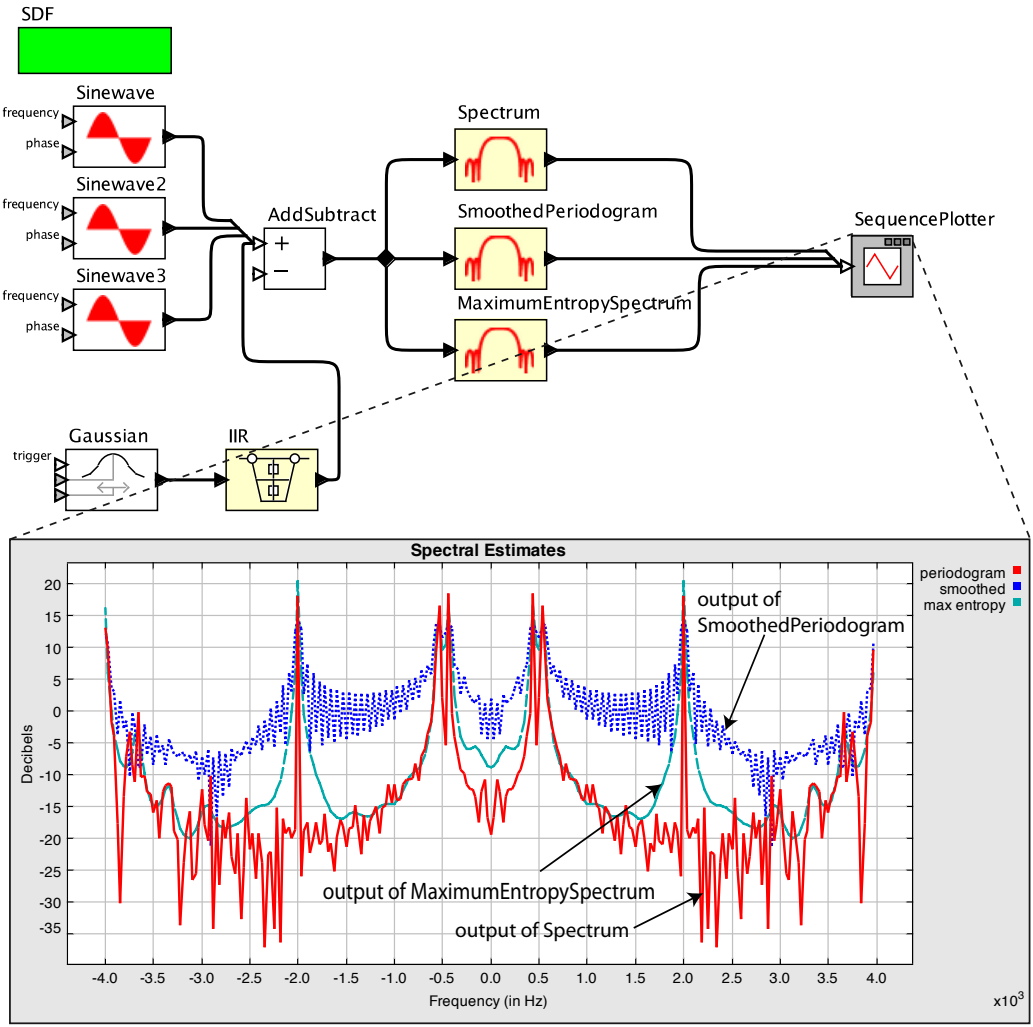


Figure 3.6: Comparison of three spectral estimation techniques described in the box on page 101. [online]

3.1.2 Feedback Loops

A feedback loop in SDF must include at least one instance of the **SampleDelay** actor (found in the `FlowControl→SequenceControl` sublibrary). Without this actor, the loop would **deadlock**; actors in the feedback loop would be unable to fire because they depend on each other for tokens. The SampleDelay actor resolves this problem by producing initial tokens on its output before the model begins firing. The initial tokens are specified by the *initialOutputs* parameter, which defines an array of tokens. These initial tokens enable downstream actors to fire and break the circular dependencies that would otherwise result from a feedback loop.

Example 3.6: Consider the model in Figure 3.7. This homogeneous SDF model generates a counting sequence using a feedback loop. The SampleDelay actor begins the process by producing a token with value of 0 on its output. This token, together with a token from the Const actor, enables the AddSubtract actor to fire. The output of that actor enables the next firing of SampleDelay. After the initial firing, the SampleDelay copies its input to its output unchanged.

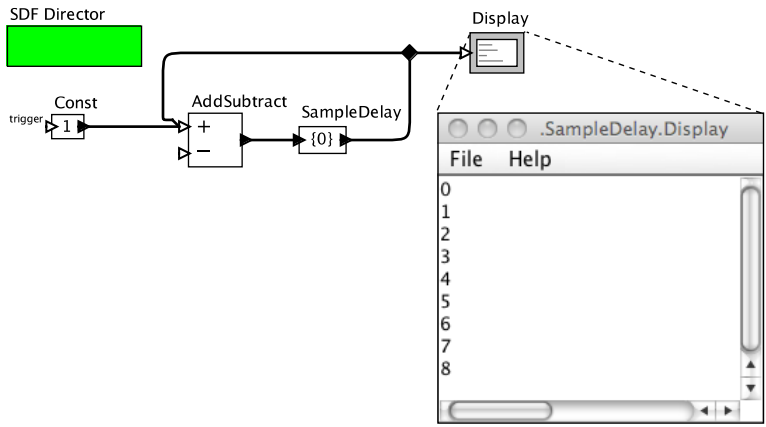


Figure 3.7: An SDF model with a feedback loop must have at least one instance of the SampleDelay actor in it. [online]

Consistency is sufficient to ensure bounded buffers, but it is not sufficient to ensure that an unbounded execution exists. The model may deadlock even if it is consistent. The SDF director analyzes a model for both consistency and deadlock. To allow feedback, it treats **delay actors** differently than other actors. A delay actor is able to produce initial output tokens before it receives any input tokens. It subsequently behaves like a normal SDF actor, consuming and producing a fixed number of tokens on each firing. In the SDF domain, the initial tokens are understood to be initial conditions for execution rather than part of the execution itself. Thus, the scheduler will ensure that all initial tokens are produced before the SDF execution begins. Conceptually, the SampleDelay actor could be replaced by initial tokens placed on a feedback connection.

Example 3.7: Figure 3.8 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 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, because 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,

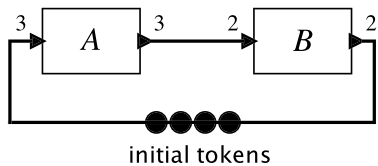


Figure 3.8: An SDF model with initial tokens on a feedback loop. In Ptolemy II, these initial tokens would be provided by a SampleDelay actor.

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 condition.

Were there any fewer than four initial tokens on the feedback path, however, 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.

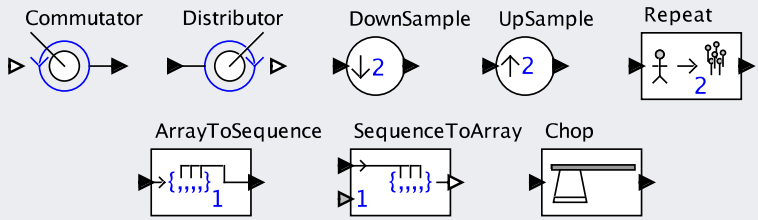
Lee and Messerschmitt (1987b) discuss the procedure for solving the balance equations, along with a procedure that will either provide a schedule for an unbounded execution or prove that no such schedule exists. Using these procedures, both bounded buffers and deadlock are **decidable** for SDF models (meaning that it is possible for Ptolemy to determine whether deadlock or unbounded buffers occur in any SDF model).

3.1.3 Time in Dataflow Models

In the SDF examples we have considered thus far, we have used the SequencePlotter actor but not the TimedPlotter actor (see Chapter 17). This is because the SDF domain does not generally use the notion of time in its models. By default, time does not advance as an SDF model executes (though the SDFDirector does contain a parameter, called *period*, that can be used to advance time by a fixed amount on each iteration of the model). Therefore, in most SDF models, the TimedPlotter actor would show the time axis as always being equal to zero. The SequencePlotter actor, in contrast, plots a sequence of values that are not time-based, and is therefore frequently used in SDF models. The discrete event (DE) and Continuous domains, discussed in Chapters 7 and 9, include a much stronger notion of time, and often use the TimedPlotter.

Sidebar: Multirate Dataflow Actors

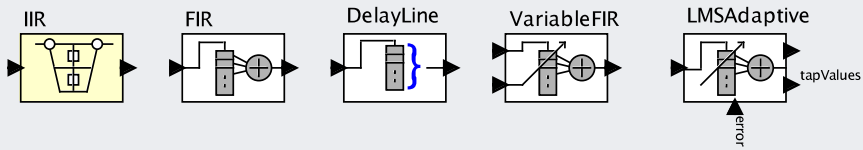
The Ptolemy II library offers a few actors that produce and/or consume multiple tokens per firing on a port. The most basic ones are shown below:



- **Commutator** and **Distributor**, in the `FlowControl→Aggregators` sublibrary, convert tokens arriving from multiple signals into a sequence of tokens and vice versa. Commutator has a multiport input, and on each firing, it reads a fixed number of tokens (given by its *blockSize* parameter) from each input channel, and outputs all the tokens from all the input channels as a sequence. Distributor reverses this process.
- **DownSample** and **UpSample**, in `SignalProcessing→Filtering`, discard or insert tokens. Downsample reads a fixed number of tokens (given by its *factor* parameter), and outputs one of those tokens (selected by its *phase* parameter). UpSample inserts a fixed number of zero-valued tokens between input tokens.
- **Repeat**, found under `FlowControl→SequenceControl`, is similar to UpSample except that instead of inserting zero-valued tokens, it repeats the input token.
- ArrayToSequence and SequenceToArray, found in the `Array` library, convert array tokens into sequences of tokens and vice versa. Both actors have an *arrayLength* parameter that specifies the length of the incoming (or outgoing) array. ArrayToSequence also has an *enforceArrayLength* parameter, which, if set to `true`, causes the actor to generate an error message if it receives an array of the wrong length. In SequenceToArray, *arrayLength* is a PortParameter, and hence the number of input tokens that are read can vary. These actors are SDF actors only when the array length is constant.
- **Chop**, in `FlowControl→SequenceControl`, reads a specified number of input tokens and produces a specified subset of those inputs, possibly padded with zero-valued tokens or previously consumed tokens.

Sidebar: Signal Processing Actors

In addition to the spectral analysis actors described on page 101, Ptolemy II includes several other key signal processing actors, as shown below.



- **IIR** implements an infinite impulse response filter, also called a recursive filter (see Lee and Varaiya (2011)). Filter coefficients are provided in two arrays, one for the numerator and one for the denominator polynomial of the transfer function.
- **FIR** implements a finite impulse response filter, also called a tapped delay line, with coefficients specified by the *taps* parameter. Whereas IIR is a homogeneous SDF (single-rate) actor, FIR is potentially a multirate actor. When the *decimation* (*interpolation*) parameters are not equal to 1, the filter behaves as if it were followed (preceded) by a DownSample (UpSample) actor (see sidebar on page 106). However, the implementation is much more efficient than it would be using UpSample or DownSample actors; a polyphase structure is used internally, avoiding unnecessary use of memory and unnecessary multiplication by zero. Arbitrary sample-rate conversions by rational factors can be accomplished in this manner.
- **DelayLine** produces an array rather than the scalar produced by FIR. Instead of a weighted average of the contents of the delay line (which is what FIR produces), DelayLine simply outputs the contents of the delay line as an array.
- **VariableFIR** is identical to FIR except that the coefficients are provided as an array on an input port (and thus can vary) rather than being defined as actor parameters.
- **LMSAdaptive** is similar to FIR, except that the coefficients are adjusted on each firing using a gradient descent adaptive filter algorithm that attempts to minimize the power of the signal at the *error* input port.

In addition to the actors described here, the signal processing library includes fixed and adaptive lattice filters, statistical analysis actors, actors for communications systems (such as source and channel coders and decoders), audio capture and playback, and image and video processing actors. See the actor index on page 632.

Sidebar: Dynamically Varying Rates

A variant of SDF that is called **parameterized SDF** (**PSDF**), introduced by Bhattacharya and Bhattacharyya (2000), allows the production and consumption rates of ports to be given by a parameter rather than being a 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.

The example in Figure 3.9 illustrates how PSDF can be achieved with the SDF director in Ptolemy II. First, notice that the director's *allowRateChanges* parameter has been set to true. This indicates to the director that it may need to compute more than one schedule, since rate parameters may change during the execution of the model.

Second, notice that the Repeat actor's *numberOfTimes* parameter is set equal to the model parameter *rate*, which initially has value zero. Hence, when this model executes its first iteration, the Repeat actor will produce zero tokens, so the Display actor will not fire. The initial output from the Ramp actor, which has value 1, will not be displayed.

During this first iteration, the Expression and SetVariable actor both fire once. The Expression actor sets its output equal to input, unless the input is equal to the value of the *iterations* parameter (which it doesn't in this first iteration). The SetVariable actor sets the value of the *rate* parameter to 1. By default, SetVariable has a *delayed* parameter with value true, which means that the *rate* parameter changes only after the current iteration is complete.

In the second iteration, the value of the *rate* parameter is 1, so the Repeat actor copies its input (which has value 2) once to its output. The Expression and SetVariable actors set the *rate* parameter now to 2, so in the third iteration, the Repeat actor copies its input (which has value 3) twice to its output. The sequence of displayed outputs is therefore $2, 3, 3, 4, 4, 4, \cdots$.

To stop the model, the *iterations* parameter of the director is set to 5. In the last iteration of the execution, the Expression actor ensures that the *rate* parameter gets reset to 0. Hence, the next time the model executes, it will start again with the *rate* parameter set to 0.

In this example, each time the *rate* parameter changes, the SDF director recomputes the schedule. In a better implementation of PSDF, it would probably precompute schedules and/or cache previously computed schedules, but this implementation does not do that. It just recomputes the schedule between iterations.

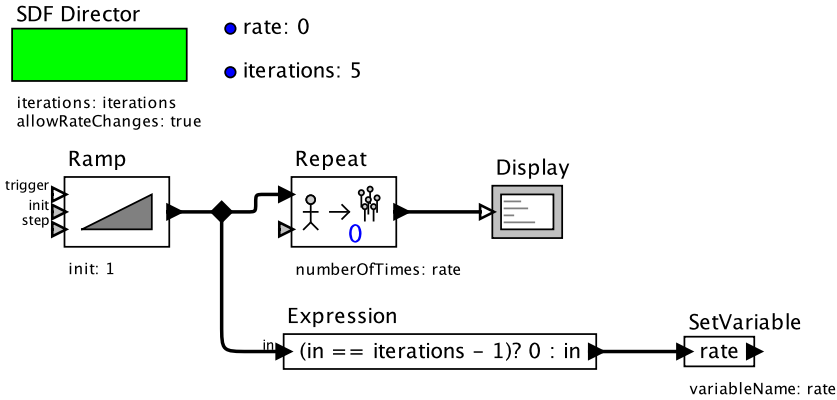


Figure 3.9: An SDF model with dynamically varying rates. [online]

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: Other Variants of 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. DDF (Section 3.2) relaxes this constraint at the cost of being able to statically precompute the firing schedule. In addition, as discussed earlier in the chapter, it is no longer possible to analyze all models for deadlock or bounded buffers (these questions are undecidable). Researchers have developed a number of variants of dataflow, however, that are more expressive than SDF but still amenable to some forms of static analysis.

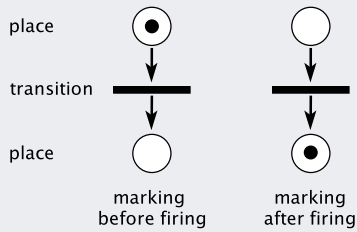
Cyclo-static dataflow (**CSDF**) allows production and consumption rates to vary in a periodic manner (Bilsen et al., 1996). An example is the **SingleTokenCommutator** actor (in `FlowControl→Aggregators`). This actor is similar to the Commutator actor (see sidebar on page 106), but instead of consuming all inputs in a single firing, it consumes inputs from only one channel in each firing, and rotates through the input channels on successive firings. For each input channel, the consumption rate alternates between zero and one. This actor is useful in feedback systems where the input to the second channel depends on the input to the first channel.

SDF can also be combined hierarchically with finite state machines (FSMs) to create modal models, described in Chapter 8. Each state of the FSM is associated with a submodel (a mode refinement, where each refinement can have different production and consumption rates). If the state transitions of the FSM are constrained to occur only at certain times, the model remains decidable. This combination was introduced by Girault et al. (1999), who called it **heterochronous dataflow** (**HDF**). **SDF Scenarios** (Geilen and Stuijk, 2010) are similar to HDF in that they also use an FSM, but rather than having mode refinements, in SDF Scenarios each state of the FSM is associated with a set of production and consumption rates for a single SDF model. Bhattacharya and Bhattacharyya (2000) introduced parameterized SDF (PSDF), where production and consumption rates can depend on input data as long as the same dependence can be represented in parameterized schedule.

Murthy and Lee (2002) introduced **multidimensional SDF** (**MDSDF**). Whereas a channel in SDF carries a sequence of tokens, a channel in MDSDF carries a multi-dimensional 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.

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.



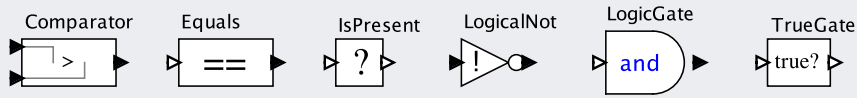
Once a transition is enabled, it can **fire**, consuming one token from each input place and depositing one token on each output place. The state of a network, called its **marking**, is the number of tokens on each place in the network. The figure above shows the marking of a simple network before and 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, and firing of a transition does not involve any computation on tokens. A firing is just the act of moving 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, resulting in nondeterminism.

There are many variants of Petri nets, at least one of which is equivalent to SDF. In particular, tokens can have values (such tokens are called **colored tokens** in the literature, where the color of a token represents its value). Transitions can manipulate the color of tokens (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. This is analogous to SDF production and consumption rates. And finally, the Petri net graph structure 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.

Sidebar: Logic Actors

The actors found in the `Logic` library are useful for building control logic:



- The **Comparator** actor compares two values of type *double* (or of any type that can be losslessly converted to *double*, as explained in Chapter 14). The available comparisons are $>$, $>=$, $<$, $<=$, and $==$. The output is a boolean.
- The **Equals** actor compares any number of input tokens of any type for equality and outputs a boolean true if they are equal and a false otherwise.
- The IsPresent actor outputs a boolean true if the input is present when it fires and false otherwise. In dataflow domains, the input is always present, so the output will always be true. This actor is more useful in the SR and DE domains (Chapters 5 and 7).
- **LogicalNot** accepts an input boolean and outputs the converse boolean.
- **LogicGate** implements the following six logic functions (the icon changes when you select the logic function):

- The TrueGate actor produces a boolean true output when the input is a boolean true. Otherwise, it produces no token at all. This is clearly not an SDF actor, but it can be used with DDF. It is also useful in SR (Chapter 5).

3.2 Dynamic Dataflow

Although the ability to guarantee bounded buffers and rule out deadlock is valuable, it comes at a price: SDF is not very expressive. It cannot directly express conditional firing, for example, such as when an actor fires only if a token has a particular value.

A number of dataflow variants have been developed that loosen the constraints of SDF; several of these are discussed in the sidebar on page 110. In this section, we describe a variant known as **dynamic dataflow** (**DDF**). DDF is much more flexible than SDF, because actors can produce and consume a varying number of tokens on each firing.

3.2.1 Firing Rules

As in other dataflow MoCs (such as SDF) DDF actors begin execution when they have sufficient input data. For a given actor to fire, its **firing rule** (that is, the condition that must be met before an actor can fire) must be satisfied. In SDF, the actors' firing rule is constant. It simply specifies the fixed number of tokens that are required on each input port before the actor can fire. In the DDF domain, however, firing rules can be more complicated, and may specify a different number of tokens for each firing.

Example 3.8: The SampleDelay actor of Example 3.6 is directly supported by the DDF MoC, without any need for special treatment of initial tokens. Specifically, the firing rule for SampleDelay states that on the first firing, it requires no input tokens. On subsequent firings, it requires one token.

Another difference is that, in SDF, actors produce a fixed number of tokens on each output port. In DDF, the number of tokens produced can vary.

Example 3.9: On its first firing, the SampleDelay actor produces the number of tokens specified in its *initialOutputs* parameter. On subsequent firings it produces a single token that is equal to the token it consumed.

The firing rules themselves need not be constant. Upon firing, a DDF actor may change the firing rules for the next firing.

A key DDF actor is the BooleanSelect, which merges two input streams into one stream according to a stream of boolean-valued control tokens (see sidebar on page 119). This actor has three firing rules. Initially, it requires one token on the *control* (bottom) port, and no tokens on the other two ports. When it fires, it records the value of the control token and changes its firing rule to require a token on one of the *trueInput* port (labeled *T*) or the *falseInput* port (labeled *F*), depending on the value of the control token. When the actor next fires, it consumes the token on the corresponding port and sends it to the output. Thus, it fires twice to produce one output. After producing an output, its firing rule reverts to requiring a single token on the *control* port.

A more general version of the BooleanSelect is the Select actor, which merges an arbitrary number of input streams into one stream according to a stream of integer-valued control tokens, rather than just two streams (see sidebar on page 119).

Whereas BooleanSelect and Select merge multiple input streams into one, BooleanSwitch and Switch do the converse; they split a single stream into multiple streams. Again, a stream of control tokens determines, for each input token, to which output stream that token should be sent. These Switch and Select actors accomplish conditional routing of tokens, as illustrated in the following examples.

Example 3.10: Figure 3.10 uses BooleanSwitch and BooleanSelect to accomplish conditional firing, the equivalent of if-then-else in an imperative programming language. In this figure, the **Bernoulli** actor produces a random stream of Boolean-valued tokens. This control stream controls the routing of tokens produced by the Ramp actor. When Bernoulli produces `true`, the output of the Ramp actor is multiplied by -1 using the Scale actor. When Bernoulli produces `false`, Scale2 is used; it passes its input through unchanged. The BooleanSelect uses the same control stream to select the appropriate Scale output.

Example 3.11: Figure 3.11 shows a DDF model that uses BooleanSwitch and BooleanSelect to realize data-dependent iteration using a feedback loop. The Ramp

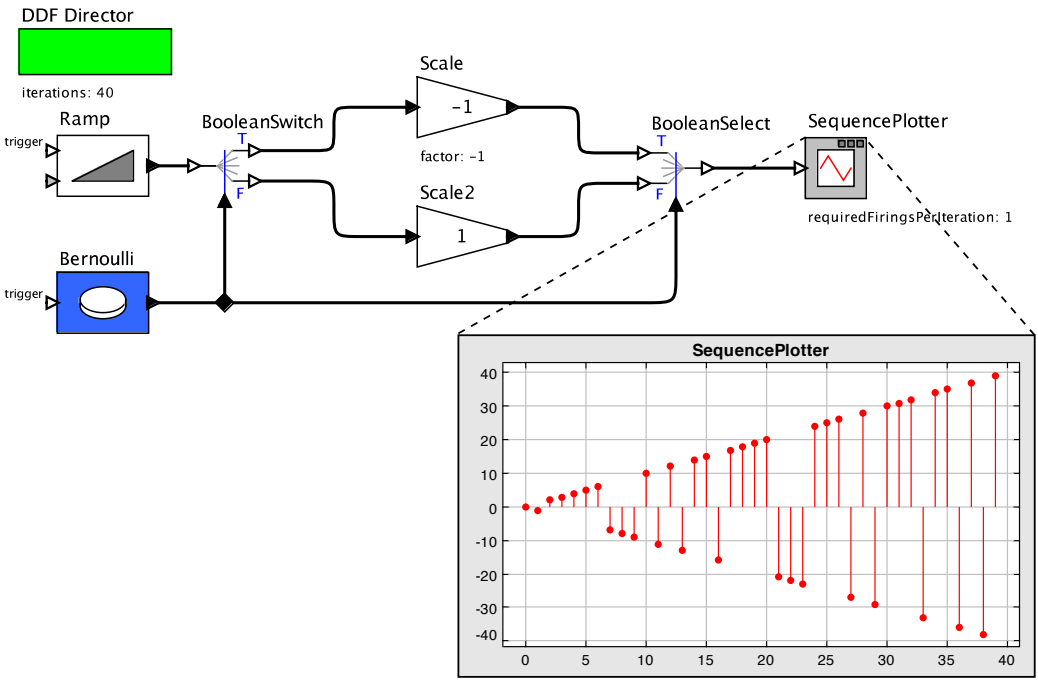


Figure 3.10: A DDF model that accomplishes conditional firing. [online]

actor feeds the loop with a sequence of increasing integers, 0, 1, 2, 3, etc. The SampleDelay initiates the loop by providing a `false` token to the *control* port of the BooleanSelect. In the full cycle, each input integer is repeatedly multiplied by 0.5 until the resulting value is less than 0.5. The Comparator actor (found in the `Logic` library) controls whether the token is routed back around the loop for another iteration or routed out of the loop to the Discard actor (the one at the right with the icon that looks like a ground symbol on an electrical circuit diagram, found in `Sinks→GenericSinks`). The Discard actor receives and discards its input, but in this case, it is also used to control what an iteration means. The parameter *requiredFiringsPerIteration* has been added to the actor and assigned a value of 1 (see Section 3.2.2 below). Hence, one iteration of the model consists of as many iterations of the loop as needed to produce one firing of Discard. This structure is analogous to a do-while loop in an imperative programming language.

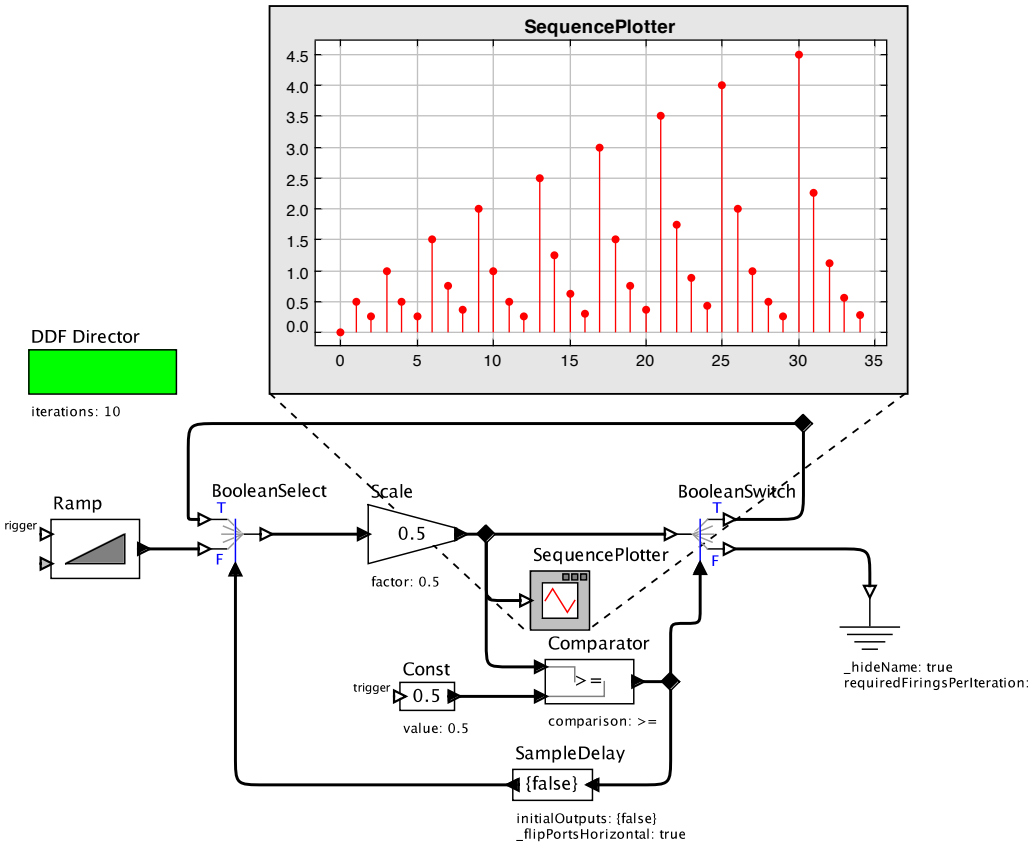


Figure 3.11: A DDF model that accomplishes data-dependent iteration. [online]

The pattern shown in Figure 3.11 is sufficiently useful that it might be used repeatedly. Fortunately, Ptolemy II includes a mechanism for storing and re-using a **design pattern**, created by Feng (2009). The pattern shown in Figure 3.12, for example, is available as a unit in the `MoreLibraries→DesignPatterns`. In fact, any Ptolemy II model can be exported to a library as a design pattern and reimported into another model as a unit by simply dragging it into the model.

The Switch and Select actors (and their boolean versions) that are part of the DDF domain provide increased flexibility and expressiveness relative to the SDF domain, but their use means that it may not be possible to determine a schedule with bounded buffers, nor is

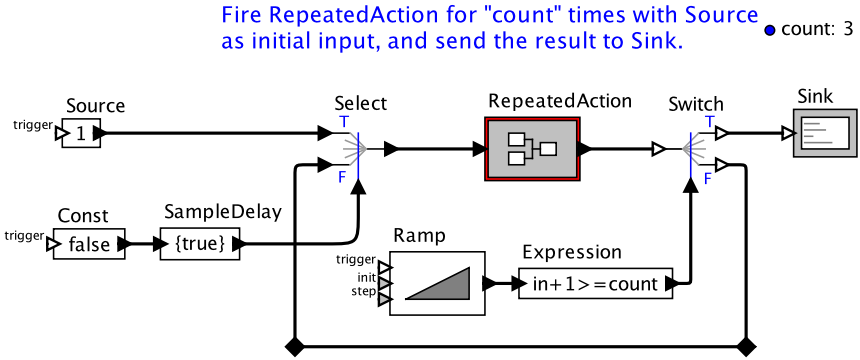


Figure 3.12: A design pattern stored as a unit in a library.

possible to ensure that the model will not deadlock. In fact, Buck (1993) showed that bounded buffers and deadlock are undecidable for DDF models. For this reason, DDF models are not as readily analyzed.

Example 3.12: A variant of the if-then-else model shown in Figure 3.10 is shown in Figure 3.13. In this case, the inputs to the BooleanSelect have been reversed. Unlike the earlier model, this model has no schedule that assures bounded buffers. The Bernoulli actor is capable of producing an arbitrarily long sequence of `true`-valued tokens, during which an arbitrarily long sequence of tokens may build up on input buffer for the *false* port of the BooleanSelect, thus potentially overflowing the buffer.

Switch and Select and their boolean cousins are dataflow analogs of the **goto** statement in imperative languages. They provide low-level control over execution by conditionally routing tokens. Like goto statements, their use can result in models that are difficult to understand. This problem is addressed using structured dataflow, described in the sidebar on page 120, and implemented in Ptolemy II using higher-order actors, described in Section 2.7.

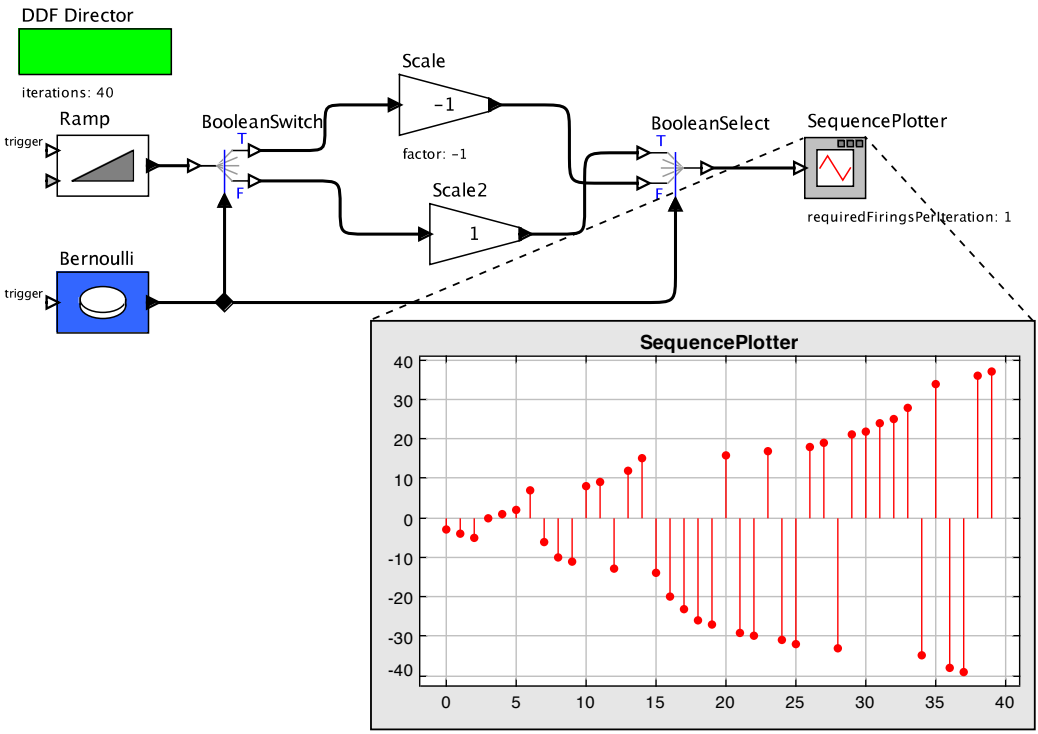


Figure 3.13: A DDF model that has no bounded buffer schedule. [online]

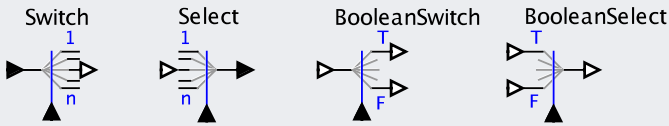
3.2.2 Iterations in DDF

One of the advantages of SDF is that a complete iteration is uniquely defined. It consists of a fixed number of firings of each of the actors in the model. It is therefore easy to control the duration of an overall execution of the model by setting the *iterations* parameter of the SDF director, which controls the number of times each actor will be executed.

The DDF director also has an *iterations* parameter, but defining an iteration is more difficult. An iteration can be defined by adding a parameter to one or more actors named *requiredFiringsPerIteration* and giving that parameter an integer value, as illustrated in the following example.

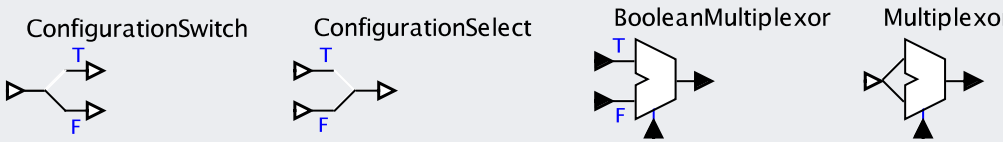
Sidebar: Token Flow Control Actors

Ptolemy II provides a number of actors that can route tokens in a model. The most basic of these are **Switch** and **Select** (and their boolean variants), shown here:



On each firing, Switch consumes one token from the input and an integer-valued token from the *control* port (on the bottom) and routes the input token to the output channel specified by the control token. All other output channels produce no tokens on that firing. Select does the converse, consuming a single token from the channel specified by the control token and sending that token to the output. The other input channels consume no tokens. **BooleanSwitch** (**BooleanSelect**) are variants where the number of outputs (or inputs) is constrained to be two, and the control token is boolean rather than integer valued.

Switch and Select can be compared to the following actors with related functionality:



ConfigurationSwitch is similar to BooleanSwitch except that instead of a *control* input port it has a *parameter* that determines which output to send data to. If the value of the parameter does not change during execution of the model (normally this is the case with parameters), then this actor is an SDF actor that always produces zero tokens on one output and one token on the other. **ConfigurationSelect** is likewise similar to BooleanSelect.

BooleanMultiplexor and **Multiplexor** are similar to BooleanSelect and Select except that they consume one token from *all* input channels. These actors discard all but one of those input tokens, and send that one token to the output. Since these two actors consume and produce exactly one token on every channel, they are homogeneous SDF actors.

Sidebar: Structured Dataflow

In an imperative language, structured programming replaces goto statements (which can be problematic, as described in Dijkstra (1968)) with nested `for` loops, `if-then-else`, `do-while`, and recursion. In structured dataflow, these concepts are adapted to the dataflow modeling environment.

Figure 3.14 shows an alternative way to accomplish the conditional firing of Figure 3.10. The result is an SDF model that has many advantages over the DDF model in Figure 3.10. The **Case** actor is an example of a higher-order actor, like those discussed in Section 2.7. Inside, it contains two sub-models (refinements), one named true that contains a Scale actor with a parameter of -1, and one named default that contains a Scale actor with a parameter of 1. When the control input to the Case actor is true, the true refinement executes one iteration. For any other control input, the default refinement executes.

This style of conditional firing is called **structured dataflow**, because, much like structured programming, control constructs are nested hierarchically. The approach avoids arbitrary data-dependent token routing (which is analogous to avoiding arbitrary branches using goto instructions). Moreover, the use of the Case actors enables the overall model to be an SDF model. In the example in Figure 3.14, every actor consumes and produces exactly one token on every port. Hence, the model is analyzable for deadlock and bounded buffers.

This style of structured dataflow was introduced in LabVIEW, a design tool developed by National Instruments (Kodosky et al., 1991). In addition to providing a conditional operation similar to that of Figure 3.14, LabVIEW provides structured dataflow constructs for iterations (analogous to `for` and `do-while` loops in an imperative language), and for sequences (which cycle through a finite set of submodels). Iterations can be achieved in Ptolemy II using the higher-order actors of Section 2.7. Sequences (and more complicated control constructs) can be implemented using modal models, discussed in Chapter 8. Ptolemy II supports structured recursion using the **ActorRecursion** actor, found in `DomainSpecific→DynamicDataflow` (see Exercise 3). However, without careful constraints, boundedness again becomes undecidable with recursion (Lee and Parks, 1995).

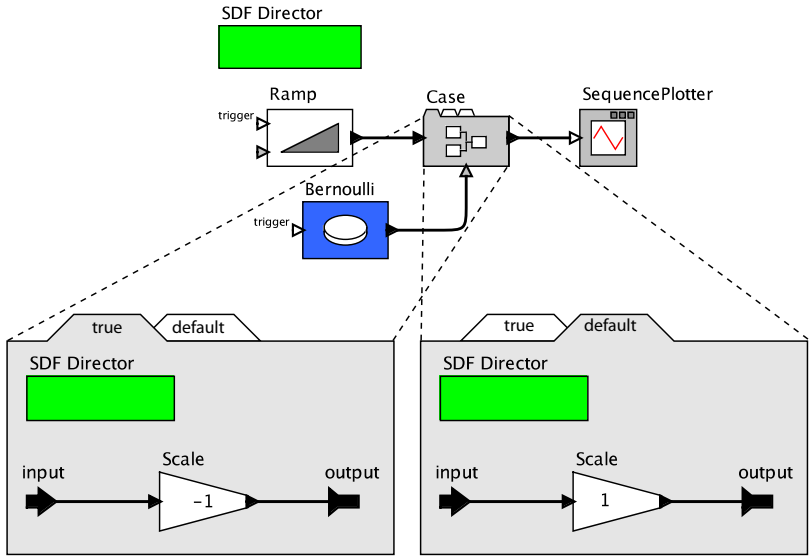


Figure 3.14: Structured dataflow approach to conditional firing. [online]

Example 3.13: Consider the if-then-else example in Figure 3.10, discussed in Example 3.10. The *iterations* parameter of the director is set to 40, and indeed the plot has 40 points. This is because a parameter named *requiredFiringsPerIteration* has been added to the SequencePlotter actor and assigned the value 1. As a consequence, each iteration must include at least one firing of the SequencePlotter. In this case, no other actor in the model has a parameter named *requiredFiringsPerIteration*, so this parameter ends up determining what constitutes an iteration.

When multiple actors within a model have parameters named *requiredFiringsPerIteration*, or when there are no such parameters, the situation is more subtle. In these cases, DDF still has a well-defined iteration, but the definition is complex, and can surprise the designer (see sidebar on page 124).

Example 3.14: Consider again the if-then-else example in Figure 3.10. If we remove the *requiredFiringsPerIteration* from the SequencePlotter, then 40 iterations of the model will produce only nine plotted points. Why? Recall from Section 3.2.1 that BooleanSelect fires twice for each output it produces. Absent any constraints in the model, the DDF director will not fire any actor more than once in an iteration.

Example 3.15: Figure 3.15 shows a DDF model that replaces all instances of the SequencePlotter actor with instances of the Test actor for all Ptolemy models in a directory (see box on page 126 for why you might want to do this). This model uses the DirectoryListing actor (see box on page 128) to construct an array of file names for actors in a specified directory. Ptolemy models are identified by files whose names match the regular expression `.*.xml`, which matches any file name that ends with `.xml`. The *firingCountLimit* parameter of the DirectoryListing ensures that this actor fires only once. It will produce one array token on its output, and then will refuse to fire again. Once the data in that array have been processed, there are no more tokens to process, so the model deadlocks, and stops execution.

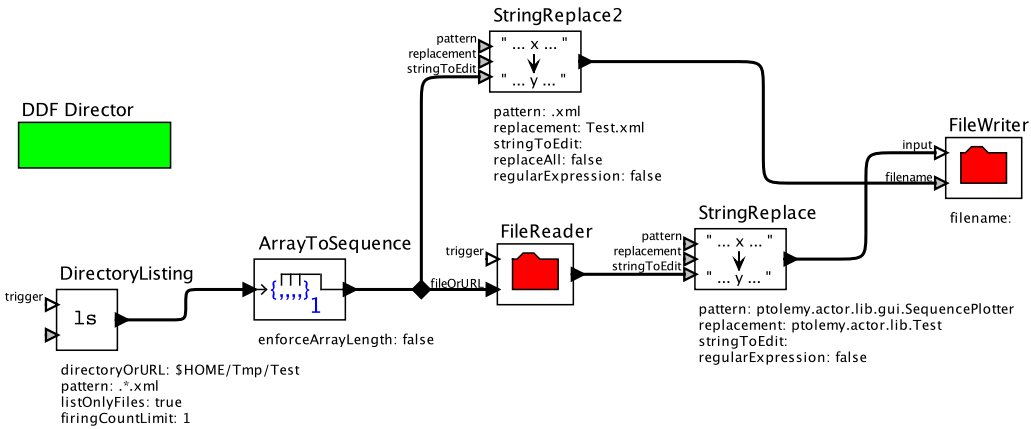


Figure 3.15: A DDF model that replaces all instances of the SequencePlotter actor with instances of the Test actor for all Ptolemy models in a directory.

The ArrayToSequence actor (see box on page 106) converts the array of file names into a sequence of tokens, one string-valued token for each file name. Notice that the *enforceArrayLength* parameter is set to false for this actor. If we were to know exactly how many XML files were in the directory in question, would could have left this parameter with its default value true, set the *arrayLength* parameter to the number of files, and used the SDF director instead of the DDF director. The ArrayToSequence actor would consume one token and produce a fixed, known number of output tokens, and hence would be an SDF actor. But since we do not know in general how many matching files there will be in the directory, the DDF director is more useful.

The FileReader actor (see box on page 128) reads the XML file and outputs its contents as a single string. The StringReplace actor (see box on page 125) replaces all instances of the full classname for the SequencePlotter actor with the full classname for the Test actor.

The second StringReplace actor, named StringReplace2, is used to create a new filename from the original file name. For example, the filename `Foo.xml` will become `FooTest.xml`. The FileWriter actor then writes the modified filename to a file with the new file name.

Note that we could have used the IterateOverArray actor and the SDF director instead (see Section 2.7.2). We leave this as an exercise (see Problem 2 at the end of this chapter).

3.2.3 Combining DDF with Other Domains

Although a system may be best modeled as DDF overall, it may contain some subsystems that can be modeled as SDF. Thus, a DDF model may contain an opaque composite actor that has an SDF director. This approach can improve efficiency and provide better control over the amount of computation done in an iteration.

Conversely, a DDF model may be placed within an SDF model if it behaves like SDF at its input/output boundaries. That is, to be used within an SDF model, a DDF opaque composite actor should consume and produce a fixed number of tokens. It is not generally possible for the DDF director to determine from the model how many tokens are produced and consumed at the boundary (this question is undecidable in general) so it is

Sidebar: Defining a DDF Iteration

An **iteration** in DDF consists of the minimum number of **basic iterations**,(defined below) that satisfy all constraints imposed by *requiredFiringsPerIteration* parameters.

In one **basic iteration**, the DDF director fires all **enabled** and **non-deferrable** actors once. An enabled actor is one that has sufficient data at its input ports, or has no input ports. A **deferrable actor** is one whose execution can be deferred because its execution is not currently required by a downstream actor. This is the case when the downstream actor either already has enough tokens on the channel connecting it to the deferrable actor, or the downstream actor is waiting for tokens on another channel or port. If there are no enabled and non-deferrable actors, then the director fires those enabled and deferrable actors that have the smallest maximum number of tokens on their output channels that will satisfy the demands of destination actors. If there are no enabled actors, then a deadlock has occurred. The above strategy was shown by Parks (1995) to guarantee that buffers remain bounded in an unbounded execution if there exists an unbounded execution with bounded buffers.

The algorithm that implements one basic iteration is as follows. Let E denote the set of enabled actors, and let D denote the set of deferrable enabled actors. One basic (default) iteration then consists of the following, where the notation $E \setminus D$ means "the set of elements in E that are not in D":

if $E \setminus D \neq \emptyset$ **then**
  fire actors in $(E \setminus D)$
else if $D \neq \emptyset$ **then**
  fire actors in minimax(D)
else
  declare deadlock
end if

The function "minimax(D)" returns a subset of D with the smallest maximum number of tokens on their output channels that satisfy the demand of destination actors. This will always include **sink actors** (actors with no output ports).

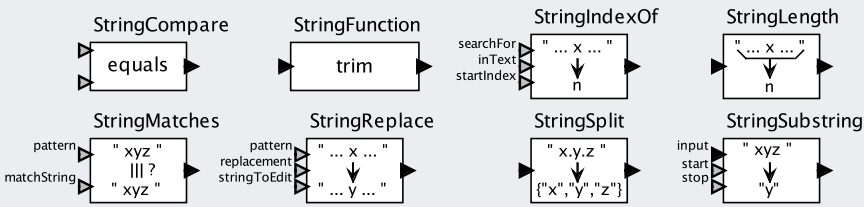
up to the model designer to assert the production and consumption rates. If they are not equal to a value of 1 (which need not be explicitly asserted), then the model designer can assert consumption and production rates by creating a parameter in each input port called *tokenConsumptionRate* and assigning it an integer value. Similarly, output ports should be given a parameter called *tokenProductionRate*.

Once the rates at the boundary are set, it is up to the model designer to ensure that they are respected at run time. This can be accomplished using the *requiredFiringsPerIteration* parameter, as explained above in Section 3.2.2. In addition, the DDF director has a parameter *runUntilDeadlockInOneIteration* that, when set to true, defines an iteration to be repeated invocations of a basic iteration (see sidebar on page 124). until deadlock is reached. If this parameter is used, it overrides any *requiredFiringsPerIteration* that may be present in the model.

DDF conforms with the loose actor semantics, meaning that if a DDF director is used in opaque composite actor, its state changes when its `fire` method is invoked. In particular,

Sidebar: String Manipulation Actors

The `String` library provides actor for manipulating strings:

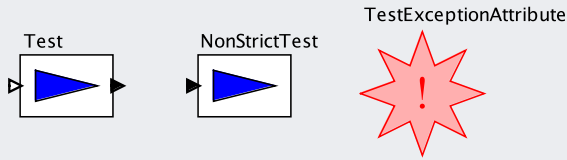


The **StringCompare** actor compares two strings, determine whether they are equal, or if one string starts with, ends with, or contains another string. The **StringMatches** actor checks whether a string matches a pattern given as a regular expression. The **StringFunction** actor can trim white space around a string or convert it to lower case or upper case. The **StringIndexOf** actor searches for a substring within a string a returns the index of that substring. The **StringLength** actor outputs the length of a string. **StringReplace** replaces a substring that matches a specified pattern with a specified replacement string. **StringSplit** divides a string at specified separators. **StringSubstring** extracts a substring, given a start and stop index.

dataflow actors **consume** input tokens in their `fire` method. Once the tokens have been consumed, they are no longer available in the input buffers. Thus, a second firing will see new data, regardless of whether `postfire` has been invoked. For this reason, DDF and SDF composite actors should not be used within domains that require strict actor semantics, such as SR and Continuous, unless the model builder can ensure that these

Sidebar: Building Regression Tests

When developing nontrivial models and when extending Ptolemy II, good engineering practice requires creating **regression tests**. Regression tests guard against future changes that may change behavior in ways that can invalidate applications that were created earlier. Fortunately, in Ptolemy II, it is extremely easy to create regression tests. Key components are found in `MoreLibraries→RegressionTest`:



The **Test** actor compares the inputs against the value specified by the *correctValues* parameter. The actor has a *trainingMode* parameter, which when set to true, simply records the inputs it receives. Therefore, a typical use is to put the actor in training mode, run the model, take the actor out training mode, and then save the model in some directory where all models are executed as part of daily testing. (This is how the vast majority of the rather extensive regression tests for the Ptolemy II itself are created.) The model will throw an exception if the Test actor receives any input that differs from the ones it recorded. Note that one of the key advantages of determinate models is the ability to construct such regression tests.

The **NonStrictTest** is similar, except that it tolerates (and ignores) absent inputs, and it tests the inputs in the postfire phase of execution rather than the fire phase. It is useful for domains such as SR and Continuous, which iterate to a fixed point.

Sometimes, a model is expected to throw an exception. A regression test for such a model should include an instance of **TestExceptionAttribute**, which also has a training mode. The presence of this attribute in a model causes the model to throw an exception if the execution of the model does *not* throw an exception, or if the exception it throws does not match the expected exception.

composite actors will not be fired more than once in an iteration of the SR of Continuous container.

Note that any SDF model can be run with a DDF Director. However, the notion of iteration may be different. Sometimes, a DDF model may be run with an SDF director even when there is data-dependent iteration. Figure 3.14 shows one example, where the Case actor facilitates this combination. However, it is sometimes possible to use this combination even when a Switch is used. The SDF scheduler will assume the Switch produces one token on every output channel, and will construct a schedule accordingly. While executing this schedule, the director may encounter actors that it expects to be ready to fire but which do not actually have sufficient input data to fire. Many actors can be safely iterated even if they have no input data. Their `prefire` method returns false, indicating to the director that they are not ready to fire. The SDF director will respect this, and will simply skip over that actor in a schedule. However, this technique is rather tricky and is not recommended. It can result in unintended sequences of actor execution.

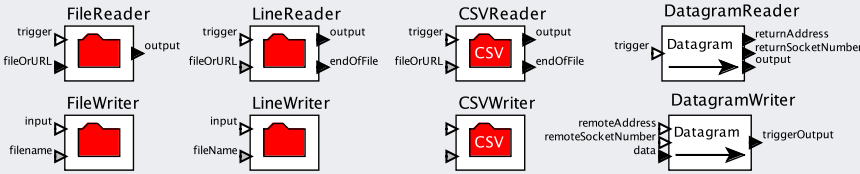
3.3 Summary

Dataflow is a simple and versatile model of computation in which the execution of actors is driven by the availability of input data. It is particularly useful for expressing streaming applications, where long sequences of data values are routed through computations, such as is common in signal processing and multimedia applications.

SDF is a simple (though restrictive) form of dataflow that enables extensive static analysis and efficient execution. DDF is more flexible, but also more challenging to control and more costly to execute, because scheduling decisions are made during run time. The two can be mixed within a single model, so the extra costs of DDF may be incurred only where they are absolutely required by the application. Both SDF and DDF are useful in modal models, as explained in Chapter 8. Using SDF and DDF within modal models provides a versatile concurrent programming model.

Sidebar: IO Actors

The following key input/output actors are in the `IO` library:



FileReader and **FileWriter** read and write files from the local disk or from a remote location specified via a URL or URI. For FileReader, the entire contents of the file is produced on a single output string token. For FileWriter, each input string token is written to a file, overwriting the previous contents of the file. In both cases, a new file name can be given for each firing. To read from standard input, specify System.in as the file name. To write to standard output, specify System.out as the file name. **LineReader** and **LineWriter** are similar, except that they read and write a line at a time.

CSVReader and **CSVWriter** read and write files or URLs that are in CSV format, or comma-separated values (actually, the separator can be anything; it need not be commas). CSV files are converted into record tokens, and record tokens are converted into CSV files. The first line of the file defines the field names of the records. To use CSVReader, you need to help the type system to determine the output type. The simplest way to do this is to enable backward type inference (see Section 14.1.4). This sets the data type of the output port of the CSVReader actor to be the most general type that is acceptable to the actors downstream. Thus, for example, if the actors downstream extract fields from the record, then the type constraints will automatically require those fields to be present and to have compatible types. You can also coerce the output type using the [`Customize→Ports`] context menu command.

The following actors are also in the `IO` library:

ArrowKeySensor produces outputs that respond to the arrow keys on the keyboard. **DirectoryListing** produces on its output an array of file names in a specified directory that match an (optional) pattern.

Exercises

1. The multirate actors described in the box on page 106 and the array actors described in the boxes on page 88 and 86 are useful with SDF to construct **collective operations**, which are operations on arrays of data. This exercise explores the implementation of what is called an **all-to-all scatter/gather** using SDF. Specifically, construct a model that generates four arrays with values:

   ```
   {"a1", "a2", "a3", "a4"}
   {"b1", "b2", "b3", "b4"}
   {"c1", "c2", "c3", "c4"}
   {"d1", "d2", "d3", "d4"}
   ```

   and converts them into arrays with values

   ```
   {"a1", "b1", "c1", "d1"}
   {"a2", "b2", "c2", "d2"}
   {"a3", "b3", "c3", "d3"}
   {"a4", "b4", "c4", "d4"}
   ```

   Experiment with the use of ArrayToElements and ElementsToArray, as well as ArrayToSequence and SequenceToArray (for the latter, you will also likely need Commutator and Distributor). Comment about the relative merits of your approaches. **Hint:** You may have to explicitly set the channel widths of the connections to 1. Double click on the wires and set the value. You may also experiment with MultiInstanceComposite.

2. Consider the model in Figure 3.15, discussed in Example 3.15. Implement this same model using the IterateOverArray actor and only the SDF director instead of the DDF director (see Section 2.7.2).

3. The DDF director in Ptolemy II supports an actor called ActorRecursion that is a recursive reference to a composite actor that contains it. For example, the model shown in Figure 3.16 implements the sieve of Eratosthenes, which finds prime numbers, as described by Kahn and MacQueen (1977).

   Use this actor to implement a composite actor that computes Fibonacci numbers. That is, a firing of your composite actor should implement the firing function

$f : \mathbb{N} \to \mathbb{N}$ defined by, for all $n \in \mathbb{N}$,

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

When ActorRecursion fires, it clones the composite actor above it in the hierarchy (i.e., its container, or its container's container, etc.) whose name matches the value of its *recursionActor* parameter. The instance of ActorRecursion is populated with ports that match those of that container. This actor should be viewed as a highly experimental realization of a particular kind of higher-order actor. It is a higher-order actor because it is parameterized by an actor that contains it. Its implementation, however, is very inefficient. The cloning of the actor it references on each firing is expensive in terms of both memory and time. A better implementation would use an approach similar to the stack frame approach used in procedural programming languages. Instead, the approach it uses is more like copying the source code at run time and then interpreting it. In an attempt to make execution more efficient, this actor avoids creating the clone if it has previously created it. Also, the visual representation of the recursive reference is inadequate. There is no way, looking only at the image in Figure 3.16, to tell what composite actor the ActorRecursion instance references. Thus, you cannot really read the program from its visual representation.

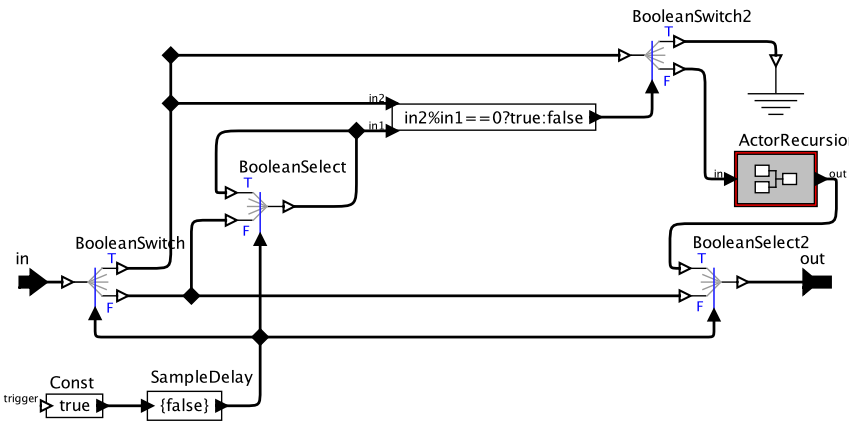


Figure 3.16: The sieve of Eratosthenes, using ActorRecursion in DDF. [online]