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.

5 Synchronous-Reactive Models

Stephen A. Edwards, Edward A. Lee, Stavros Tripakis, Paul Whitaker
In memory of Paul Caspi

Contents

The **synchronous-reactive** (**SR**) model of computation is designed for modeling systems that involve synchrony, a fundamental concept in concurrent systems (see sidebar on page 165). It is an appropriate choice for modeling applications with complicated control logic where many things are happening at once (concurrently) and yet determinism and precise control are important. Such applications include embedded control systems, where safety must be preserved. SR systems are good at orchestrating concurrent actions, managing shared resources, and detecting and adapting to faults in a system. Whereas dataflow models are good for managing streams of data, SR systems are good at managing sporadic data, where events may be present or absent, and where the absence of events has meaning (more than just transport delay). For example, detecting the absence of an event may be an essential part of a **fault management** system. SR is also a good model of computation for coordinating finite state machines, described in Chapters 6 and 8, which can be used to express the control logic of the individual actors that are concurrently executed.

The Ptolemy II SR domain has been influenced by the family of so-called **synchronous languages** (see sidebar on page 166) and in particular **dataflow synchronous languages** such as **Lustre** (Halbwachs et al., 1991) and **Signal** (Benveniste and Le Guernic, 1990). SR primarily realizes the model of **synchronous block diagrams** as described by Edwards and Lee (2003b). The model of computation is closely related to synchronous digital circuits. In fact, this chapter will illustrate some of the ideas using circuit analogies, although the SR domain is intended more for modeling embedded software than circuits.

SR can be viewed as describing **logically timed** systems. In such systems, time proceeds as a sequence of discrete steps, called **reactions** or **ticks**. Although the steps are ordered, there is not necessarily a notion of "time delay" between steps like there is in discrete time systems; and there is no *a priori* notion of real time. Thus, we refer to time in this domain as **logical time** rather than discrete time.

The similarities and differences with dataflow models are:

1. Like homogeneous SDF, an iteration of an SR model consists of one iteration of each actor in the model. Each iteration of the model corresponds to one tick of the logical clock. Indeed, most of the SDF models considered in Chapter 2 could just as easily have been SR models. For example, the behavior of the channel model in Figure 2.29 and all of its variants would behave identically under the SR director.
2. Unlike dataflow and process networks, there is no buffering on the communication between actors. In SR, and output produced by one actor is observed by the destination

actors in the same tick. Unlike rendezvous, which also does not have buffered communication, SR is determinate.

3. Unlike dataflow, an input or output may be **absent** at a tick. In dataflow, the absence of an input means simply that the input hasn't arrived yet. In SR, however, an absent signal has more meaning. Its absence is not a consequence of accidents of scheduling or of the time that computation or communication may take. Instead, the absence of a signal at a tick is *defined* deterministically by the model. As a consequence, in SR, actors may react to the *absence* of a signal. This is quite different from dataflow, where actors react only the *presence* of a signal.
4. As we will explain below, in SR, an actor may be fired multiple times between invocations of postfire. That is, one iteration of an actor may consist of more than invocation of the fire method. For simple models, particularly those without feedback, you will never notice this. Sometimes, however, significant subtleties arise. We focus on such models in this chapter.

5.1 Fixed-Point Semantics

Consider a model with three actors with the structure shown in Figure 5.1(a). Let n denote the tick number. The first tick of the local clock corresponds to $n = 0$, the second to $n = 1$, etc. At each tick, each actor implements an input-output function (which typically changes from tick to tick, possibly in ways that depend on previous inputs). For example, actor 1, in tick 0, implements the function $f_1(0)$. That is, given an input value $s_1(0)$ on port *p1*, it will produce output value $s_2(0) = (f_1(0))(s_1(0))$ on output port *p5*.

At any tick, an input may be absent; in this view, "absent" is treated like any other value. The actor can respond to an absent input, and it may assert an absent output or assign the output some value compatible with the data type of the output port.

Each actor thus produces a sequence of values (or absent values), one at each tick. Actor 1 produces values $s_2(0), s_2(1), \cdots$, while actor 2 produces $s_1(0), s_2(1), \cdots$, and actor 3 produces $s_3(0), s_3(1), \cdots$, where any of these can be absent. The job of the SR director is to find these values (and absences). This is what it means to execute the model.

As illustrated in Figures 5.1(b) through (d), any SR model may be rearranged so that it becomes a single actor with function $f(n)$ at tick n. The domain of this function is a tuple of values (or absences) $s(n) = (s_1(n), s_2(n), s_3(n))$. So is the codomain. Therefore, at

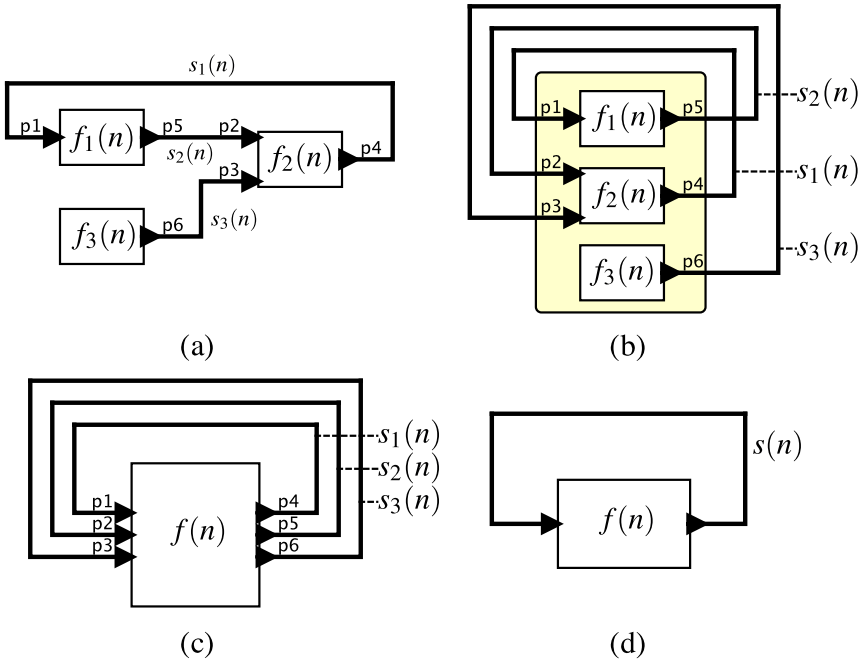


Figure 5.1: An SR model is reducible to a fixed point problem at each tick of he logical clock.

tick n, the job of the director is to find the tuple $s(n)$ such that

$$s(n) = (f(n))(s(n)).$$

At each tick of the logical clock, the SR director finds a fixed point $s(n)$ of the function $f(n)$. The subtleties around SR models concern whether such a fixed point exists and whether it is unique. As we will see, in a well constructed SR model, there will be a unique fixed point that can be found in a finite number of steps.

Logically, as SR model can be conceptualized as a **simultaneous and instantaneous** reaction of all actors at each tick of the clock. The "simultaneous" part of this asserts that the actors are reacting all the same time. The "instantaneous" part means that the outputs of each actor are simultaneous with its inputs. The inputs and outputs are all part of the same fixed point solution. This mental model is called the **synchrony hypothesis**, where one thinks of actors as executing in zero time. But it's a bit more subtle than just that,

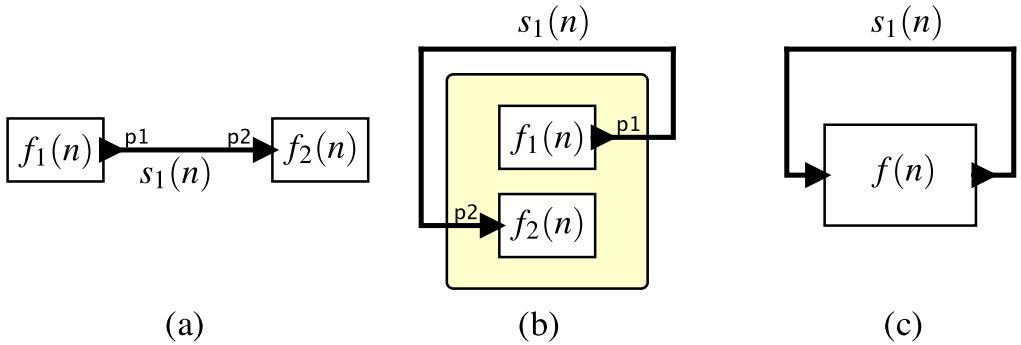


Figure 5.2: Even an SR model without feedback is reducible to a fixed point problem at each tick of he logical clock.

because when there is feedback, an actor may be reacting to an input *that is a function of its own output*. Clearly, one can get trapped in **causality** problems, where the input is not known until the output is known, and the output can't be known until the input is known. Indeed, such causality problems are the major source of subtlety in SR models.

A simple case of SR is a model without feedback, as shown in Figure 5.2. Even such a model is reducible to a fixed-point problem, but in this case it becomes a rather simple problem. The function $f_1(n)$ at tick n only needs to be evaluated once at each tick, and it immediately finds the fixed point. The function $f_2(n)$ never needs to be evaluated (from the perspective of the SR director), but the SR director fires and postfixes actor 2 anyway because of the side effects it may have (e.g. updating a display). But actor 2 plays no role in finding the fixed point.

Once the director has found the fixed point, it can then allow each actor to update its function to $f(n+1)$ in preparation for the next tick. Indeed, this is what an actor does in its postfire phase of execution. An iteration of the model, therefore, consists of some number of firings of the actors, until a fixed point is found, followed by one invocation of postfire, allowing the actor update its state in reaction to the inputs provided by the fixed point that was found. The details of how this execution is carried out are described below in Section 5.3, but first, we consider some examples.

5.2 SR Examples

5.2.1 Acyclic Models

SR models without feedback are much like homogeneous SDF models without feedback, except that signals may be absent. The ability to have absent signals can be convenient for controlling the execution of actors.

Example 5.1: Recall the if-then-else of Figure 3.10, which uses dynamic dataflow to conditionally route tokens to the computations to be done. A similar effect can be achieved in SR using When and Default (see sidebar on page 167), as shown in Figure 5.3. This model operates on a stream produced by the Ramp actor in one of

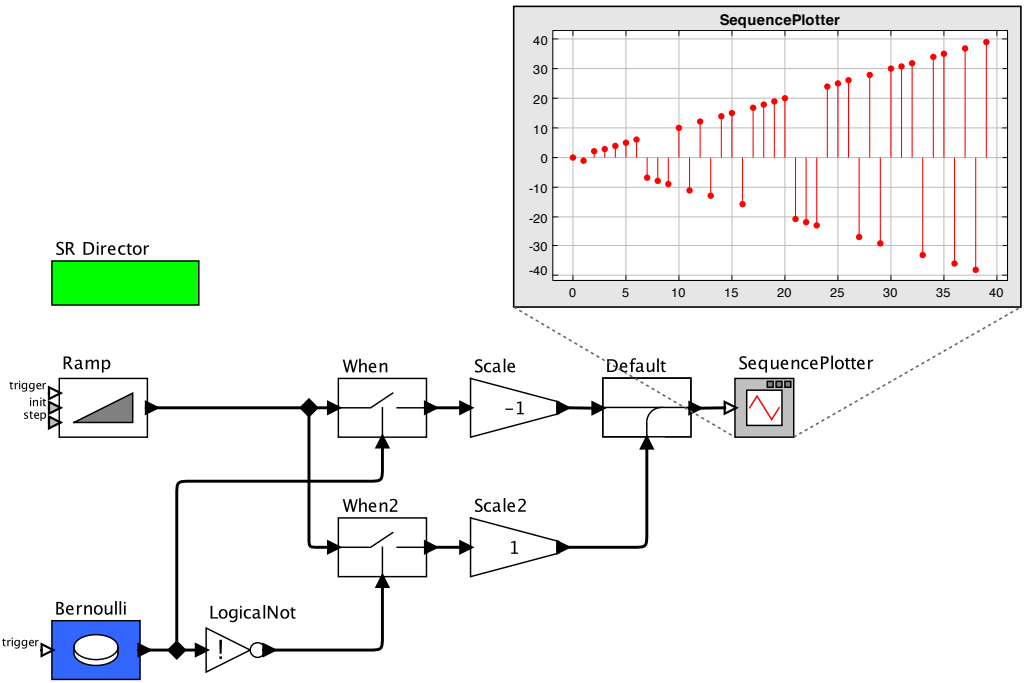


Figure 5.3: A model accomplishing conditional execution using SR. [online]

two (rather trivial) ways. Along the top path, it multiplies the stream by -1. Along the bottom path, it multiplies by 1. Such a pattern might be used, for example, to model intermittent failures in a system.

The Bernoulli actor generates a random boolean that is used to control two instances of When. The top When actor will convey the output from the Ramp to its output when the boolean is true. The bottom When actor will convey the output from the Ramp to its output when the boolean is false. When the output of a When actor is absent, then the downstream Scale actor will also have an absent output. Hence, the Default actor will have only one present input in each tick, and it will convey that input to its output. Finally, the SequencePlotter plots the result.

Whereas with dataflow models, it is possible to make wiring errors that will result in unbounded buffers, as for example in Figure 3.13, in SR, execution is always bounded. Every connection between actors stores at most one token on each tick of the clock. Hence, there is no mechanism for memory usage to become unbounded (unless, of course, an actor does so internally).

5.2.2 Feedback

More interesting SR models involve feedback (directed cycles in the graph), as in Figure 5.1. With such feedback systems, causality becomes a concern. Consider in particular the relationship between actors 1 and 2 in Figure 5.1(a). At a tick n of the logical clock, it would seem that we need to know $s_1(n)$ in order to evaluate function $f_1(n)$. But to know $s_1(n)$, it seems we need to evaluate $f_2(n)$. But to evaluate $f_2(n)$, it seems we need to know $s_2(n)$, which requires evaluating $f_1(n)$. We appear to have gotten stuck in a **causality loop**.

Causality loops must be broken by **non-strict actors**. An actor is said to be **strict** if it requires knowledge of all its inputs in order to provide outputs. If it can provide outputs without full knowledge of the inputs, then it is non-strict. The simplest non-strict actor is the NonStrictDelay (see box on page 167). It can be used to break causality loops, as illustrated in the following example.

Sidebar: About Synchrony

The general definitions of the term **synchronous** are (1) occurring or existing at the same time or (2) moving or operating at the same rate. In engineering and computer science, the term has a number of meanings that are mostly consistent with these definitions, but oddly inconsistent with one another. In referring to concurrent software using threads or processes, synchronous communication refers to a rendezvous style of communication, where the sender of a message must wait for the receiver to be ready to receive, and the receiver must wait for the sender. Conceptually, the communication occurs at the same time from the perspective of each of the two threads, consistent with definition (1). In Java, however, the keyword `synchronized` defines blocks of code that are *not* permitted to execute simultaneously, which is inconsistent with both definitions.

There is yet a third meaning of the word synchronous, which is the definition we use in this chapter. This third meaning underlies synchronous languages (see box on page 166). Two key ideas govern these languages. First, the outputs of components in a program are (conceptually) simultaneous with their inputs (this is called the synchrony hypothesis). Second, components in a program execute (conceptually) simultaneously and instantaneously. Even though this cannot occur in reality, a correct execution must behave as though it did. This interpretation is consistent with *both* definitions (1) and (2) above, since executions of components occur at the same time and operate at the same rate.

In circuit design, the word synchronous refers to a style where a clock signal that is distributed throughout a circuit causes circuit components called "latches" to record their inputs on the rising or falling edges of the clock. The time between clock edges needs to be sufficient for circuit gates between latches to settle. Conceptually, this model is very similar to the model in synchronous languages. Assuming that the gates between latches have zero delay is equivalent to the synchrony hypothesis, and global clock distribution gives simultaneous and instantaneous execution of those gates. Hence, the SR domain is often useful for modeling digital circuits.

In power systems engineering, synchronous means that electrical waveforms have the same frequency and phase. In signal processing, synchronous means that signals have the same sample rate, or that their sample rates are fixed multiples of one another. The term synchronous dataflow, described in Chapter 3.1, is based on this latter meaning of the word synchronous. This usage is consistent with definition (2).

Sidebar: Synchronous-Reactive Languages

The synchronous-reactive model of computation dates back to at least the mid-1980s, when a number of programming languages were developed. The term "reactive" comes from a distinction in computational systems between **transformational systems**, which accept input data, perform a computation, and produce output data, and **reactive systems**, which engage in an ongoing dialog with their environment (Harel and Pnueli, 1985). Manna and Pnueli (1992) state

> "The role of a reactive program ... is not to produce a final result but to maintain some ongoing interaction with its environment."

The distinctions between transformational and reactive systems led to the development of a number of innovative programming languages. The **synchronous languages** (Benveniste and Berry, 1991) take a particular approach to the design of reactive systems, in which pieces of the program react simultaneously and instantaneously at each tick of a global clock. Primary among these languages are Lustre (Halbwachs et al., 1991), Esterel (Berry and Gonthier, 1992), and Signal (Le Guernic et al., 1991). Statecharts (Harel, 1987) and its implementation in Statemate (Harel et al., 1990) also have a strongly synchronous flavor.

SCADE (Berry, 2003) (Safety Critical Application Development Environment), a commercial product of Esterel Technologies, builds on Lustre, borrows concepts from Esterel, and provides a graphical syntax in which state machines similar to those in Chapter 6 are drawn and actor models are composed synchronously. One of the main attractions of synchronous languages is their strong formal properties that facilitate formal analysis and verification techniques. For this reason, SCADE models are used in the design of safety-critical flight control software systems for commercial aircraft made by Airbus.

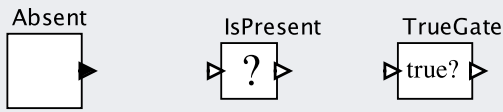
In Ptolemy II, SR is a form of **coordination language** rather than a programming language, (see also ForSyDe (Sander and Jantsch, 2004), which also uses synchrony in a coordination language). This allows for "primitives" in a system to be complex components rather than built-in language primitives. This, in turn, enables heterogeneous combinations of MoCs, since the complex components may themselves include components developed under another model of computation.

Sidebar: Domain-Specific SR Actors

The SR actors in `DomainSpecific→SynchronousReactive` below are inspired by the corresponding operators of the synchronous languages Lustre and Signal.

- **Current** outputs the most recently received non-absent input. If no input has been received, then the output is absent.
- **Default** merges two signals with a priority. If the preferred input (on the left) is present, then the output is equal to that input. If the preferred input is absent, then the output is equal to the alternate input (on the bottom, whether it is absent or not).
- **NonStrictDelay** provides a one-tick delay. On each firing, it produces on the output port whatever value it read on the input port in the previous tick. If the input was absent on the previous tick of the clock, then the output will be absent. On the first tick, the value may be given by the *initialValue* parameter. If no value is given, the first output will be absent.
- **Pre** outputs the previously received (non-absent) input. When the input is absent, the output is absent. The first time the input is present, the output is given by the *initialValue* parameter of the actor (which by default is absent). It is worth noting that, contrary to NonStrictDelay, Pre is strict, meaning that the input must be known before the output can be determined. Thus, it will not break a causality loop. To break a causality loop, use NonStrictDelay.
- **When** filters a signal based on another. If the control input (on the bottom) is present and true, then the data input (on the left) is copied to the output. If control is absent, false, or true with the data input being absent, then the output is absent.

The Ptolemy II library also offers several actors to manipulate absent values:



- **Absent**. Output is always absent.
- **IsPresent** outputs true if its input is present and false otherwise.
- **TrueGate** outputs true if its input is present and true; otherwise, absent.

Example 5.2: A simple model of a digital circuit is shown in Figure 5.4. It is a model of a 2-bit, modulo-4 counter that produces the integer sequence $0, 1, 2, 3, 0, 1, \ldots$. The feedback loops use NonStrictDelay actors, each of which models a latch (a latch is a circuit element that captures a value and holds it for some period of time). It also includes two actors that model logic gates, the LogicalNot and LogicGate (see box on page 112).

The upper loop, containing the LogicalNot, models the low-order bit (**LOB**) of the counter. It starts with value false, the initial output of the NonStrictDelay, and the alternates between true and false in each subsequent tick.

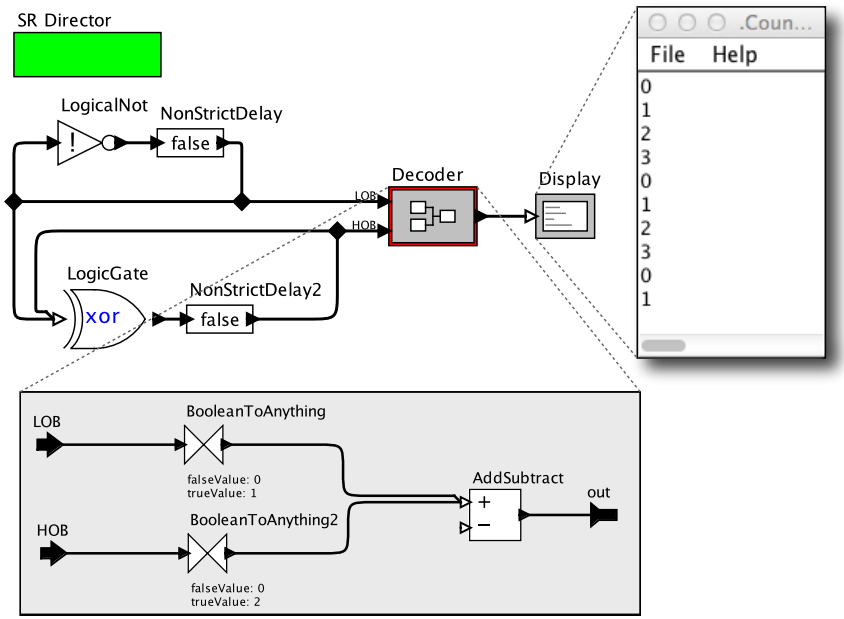


Figure 5.4: A model of a 2-bit counter in SR. The top-level model includes a Decoder composite actor that translates the boolean data into integers. [online]

The lower loop, containing the LogicGate, models a carry circuit, implementing the high-order bit (**HOB**) of the counter. It also starts with false, and toggles between true and false in each tick where the the LOB is true.

The Decoder is a composite actor provided just to generate a more readable display. It converts the two Boolean values into a numerical value from 0 to 3 by assigning values to the LOB It contains two **BooleanToAnything** actors that convert the Boolean values to the values of the LOB and HOB, which are then added together.

The NonStrictDelay actors in Figure 5.4 are non-strict. They are able to produce outputs without knowing the inputs. On the first tick, the values of the outputs are given by the *initialValue* parameters of the actors. In subsequent ticks, the values of the outputs are given by the input *from the previous tick*, which has been found by identifying the fixed point. Thus, these actors break the potential causality loops.

Notice that it would not work to use Pre instead of NonStrictDelay (see box on page 5.2.1). The Pre actor is strict, because it has to know whether the input is present or not in order to determine whether the output is present or not.*

The model of Figure 5.4 is rather simple and does not illustrate the full power of SR. In fact, the same model would work with an SDF director, provided that NonStrictDelay actors are replaced by SampleDelay actors.†

Every directed cycle in SR is required to contain at least one non-strict actor. But NonStrictDelay is not the only non-strict actor. Another example of a non-strict actor is the **NonStrictLogicGate** actor, which can be parameterized to implement functions such as non-strict logical AND, also called a **parallel AND**. The truth table of the non-strict AND with two inputs is shown below (the actor can in fact accept an arbitrary number of in-

*The Lustre synchronous language (Halbwachs et al., 1991) is able to make Pre non-strict by using a **clock calculus**, which analyzes the model to determine in which ticks the inputs will be present. Thus, in Lustre, Pre *does not execute* in ticks where its input is absent. As a consequence, when it does execute, it knows that the input is present, and even though it does not know the value of the input, it is able to produce an output. The SR director in Ptolemy II does not implement a clock calculus, adopting instead the simpler clocking scheme of Esterel (Berry and Gonthier, 1992).

†SampleDelay produces initial outputs during the initialize phase of execution. In dataflow domains, those initial outputs are buffered and made available during the execution phase. In SR, however, there is no buffering of data, and any outputs produced during initialize are lost. Hence, SampleDelay is not useful in SR.

puts):

inputs	$\perp$	$true$	$false$
$\perp$	$\perp$	$\perp$	$false$
$true$	$\perp$	$true$	$false$
$false$	$false$	$false$	$false$

Here, the symbol $\perp$ means **unknown**. Observe that when one input is known to be false, the output is false, even if the other input is unknown.

Example 5.3: The model shown in Figure 5.5 results in non-ambiguous semantics despite its feedback loop. The NonStrictLogicGate implements the AND logic function, and outputs a Boolean "false" value at every tick because one of the inputs is always false.

A practical example that also has cycles without NonStrictDelay is next.

Example 5.4: Figure 5.6 shows an SR realization of **token-ring media access control** (**MAC**) protocol given by Edwards and Lee (2003b). The top-level model has three instances of an **Arbiter** class connected in a cycle. It also has a ComposeDisplay composite actor used to construct a human-readable display of the results of execution, shown at the bottom.

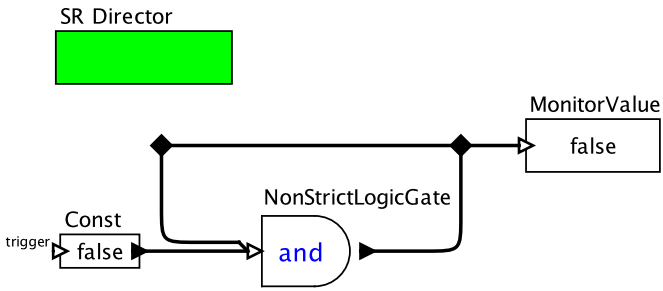


Figure 5.5: A non-ambiguous model which uses a non-strict logical AND. [online]

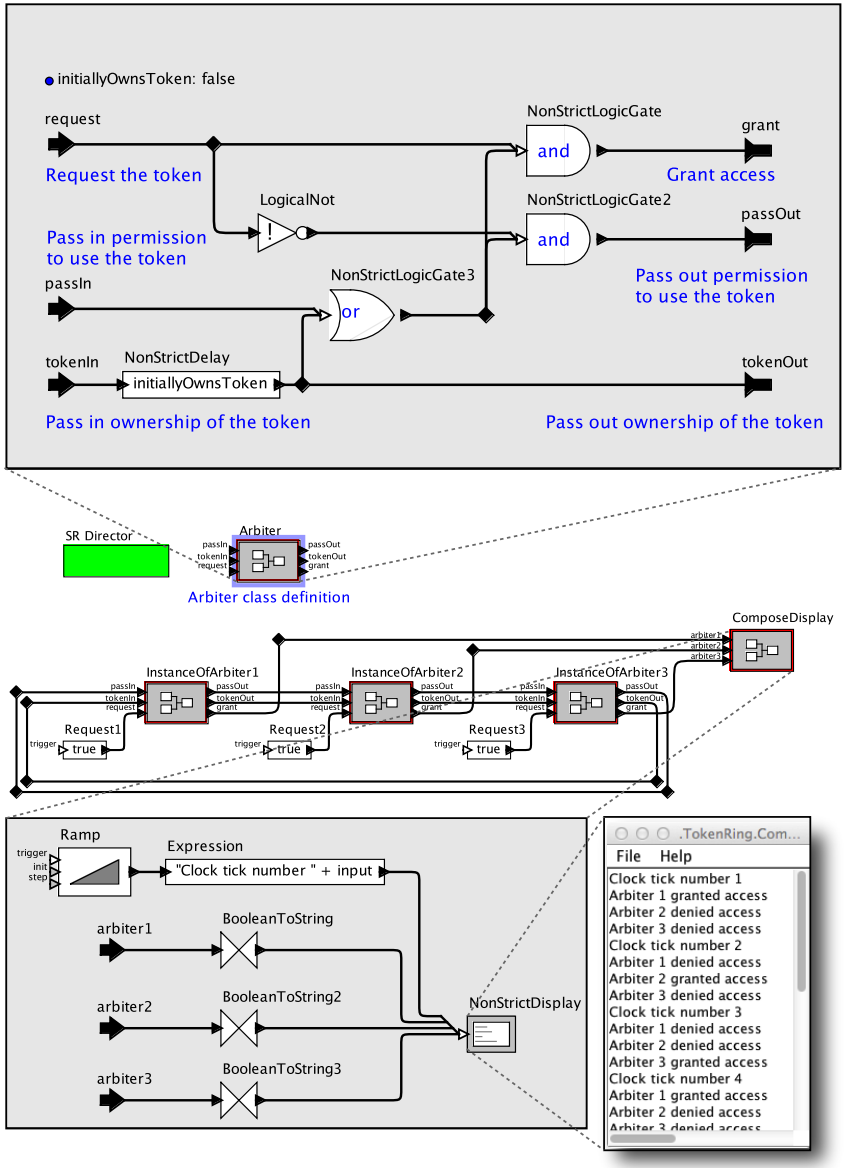


Figure 5.6: A token-ring media access protocol implemented using SR. From Edwards and Lee (2003b). [online]

The goal of this system is to arbitrate fairly among requests for exclusive access to a shared resource by marching a token around a ring. At each tick of the logical clock, the arbiter grants access to requestor holding the token, if it requests access. If it does not request access, then the model grants access to the first requestor downstream of the block with the token that requests access. In the figure, all three requestors are always requesting access, and in the display at the bottom, you can see that access is granted fairly in a round-robin fashion. In this model, InstanceOfArbiter1 starts with the token (see the parameter of the instance).

The three arbiters are instances of the actor-oriented class shown at the top of the figure. This class has three inputs and three outputs. It has an instance of NonStrictDelay that outputs true for the arbiter that currently holds the token. Exactly one of the three is initialized with value true. At each tick of the clock, the arbiter passes the token down to the next arbiter. This forms a cycle that include three instances of NonStrictDelay.

However, there are another cycles that have no instances of NonStrictDelay, for example the cycle passing through each *request* input and *grant* output. This cycle has three instances of NonStrictLogicGate, configured to implement the parallel AND. This logic gate will grant access to the requestor if it has a request and it either holds the token or its *passIn* input is true (meaning that the upstream arbiter has the token but does not have a request). Although it is far from trivial to see at glance, every cycle of logical gates can be resolved without full knowledge of the inputs, so the model does not suffer from a causality loop.

Another example of a non-strict actor is the Multiplexor or BooleanMultiplexor (see box on page 119). These require their control input (at the bottom of the icon) to be known; the value of this input then determines which of the data inputs are to be forwarded to the output. Only that one data input needs to be known for the actor to able to produce an output.

Example 5.5: An interesting example, shown in Figure 5.7, calculates either $\sin(\exp(x))$ or $\exp(\sin(x))$, depending on a coin toss from the Bernoulli actor. Malik (1994) called examples like this **cyclic combinational circuit**s, because, although there is feedback, there is actually no state stored in the system. The output (each value plotted) depends only on the current inputs (the data from the Ramp

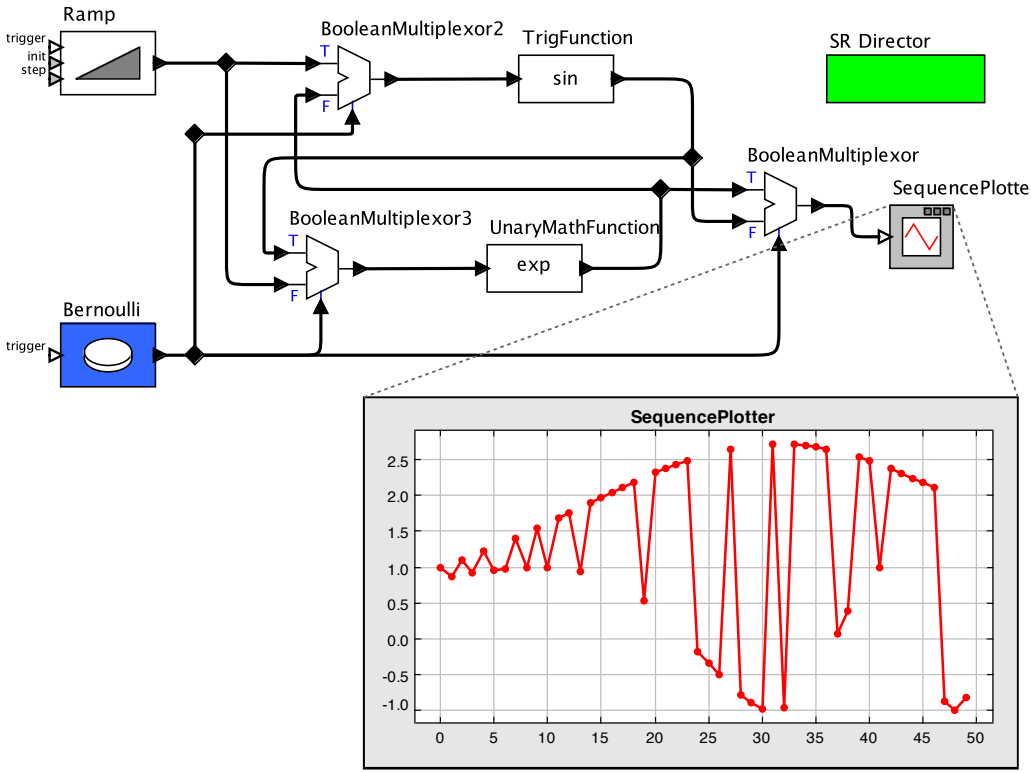


Figure 5.7: A model of a cyclic combinational circuit in SR. From Malik (1994). [online]

and Bernoulli actors). A circuit whose output depends only on the current inputs and not on the past history of inputs is called a **combinational** circuit. Most circuits with feedback are not combinational. The output depends not only on the current inputs, but also on the current state, and the current state changes over time.

In this case, feedback is being used to avoid having to have two copies of the actors that do the actual computation, the TrigFunction and UnaryMathFunction (see box on page 58). An equivalent model that uses two copies of these actors is shown in Figure 5.8. If these models are literally implemented in circuits, with a separate cir-

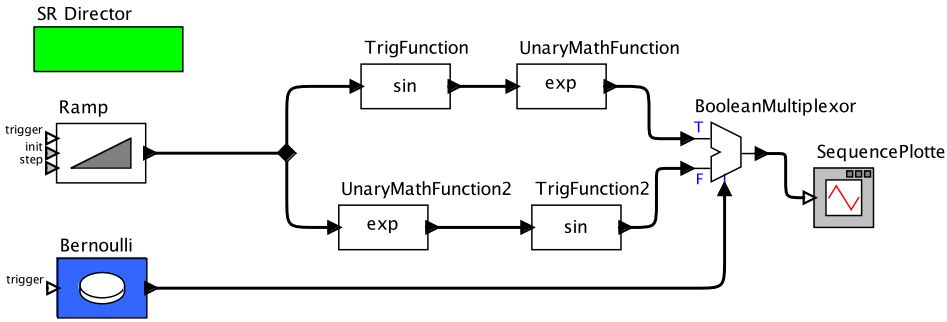


Figure 5.8: Acyclic version of the model of Figure 5.7 that uses two copies of each of the math actors. [online]

cuit for each actor, then the model in Figure 5.7 may be considerably less expensive than the one in Figure 5.8.

The model has three BooleanMultiplexor actors. These actors send either their "T" or "F" input value to the output port depending depending on whether the control input (at the bottom of the actor) is set to true or false. At each tick, one of the two BooleanMultiplexor actors on the left will be able to provide an output (once it is provided with an input from the Ramp). That one BooleanMultiplexor, therefore, breaks the causality loop and enables finding a fixed point.

5.2.3 Causality Loops

Not all SR models are executable. In particular, it is possible to construct feedback models that exhibit a causality loop, as illustrated by the following examples.

Example 5.6: Two examples of loops with unresolvable cyclic dependencies are shown in Figure 5.9. Both the Scale and the LogicalNot actors are strict, and hence their inputs must be know for the outputs to be determined. But the outputs are equal to the inputs in these models, so the inputs cannot be known. The SR director will reject these models, reporting an exception

```
IllegalActionException: Unknown inputs remain. Possible
causality loop:
  in Display.input
```

5.2.4 Multiclock Models

The logical clock in the SR domain is a single, global clock. Every actor under the control of an SR director will be fired on every tick of this clock. But what we want some actors to be fired more or less frequently? Fortunately, the hierarchy mechanism in Ptolemy II makes it relatively easy to construct models with multiple clocks proceeding at different rates. The **EnabledComposite** actor is particularly useful for building such **multiclock** models.

Example 5.7: Consider the **guarded count** model of Figure 5.10, which counts down to zero from some initial value and then restarts the count from some new value. At the top level, the model has two composite actors and two Display actors. The CountDown composite actor uses SR primitive actors to implement the following count-down behavior: whenever it receives a non-absent value n (an integer) at its *start* input port, it (re)starts a count-down from n; that is, it outputs the sequence of values $n, n-1, ..., 0$ at its *count* output port. When the count reaches 0, the *ready* port outputs a value true, signaling that the actor is ready for a new count down.

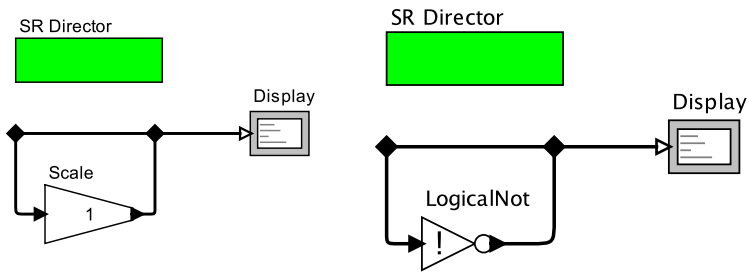


Figure 5.9: Two SR models with invalid loops.

The ready signal controls the firing of the EnabledComposite actor. Within this composite, a reaction only occurs when a true value is provided on the *enable* input port (the port at the bottom of the actor). Note that the ready signal is initially true, due to the NonStrictDelay actor used inside CountDown.

The clock of the SR director inside EnabledComposite progresses at a slower rate than the clock of the top-level SR director. In fact, the relationship between these rates is determined dynamically by the data provided by the Sequence actor.

5.3 Finding the Fixed-Point

For acyclic models (such as the one shown Figure 5.8) or cyclic models where every cycle is "broken" by a NonStrictDelay actor (such as the model shown in Figure 5.4), executing the model efficiently is easy. The actors of the model can be ordered according to their dependencies (e.g., using a topological sorting algorithm) and then fired according to that order. In this case, each actor only needs to be fired once at each tick of the logical clock.

However, this strategy will not work with models like those in Figures 5.6 or 5.7, because the order in which the actors have to be fired depends on data computed by some of the actors. Fortunately, there is a simple execution strategy that works. The key is to start each tick of the logical clock by assigning a special value called unknown, denoted $\perp$, to all signals. The director can then simply evaluate actors in arbitrary order until no more progress is made. For strict actors, if there are any unknown inputs, then the outputs will remain unknown. For non-strict actors, even when some inputs are unknown, some outputs may become known. This procedure is said to have converged when no firing of any actor changes the state of any signal. If the actors all follow the strict actor semantics (see box on page 433), then it can be proven that this procedure converges in a finite number of steps (see, for example, Edwards and Lee (2003b)).

Upon convergence, either all signals will be known, or some signals will remain unknown. If every iteration results in all signals being known for all possible inputs, then the model is said to be **constructive** (that is, a solution can be "constructed" in a finite number of steps). Otherwise, the model is declared to be **non-constructive**, and it is rejected.

Note that in the Ptolemy II SR domain, the causality analysis is performed dynamically, at run-time. This is in contrast to languages such as Esterel, where the compiler attempts

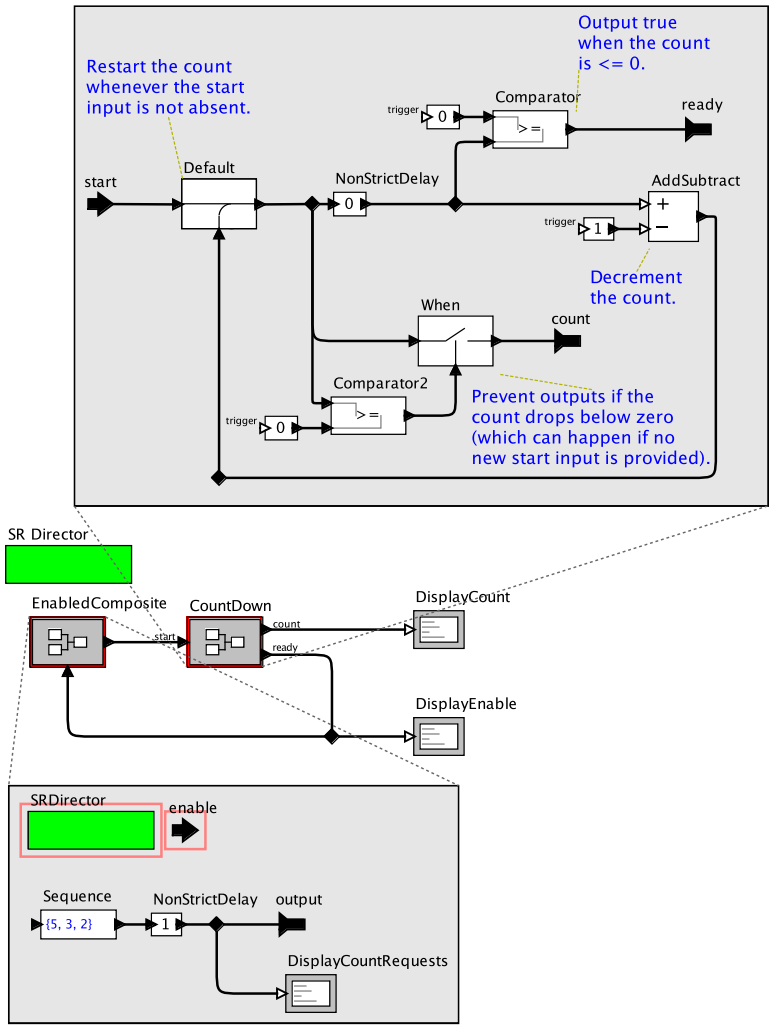


Figure 5.10: Multiclock model in SR. [online]

to prove statically (i.e., at compile-time) that the program is constructive (see the sidebar on page 179).

SR can only work correctly with actors that follow the strict actor semantics. To understand this, we can model an actor as a state machine. Let $\vec{x}$, $\vec{y}$ and $\vec{s}$ denote the vectors of inputs, outputs, and states, respectively. Then the behavior of the state machine can be described as

$$\vec{y}(n) = f(\vec{x}(n), \vec{s}(n)) \tag{5.1}$$

$$\vec{s}(n+1) = g(\vec{x}(n), \vec{s}(n)), \tag{5.2}$$

where n indexes the ticks of the logical clock, f models the fire method that computes outputs from current inputs and current state, and g models the postfire method that computes the next state from current inputs and current state. The key here is that the fire method does not change the state of the actor. Hence, the fire method can be invoked repeatedly, and each time, given the same inputs, it will produce the same outputs.

An additional condition on actors is that they be monotonic (see box on page 182). Although the mathematical underpinnings of this constraint are quite sophisticated, the practical manifestation of the constraint is simple. An actor is monotonic if it does not change its mind about outputs given more information about inputs. Specifically, if the fire method is invoked with some inputs unknown, then if the actor is non-strict, it may be able to produce outputs. Suppose that it does. Then the actor is monotonic if given more information about the inputs (fewer inputs are unknown) does not cause it to produce a *different* output than the one it produced with less information.

Most Ptolemy II actors conform to the strict actor semantics and are monotonic and therefore can be used in SR.

5.4 The Logic of Fixed Points

Recall the two models of Figure 5.9, both of which exhibit causality loops. These models, however, are different from one another in an interesting way. They exhibit the difference between a deterministic and a constructive semantics of synchronous models. The constructive semantics is based on ideas from **intuitionistic logic**, and although it is also deterministic, it rejects some models that would be accepted by a broader deterministic semantics based on classic logic.

Sidebar: Causality in Synchronous Languages

The problem of how to resolve cyclic dependencies, the **causality problem**, is one of the major challenges in synchronous languages. We briefly summarize several solutions here, and refer the reader to research literature and survey articles such as Caspi et al. (2007) for more details.

The most straightforward solution to the causality problem is to forbid cyclic dependencies altogether. This is the solution adopted by the Lustre language, which requires that every dataflow loop must contain at least one **pre** operator. The same effect could be achieved by the Ptolemy II SR director by requiring that every loop contain at least one NonStrictDelay actor. This actor breaks the instantaneous cyclic dependency. The Lustre compiler statically checks this condition and rejects those programs that violate it. The same policy is followed in SCADE.

Another approach is to accept a broader set of constructive programs, as is the case with the Ptolemy II SR domain. This approach was pioneered by Berry (1999) for Esterel. A key difference between Esterel and SR is that in SR a fixed-point is computed at run-time (at each tick of the logical clock), while the Esterel compiler attempts to prove that a program is constructive at compile-time. The latter is generally more difficult since the inputs to the program are generally unknown at compile-time. On the other hand, statically proving that a program is constructive has two key benefits. First, it is essential for safety-critical systems, where run-time exceptions are to be avoided. Second, it allows generation of implementations that minimize the run-time overhead of fixed-point iteration.

Yet another approach is to accept only deterministic programs, or conversely, reject programs that, when interpreted as a set of constraints, do not yield unique solutions. This approach is followed in Signal (Benveniste and Le Guernic, 1990) and Argos (Maraninchi and Rémond, 2001). One drawback with this approach is that it sometimes accepts dubious programs. For instance, consider a program representing the system of equations

$$Y = X \wedge \neg Y.$$

Although this system admits a unique solution in classic two-valued logic, namely, $X = Y =$ false, it is unclear whether the corresponding implementation is meaningful. In fact, a straightforward combinational circuit implementation is unstable; it oscillates.

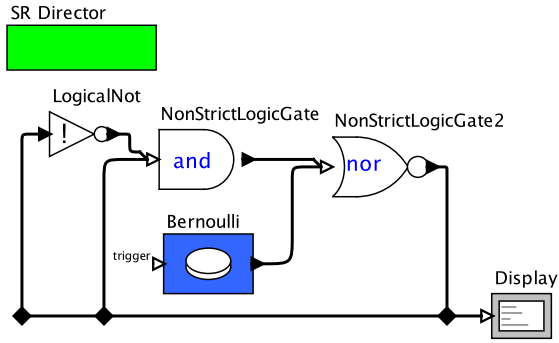


Figure 5.11: Non-constructive example with a unique fixed point. [online]

In particular, we could have interpreted the left model of Figure 5.9 as defining an equation between the input and output of the Scale actor, say x, as follows:

$$x = 1 \cdot x$$

In the classic logic interpretation, the above equation has multiple solutions, e.g., $x = 0$, $x = 1$, and so on. A non-deterministic semantics based on classic logic would accept any of these solutions as a valid behavior of the system. A deterministic semantics would declare the model ambiguous, and thus invalid. In the SR semantics, the above equation has a unique least fixed-point solution, namely, $x = \bot$, unknown. Hence, SR also rejects this model.

The right model of Figure 5.9 can be seen as defining the equation

$$x = \neg x$$

where $\neg$ denotes logical negation. In this case, in the classic logical interpretation, there is no solution at all, quite a different situation. A deterministic semantics may again reject this model. In the case of SR, the solution is again $x = \bot$, unknown, resulting in rejection of the model.

A third situation, due to Malik (1994) and shown in Figure 5.11, however, could be accepted by a deterministic semantics, but is rejected by a constructive semantics. Logically, the output of the AND gate should always be false, and hence the output sent to the Display actor should be equal to the negation of the input value produced by the Bernoulli

actor. Hence, there is a single unique behavior for all possible inputs. The model, however, is rejected by the Ptolemy II SR director as non-constructive whenever the Bernoulli actor produces a false. In that case, all signals in the loop remain unknown. In the constructive SR semantics, this solution with unknowns is the *least* fixed point, and hence is the behavior selected, even though there is a unique fixed point with no unknowns.

Even though the circuit in Figure 5.11 seems to have a logically consistent behavior for every input, there are good reasons for rejecting it. If this were actually implemented as a circuit, then time delays in the logic gates would cause the circuit to oscillate. It would not, in fact, realize the logic specified by the model. To realize such circuits in software, the only known technique for finding the unique fixed point and verifying that it is unique, in general, is to exhaustively search over all possible signal assignments. In a small model like this, such an exhaustive search is possible, but it becomes intractable for larger models, and it becomes impossible if the data types have an infinite number of possible values. Thus, the fact that the model is non-constructive reveals very real practical problems with the model.

We now give a brief introduction to the theoretical foundation of the SR semantics. This is a rather deep topic, and our coverage here is meant only to whet the appetite of the reader to learn more. The SR semantics is based on the theory of continuous functions over complete partially ordered sets (CPOs) (see box on page 182). In the case of SR, the key CPO is a so-called flat CPO shown in Figure 5.12. This CPO consists of the minimal element $\perp$ and all "legal" values of Ptolemy models, such as booleans, integer and real numbers, but also tuples, records, lists, and so on (see Chapter 14). Any legal value is considered to be greater than $\perp$ in the CPO order, but the legal values are incomparable among themselves, leading to the term "flat".

Now, consider an SR model. The output of every actor in the model can be seen as a variable taking values in the above flat CPO. The vector of all output variables can be seen as taking values in the product CPO obtained by forming the cartesian product of all individual CPOs, with element-wise ordering. For simplicity, let us suppose that

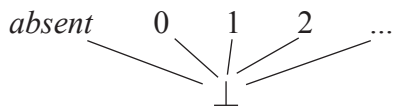


Figure 5.12: The flat CPO ensuring existence of a unique least fixed-point in SR.

Sidebar: CPOs, Continuous Functions and Fixed Points

The SR semantics is based on order theory, which we summarize here; see Davey and Priestly (2002) for a more thorough explanation

Consider a set S. A **binary relation** on S is a subset $\sim \subseteq S \times S$. We often write $x \sim y$ instead of $(x, y) \in \sim$. A **partial order** on S is a binary relation $\sqsubseteq$ which is **reflexive** (i.e., $\forall x \in S : x \sqsubseteq x$), **antisymmetric** (i.e., $\forall x, y \in S : x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$), and **transitive** (i.e., $\forall x, y, z \in S : x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$). A **partially ordered set** or **poset** is a set equipped with a partial order.

Let $X \subseteq S$. An **upper bound** of X is an element $u \in S$ such that $\forall x \in X : x \sqsubseteq u$. A **least upper bound** of X, denoted $\sqcup X$, is an element $\ell \in S$ such that $\ell \sqsubseteq u$ for all upper bounds u of X. A **chain** of S is a subset $C \subseteq S$ which is **totally ordered**: $\forall x, y \in C : x \sqsubseteq y$ or $y \sqsubseteq x$. A **complete partial order** or CPO is a poset S such that every chain of S has a least upper bound in S. This condition also guarantees that every CPO S has a **bottom element** $\bot$, such that $\forall x \in S : \bot \sqsubseteq x$. (Indeed, the empty chain must have a least upper bound in S, and the set of upper bounds of the empty subset of S is the entire S.)

To illustrate the above concepts, consider the set of natural numbers $\mathbb{N} = \{0, 1, 2, ...\}$. $\mathbb{N}$ is a poset with the usual (total, and therefore also partial) order $\leq$. Because $\leq$ is a total order, $\mathbb{N}$ is a chain. The least upper bound of $\mathbb{N}$ can be defined to be a new number ω such that $n < \omega$ for all $n \in \mathbb{N}$. ω is not a natural number, therefore, $\mathbb{N}$ is not a CPO. On the other hand, the set $\mathbb{N}^\omega = \mathbb{N} \cup \{\omega\}$ is a CPO. The bottom element of $\mathbb{N}^\omega$ is 0.

Every poset whose chains are all finite is a CPO. This is because the greatest element in a chain is also the least upper bound of the chain. This is why the "flat" poset of Figure 5.12 is a CPO.

Consider two CPOs X and Y. A function $f : X \to Y$ is **Scott-continuous** or simply **continuous** if for all chains $C \subseteq X$, $f(\sqcup C) = \sqcup\{f(c) \mid c \in C\}$. It can be shown that every continuous function is also **monotonic**, i.e. it satisfies: $\forall x, y \in X : x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. However, not all monotonic functions are continuous. For example, consider the function $f : \mathbb{N}^\omega \to \mathbb{N}^\omega$ such that $f(n) = 0$ for all $n \in \mathbb{N}$ and $f(\omega) = \omega$. Then $f(\sqcup\mathbb{N}) = f(\omega) = \omega$, whereas $\sqcup\{f(n) \mid n \in \mathbb{N}\} = \sqcup\{0\} = 0$. The following **fixed-point theorems** are well-known results of order theory: (A) Every monotonic function $f : X \to X$ on a CPO X has a least fixed-point x^*. (B) If f is also continuous then $x^* = \bigsqcup_{i \geq 0} f^i(\bot)$, where $f^0(\bot) = \bot$ and $f^{i+1}(\bot) = f(f^i(\bot))$. (B) is used to obtain an effective procedure for computing the semantics of an SR model.

the model is *closed*, in the sense that every input port of every actor in the model is connected to some output port (the theory also works for open models, but is slightly more complicated; we refer the reader to Edwards and Lee (2003b) for a more detailed explanation). The SR model then defines a function F which has both as domain and co-domain this product CPO: this is because the model is closed, so every input is also an output. Thus, F takes as input a vector $\vec{x}$ and returns as output another vector $\vec{y}$. The latter is obtained by firing all actors in the model once. Given this interpretation, a closed SR model defines the equation

$$\vec{x} = F(\vec{x})$$

This equation has a unique least solution $\vec{x}^*$, provided that F is monotonic; that is, provided that $\vec{x} \leq \vec{y}$ implies $F(\vec{x}) \leq F(\vec{y})$. (The precise condition is for the function to be continuous, but in the case of flat CPOs, monotonicity is equivalent to continuity.) The solution $\vec{x}^*$ is called a fixed-point because it satisfies $\vec{x}^* = F(\vec{x}^*)$. It is 'least' in the sense that it is smaller in the CPO ordering than every other solution of the above equation. That is, for any $\vec{y}$ such that $\vec{y} = F(\vec{y})$, it must be $\vec{x}^* \leq \vec{y}$.

Moreover, the least fixed-point can be computed effectively in a finite number of iterations, in fact, at most N iterations, where N is the total number of outputs in the model. Indeed, starting with all outputs set to $\perp$, every iteration that fires all actors without reaching the fixed-point is guaranteed to update at least one output. The first time an output is updated, it changes from $\perp$ to some legal value v. Because F is monotonic, the same output can no longer change from v to $\perp$ or any other v', since $v > \perp$ and v is incomparable with any $v' \neq v$. Therefore, each output can be updated at most once. As a result, the fixed-point must be reached after at most n iterations.

The monotonicity of F is ensured by ensuring that every individual actor is monotonic; that is, that its fire method is monotonic. Monotonicity of F then follows from the fact that composition of monotonic functions results in a monotonic function. Monotonicity of atomic actors is ensured in Ptolemy by construction. The key is to ensure that if an actor outputs a known value, say v, in the presence of unknown inputs, then if those inputs become known, the actor will not "change its mind" and output a different value v'. A straightforward way to ensure this property is by making an actor strict, in the sense that it requires all inputs to be known, otherwise, it produces unknown outputs. Most actors in

Ptolemy are strict, but a few key ones that we have discussed are non-strict. Every cycle in an SR model requires some non-strict actors.

5.5 Summary

This chapter has introduced the SR domain in Ptolemy II. In SR, execution is governed by a logical clock, and at each tick of the clock, actors execute, conceptually, simultaneously and instantaneously. We have explained how this results in a fixed-point semantics, and have given examples of both cyclic and acyclic models. We have shown that SR admits multiple clock domains, where clocks progress at different rates. Finally, we have given a brief introduction to the (rather deep) mathematical foundations behind the semantics of SR models.

Exercises

1. This exercise studies the use of absent events in SR.

   (a) As a warmup, use Sequence and When to construct an SR model that generates a sequence of values *true* interspersed with *absent*. For example, produce the sequence

   $$(true, absent, absent, true, absent, true, true, true, absent) .$$

   Make sure your model adequately displays the output. In particular, *absent* should be visible in the display.[‡]

   (b) Use Default and When to create a composite actor **IsAbsent** that given any input sequence, produces an output *true* at every tick when the input is *absent*, and otherwise produces the output *absent*.[§]

   (c) Create a composite actor that can recognize the difference between single and double mouse clicks. Your actor should have an input port named *click*, and two output ports, *singleClick* and *doubleClick*. When a `true` input at *click* is followed by N *absent*s, your actor should produce output `true` on *singleClick*, where N is a parameter of your actor. If instead a second `true` input occurs within N ticks of the first, then your actor should output a `true` on *doubleClick*.

   How does your model behave if given three values *true* within N ticks on input port *click*?

   (d) **Extra credit**: Redo (a)-(c) by writing a custom a Java actor for each of the three functions above. How does this design compare with the design implemented using primitive SR actors? Is it more or less understandable? Complex?

2. The token-ring model of Figure 5.6 is constructive under the assumption that exactly one of the instances of the Arbiter initially owns the token (it has its *initiallyOwnsToken* parameter set to true). If no instance of Arbiter initially owns the token, then is the model still constructive? If so, explain why. If not, given a set of values of the Request actors that exhibits a causality loop.

[‡]Note that you could use TrueGate to implement this more simply, but part of the goal of this exercise is to fully understand When.

[§]Again, a simpler implementation is available using IsPresent, but the goal of this exercise is to fully under Default.