Concurrent Models of Computation

Edward A. Lee

Robert S. Pepper Distinguished Professor, UC Berkeley

EECS 290n – Advanced Topics in Systems Theory

Concurrent Models of Computation

Spring 2009

Week 6: Synchronous/Reactive Models

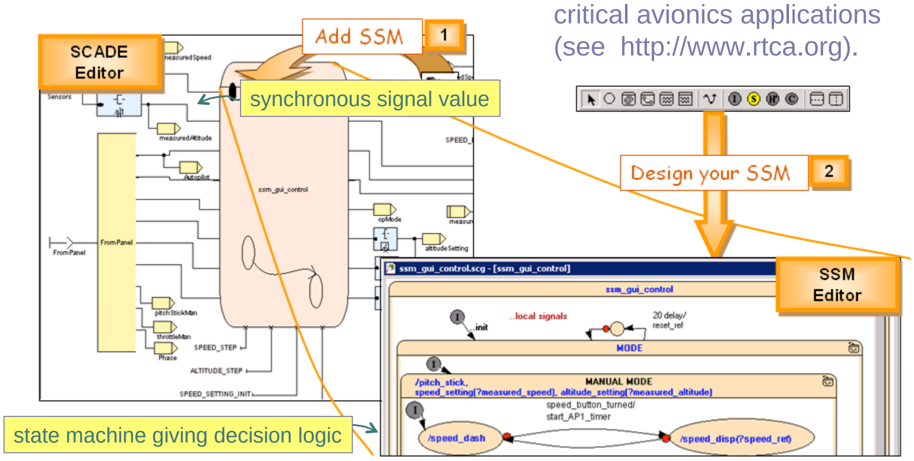
Synchronous Languages

- Esterel
- Lustre
- SCADE (visual editor for Lustre-ish/Esterel-ish lang.)
- Signal
- Statecharts (some variants)
- Ptolemy II SR domain

The model of computation is called *synchronous reactive* (SR). It has strong formal properties (many key questions are decidable).

Lustre/SCADE

The SCADE tool has a code generator that produces C or ADA code that is compliant with the DO-178B Level A standard, which allows it to be used in critical avionics applications (see http://www.rtca.org).



SCADE Editor

Add SSM 1

synchronous signal value

Design your SSM 2

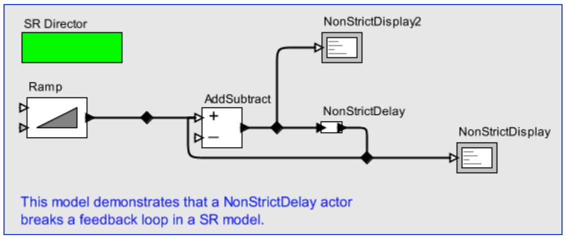
SSM Editor

state machine giving decision logic

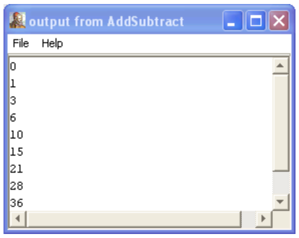
from http://www.esterel-technologies.com/

SR Domain in Ptolemy II

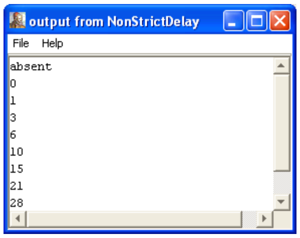
At each tick of a global "clock," every signal has a value or is absent.



This model demonstrates that a NonStrictDelay actor breaks a feedback loop in a SR model.



```
0
1
3
6
10
15
21
28
36
```



```
absent
0
1
3
6
10
15
21
28
```

The job of the SR director is to find the value at each tick.

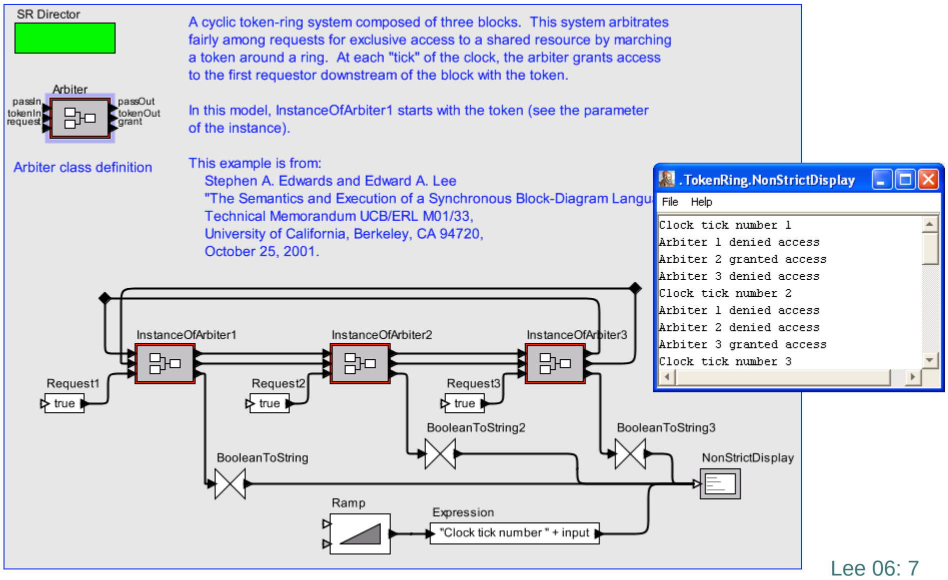
The Synchronous Abstraction

- “Model time” is discrete: Countable ticks of a clock.
- WRT model time, computation does not take time.
- All actors execute “simultaneously” and “instantaneously” (WRT to model time).
- There is an obviously appealing mapping onto real time, where the real time between the ticks of the clock is constant. Good for specifying periodic real-time tasks.

Properties

- Buffer memory is bounded (obviously).
- Hence the model of computation is not Turing complete.
  - … or bounded memory would be undecidable …
- Causality loops are possible, where at a tick, the value of one or more signals cannot be determined.

Practical Application – Token Ring Arbitration



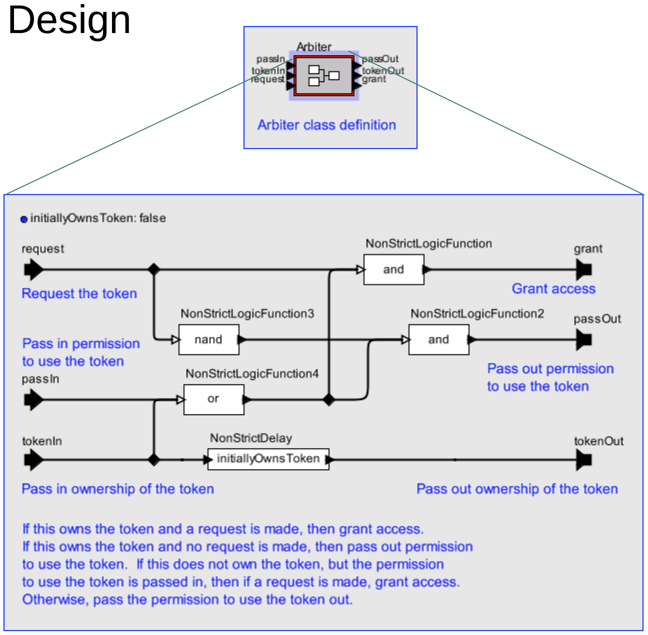
A cyclic token-ring system composed of three blocks. This system arbitrates fairly among requests for exclusive access to a shared resource by marching a token around a ring. At each "tick" of the clock, the arbiter grants access to the first requestor downstream of the block with the token.

In this model, InstanceOfArbiter1 starts with the token (see the parameter of the instance).

This example is from:
Stephen A. Edwards and Edward A. Lee
"The Semantics and Execution of a Synchronous Block-Diagram Langu
Technical Memorandum UCB/ERL M01/33,
University of California, Berkeley, CA 94720,
October 25, 2001.

```
Clock tick number 1
Arbiter 1 denied access
Arbiter 2 granted access
Arbiter 3 denied access
Clock tick number 2
Arbiter 1 denied access
Arbiter 2 denied access
Arbiter 3 granted access
Clock tick number 3
```

Arbiter Design



Request the token

Grant access

Pass in permission to use the token

Pass out permission to use the token

Pass in ownership of the token

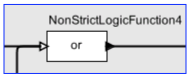
Pass out ownership of the token

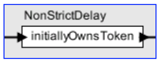
If this owns the token and a request is made, then grant access.
If this owns the token and no request is made, then pass out permission to use the token. If this does not own the token, but the permission to use the token is passed in, then if a request is made, grant access.
Otherwise, pass the permission to use the token out.

Cycles

Note that there are cycles in this graph, so that if you require that all inputs be known to find the output, then this cannot execute.

The “non strict” actors are key: They do not need to know all their inputs to determine the outputs.





Simple Execution Policy

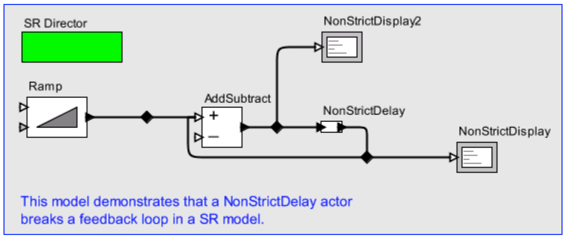
At each tick, start with all signals “unknown.” Evaluate non-strict actors and source actors. Then keep evaluating any actors that can be evaluated until all signals become known or until no further progress can be made.

Q: How do we know this will work?

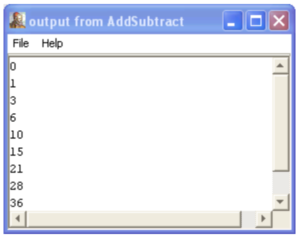
A: Least fixed point semantics.

SR Domain in Ptolemy II

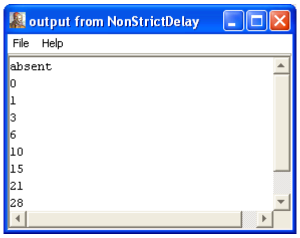
At each tick of a global “clock,” every signal has a value or is absent.



This model demonstrates that a NonStrictDelay actor breaks a feedback loop in a SR model.



```
0
1
3
6
10
15
21
28
36
```



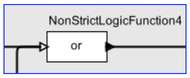
```
absent
0
1
3
6
10
15
21
28
```

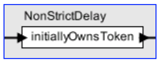
The job of the SR director is to find the value at each tick.

Cycles

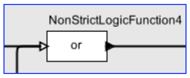
Note that there are cycles in this graph, so that if you require that all inputs be known to find the output, then this cannot execute.

The “non strict” actors are key: They do not need to know all their inputs to determine the outputs.





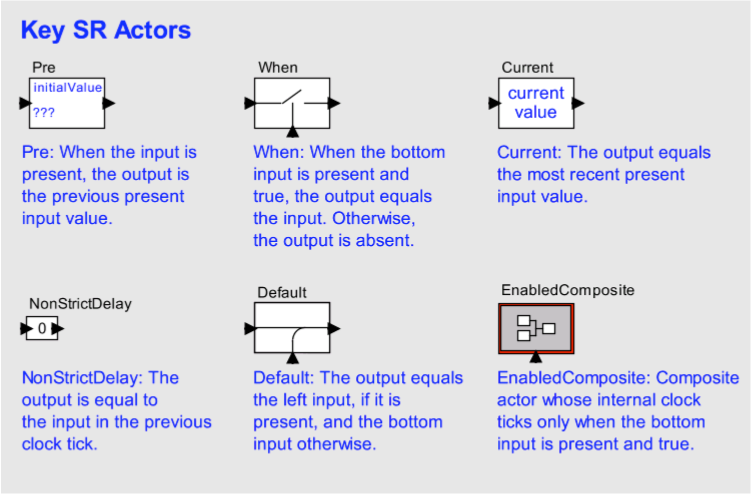
Non-Strict Logical Or



The non-strict or (often called the “parallel or”) can produce a known output even if the input is not completely know. Here is a table showing the output as a function of two inputs:

		input 1			
input 2		⊥	ε	F	T
	⊥	⊥	⊥	⊥	T
	ε	⊥	ε	F	T
	F	⊥	F	F	T
	T	T	T	T	T

More Synchronous/Reactive Actors



Key SR Actors

Pre: When the input is present, the output is the previous present input value.

When: When the bottom input is present and true, the output equals the input. Otherwise, the output is absent.

Current: The output equals the most recent present input value.

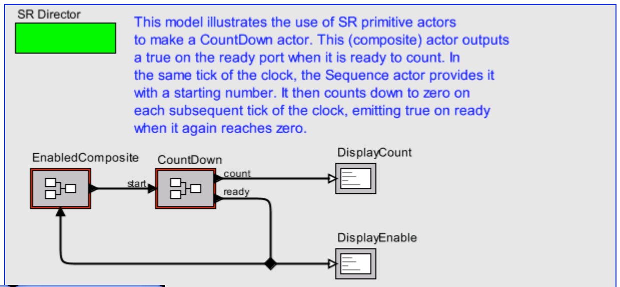
NonStrictDelay: The output is equal to the input in the previous clock tick.

Default: The output equals the left input, if it is present, and the bottom input otherwise.

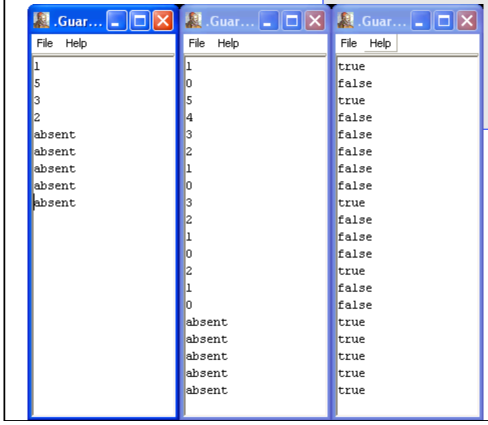
EnabledComposite: Composite actor whose internal clock ticks only when the bottom input is present and true.

Use of some of these can be quite subtle.

Design in SR: Example



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.



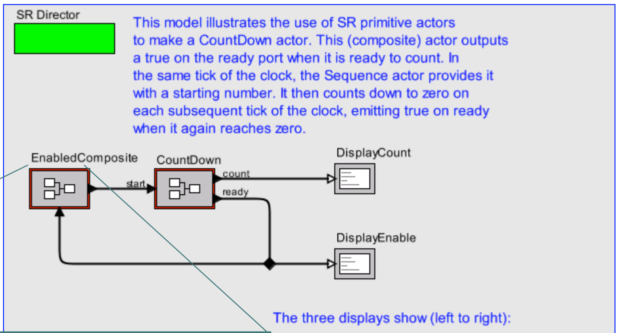
```
1        1        true
5        0        false
3        5        true
2        4        false
absent   3        false
absent   2        false
absent   1        false
absent   0        false
absent   3        true
         2        false
         1        false
         0        false
         2        true
         1        false
         0        false
         absent   true
         absent   true
         absent   true
         absent   true
         absent   true
```

The three displays show (left to right):

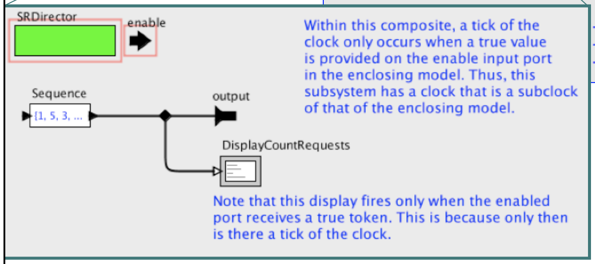
- Requested numbers to count down from.
- The count down for these numbers.
- The enable signal for the EnabledComposite actor.

In this example, the CountDown composite issues a “ready” signal to the EnabledComposite, which then issues a number. The CountDown composite counts down from that number to 0, then issues another ready.

Design in SR: Example



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.



Within this composite, a tick of the clock only occurs when a true value is provided on the enable input port in the enclosing model. Thus, this subsystem has a clock that is a subclock of that of the enclosing model.

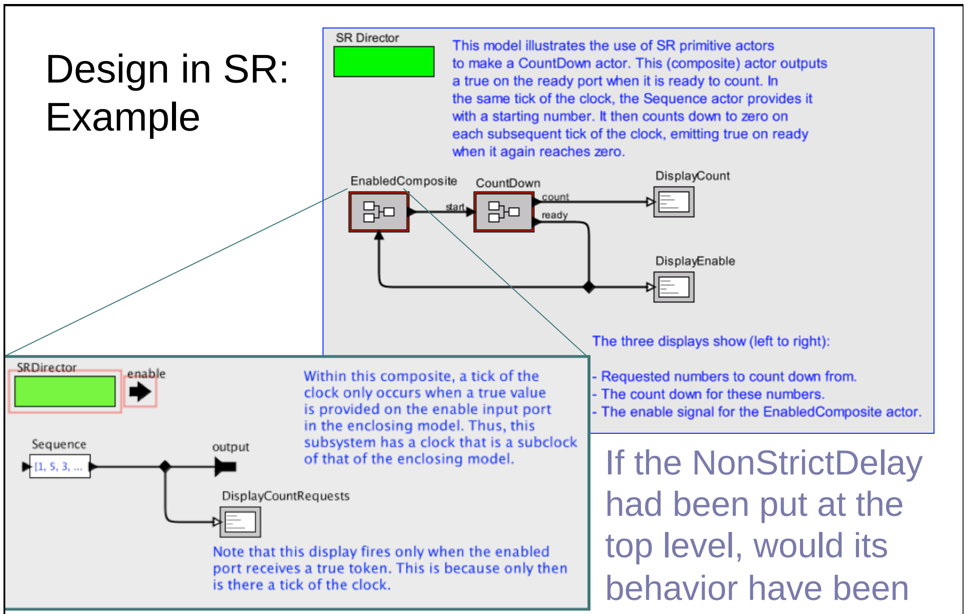
Note that this display fires only when the enabled port receives a true token. This is because only then is there a tick of the clock.

The three displays show (left to right):

- Requested numbers to count down from.
- The count down for these numbers.
- The enable signal for the EnabledComposite actor.

The EnabledComposite has a clock that ticks only when the enable input is present and true. It issues the sequence 1, 5, 3, 2, followed by absent henceforth.

Design in SR: Example



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

The three displays show (left to right):

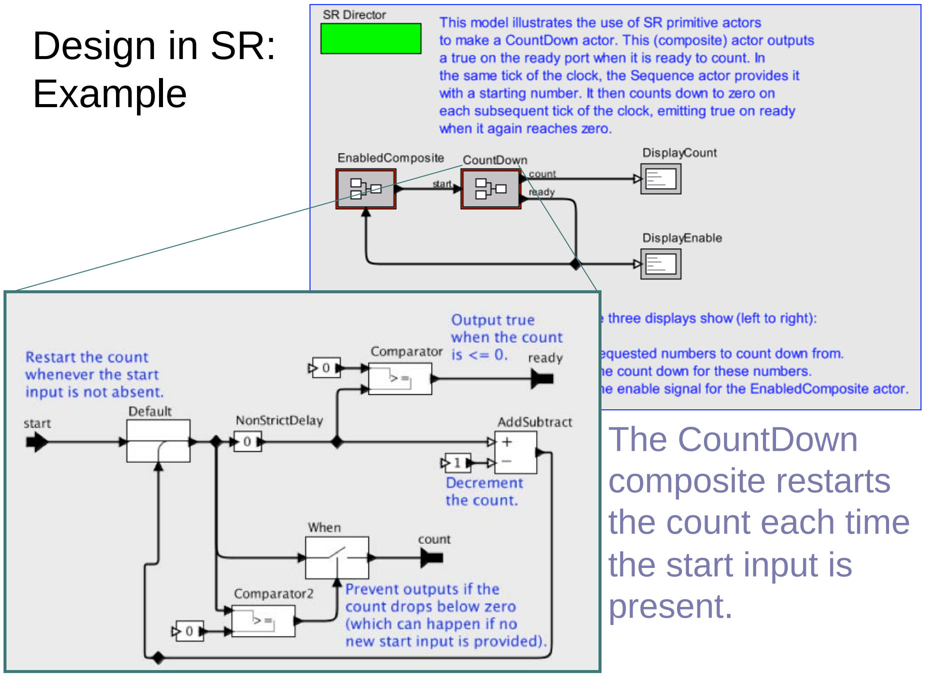
- Requested numbers to count down from.
- The count down for these numbers.
- The enable signal for the EnabledComposite actor.

Within this composite, a tick of the clock only occurs when a true value is provided on the enable input port in the enclosing model. Thus, this subsystem has a clock that is a subclock of that of the enclosing model.

Note that this display fires only when the enabled port receives a true token. This is because only then is there a tick of the clock.

If the NonStrictDelay had been put at the top level, would its behavior have been the same?

Design in SR: Example



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

three displays show (left to right):

- equested numbers to count down from.
- he count down for these numbers.
- he enable signal for the EnabledComposite actor.

Restart the count whenever the start input is not absent.

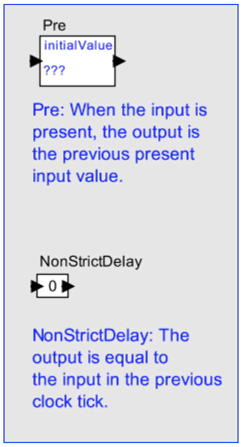
Output true when the count is <= 0.

Decrement the count.

Prevent outputs if the count drops below zero (which can happen if no new start input is provided).

The CountDown composite restarts the count each time the start input is present.

Subtleties: Pre vs. NonStrictDelay



Pre: When the input is present, the output is the previous present input value.

NonStrictDelay: The output is equal to the input in the previous clock tick.

Pre: True one-sample delay. The behavior is not affected by insertion of an arbitrary number of ticks with “absent” inputs between present inputs.

NonStrictDelay: One-tick delay (vs. one-sample). The output in each tick equals the input in the previous tick (whether absent or not).

Illustration of this Subtlety

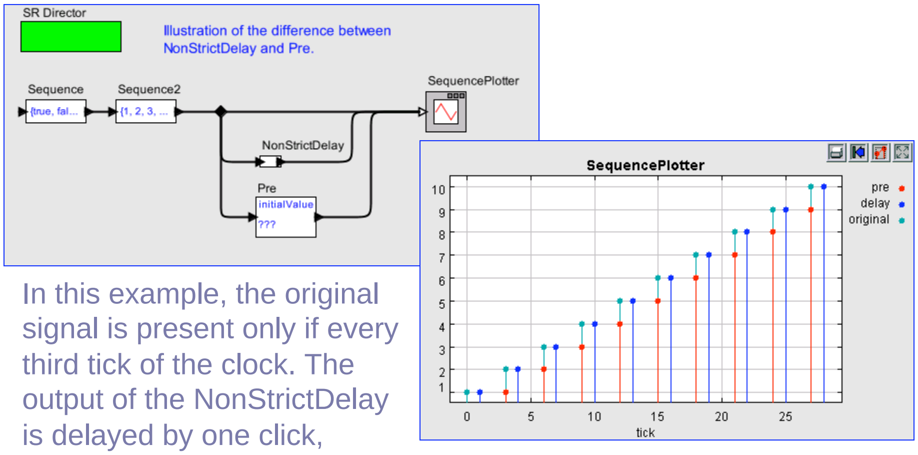
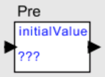


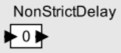
Illustration of the difference between NonStrictDelay and Pre.

In this example, the original signal is present only if every third tick of the clock. The output of the NonStrictDelay is delayed by one click, whereas the output the Pre actor is delayed by one (present) sample.

Consequences: Pre vs. NonStrictDelay



Pre: When the input is present, the output is the previous present input value.

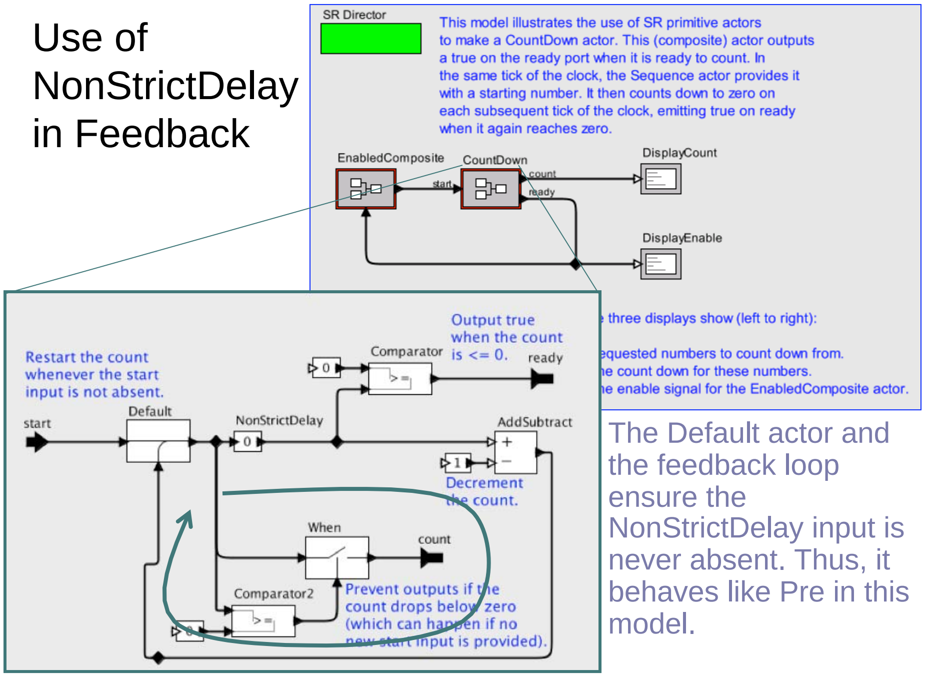


NonStrictDelay: The output is equal to the input in the previous clock tick.

Pre: This actor is *strict*. It must know whether the input is present before it can determine the output. Hence, it cannot be used to break feedback loops.

NonStrictDelay: This actor is *nonstrict*. It need not know whether the input is present nor what its value is before it can determine the output. Hence, it can be used to break feedback loops.

Use of NonStrictDelay in Feedback



This model illustrates the use of SR primitive actors to make a CountDown actor. This (composite) actor outputs a true on the ready port when it is ready to count. In the same tick of the clock, the Sequence actor provides it with a starting number. It then counts down to zero on each subsequent tick of the clock, emitting true on ready when it again reaches zero.

three displays show (left to right):

equested numbers to count down from.
he count down for these numbers.
he enable signal for the EnabledComposite actor.

Restart the count whenever the start input is not absent.

Output true when the count is <= 0.

Decrement the count.

Prevent outputs if the count drops below zero (which can happen if no new start input is provided).

The Default actor and the feedback loop ensure the NonStrictDelay input is never absent. Thus, it behaves like Pre in this model.

The Flat CPO

Consider a set of possible values $T = \{t_1, t_2, \dots\}$. Let

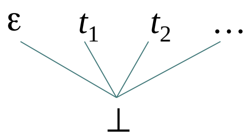
$$A = T \cup \{ \perp, \varepsilon \}$$

where $\perp$ represents “unknown” and ε represents “absent.”

Let $(A, \leq)$ be a partial order where:

- $\perp \leq \varepsilon$
- for all t in T, $\perp \leq t$
- all other pairs are incomparable

Hasse Diagram for the Flat CPO



Note that this is obviously a CPO
(all chains have a LUB)

All chains have length 2.

Monotonic Functions on This CPO

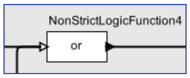
In this CPO, any function $f\colon A \to A$ is monotonic if

$$f(\bot) = a \neq \bot \;\Rightarrow\; f(b) = a \text{ for all } b \in A$$

I.e., if the function yields a “known” output when the input is unknown, then it will not change its mind about the output once the input becomes known.

Since all chains are finite, every monotonic function is continuous.

Non-Strict Logical Or is Monotonic



The non-strict or is a monotonic function $f : A \times A \to A$ where $A = \{\bot, \varepsilon, \mathrm{T}, \mathrm{F}\}$ as can be verified from the truth table:

input 1

input 2

	⊥	ε	F	T
⊥	⊥	⊥	⊥	T
ε	⊥	ε	F	T
F	⊥	F	F	T
T	T	T	T	T

Recall: Fixed Point Theorem 1

Let $(A, \leq)$ be a CPO

Let $f : A \to A$ be a monotonic function

Let $C = \{ f^n(\bot), n \in N \}$

- If $\vee C = f(\vee C)$, then $\vee C$ is the *least* fixed point of f
- If f is continuous, then $\vee C = f(\vee C)$

Intuition: The least fixed point of a continuous function is obtained by applying the function first to the empty sequence, then to the result, then to that result, etc.

Lee 06: 27

Recall: Fixed Point Theorem 2

Let $f : A \to A$ be a monotonic function on CPO $(A, \leq)$.

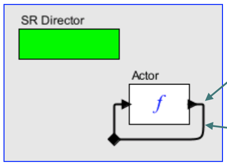
Then f has a least fixed point.

Intuition: If a function is monotonic (but not continuous), then it has a least fixed point, but the execution procedure of starting with the empty sequence and iterating may not converge to that fixed point.

This is obvious, since monotonic but not continuous means it waits forever to produce output.

Lee 06: 28

Applying Fixed Point Theorem 1



type T

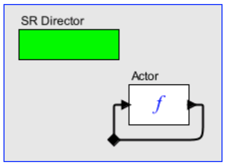
value in $A = T \cup \{ \perp, \varepsilon \}$

At each tick of the clock

- Start with signal value $\perp$
- Evaluate $f(\perp)$
- Evaluate $f(f(\perp))$
- Stop when a fixed point is reached

Unlike PN, a fixed point is always reached in a finite number of steps (one, in this case).

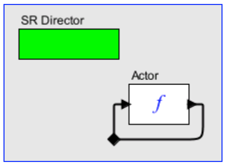
Causality Loops



What is the behavior in the following cases?

- f is the identity function.
- f is the logical NOT function.
- f is the nonstrict delay function with initial value 0.
- f is the nonstrict delay function with no initial value.

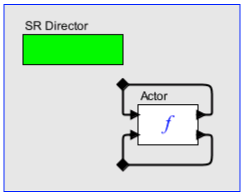
Causality Loops

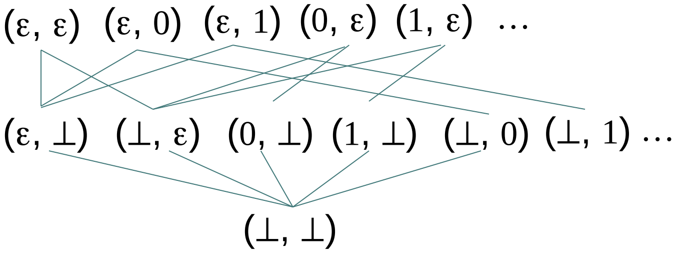


What is the behavior in the following cases?

- *f* is the identity function: ⊥
- *f* is the logical NOT function: ⊥
- *f* is the nonstrict delay function with initial value 0: 0
- *f* is the nonstrict delay function with no initial value: ε

Generalizing to Multiple Signals





(ε, ε) (ε, 0) (ε, 1) (0, ε) (1, ε) …

(ε, ⊥) (⊥, ε) (0, ⊥) (1, ⊥) (⊥, 0) (⊥, 1) …

(⊥, ⊥)

product CPO assuming $T = \{0, 1\}$.

- The Cartesian product of flat CPOs under pointwise ordering is also a CPO.
- All chains are still finite.
- Can now apply to any composition, as done with PN.

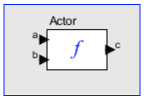
Compositional Reasoning

So far, with both PN and SR, we deal with composite systems by reducing them to a monotonic function of all the signals.

An alternative approach is to convert an arbitrary composition to a continuous function.

Example to Use for Compositional Reasoning

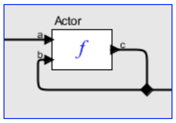
Consider an actor:



Assume $a \in A$, $b \in B$, $c \in C$, all CPOs.

Assume that the actor function $f : A \times B \to C$ is continuous

Consider the following composition:

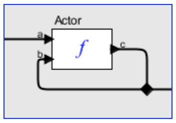


We would like to consider this a function from a to c.

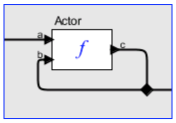
First Option: Currying (Named after Haskell Curry)

Given a function $f : A \times B \to C$, we can alternatively think of this in stages as $f_1 : A \to [B \to C]$, where $[B \to C]$ is the set of all functions from B to C.

For the following example, for each given value of a we get a new function $f_1(a)$ for which we can find the least fixed point. That least fixed point is the value of c.



Example: Non-Strict OR



Suppose f is a non-strict logical OR function. Then:

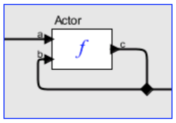
- If $a = true$, then the resulting function $f_1(a)$ always returns $true$, for all values of the input b.

  In this case, the least fixed point yields $c = true$.

- If $a = false$, then the resulting function $f_1(a)$ is the identity function.

  In this case, the least fixed point yields $c = \perp$.

Second Option: Lifting (Named after Heavy Lifting)



Given a function $f : A \times B \to C$, we are looking for a function $g : A \to C$ such that

$$c = g(a)$$

In the model we have $b = c$ and $c = f (a, b)$ so

$$g(a) = f (a, g(a))$$

This looks like a fixed point problem, but the “unknown” on both sides is g, a function not a value. If we can find the function g that satisfies this equation, then we can use it always to calculate c given a.

Posets of Functions

Suppose $(A, \leq)$ and $(C, \leq)$ are CPOs.

Consider functions $f, g \in [A \to C]$.

Define the *pointwise order* on these functions to be

$$f \leq g \Leftrightarrow \forall\, a \in A,\ \ f (a) \leq g(a)$$

Let $X \subset [A \to C]$ be the set of all continuous total functions from A to C.

Theorem: $(X , \leq)$ is a CPO under the pointwise order.

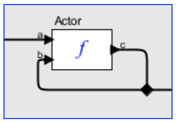
Proof: See handout.

Least Function in the CPO of Functions

Let $X \subset [\, A \to C \,]$ be the set of all continuous total functions from A to C. Since X is a CPO, it must have a bottom. The bottom is a function $\perp_X : A \to C$ where for all $a \in A$,

$$\perp_X(\, a \,) = \perp_C \in C$$

Consequence of this Theorem



Given a continuous function $f : A \times B \to C$, the function $g : A \to C$ such that

$$c = g(\, a \,)$$

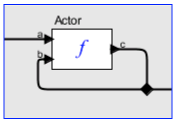
is the least fixed point of a continuous function

$$F : X \to X$$

where $X \subset [\, A \to C \,]$ is the set of all continuous total functions from A to C.

We need to now determine the continuous function F .

Consequence of this Theorem (Continued)



We need to find a function that g satisfies:

$$g(\, a\,) = f\,(\, a,\, g(\, a\,))$$

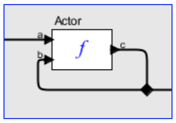
Let $X \subset [\, A \rightarrow C\,]$ be the set of all continuous total functions from A to C and let F be a continuous function $F : X \rightarrow X$.

Then $g \in X$ is the least function such that $F\,(\, g\,) = g$ where for all $a \in A$,

$$(F\,(\, g\,))(\, a\,) = f\,(\, a,\, g(\, a\,))$$

The theorem, with fixed point theorem 1, tells us that F has a least fixed point, and tells us how to find it.

Example: Non-Strict OR



Suppose f is a non-strict logical OR function. Then:

$$(F(g))(a) = \begin{cases} \mathit{true} & \text{if } \; a = \mathit{true} \\ \bot & \text{if } \; a = \bot \text{ and } g(a) = \mathit{false} \\ g(a) & \text{otherwise} \end{cases}$$

The least fixed point of this is the function g given by:

$$g(a) = \begin{cases} \mathit{true} & \text{if } \; a = \mathit{true} \\ \bot & \text{otherwise} \end{cases}$$

To find this, start with $F\,(\,\bot\,)$, then find $F\,(\,F\,(\,\bot\,))$, etc., until you get a fixed point (which happens immediately).

Showing that F is Continuous

Need to show that given a chain of continuous total functions $C = \{\, g_1, g_2 \ldots \,\}$ that:

$$F(\vee C) = \vee \hat{F}(C)$$

For all $a \in A$:

$$\begin{aligned}
(F(\vee C))(a) &= f(a, (\vee C)(a)) \\
&= f(a, \vee\{g_1(a), g_2(a), \ldots\}) && \text{because each } g_i \text{ is continuous} \\
&= \vee \hat{f}(a, \{g_1(a), g_2(a), \ldots\}) && \text{because } f \text{ is continuous} \\
&= (\vee \hat{F}(C))(a) && \text{QED}
\end{aligned}$$

Summary

- In SR, fixed point semantics is simpler than in PN because the CPO has only finite chains.
- The fancier techniques of Currying and Lifting can be applied equal well to PN, but we introduce them here because the simpler CPO makes them easier to understand.

- The fixed point semantics of SR talks only about the behavior at a tick of the clock. The behavior across ticks of the clock will require a *clock calculus*.