# Embedded Systems Design and Modeling

Chapter 6

Concurrent Models of Computation

# Outline

- Introduction
- Structure of models
- Synchronous Reactive systems
- Dataflow MoC:
  - Principles,
  - Synchronous DF
  - Dynamic DF
  - Structured DF
  - Process Networks
- Timed MoCs

# Introduction

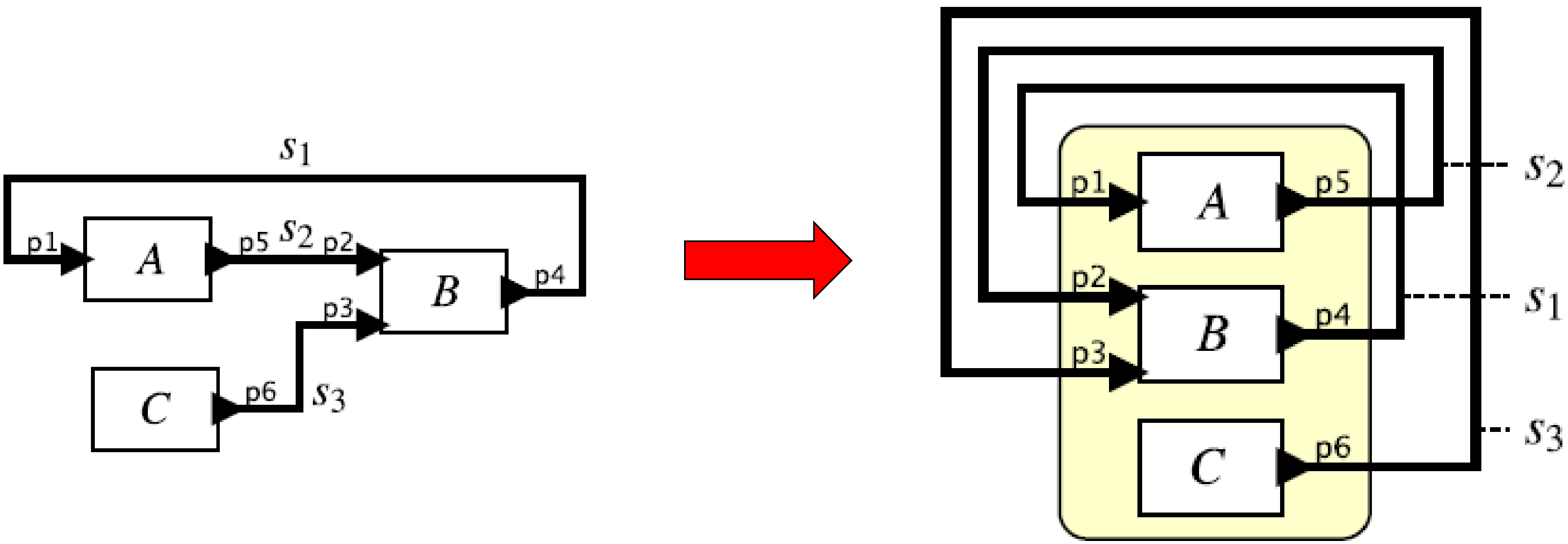
- Concurrent system: all of the different components and parts of the system operate simultaneously
- Simultaneity is mainly conceptual
- In reality the parts of the systems might only “appear” to operate simultaneously like a multithread software
- The semantics of a concurrent system is determined by its model of computation (MoC)

# Recalling MoC

- An MoC consists of three rules:
  1. What are the components and their structural relationship?
  2. What mechanism is used to generate concurrency or at least the appearance of it?
  3. What mechanisms are used for communication between the components?
- We will start with the common structures
- Any actor network can be a feedback structure ...

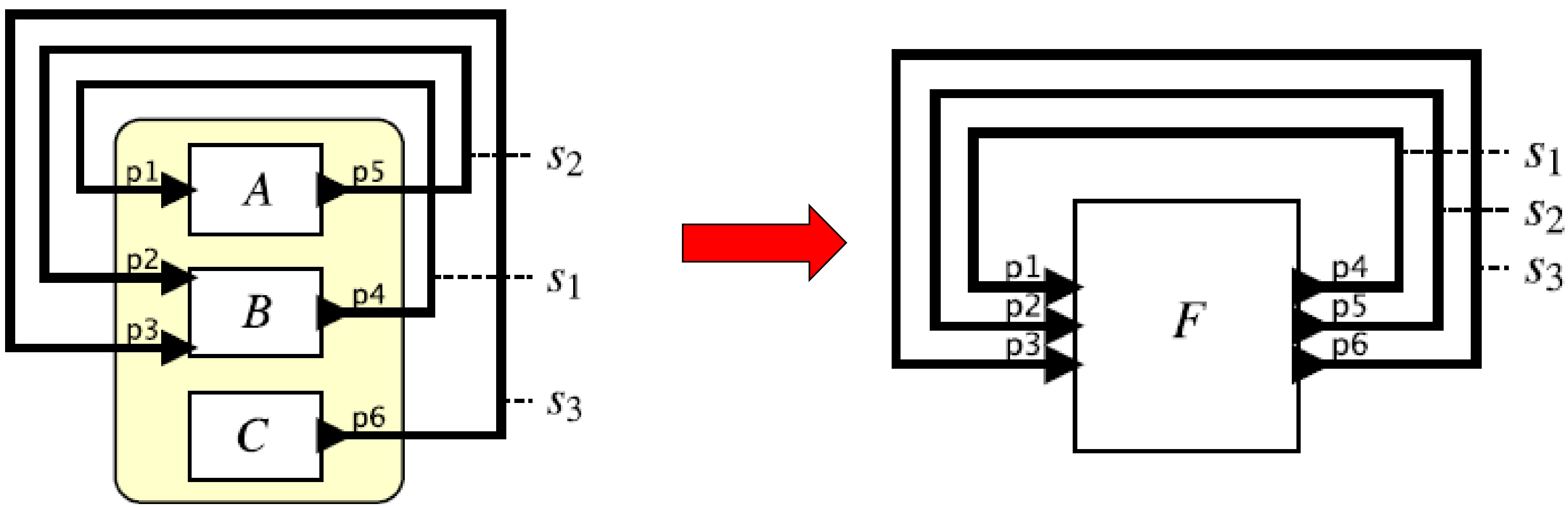
# Feedback Structure

- First, we need to rearrange all of the blocks as shown below
- The resulting composition is side-by-side



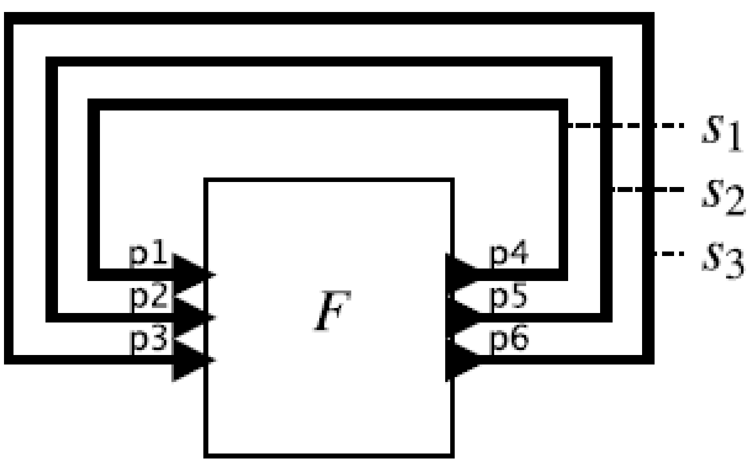
# Single Actor Representation

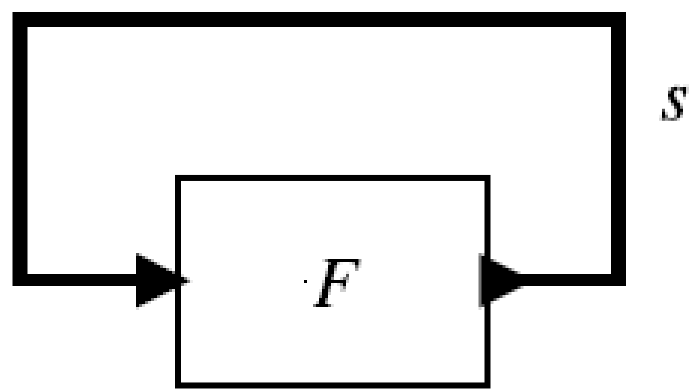
- Then, the side-by-side composition can be integrated into a single actor F



# Single Actor Representation (Cont'd)

- Finally, the single actor can be viewed as a feedback system without its details as shown below
- Pay close attention to how the connections are preserved



# Synchronous-Reactive MoC

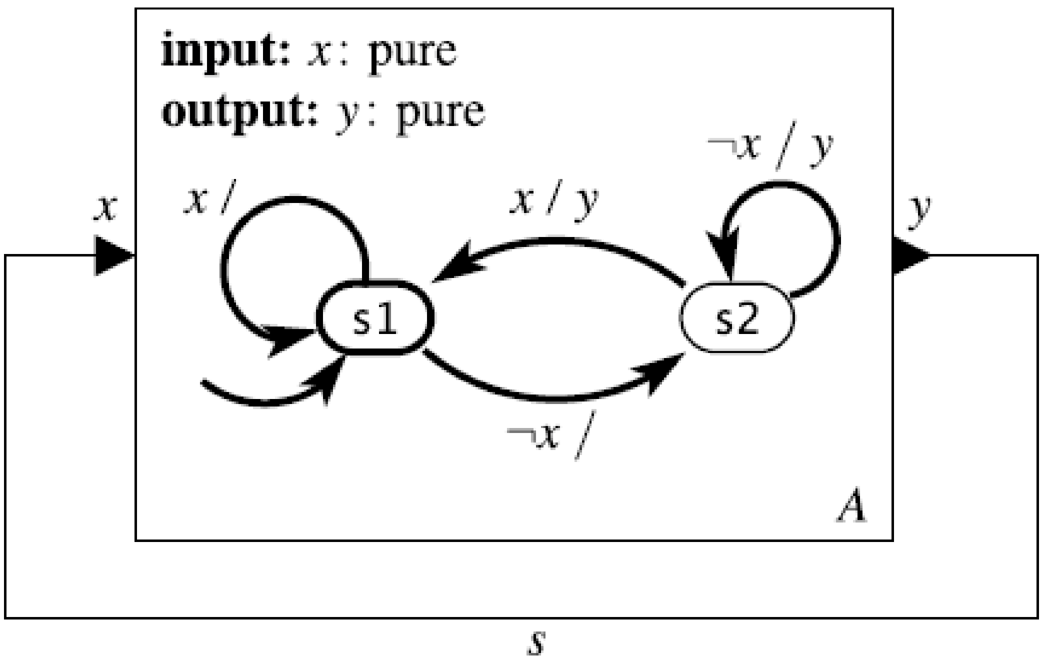
- Feedback systems have a challenge:
  - To know the output, one needs to know the input
  - To know the input, one needs to know the output
- Synchronous-reactive (SR) MoC can resolve this
- SR is a discrete system in which the signals (input and output) are always absent except at ticks of a global clock

# Synchronous-Reactive (Cont'd)

- The model runs at the ticks only
- At each tick, each component or actor reacts to the signals at its inputs
- All reactions are simultaneous and instantaneous (i.e., time delays in computations are irrelevant)
- SR is synchronous in the same sense as synchronous digital circuits
- So, one needs to look at the reactions at the ticks only

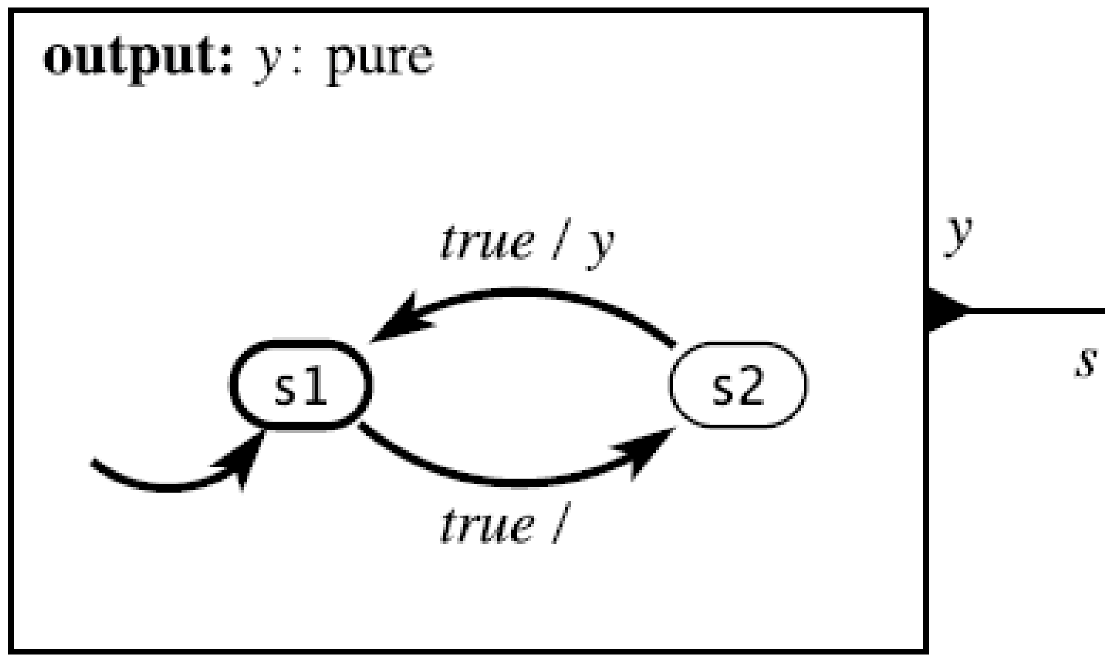
# SR Example

- Consider this example:
- Due to the feedback input equals output
- Detail examination shows that the inputs and outputs alternate between absent and present



# SR Semantics

- Based on our analysis, one can say that the semantics of the synchronous-reactive model is like this:
- Note that there is no input anymore because it is not really needed!

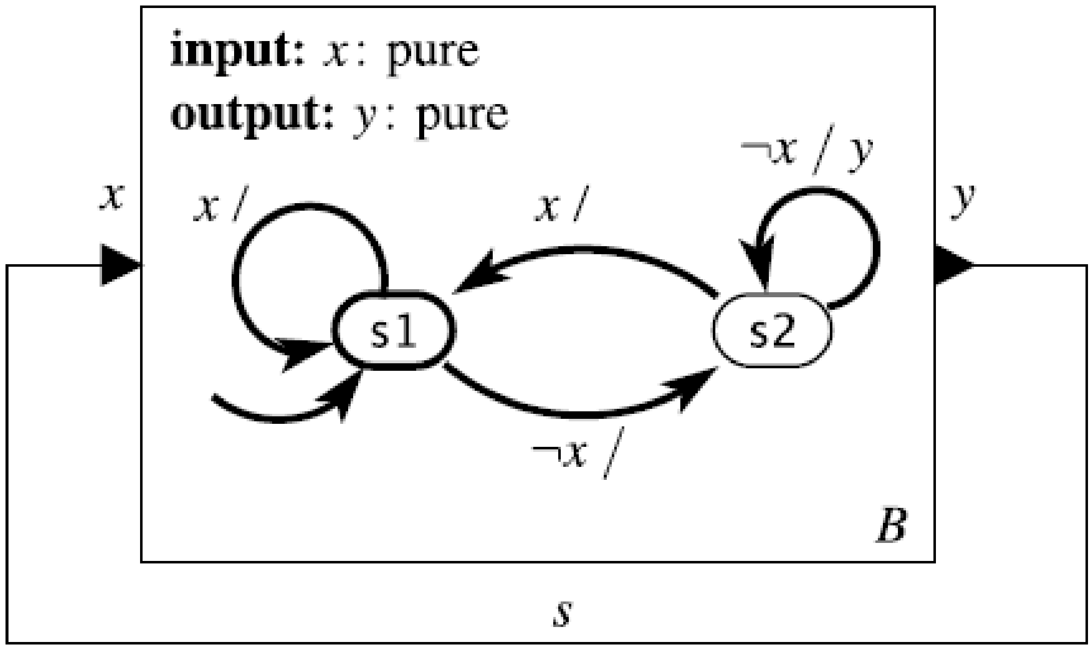


# Fixed Point Definition

- The previous example has an important characteristic:
  - If we assumed one form for the input (absent or present), we would get the same form for the output
  - No contradiction between the input and output
- This is called a “fixed point”
- Do all systems have a single fixed point?
- No, there may be no fixed point or more than one ...

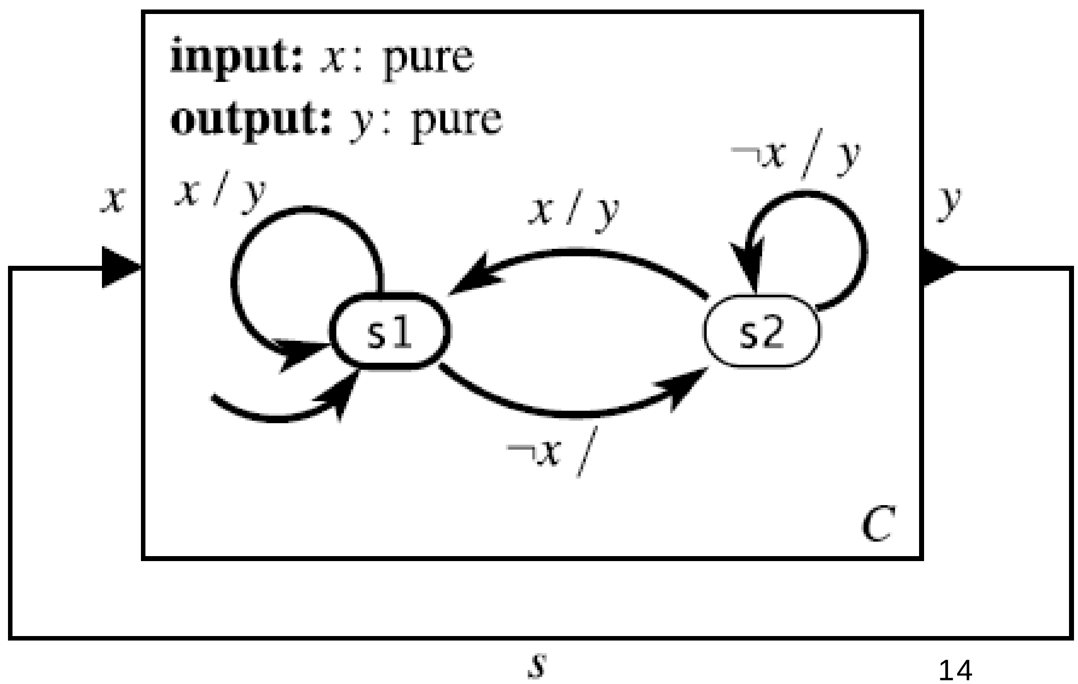
# No Fixed Point Example

- In this example (B), if we assume the input to be absent at S1, the output will be absent too => a single fixed point at S1
- If we assume the input to be absent at S2, the output will be present and vice versa => no fixed point at S2



# Two Fixed Points Example

- In system C, if we assume the input to be absent at S1, the output will be absent too
- If we assume input to be present at S1, the output will be present too
- There are TWO fixed points at S1
- S2 still has only one fixed point (like system B)



# Ill/Well-Formed Systems

- Dealing with feedback systems with no fixed point or more than one fixed point at a state doesn't provide a deterministic and unambiguous behavior
- The exception is when the state is unreachable
- So we define: if a state is <u>reachable</u> and has no fixed point or more than one, the system is called "ill-formed"
- Otherwise, it is called "well-formed"

# Ill/Well-Formed Determination

- Systems B and C in the previous examples are ill-formed
- How can we determine if a system is well-formed or not?
- Usually, we have to do exhaustive search which means trying all possible cases
  - Exhaustive search is possible only if the data types are finite
  - Exhaustive search is practical only if the search space is small
- Hence the need for another approach

# Constructive Systems

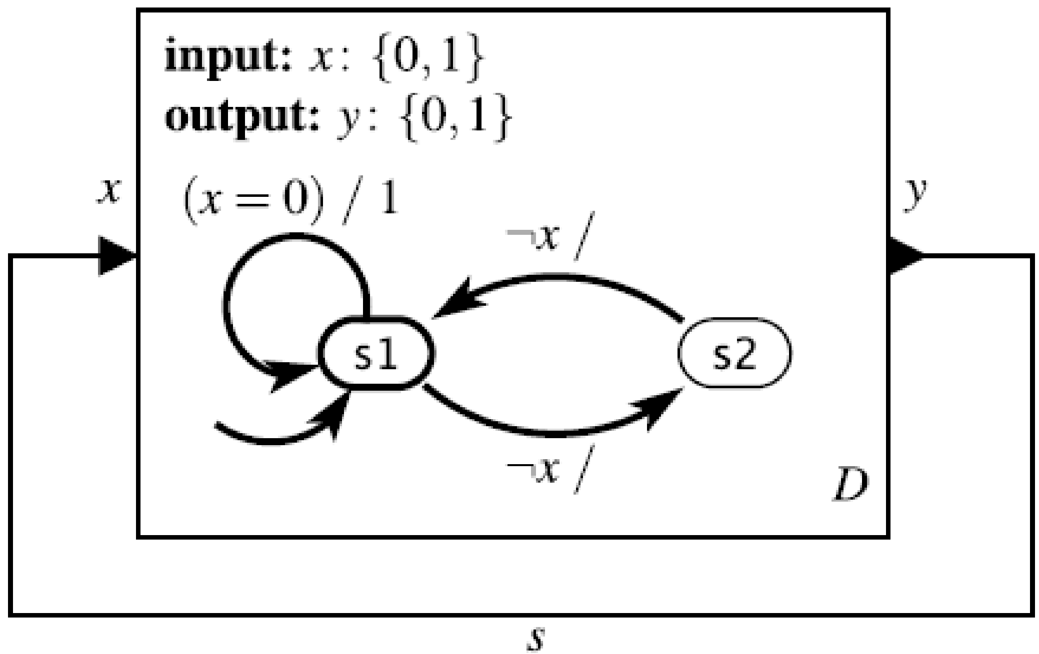
- An alternative procedure to determine if a system is well-formed or not:
  - For each reachable state i:
    1. Assume the output s(n) is unknown
    2. Determine as much as possible about $f_i(s(n))$
    3. Repeat until s(n) becomes known (if it is present or not, and if it is, what its value is) or no progress can be made
    4. If unknown values remain, reject the model
- A state machine is called “constructive” if this procedure works, otherwise it is called “non-constructive”

# Non-Constructive But Well-Formed

- A constructive machine is well-formed
- But a machine that fails this procedure (i.e., a non-constructive machine) can also be well-formed as shown in the next example
- The system D has one input and one output
- They are not pure, i.e., they have values
- So if they are present, we need to know their values

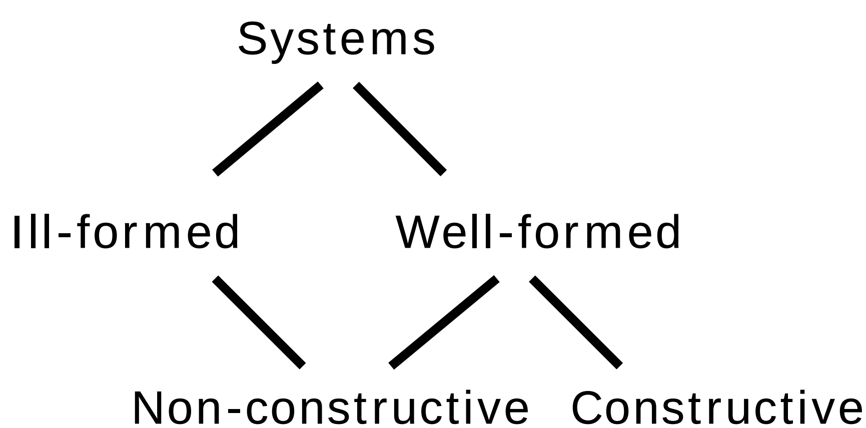
# Example Continued

- If we assume the input to be absent at state S1, the output will be absent too => there is one fixed point at S1
- If we assume the input is present at S1, the output value will not be known
- Observe that D has only one fixed point at state S2



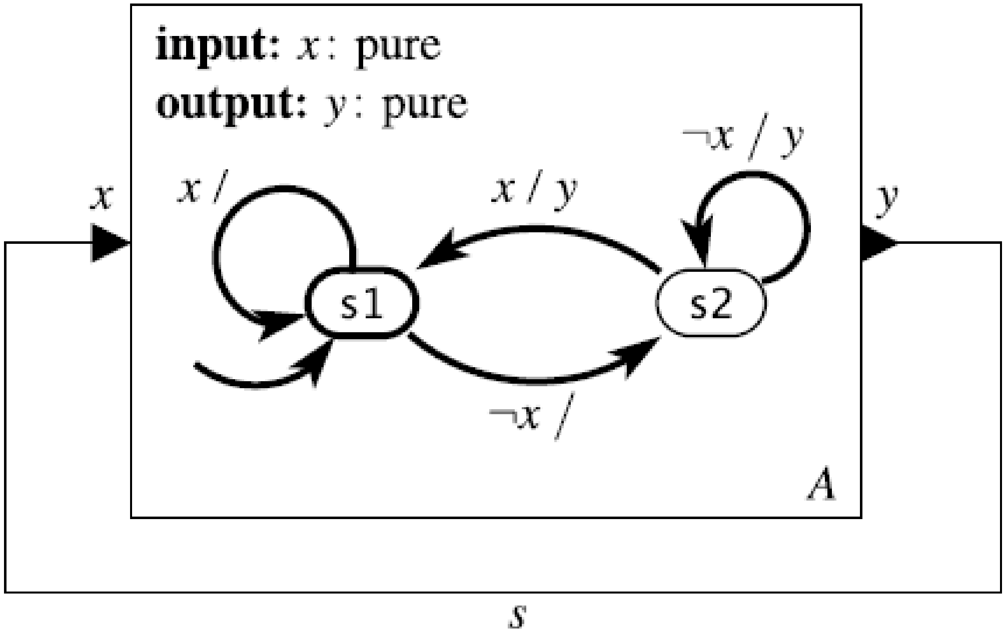
# Putting It Together

- System D is well-formed but it is non-constructive because the output is not known for all the cases in state S1 and it fails the procedure
- So for non-constructive systems, we have to do exhaustive search to see if they are well-formed or not



# Must-May Analysis

- The tools cannot do exhaustive search
- They do a <u>must</u>-<u>may</u> analysis instead
- Example: system A <u>may</u> not produce an output in state S1
  - So input is absent
- In S2, it <u>must</u> have an output
  - So input is present
- No need to check all cases



# SR Conclusions

- As mentioned, in SR models actors react simultaneously and instantaneously
- This model may not always be a realistic model of the physical world and requires tight coordination of the actors
- There are other concurrent MoC’s that do not require this tight coordination and will be discussed next

# Dataflow MoC

- Lifting the requirement for reactions to occur simultaneously allows decentralized decisions => the need for another MoC
- In general, the reactions can be independent of each other in terms of timing
- But they are data dependent, i.e., they are based on the flow of data
- Dataflow: an MoC in which constraints on reactions are based on data dependencies

# Dataflow Variations

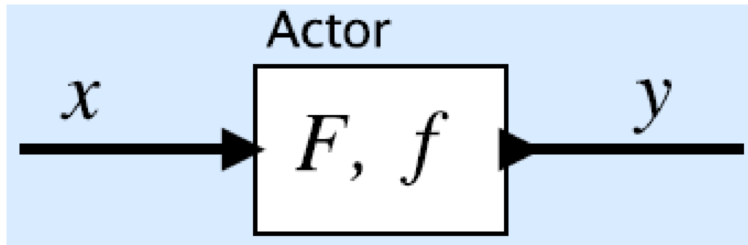
- There are many forms of dataflow but we will only consider:
  - Synchronous dataflow or SDF
  - Dynamic dataflow
  - Structured dataflow
  - Process networks
- But we first describe the main principles that are common among the various forms

# Dataflow Principles

- In dataflow MoC a signal is a sequence of messages
- Each message is called a token
- Two functions describe the system:
  - Actor function which maps the entire input sequences to the entire output sequences
  - Firing function which maps a finite portion of the input sequences to output sequences
- The difference between these two will be shown in the next example

# Dataflow Example

- Consider the following actor:




- If this is a scale actor which multiplies its input sequence by "a", then:

$$F(x_1, x_2, x_3, \cdots) = (ax_1, ax_2, ax_3, \cdots)$$

- If the actor performs one multiplication upon firing, then:

$$f(x_1, x_2, x_3, \cdots) = f(x_1) = (ax_1)$$

- So the output sequence is of length one

# Firing Function and Rules

- The firing function doesn't need to wait for a whole sequence to start
- Instead, as soon as enough tokens arrive at the input, the firing can occur
- A firing rule specifies how many tokens must be received at each input to fire the actor
- There may be too many tokens arriving at the input to be responded immediately => need some form of buffering mechanism

# Firing Function and Rule (Cont'd)

- When firing occurs, the required tokens will be consumed
- After a token is consumed, it can be discarded
- An unconsumed token remains in the buffer until it is consumed
- This means if the rate of token arrival is higher than the rate of token consumption, a buffer overflow may happen over time

# Unbounded Execution

- A model should be able to run for a very long time which is called “unbounded execution”
- 1st problem is to have scheduling policies that guarantee unbounded execution with bounded buffers
- 2nd problem is to prevent a “deadlock” which is when there is not enough tokens to start the system
- A delay actor can help with deadlocks

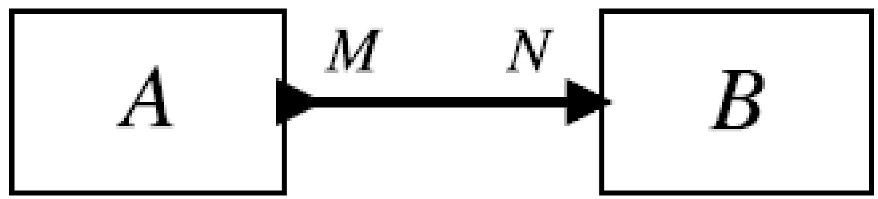
# Synchronous Dataflow

- In general, the two problems mentioned earlier are undecidable, i.e., there is no algorithm to find their answer
- But with some restrictions on the dataflow, the problem can become decidable
- For example, each actor in a Synchronous Dataflow (SDF) consumes a fixed and known number of input tokens and produces a fixed number of output tokens

# Clarification

- The terminology is rather misleading and confusing:
  - Synchronous dataflow is NOT synchronous (like the SR MoC)
  - There is no global clock in SDF in contrast to SR
  - There is no particular timing requirement for the production and/or consumption of tokens
  - The overall rate of production and consumption should be the same

# SDF Example




- Consider this SDF:
- When $A$ fires once, $M$ tokens are produced. If it fires $q_A$ times, $q_A M$ tokens are produced
- If $B$ fires once, $N$ tokens are consumed. If it fires $q_B$ times, $q_B N$ tokens are consumed
- The balance equation that ensures unbounded execution is therefore:

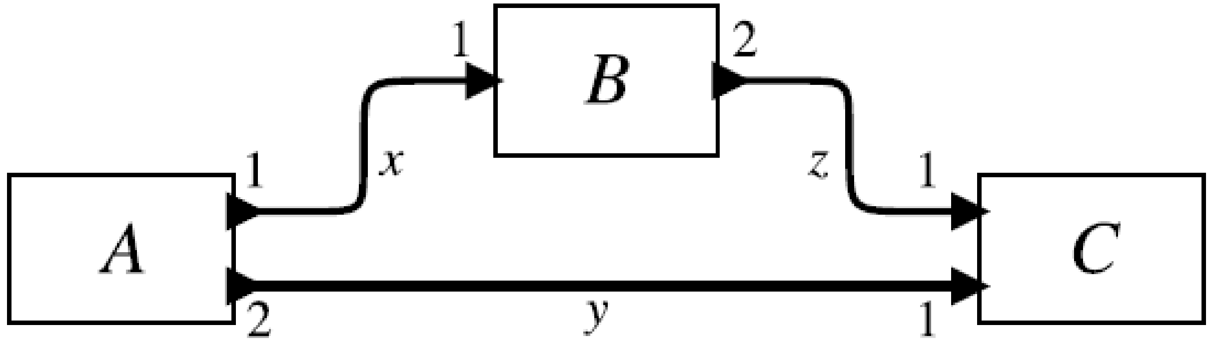
$$q_A M = q_B N$$

# Balance Equation Example

- For $M=2$ and $N=3$, $q_A$ can be 3 and $q_B$ can be 2 to satisfy the balance equation
- Any order of tokens that keeps $q_A$ and $q_B$ can be repeated forever
- Examples: A,A,A,B,B or A,A,B,A,B
- The difference between different orders is the amount of memory buffer they need
- Generally, the least positive integer values for $q_A$ and $q_B$ are the best

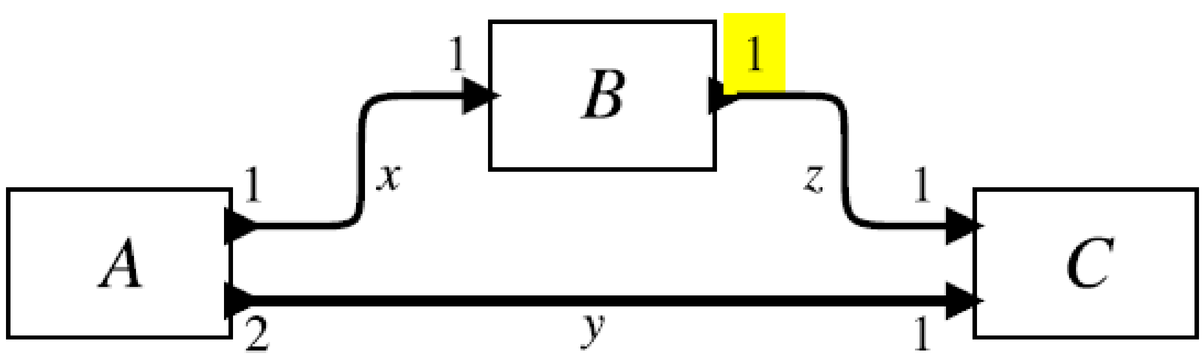
# Example With Three Actors

- In this example, there are 3 connections and 3 balance equations
- These equations have a non-zero set of solutions
- The least positive integer solution is $q_A = q_B = 1$ and $q_C = 2$
- So this SDF is called “consistent”



# Consistent vs. Inconsistent SDFs

- The three balance equations in this example don’t have a non-zero set of solutions
- There won’t be an unbounded execution with bounded buffers
- So this SDF is called “inconsistent”

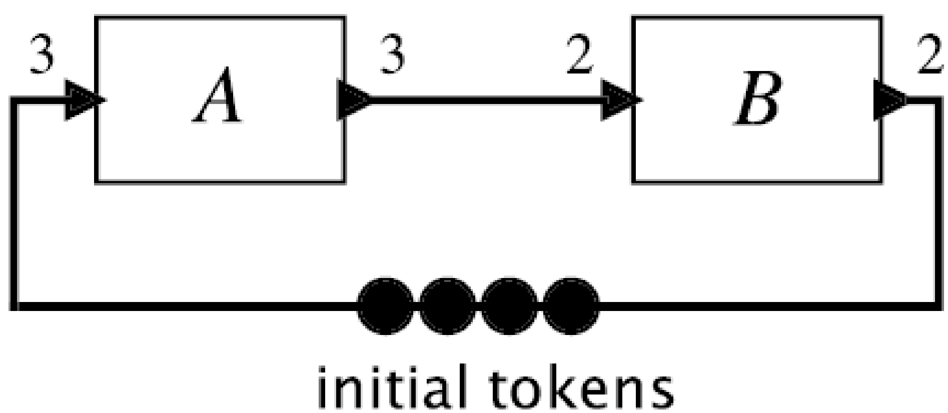


# Consistency Observations

- The balance equations solve the 1st problem:
  - If the balance equations have a non-zero solution, a positive integer solution exists
  - If so, the scheduling problem for SDFs can be solved (consistent SDF)
- Consistency is necessary and sufficient condition for bounded buffer
- Consistency is necessary but not sufficient condition for unbounded execution

# Deadlock Problem

- The 2nd problem (deadlock) can also prevent the unbounded execution
- Delay actors which can be viewed as initial tokens can solve this problem (why?)
- This SDF requires at least 4 initial tokens to start

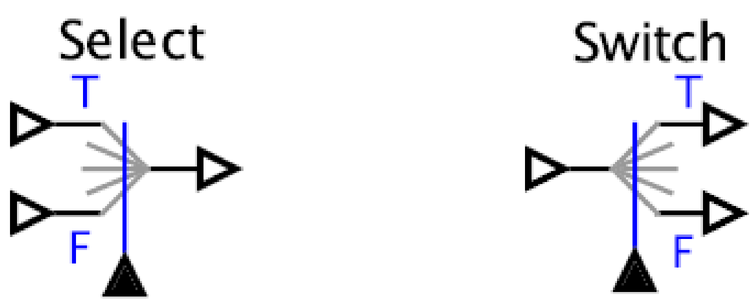


# Dynamic Dataflow

- SDF is decidable but not very expressive:
  - Cannot express conditional firing
- Dynamic dataflow (DDF) addresses this shortcoming:
  - DDF actors can have multiple firing rules
  - The number of output tokens can be different depending on the input token values
- Other than delay, two other basic actors:
  - Switch
  - Select

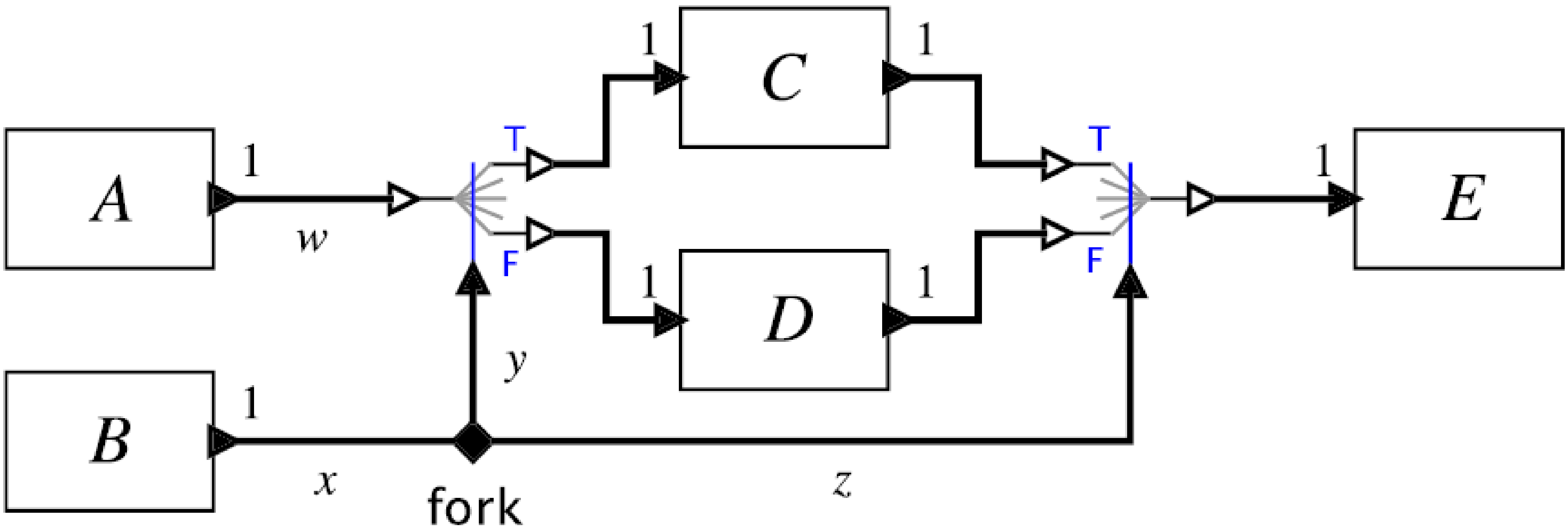
# Select And Switch Actors

- Select actor requires one Boolean (true/false) token at the bottom input:
  - If true, the top left input port is activated
  - If false, the bottom left input is activated
- Switch actor has the complementary function



# DDF Example

- In this system, actor B produces the conditional tokens for select and switch
- The fork repeats the same token twice
- Acts like if-then-else in software

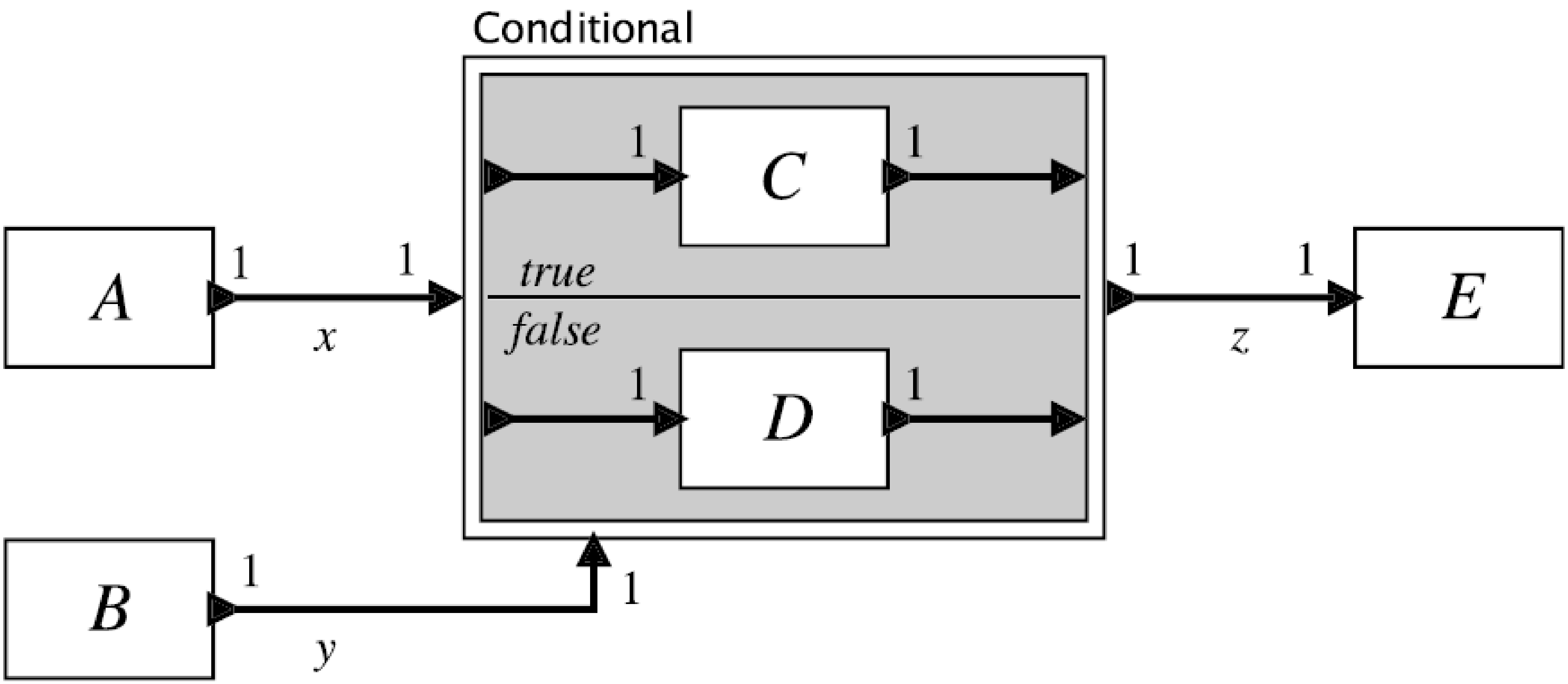


# Structured Dataflow

- While DDF models support conditional firing, they are not decidable
- Structured dataflow models support conditional firing and they are decidable:
  - Conditional control is achieved hierarchically
  - Arbitrary data-dependent token routing is avoided (like “goto” statements)
  - The overall model is still an SDF
- Hence, the model is decidable
- Consider next example...

# Structured Dataflow Example

- Each actor produces/consumes 1 token => unbounded execution + no deadlock



# Process Networks

- Kahn process networks or PN:
  - All actors execute simultaneously (this is different than SR which reacts simultaneously and SDF which waits for an input token)
  - No firing functions, i.e., each actor continuously reads data tokens from input and writes data tokens to output
  - A key to ensure determinism is blocking reads which mean the actor stops until data is ready
  - The write to output(s) is non-blocking
- Original PN is not decidable (revisions are)

# Concurrent MoC’s And Time

- Notion of time in concurrent MoC’s:
  - SR: Time is represented by a global clock only, but the clock just determines the sequence of events, so time is irrelevant
  - Global clock may not be periodic or its rate may vary without any effect on the system
  - DF, SDF, DDF, Structured DF: Time plays no role at all
- In many cases, modeling a cyber-physical system requires explicit notions that represent the actual passage of time

# Timed MoC's

- Hence the need for models which can represent the actual and exact passage of time explicitly
- Compared with the previous MoC's:
  - A time-triggered model is somewhat similar to SR as there is a notion of time
  - But events are NOT simultaneous and instantaneous, i.e., computations have an execution time

# Timed MoC's (Continued)

- Specific considerations for timed MoC's:
  - The inputs are available at the tick of a global clock
  - Computations have as much time as the duration of the clock to finish
  - The outputs will be available at the tick of the next clock
  - No interaction among actors occurs b/w clocks
  - No room for the race condition
  - Feedback is allowed => all models are constructive

# Modeling Time

- Depending on the nature of the system, the time can be modeled in two different forms:
  1. As a mere sequence of events in which the passage of time is not important but the arrival of time is
     - This form is called Discrete Event systems (or DE)
  2. Continuous time systems in which the specific passage of time plays a role in determining the behavior of the system

# Discrete Event MoC

- In DE:
  - An event is a signal that occurs at a specific point in time
  - Events carry a value and a time stamp which
    - indicates the time at which the event occurs
    - is typically generated by the actor that produces the event
    - is determined by the time stamp of input events and the latency of the block
  - Global time is known simultaneously throughout the system

# Discrete Event MoC (Cont'd)

- The system is seen as a network of actors that react according to their specific time stamps
- The time stamps are compared and the actor with appropriate time stamp is executed
- DE MoC's are also called "event-driven models"
- DE MoC's useful for modeling distributed/parallel HW or SW and their communication infrastructure
- A DE simulator (like event-driven HDL tools) needs to:
  - maintain a global event queue that sorts the events by their time stamps
  - chronologically process each event by sending it to the appropriate actor which reacts to the event

# DE Implementation

- Usual implementation of DE:
  - Prepare a list of events to be sorted based on time stamps (called “event queue”)
  - Populate the queue with the initial events
  - Start the execution by shifting the time to the first event in the list
  - Execute the first event, if a value is generated, present it to the receiving actor
  - The receiving actor might fire and produce an output
  - Update the event queue
  - Execute the next event in the list

# DE Challenges

- Important questions:

1. Can an actor have a zero delay (i.e., instantaneous reaction)?
   - Answer: some variants don’t allow that but some do
     - If they don’t allow that, there won’t be any problem with the feedbacks
     - If they do, there will be some problems with the feedback
   - Have to resolve those problems like a fixed point in the SR systems

# DE Challenges (Continued)

2. What would happen if two or more events have the same time stamp, i.e., when two events occur simultaneously?
   - If one actor output has the same time stamp as input, the same as last question
   - If more than one event is ready to be fired at the time stamp, then fire them all
   - When there is no feedback, no difficulties
   - If there is a feedback among the actors, then treat them like SR and look for a fixed point
   - Might require instantaneous firing

# Other Timed MoCs

- There are other types of discrete timed-MoC's:
  - The most important one is Petri Nets
  - The book doesn't cover it well
  - It will be covered separately at the end
- Continuous-time MoC's:
  - The most important one is ordinary differential equations (ODE's)
  - Cannot be solved easily using computers
  - Have to convert them into difference equations to solve them

# Concluding Remarks

- Chapters 7-10 deal with the implementation issues in the design of embedded systems
- Often covered in BS courses, won’t be covered in this course
- Next topic will be Petri Nets
- Then will move on to multitasking through multithreading (Chapter 11)
- Chapter 6 homework: 1, 3, 4, 8, 9, 10
- Due date Tuesday 1403/2/18