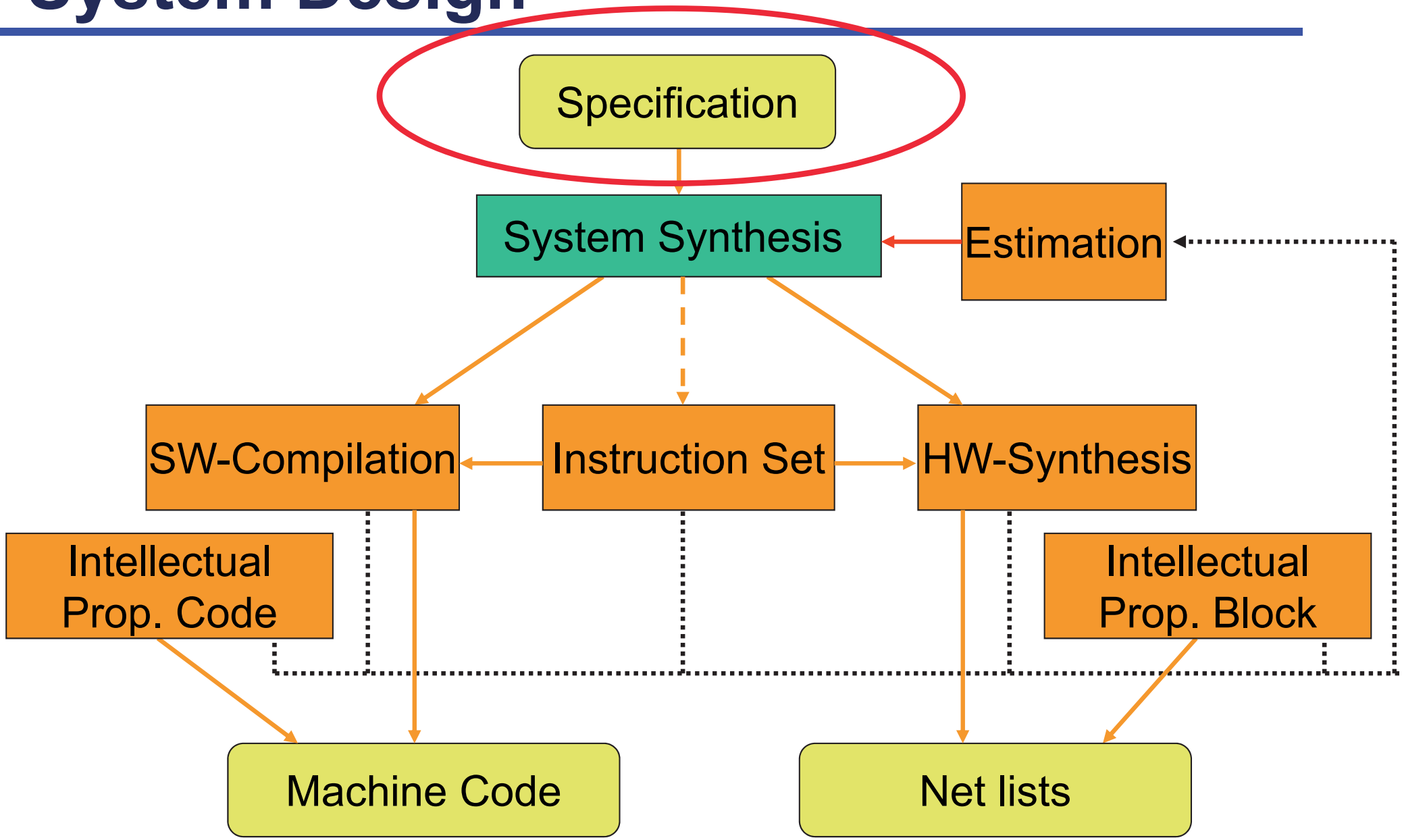
Hardware Software Codesign

2. Specification and Models of Computation

Lothar Thiele

System Design



Consider a simple example

"The Observer pattern defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically."

Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addision-Wesley, 1995

Example: Observer pattern in Java

```
public void addListener(listener) {…}

public void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<myListeners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

adds observer *listener* to data structure *myListeners*

changes subject state

changes observer state

Will this work in a multithreaded context?

Observer pattern with mutexes

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<myListeners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

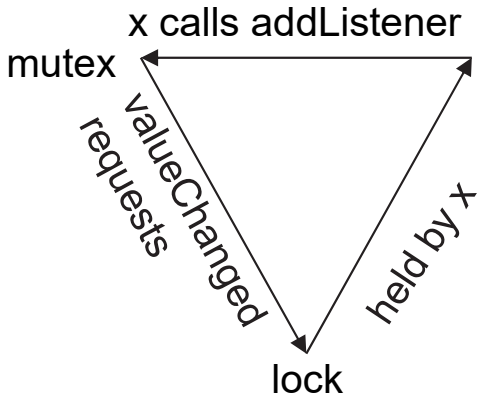
Resolves race condition between addListener and setValue

Javasoft recommends against this. What's wrong with it?

Mutexes are minefields

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<myListeners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```



valueChanged() may attempt to acquire a lock on some other (independent) object and stall. If the holder of that lock calls addListener(): deadlock!

Simple observer pattern gets complicated

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
      listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make a copy of listeners to avoid race conditions

notify each listener outside of the synchronized block to avoid deadlock

This still isn't right.
What's wrong with it?

Simple observer pattern: How to make it right?

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
      listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object (in myValue), but listeners may be notified in the opposite order. Listeners have different value than object.

Problems with thread-based concurrency

- *Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.*

☞ Search for non-thread-based models: which are the requirements for appropriate specification techniques?

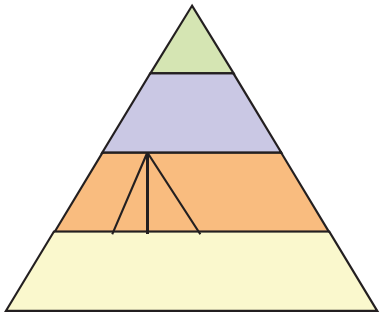
Contents

- ***Models of Computation***
- StateCharts
- Data-Flow Models

Requirements for Specification Techniques (1)

- ***Represent hierarchy***
  Humans not capable to understand systems containing more than a few objects.

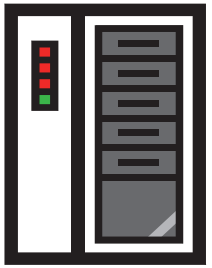
  Most actual systems require more objects
  ☞ Hierarchy



  - ***Behavioral hierarchy***
    Examples: states, processes, procedures.

```
proc
  proc
    proc
```

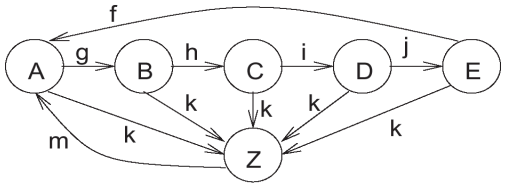
  - ***Structural hierarchy***
    Examples: processors, racks, printed circuit boards



Requirements for Specification Techniques (2)

- ***Represent timing behavior/requirements***

- ***Represent state-oriented behavior***
  Required for reactive systems.

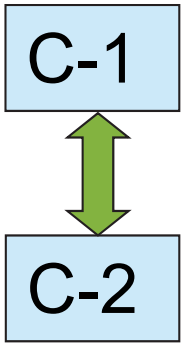


- ***Represent dataflow-oriented behavior***
  Components send streams of data
  to each other.

- **No obstacles for *efficient implementation***

Models of Computation: Definition

- **What does it mean, "to compute"?**
- **Models of computation define**:
  - **Components** and an execution model for computations for each component
  - **Communication** model for exchange of information between components.
    - Shared memory
    - Message passing
    - …



Communication: Shared memory

- Potential race conditions (☞inconsistent results possible)
  ☞ Critical sections = sections at which exclusive access to resource *r* (e.g. shared memory) must be guaranteed.

```
process a {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

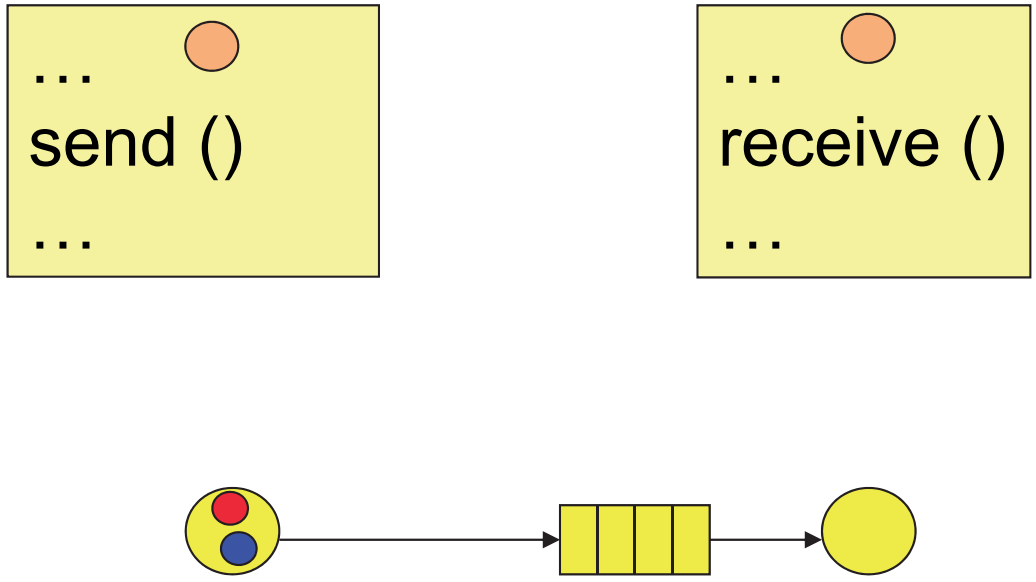
```
process b {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

race-free access to shared memory protected by S

This model may be supported by cache coherency protocols in order to efficiently implement polling of the shared lock-variable.

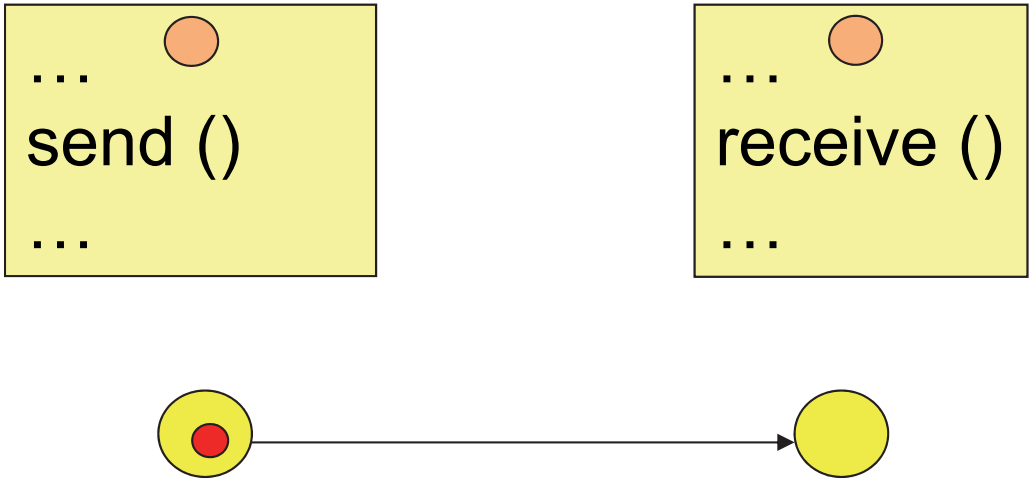
Non-blocking/asynchronous message passing

- Sender does not have to wait until message has arrived; potential problem: buffer overflow (will be discussed later)



Blocking/synchronous message passing

- Sender will wait until receiver has received message



Synchronous message passing: CSP

- **CSP** (communicating sequential processes) [Hoare, 1985], *rendez-vous*-based communication.

Example:

```
process A
..
var a ...
  a:=3;
  c!a; -- output
end
```

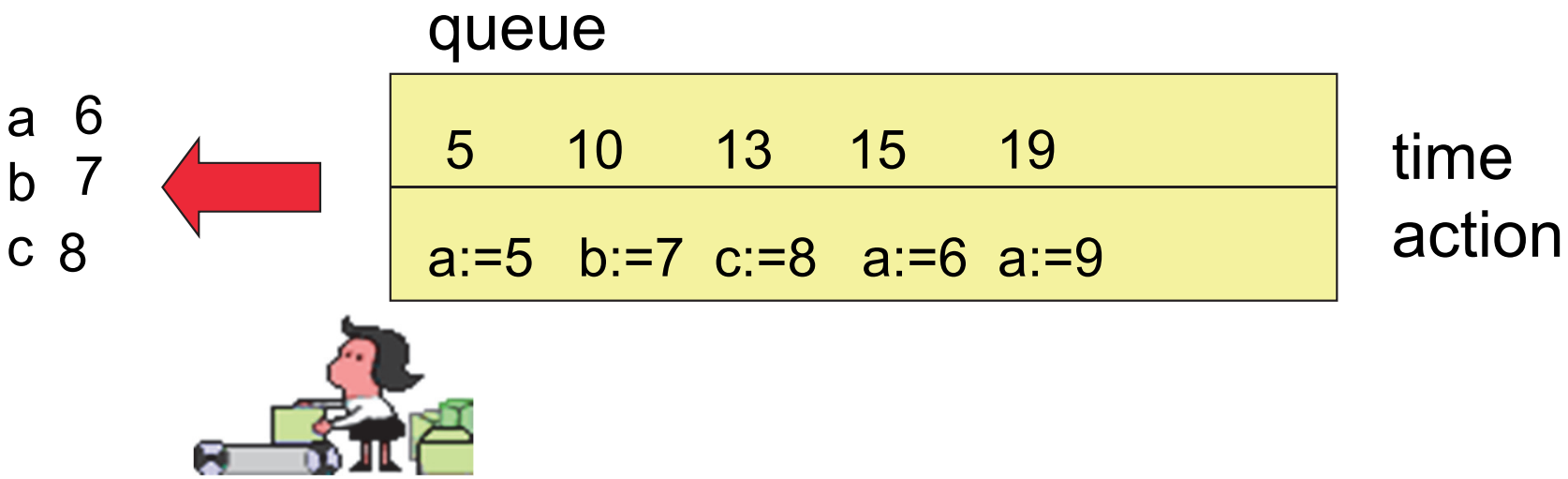
```
process B
..
var b ...
  ...
  c?b; -- input
end
```

Components (1)

- Von Neumann model

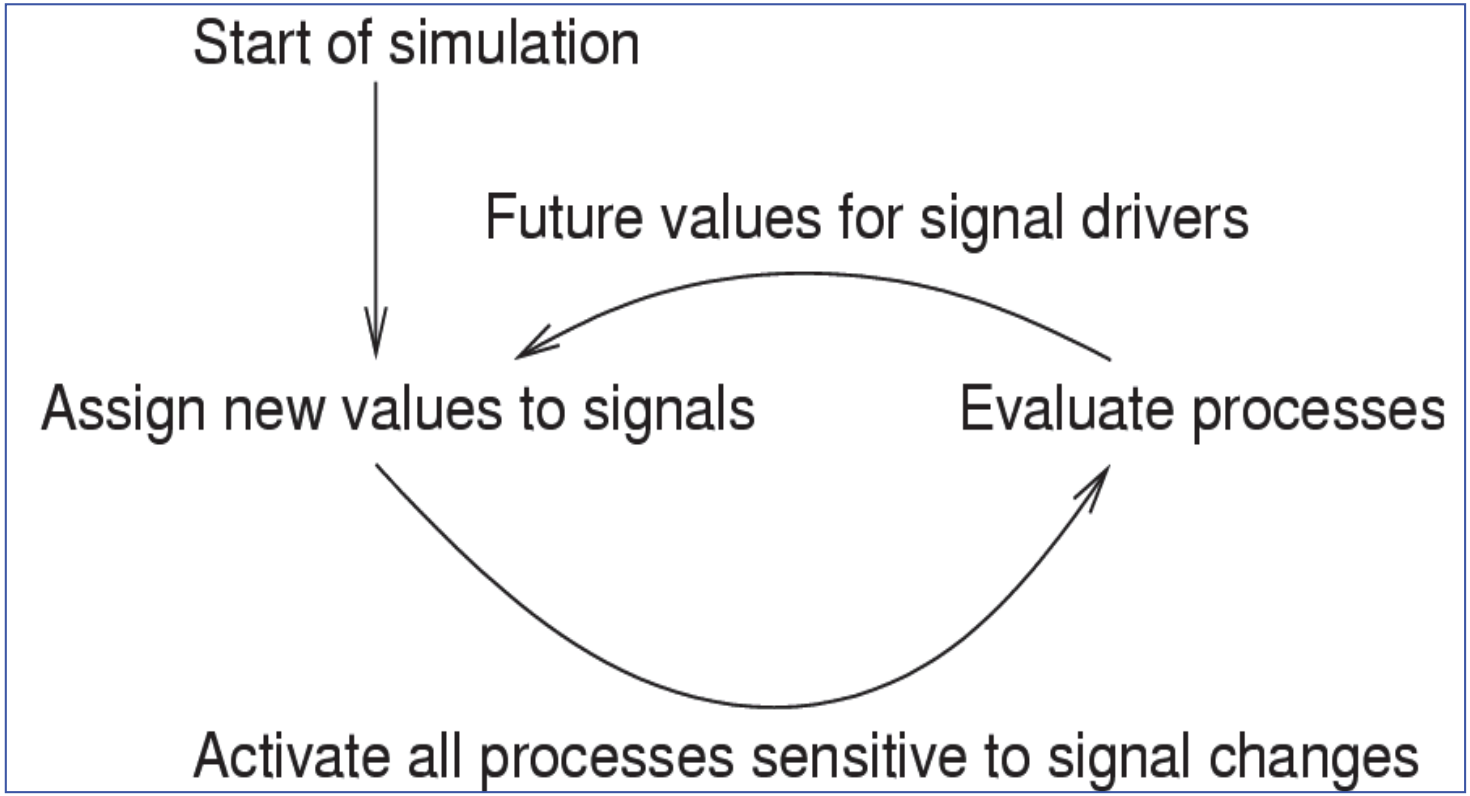
  Sequential execution, program memory etc.

- Discrete event model



Example Discrete Event: VHDL

- **VHDL** (hardware description language) is commonly used as a design-entry language for digital circuits (will be discussed later in more detail).



Sensitivity lists in VHDL

- Sensitivity lists are a shorthand for a single **wait on**-statement at the end of the process body:

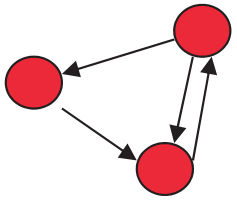
```
process (x, y)
    begin
        prod <= x  and y ;
    end process;
```

is equivalent to

```
process
    begin
        wait on x,y;
        prod <= x  and y ;
    end process;
```

Components (2)

- Finite state machines



- Differential equations

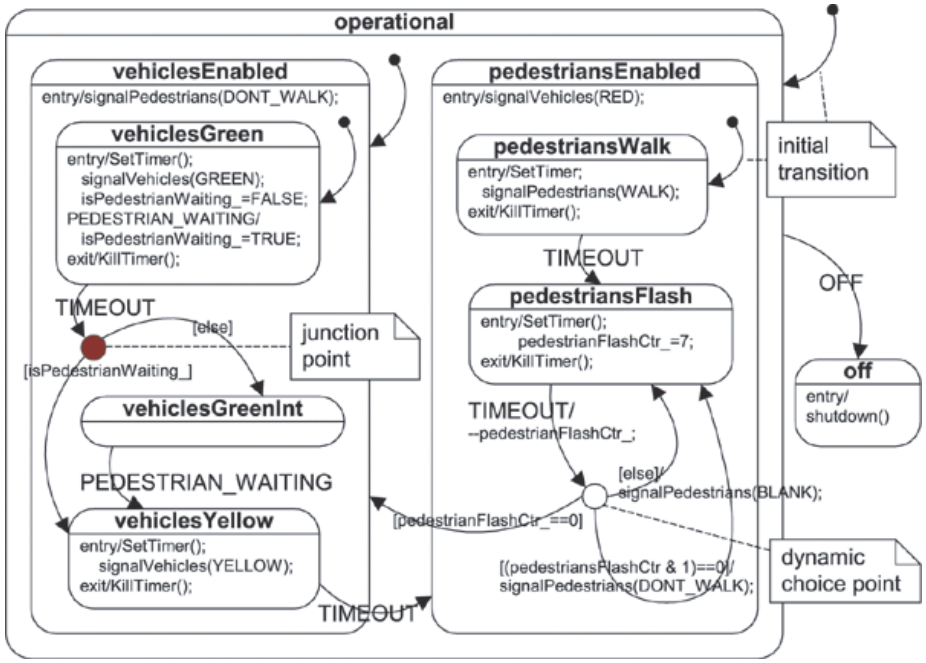
$$\frac{\partial^2 x}{\partial t^2} = b$$

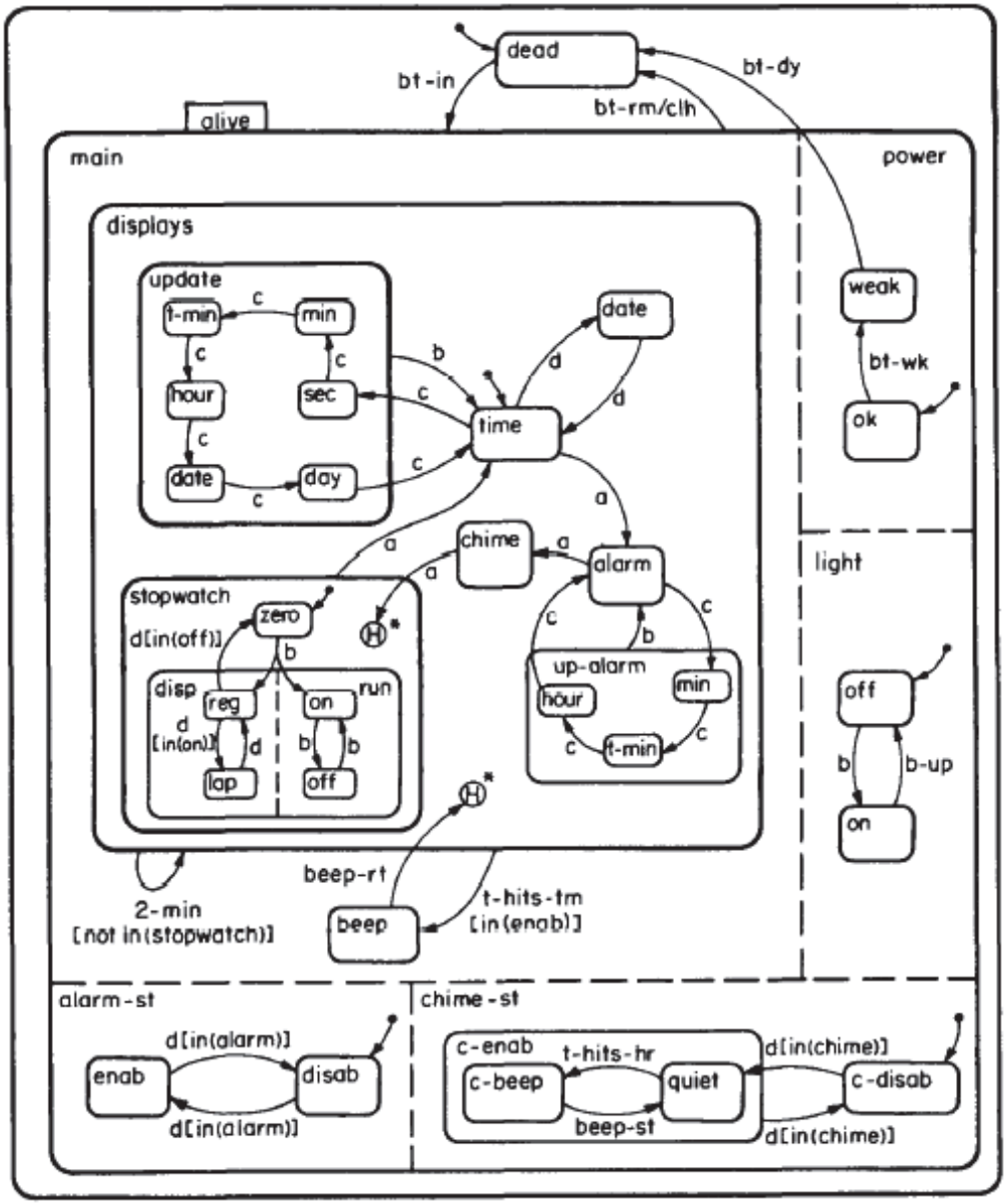
No language that meets all language requirements

☞ using compromises

Contents

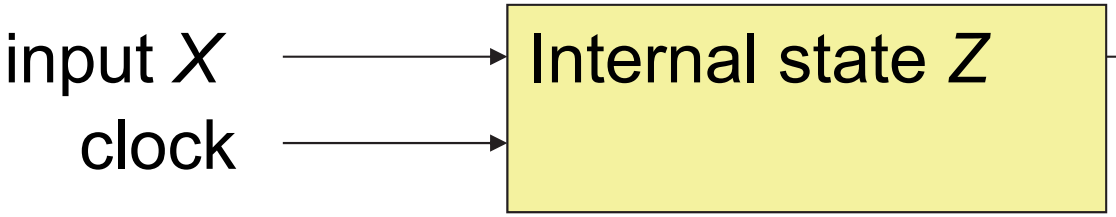
- Models of Computation
- *StateCharts*
- Data-Flow Models





Classical Automata

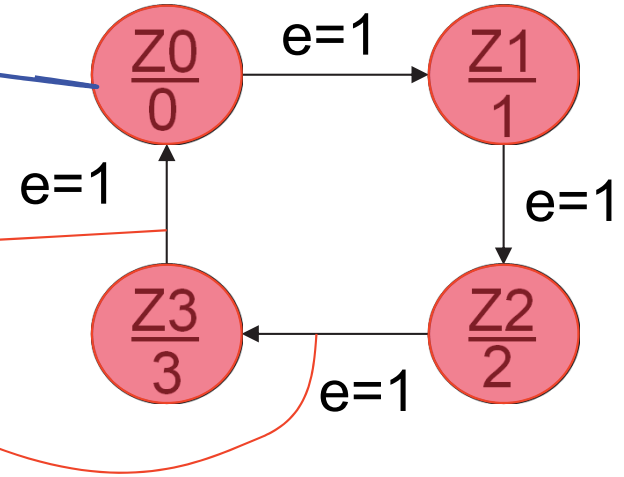
Classical automata:



Next state Z^+ computed by function δ
Output computed by function λ

Moore- + Mealy automata=finite state machines (FSMs)

- Moore-automata:
  $Y = \lambda(Z)$; $Z^+ = \delta(X, Z)$
- Mealy-automata
  $Y = \lambda(X, Z)$; $Z^+ = \delta(X, Z)$

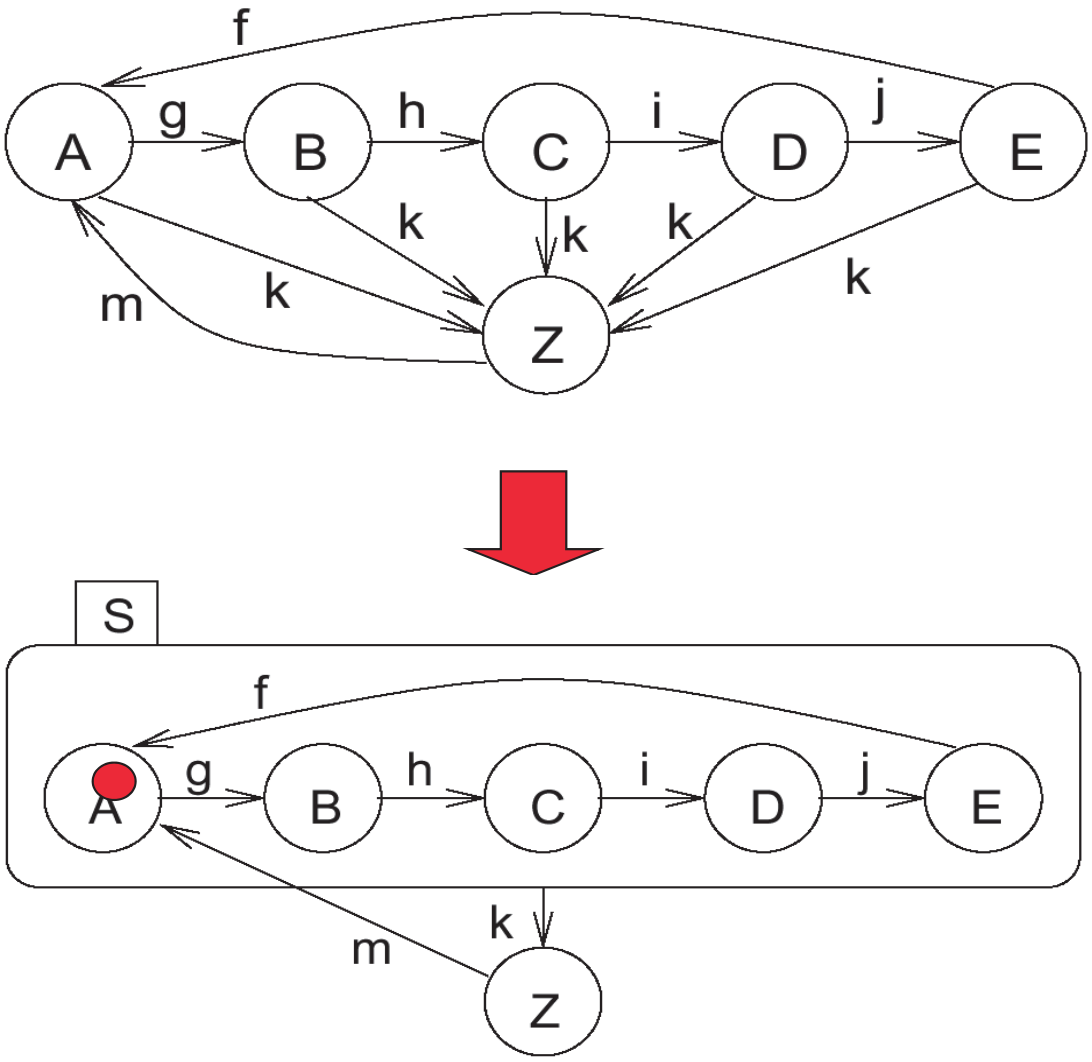


StateCharts

Classical automata not useful for complex systems (complex graphs cannot be understood by humans).

☞ ***Introduction of hierarchy*** ☞ StateCharts [Harel, 1987]

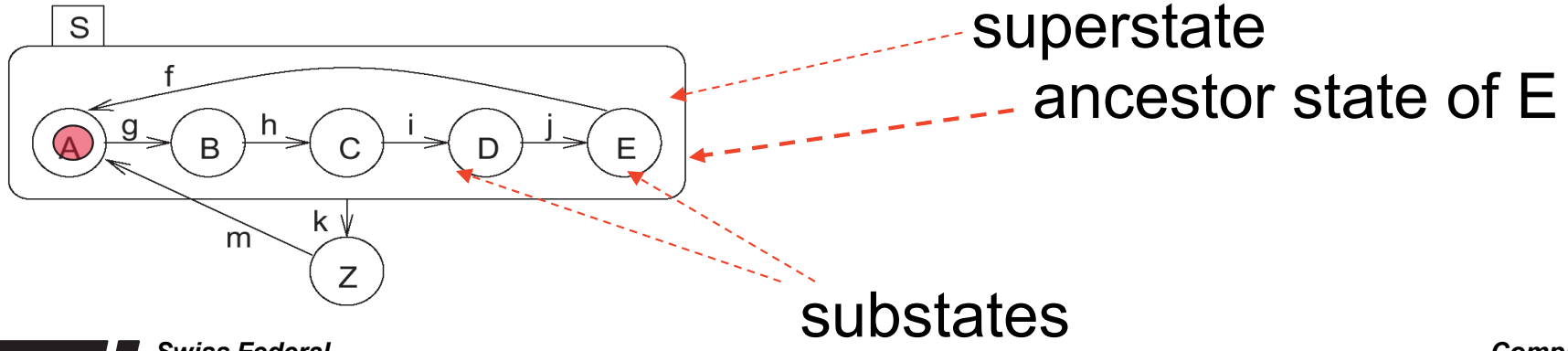
Introducing Hierarchy



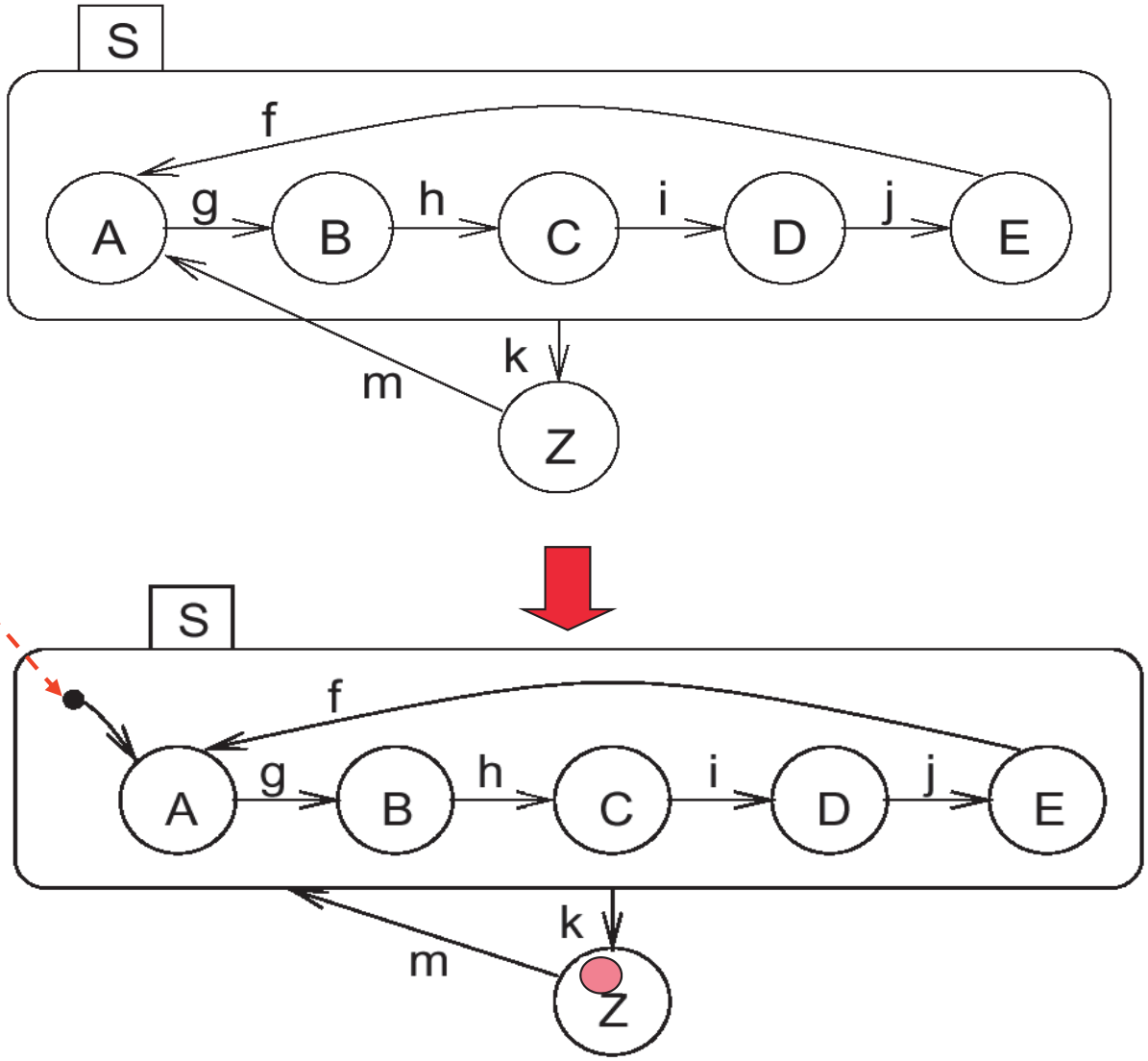
FSM will be **in** exactly one of the substates of S if S is **active**
(either in A or in B or ..)

Definitions

- Current states of FSMs are also called ***active*** states.
- States which are not composed of other states are called ***basic states***.
- States containing other states are called ***super-states***.
- For each basic state s, the super-states containing s are called ***ancestor states***.
- Super-states S are called ***OR-super-states***, if exactly one of the sub-states of S is active whenever S is active.



Default State Mechanism

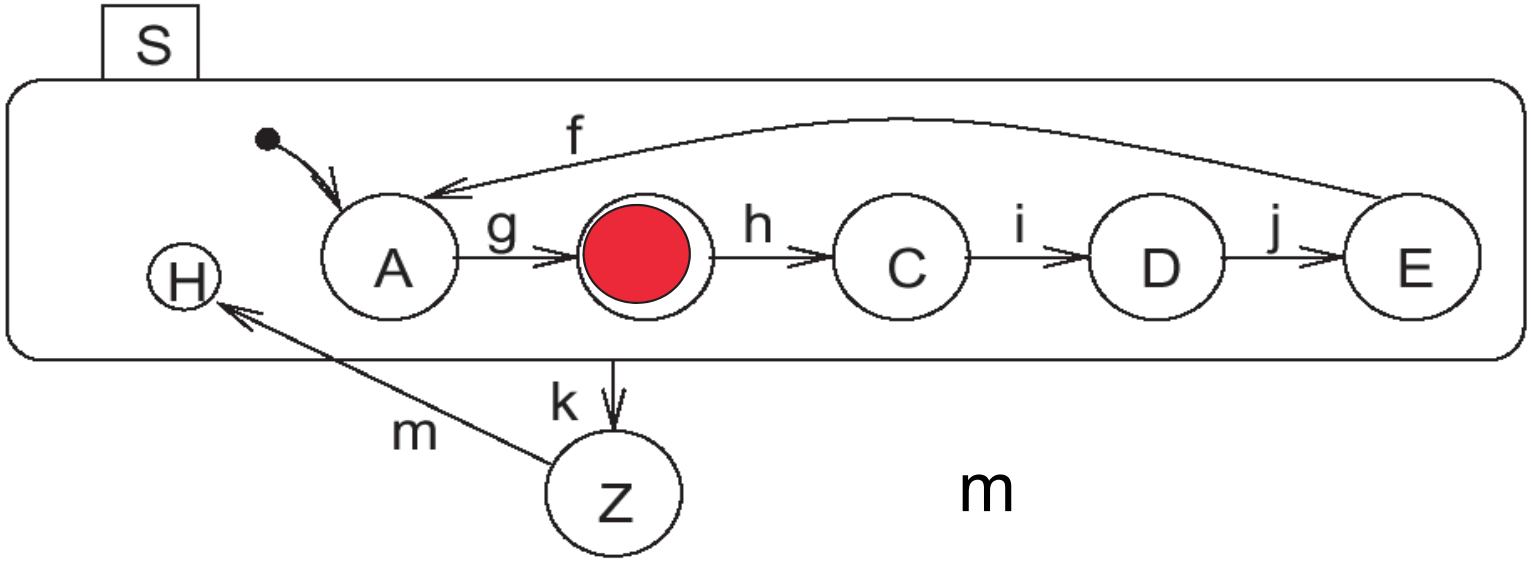


☞ Default state

Filled circle indicates sub-state entered whenever super-state is entered.

Not a state by itself!

History Mechanism

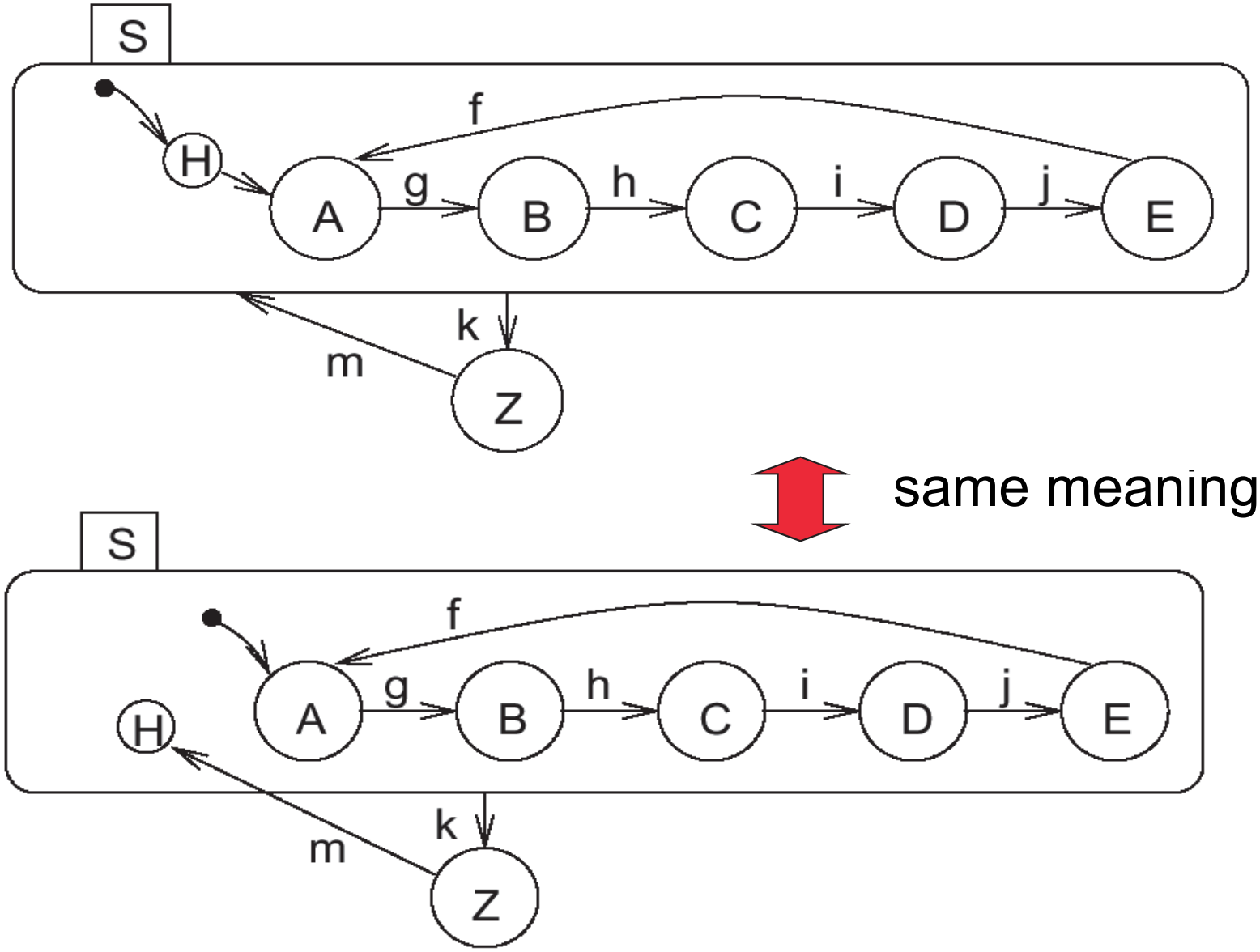


m

(behavior different from last slide)

For input m, S enters the state it was in before S was left (can be A, B, C, D, or E). If S is entered for the very first time, the default mechanism applies.

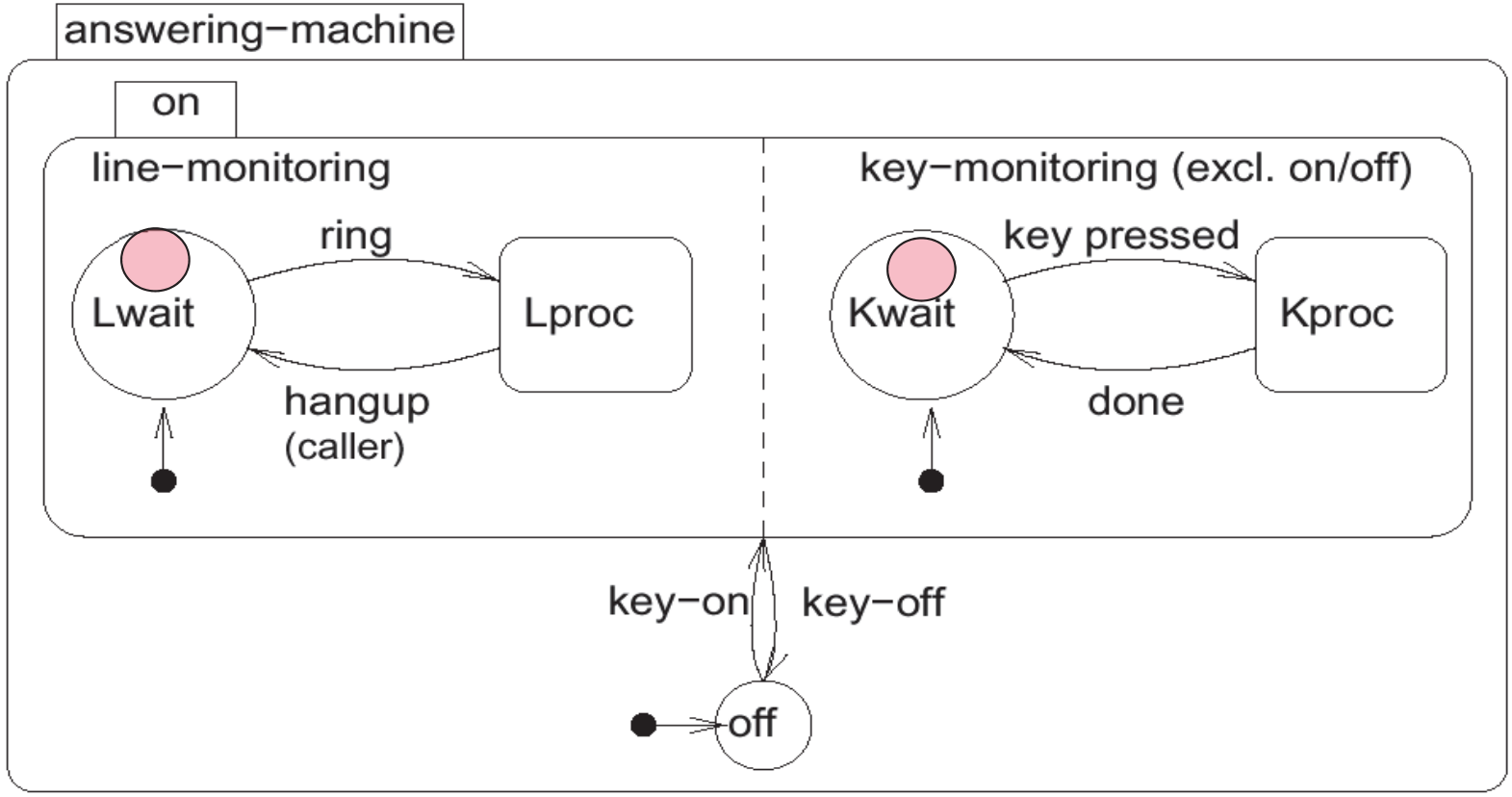
Combining History and Default State



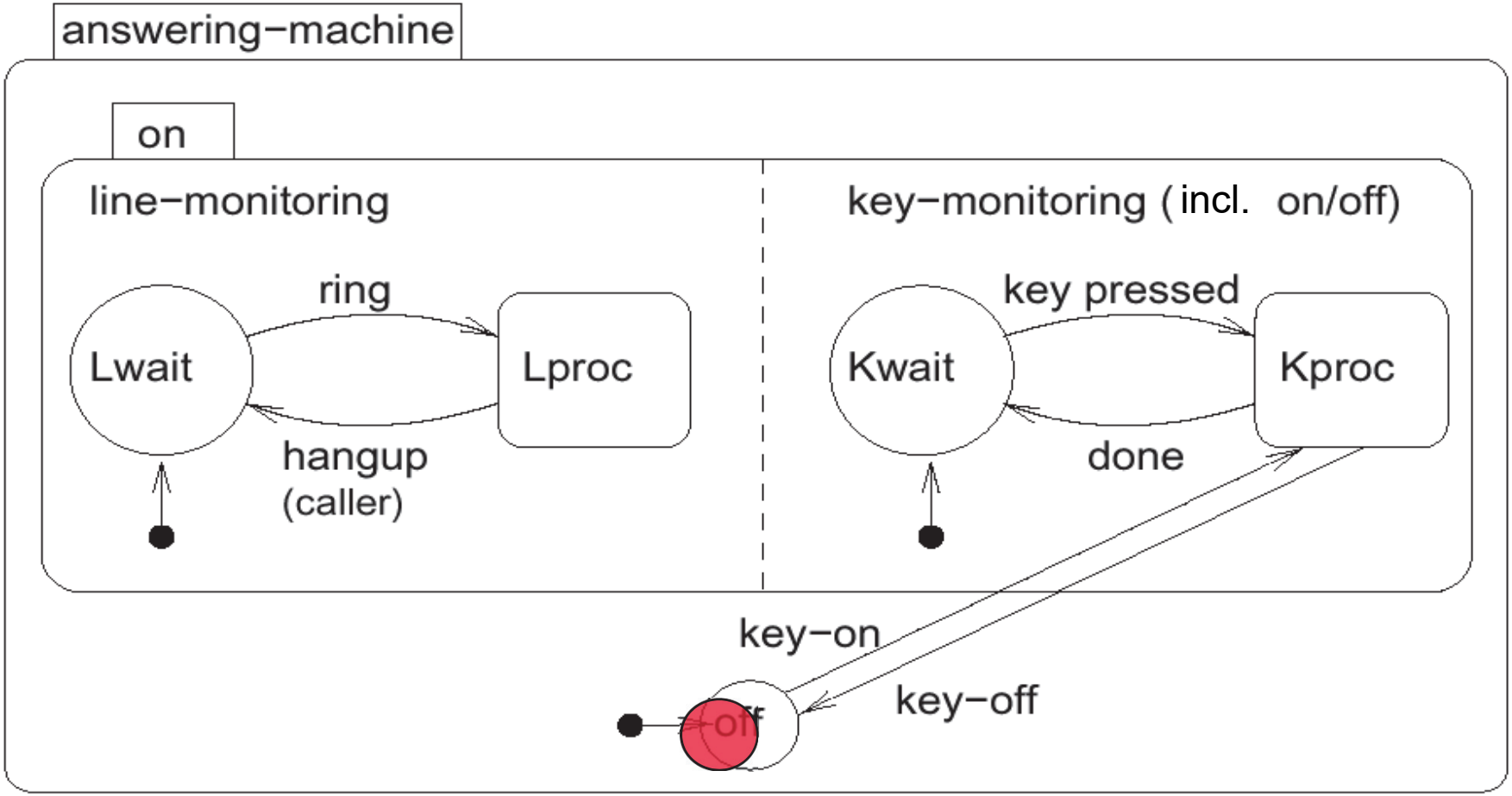
Concurrency

Convenient ways of describing concurrency are required.

AND-super-states: FSM is in **all** (immediate) sub-states of a super-state.

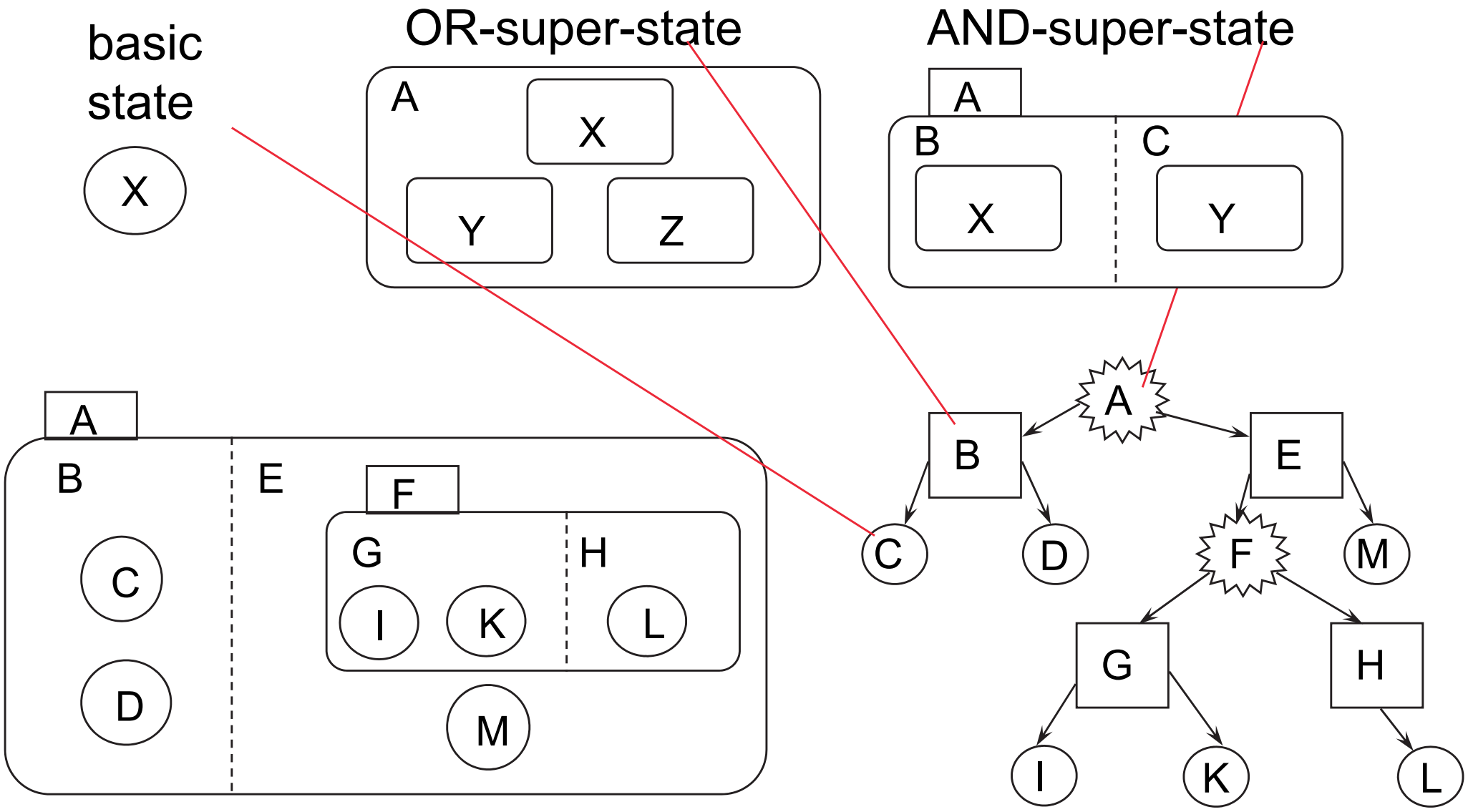


Entering and Leaving AND-Super-States



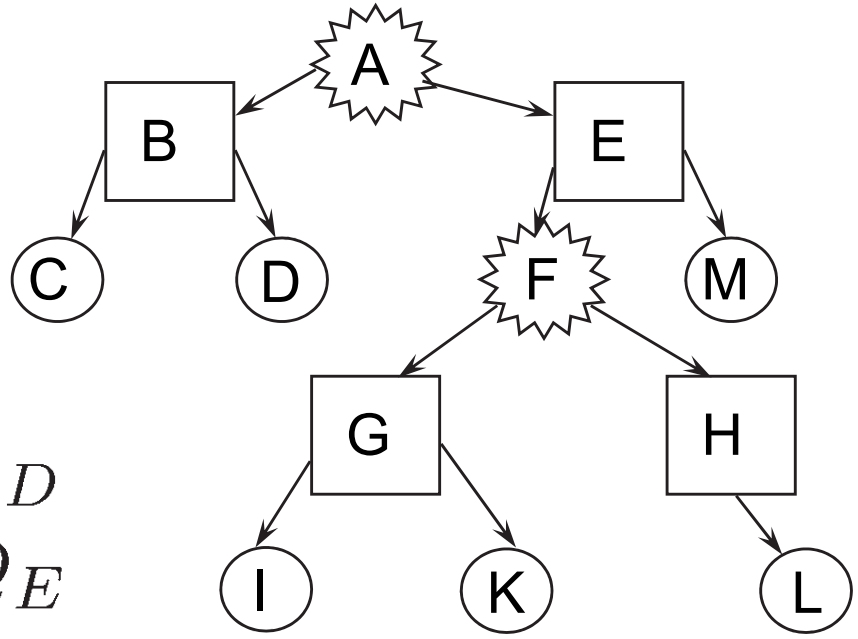
Line-monitoring and key-monitoring are entered and left, when service switch is operated.

Tree representation of state sets



Computation of state sets

- Computation of state sets by ***traversing the tree*** from leaves to root:
  - basic states: state set = state
  - OR-super-states: state set = union of children
  - AND-super-states: state set = Cartesian product of children



$$Q_H = Q_L,\ Q_G = Q_I \cup Q_K$$
$$Q_F = Q_G \times Q_H,\ Q_B = Q_C \cup Q_D$$
$$Q_E = Q_F \cup Q_M,\ Q_A = Q_B \times Q_E$$
$$Q_A = (Q_C \cup Q_D) \times (Q_M \cup ((Q_I \cup Q_K) \times Q_L))$$

Types of States

In StateCharts, states are either

***Basic states*, or**

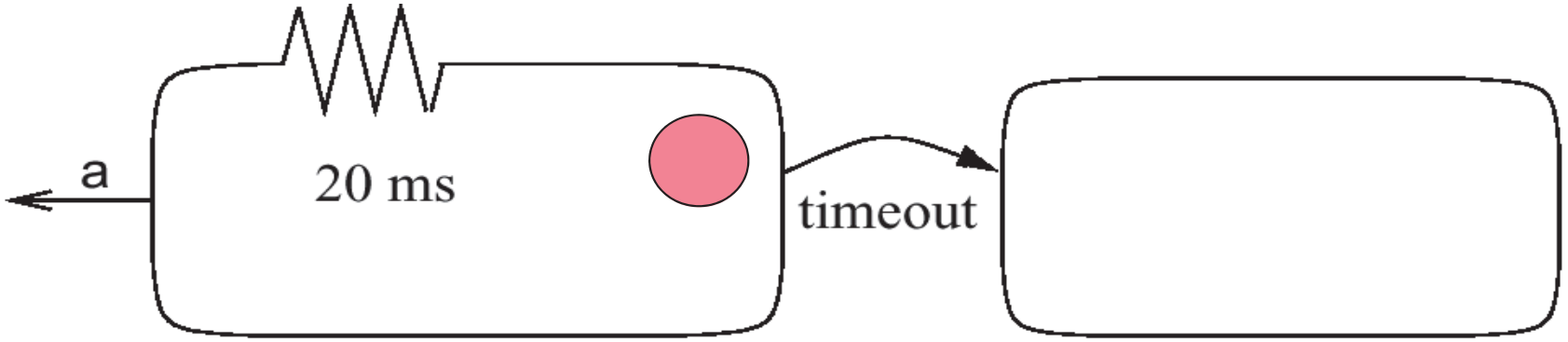
***AND-super-states*, or**

***OR-super-states*.**

Timers

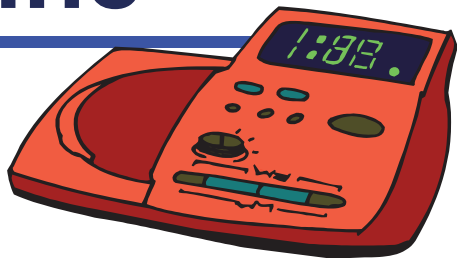
Since time needs to be modeled in embedded systems, timers need to be modeled.

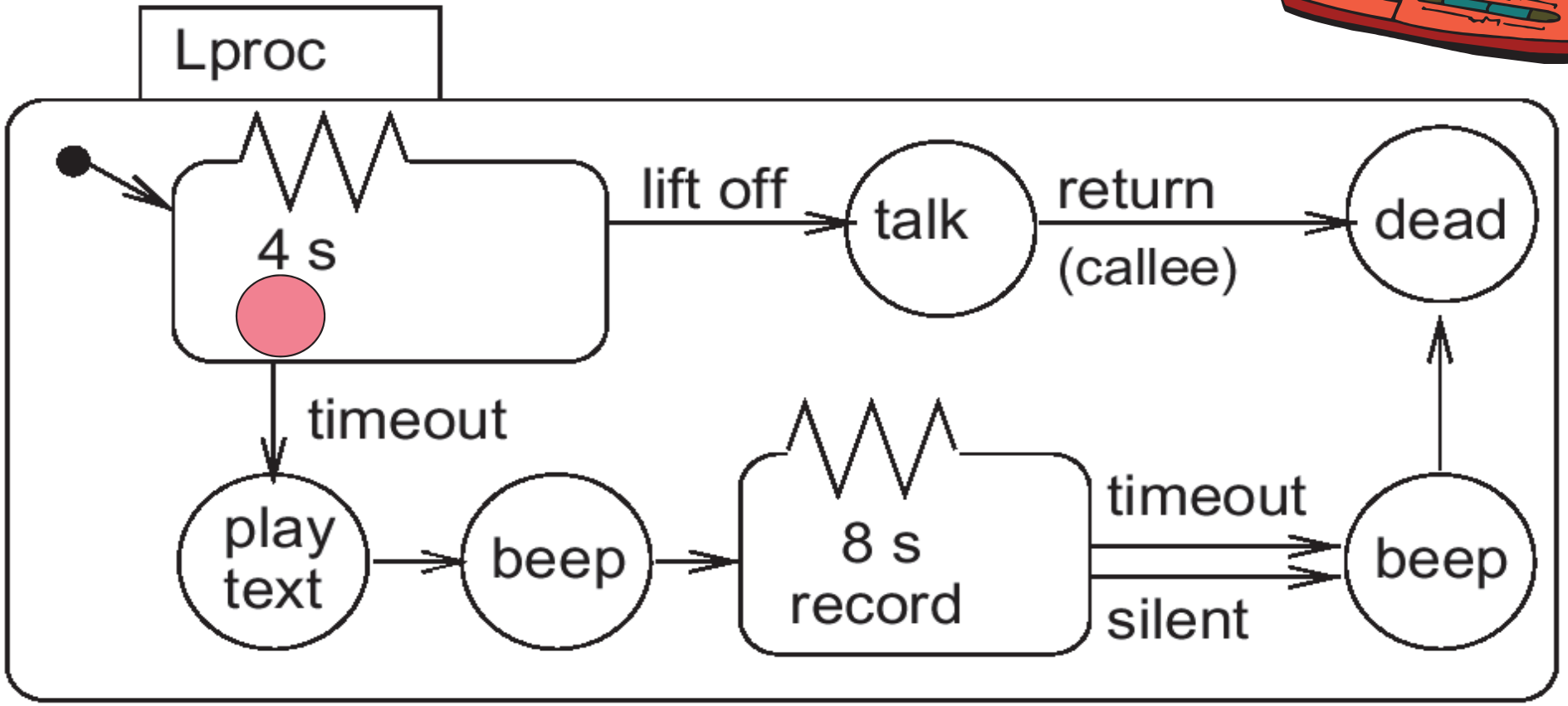
In StateCharts, special edges can be used for timeouts.



If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

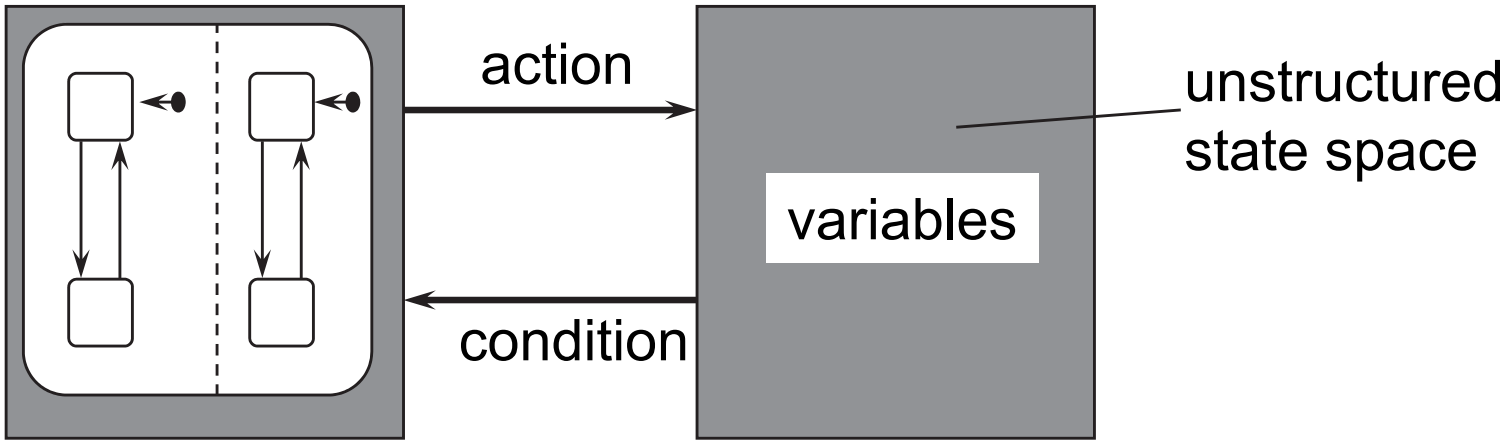
Using Timers in Answering Machine





Representation of computations

- Besides states, arbitrary many other variables can be defined. This way, not all states of the system are modeled explicitly.
- These variables can be changed as a result of a state transition ("***action***"). State transitions can be dependent on these variables ("***condition***").



General Form of Edge Labels

Events:

Exist only for the next evaluation of the model

Can be either internally or externally generated

Conditions:

Refer to values of variables that keep their value until **they are reassigned**

Actions:

Can either be assignments for variables or creation of events

Example:

service-off [a <= 7] / service:=0

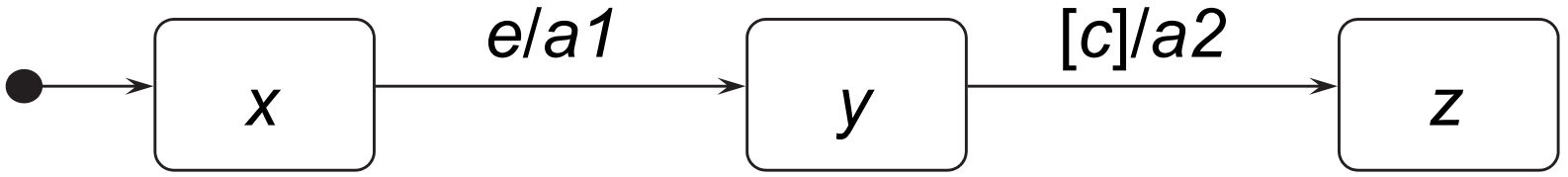
Events and actions

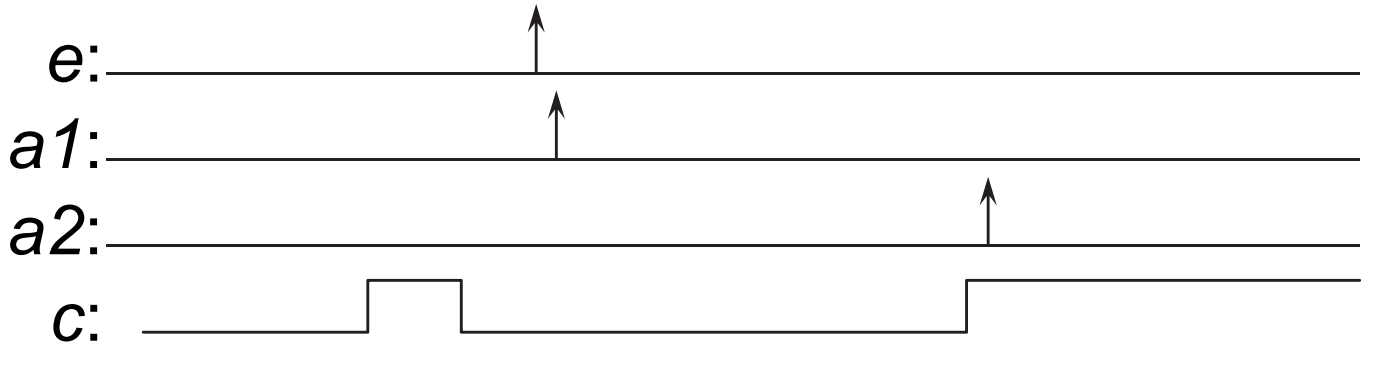
- "***event***" can be composed of several events:
  - **(*e1* and *e2*)** : event that corresponds to the simultaneous occurrence of e1 and e2.
  - **(*e1* or *e2*)** : event that corresponds to the occurrence of either e1 or e2, or both.
  - **(not *e*)** : event that corresponds to the absence of event e.

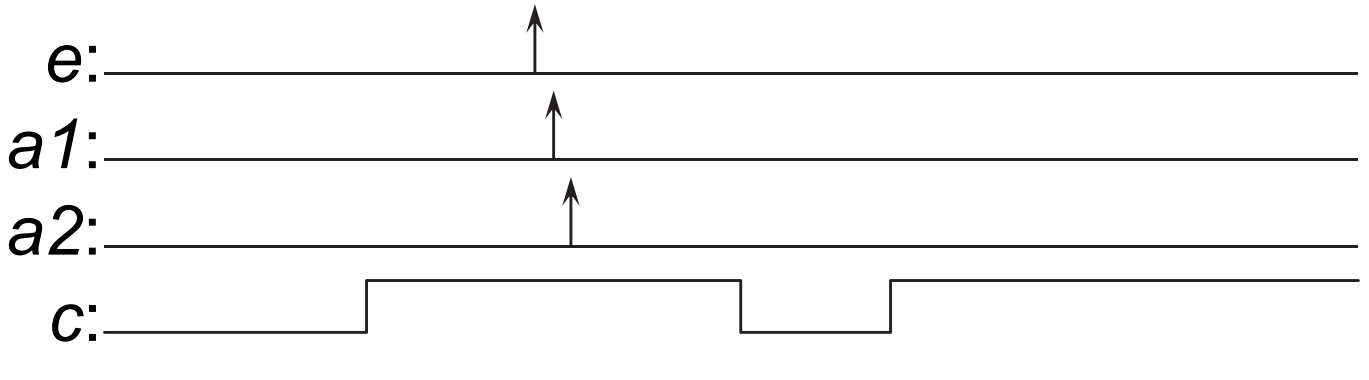
- „***action***" can also be composed:
  - **(*a1*; *a2*)** : actions a1 und a2 are executed in parallel.

- All events, states and actions are globally visible.

Example







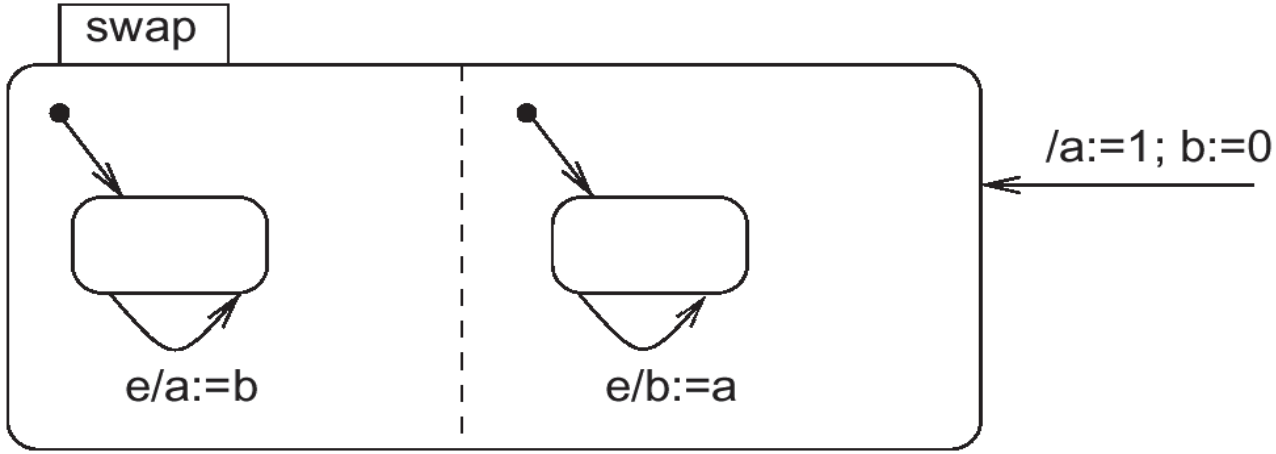
The StateCharts Simulation Phases

How are edge labels evaluated?

Three phases:

1. Effect of external changes on events and conditions is evaluated,
2. the set of transitions to be made in the current step and right hand sides of assignments are computed,
3. transitions become effective, variables obtain new values.

Example

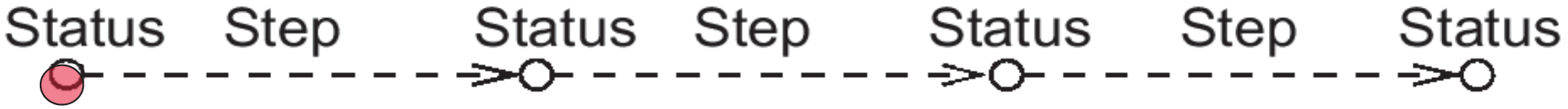


In phase 2, variables a and b are assigned to temporary variables. In phase 3, these are assigned to a and b. As a result, variables a and b are swapped.

In a single phase environment, executing the left state first would assign the old value of b (=0) to a and b. Executing the right state first would assign the old value of a (=1) to a and b. The execution would be non-deterministic.

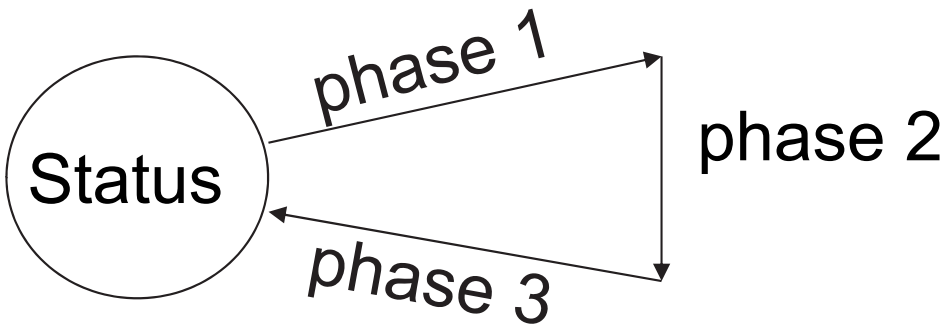
Steps

Execution of a StateChart model consists of a sequence of (status, step) pairs

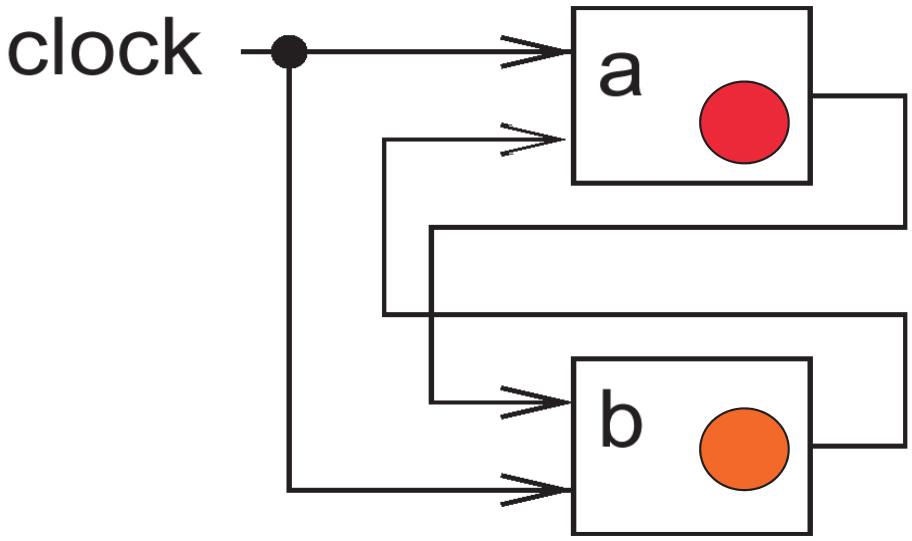


Status= values of all variables + set of events + current time

Step = execution of the three phases



Reflects Model of Clocked Hardware

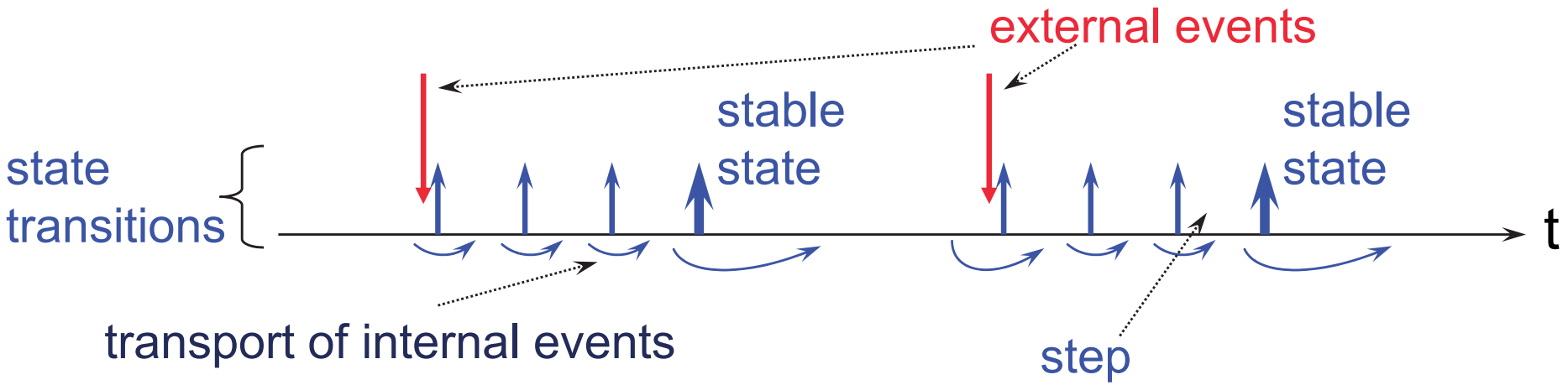


In an actual clocked (synchronous) hardware system, both registers would be swapped as well.

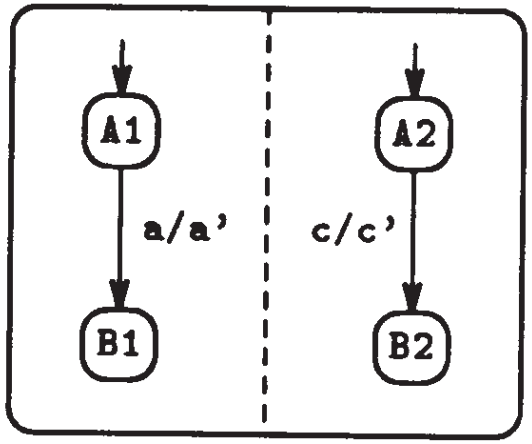
Same separation into phases found in other languages as well, especially those that are intended to model hardware.

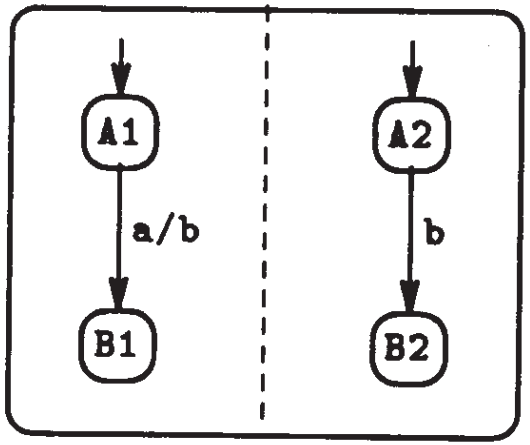
More on semantics of StateCharts

- Unfortunately, there are several time-semantics of StateCharts in use. This is another possibility:
  - A step is executed in arbitrarily small time.
  - Internal (generated) events exist only within the next step.
  - *Difference: External events can only be detected after a stable state has been reached. In the other model, events are detected after each step.*



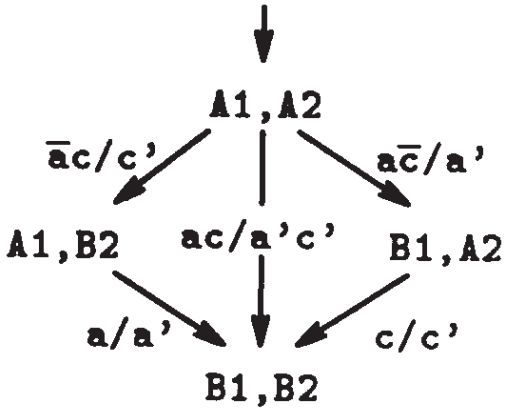
Examples

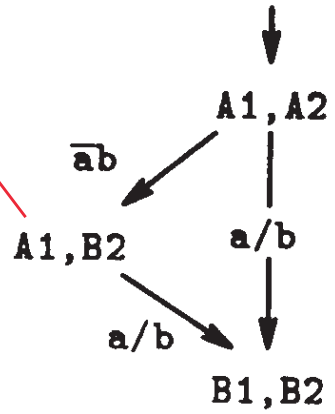




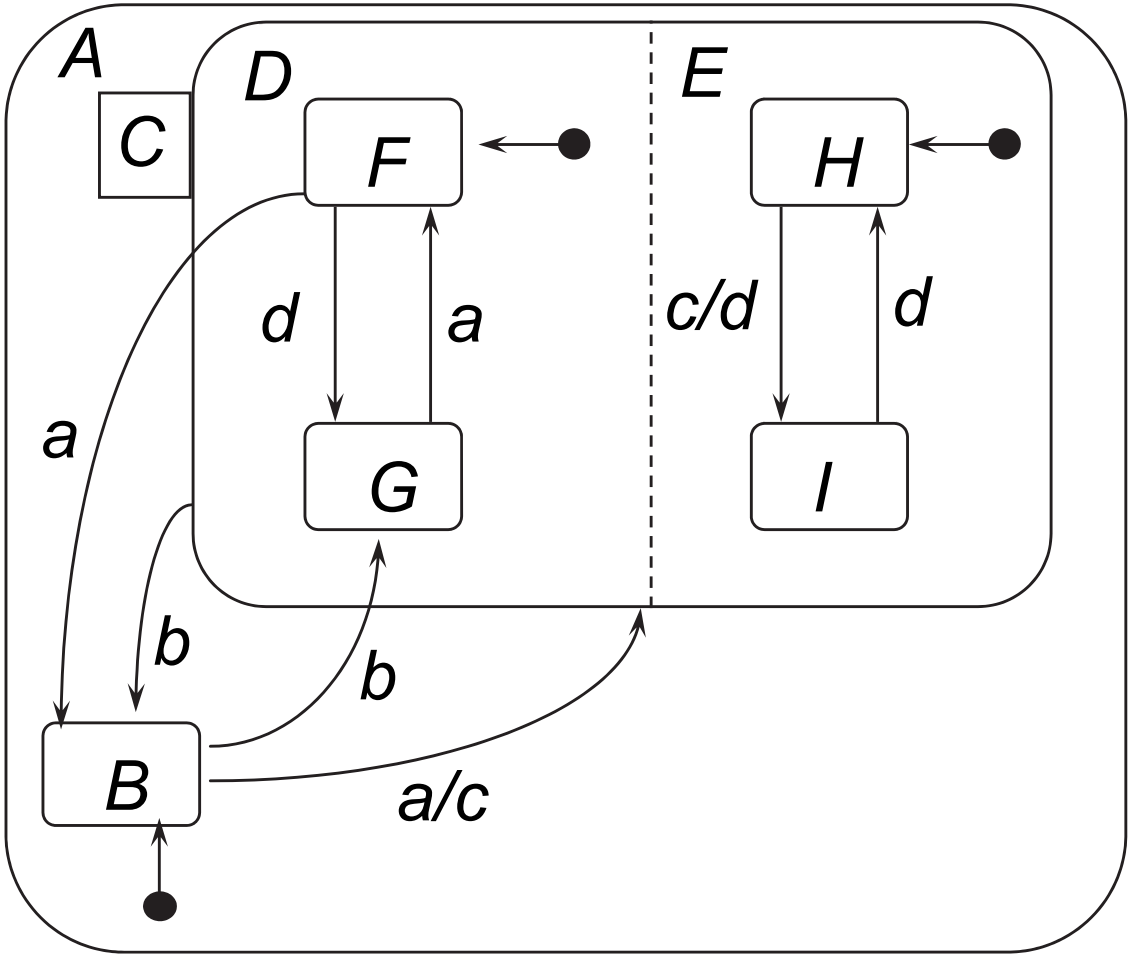
state diagram:

stable states

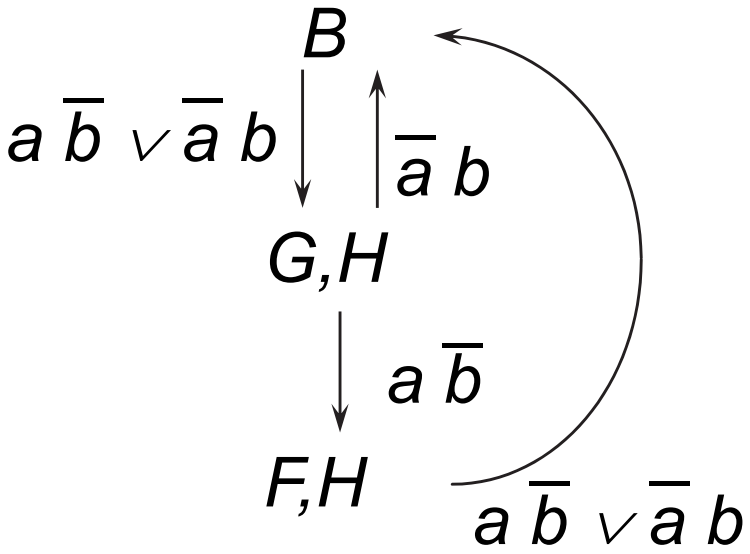




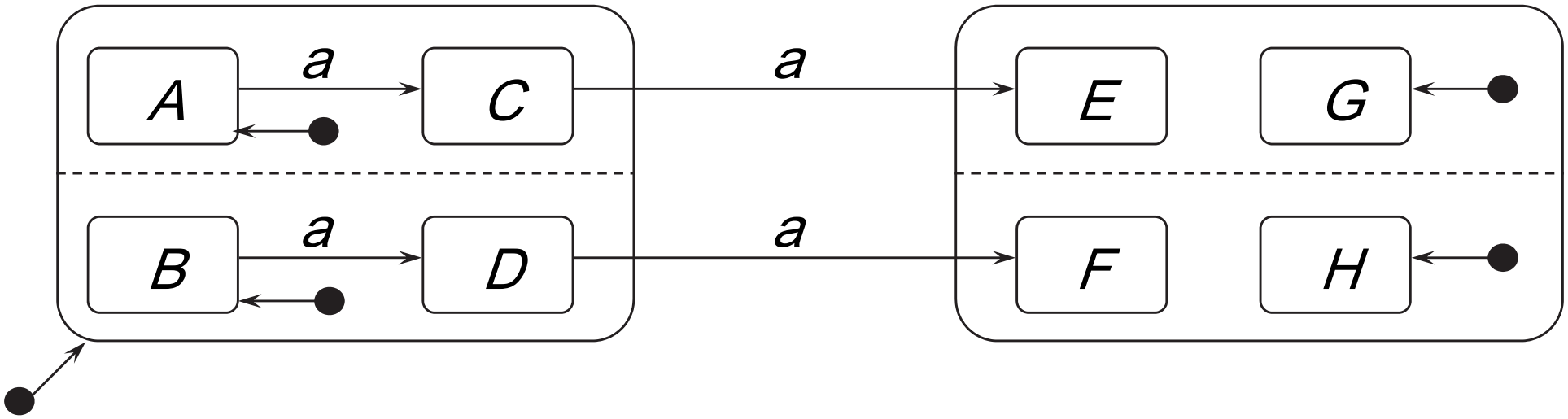
Example



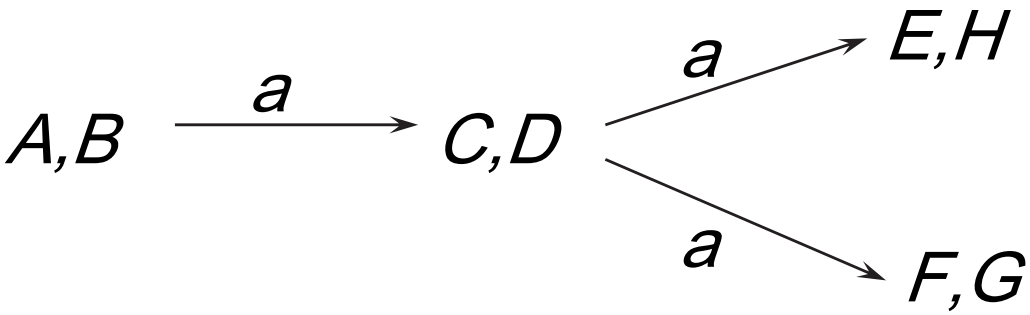
state diagram (only stable states are represented, only a and b are external):



Example



state diagram:



Evaluation of StateCharts (1)

- ***Pros***:
  - ***Hierarchy*** allows arbitrary nesting of AND- and OR-super states.
  - ***Semantics defined*** in a follow-up paper to original paper.
  - Large number of commercial simulation ***tools available*** (StateMate, StateFlow/Matlab, BetterState, UML, ...)
  - Available „back-ends“ translate StateCharts into ***C or VHDL***, thus enabling software or hardware implementations.

Evaluation of StateCharts (2)

- ***Cons***:
  - Generated C ***programs frequently inefficient***,
  - Not useful for ***distributed*** applications,
  - No ***object-orientation***,
  - No description of ***structural hierarchy***.

SDL

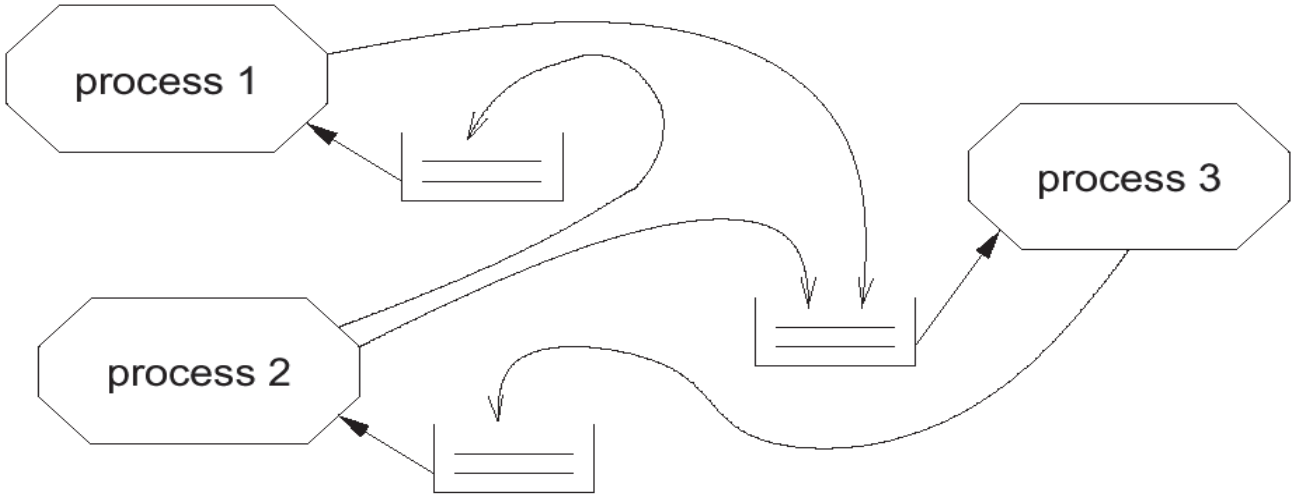
- ***Specification and Description Language*** (SDL) is a specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems.

- Used here as a (prominent) example of a model of computation based on **asynchronous message passing**.

- Appropriate also for distributed systems

Communication among SDL-FSMs

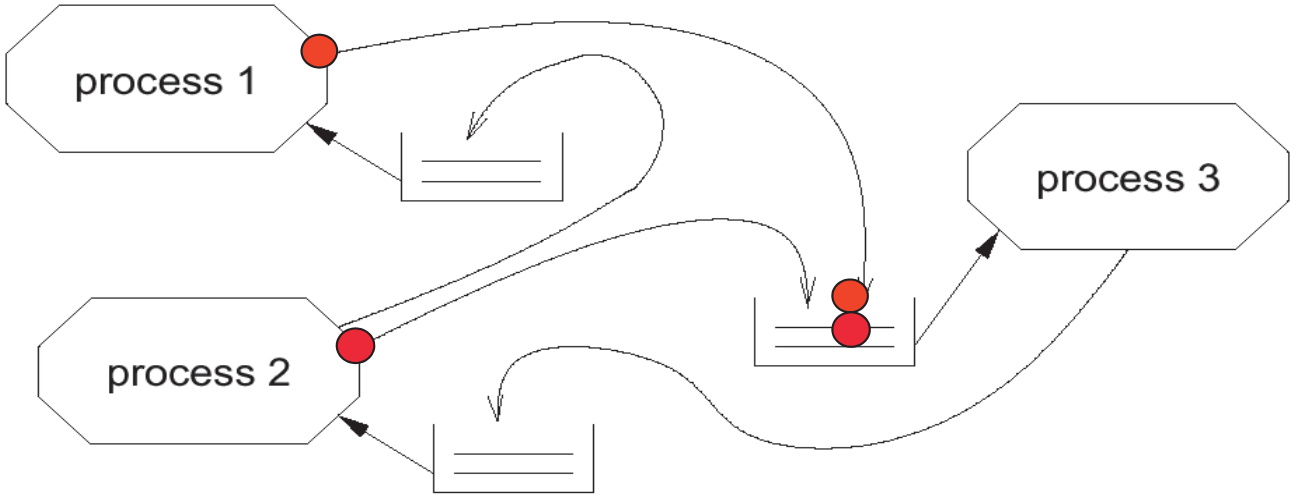
- Communication between FSMs (or “processes“) is based on **message-passing**, assuming a **potentially indefinitely large FIFO-queue**.



- Each process fetches next entry from FIFO,
- checks if input enables transition,
- if yes: transition takes place,
- if no: input is discarded (exception: SAVE-mechanism).

Deterministic?

- If tokens arrive at FIFO at the same time, the order in which they are stored, is unknown:



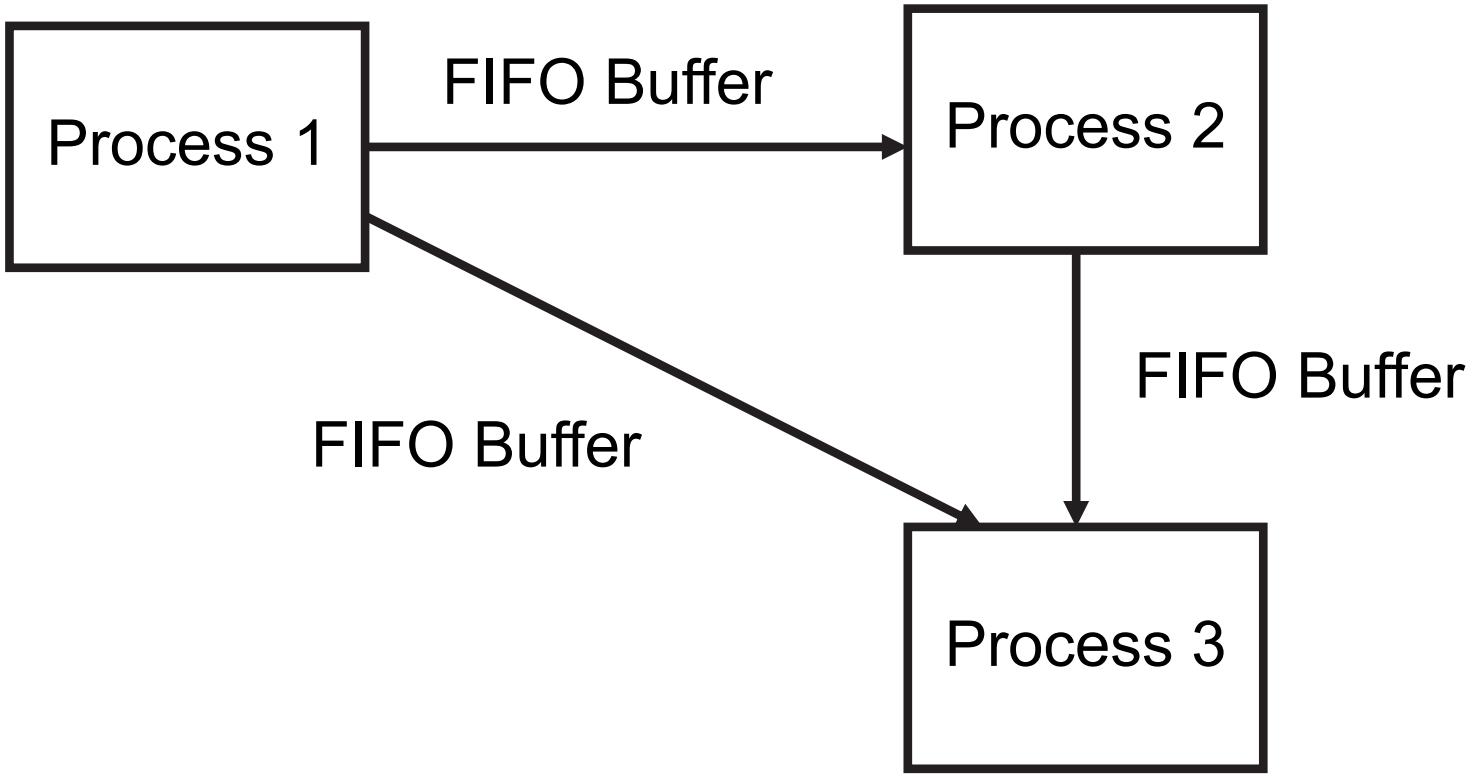
  - All orders are legal: simulators can show different behaviors for the same input, all of which are correct.
  - Behavior in an actual implementation may deviate from simulation.

Contents

- Models of Computation
- StateCharts
- ***Data-Flow Models***

Dataflow Language Model

- ***Processes*** communicating through ***FIFO buffers***



Philosophy of Dataflow Languages

- ***Drastically different way of looking at computation***:
  - Dataflow language: movement of data is the priority
  - Scheduling responsibility of the system, not the programmer

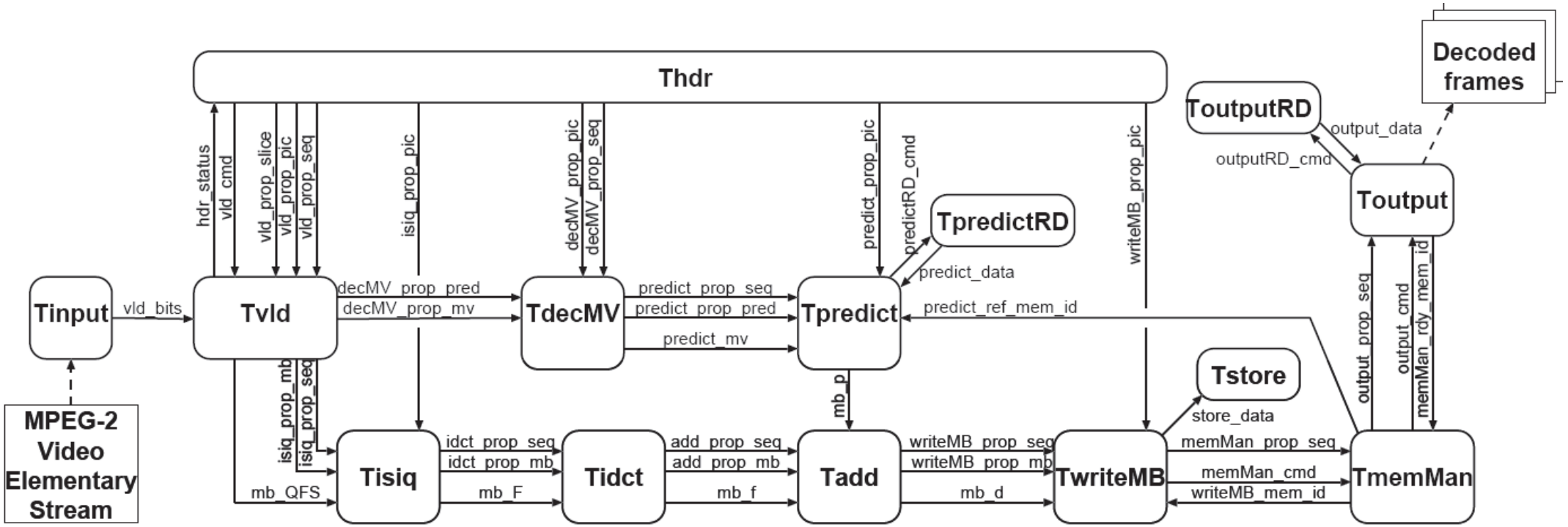
- ***Basic characteristic***:
  - All processes execute concurrently
  - Processes can be described with imperative code
  - Processes can *only* communicate through buffers
  - The buffers have FIFO (first in first out) semantics: The sequence of read tokens is identical to the sequence of written tokens

Dataflow Languages

- Appropriate for applications that deal with ***streams of data***:
  - Fundamentally concurrent: maps easily to parallel hardware
  - Perfect fit for block-diagram specifications (control systems, signal processing)
  - Matches well current and future trend towards multimedia applications

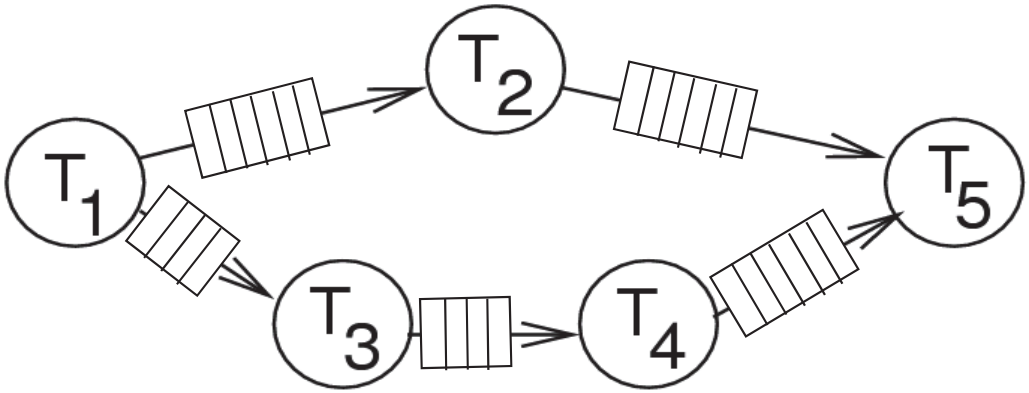
- ***Representation***:
  - Host Language (process description), e.g. C, C++, Java, ...
  - Coordination Language (network description), usually ‘home made’, for example XML.

Example: MPEG-2 video decoder



An MPEG-2 video decoder application structured as a Kahn Process Network.

Kahn Process Networks

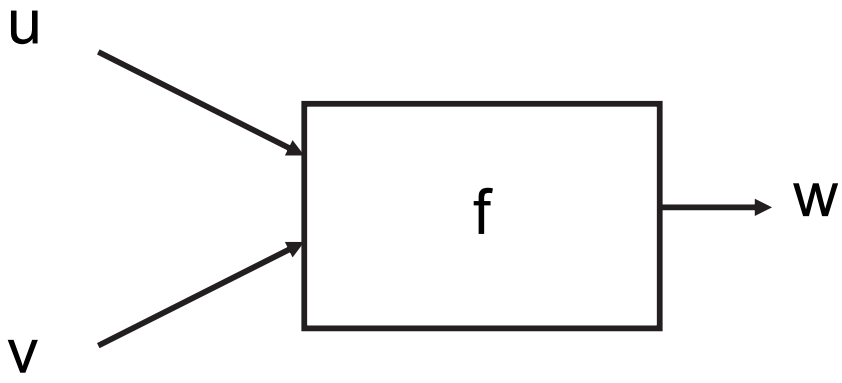


- Proposed by Kahn in 1974 as a general-purpose scheme for parallel programming:
  - ***read***: destructive and blocking (reading an empty channel blocks until data is available)
  - ***write***: non-blocking
  - ***FIFO***: infinite size
- Unique attribute: ***determinate***

A Kahn Process

- From Kahn's original 1974 paper

```
process f(in int u, in int v, out int w)
{
  int i; bool b = true;
  for (;;) {
    i = b ? wait(u) : wait(v);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}
```



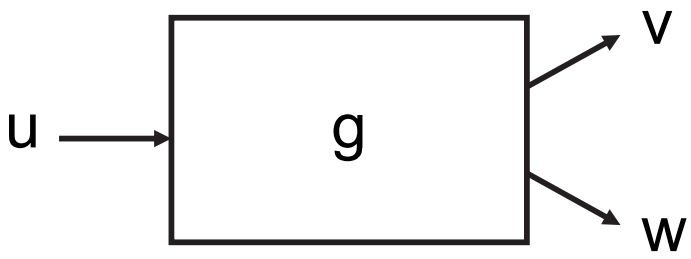
What does this do?

Process alternately reads from u and v, prints the data value, and writes it to w

A Kahn Process

- From Kahn's original 1974 paper:

```
process g(in int u, out int v, out int w)
{
  int i; bool b = true;
  for(;;) {
    i = wait(u);
    if (b) send(i, v); else send(i, w);
    b = !b;
  }
}
```



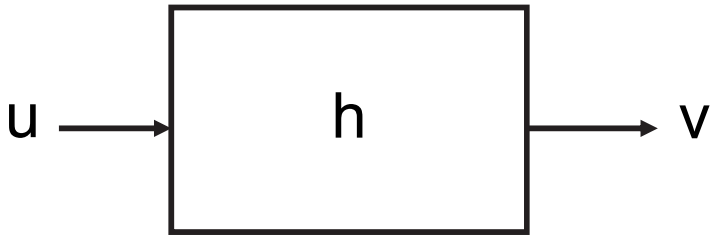
What does this do?

Process reads from u and alternately copies it to v and w

A Kahn Process

- From Kahn's original 1974 paper:

```
process h(in int u, out int v, int init)
{
  int i = init;
  send(i, v);
  for(;;) {
    i = wait(u);
    send(i, v);
  }
}
```



What does this do?

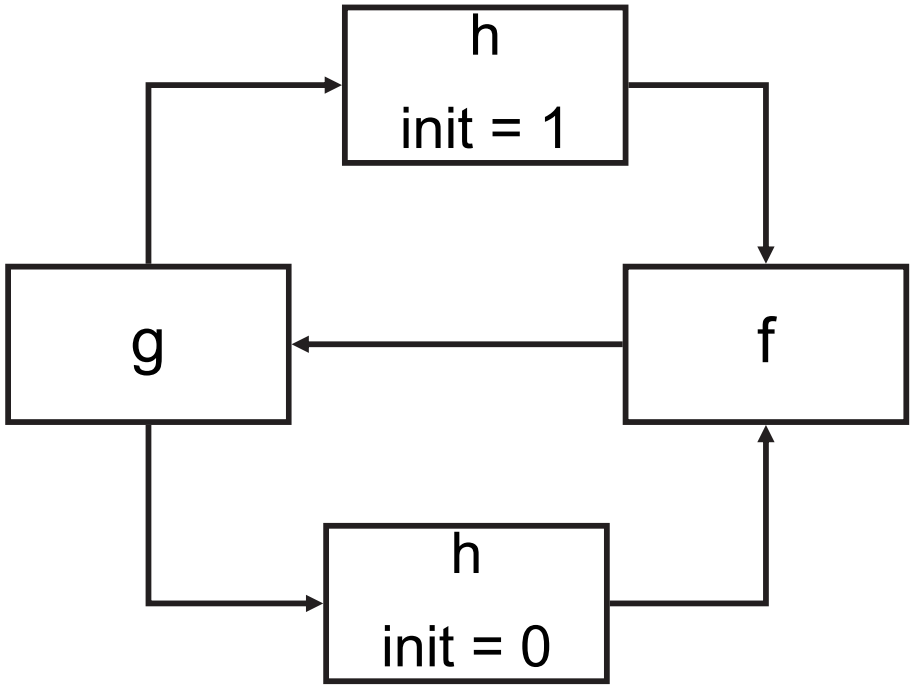
Process sends initial value, then passes through values.

A Kahn Process Network

- What does this do?

  Prints an alternating sequence of 0's and 1's.

Emits a 1 once and then copies input to output

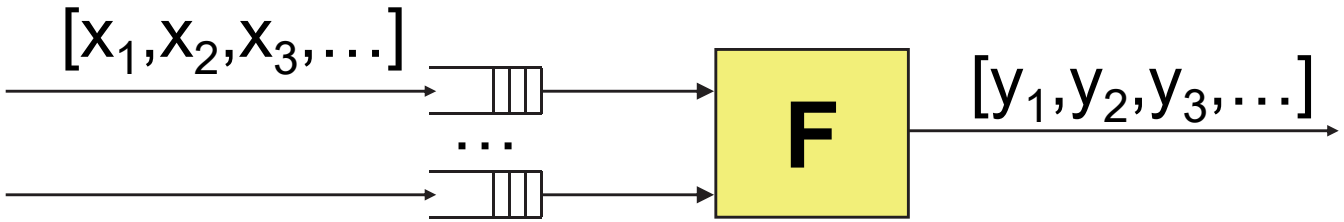


Emits a 0 once and then copies input to output

Determinacy

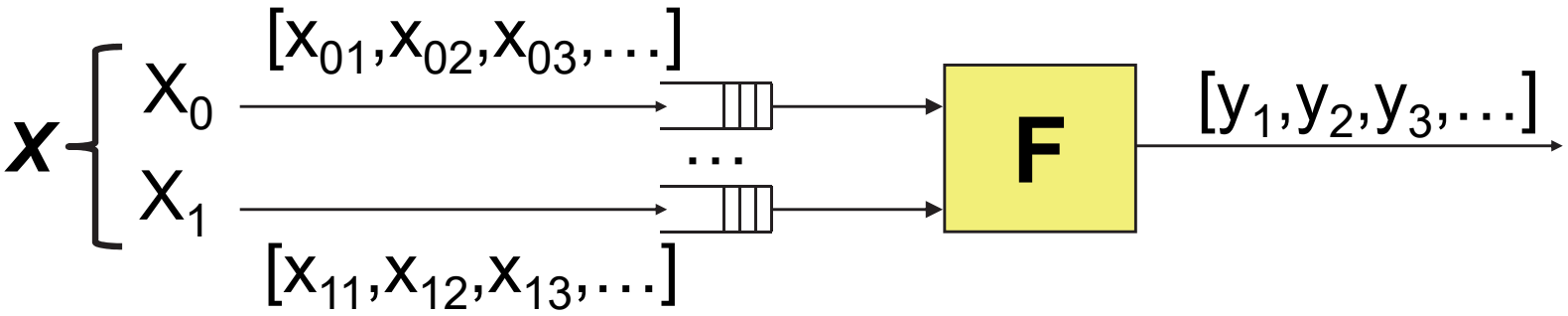
- ***Random***:
  - A system is random if the information ***known*** about the system and its inputs is not sufficient to determine its outputs.
- ***Determinate***:
  - Define the *history* of a channel to be the complete ***sequence*** of tokens that have been both written and read. A process network is said to be *determinate* if the histories of all channels depend ***only*** on the histories of the input channels.
- ***Importance***:
  - Functional behavior is independent of timing (scheduling, communication time, execution time of processes).
  - Separation of functional properties and timing.

Determinacy



- Kahn Process: "***monotonic mapping***" of input sequence to output sequences:
  - Process uses prefix of input sequences to produce prefix of output sequences. In other words, let us consider a process with two inputs and two different scenarios: The second scenario has just additional input tokens in comparison to the first one. Then the output sequence of the second scenario equals that of the first one, but with zero or more additional tokens appended.
  - Process will not wait forever before producing an output (i.e., it will not wait for completion of an infinite input sequence).

Determinacy



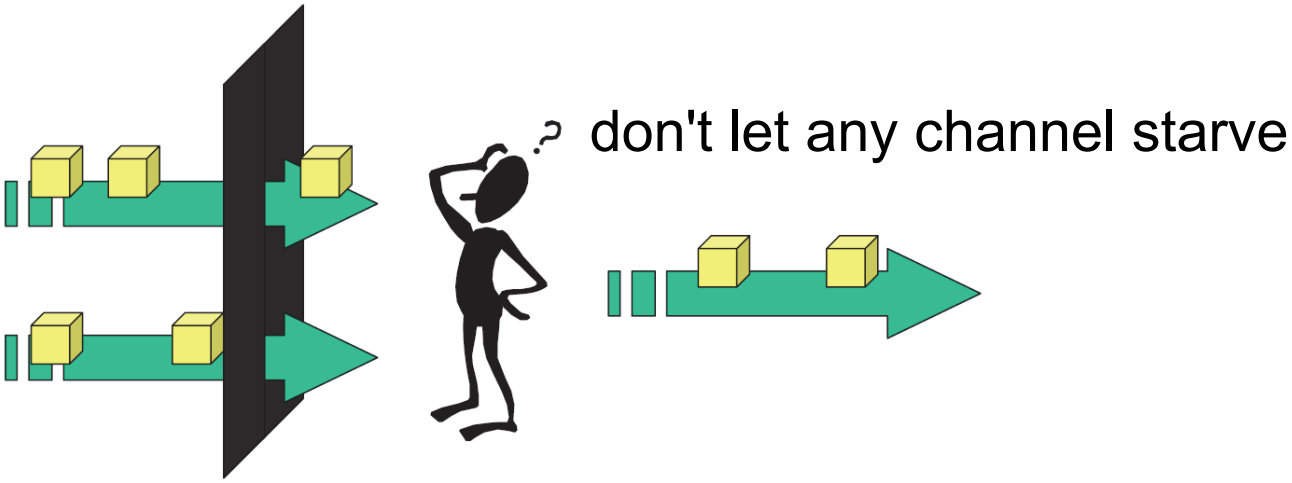
- ***Formal definition***:
  - sequence (stream): $X = [x_1, x_2, x_3, ...]$
  - prefix ordering: $[x_1] \subseteq [x_1, x_2] \subseteq [x_1, x_2, x_3, ...]$
  - p-tuple of sequences: $\boldsymbol{X} = (X_0, X_1, ... , X_p) \in S^p$
  - ordered set of sequences: $\boldsymbol{X} \subseteq \boldsymbol{X}'$ if $(\forall\ i : X_i \subseteq X_i')$
  - process: $\mathbf{F}: S^p \rightarrow S^q$
  - monotonic process: $\boldsymbol{X} \subseteq \boldsymbol{X}' \Rightarrow \mathbf{F}(\boldsymbol{X}) \subseteq \mathbf{F}(\boldsymbol{X}')$

Proof of Determinacy

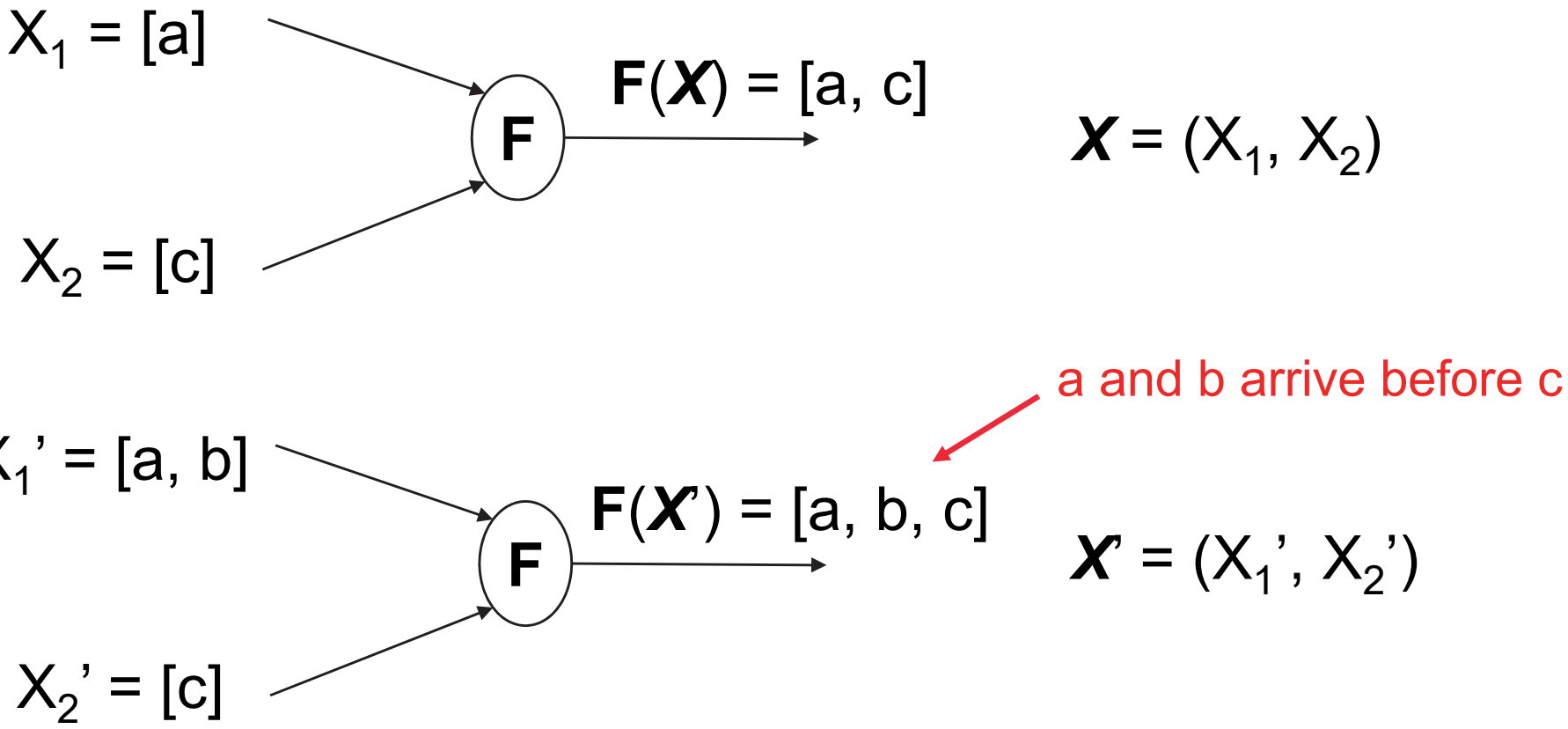
- ***Why*** is a Kahn Process Network (consisting of monotonic processes) ***determinate***?
  - A network of monotonic processes itself defines a monotonic process.
  - A monotonic process is clearly determinate.
- ***Reasoning***:
  - If I'm a process, I am only affected by the sequence of tokens on my inputs
  - I can't tell whether they arrive early, late, or in what order
  - I will behave the same in any case
  - Thus, the sequence of tokens I put on my outputs is the same regardless of the timing of the tokens on my inputs

Adding Non-determinacy

- There are several ways to introduce non-monotonic behavior, for example:
  - Allow processes to test for emptiness of input channels (buffers)
  - Allow more than one process to read from or to write to a single channel
  - Allow processes to share variables (data)
- ***Example*** (fair merge):



Adding Non-determinacy



$\boldsymbol{X} \subseteq \boldsymbol{X}'$ as ([a], [c]) $\subseteq$ ([a, b], [c])

$\mathbf{F}(\boldsymbol{X}) \not\subseteq \mathbf{F}(\boldsymbol{X}')$ as [a, c] $\not\subseteq$ [a, b, c]

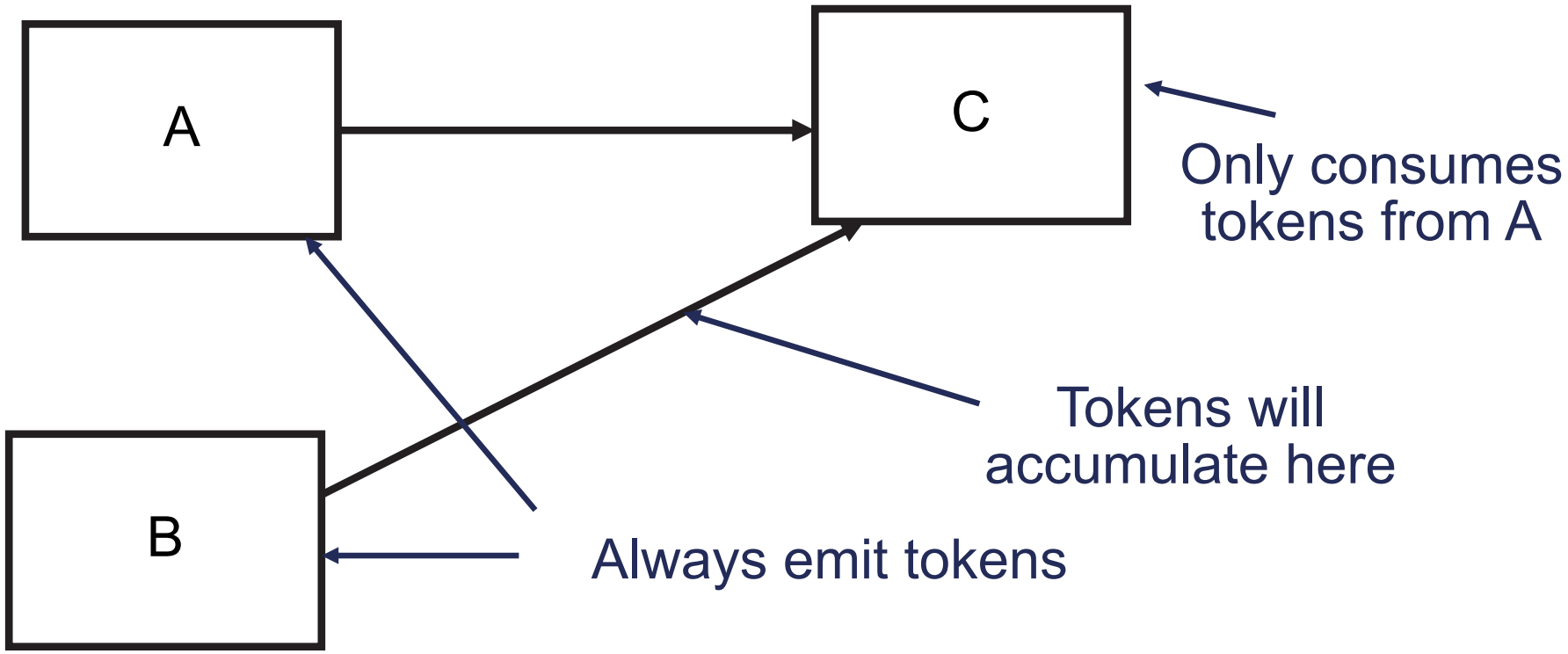
Philosophy of Dataflow Languages

- ***Drastically different way of looking at computation***:
  - Dataflow language: movement of data is the priority
  - ***Scheduling responsibility of the system, not the programmer***

- ***Basic characteristic***:
  - All processes run “simultaneously”
  - Processes can be described with imperative code
  - Processes can *only* communicate through buffers
  - Sequence of read tokens is identical to the sequence of written tokens

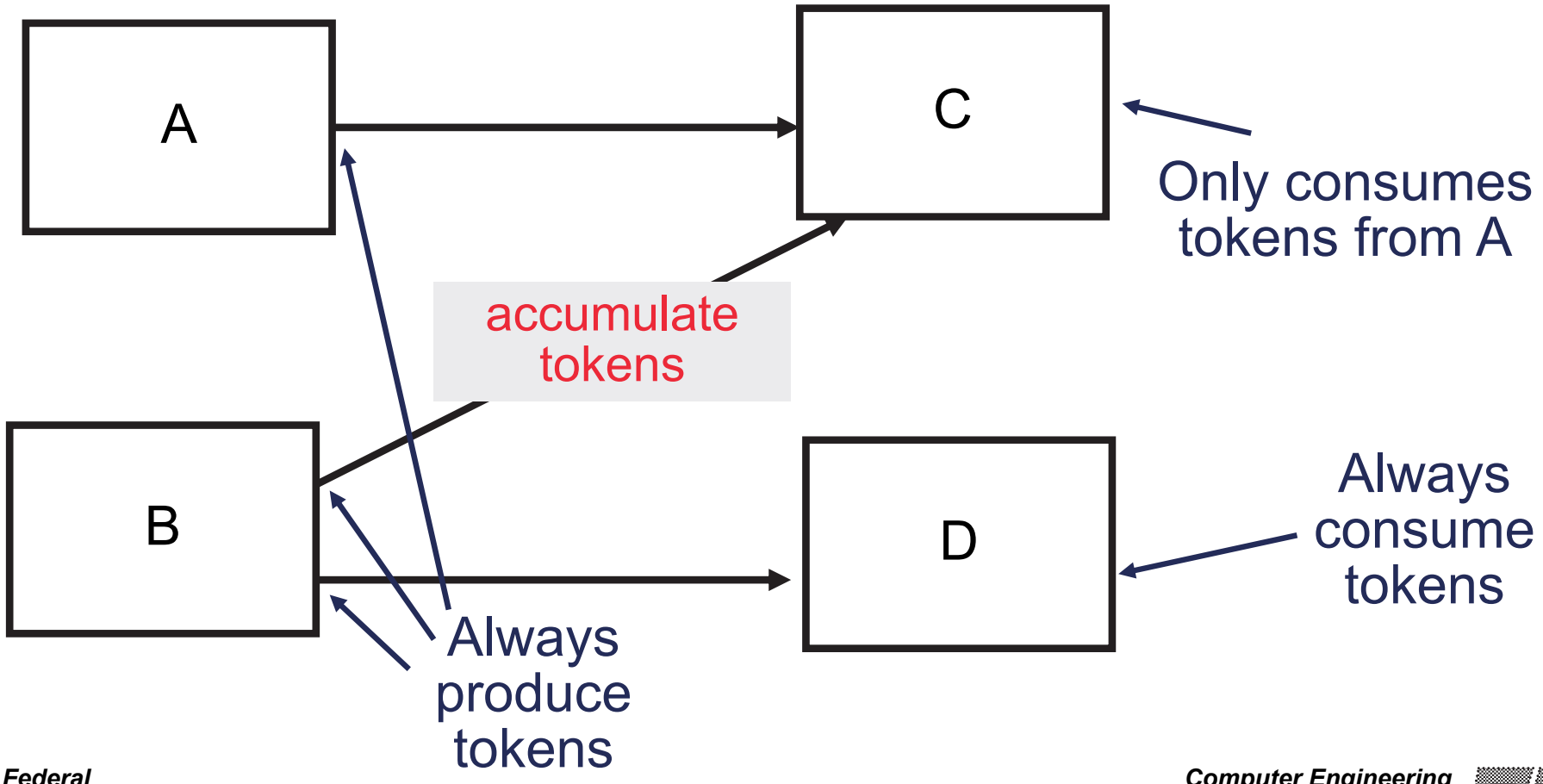
Scheduling Kahn Networks

- Challenge is running processes without accumulating tokens and without producing a deadlock (none of the processes can execute; they all block on missing input).



Demand-driven Scheduling?

- Idea: Only run a process whose outputs are being actively solicited.
- However, it does not solve the problem …



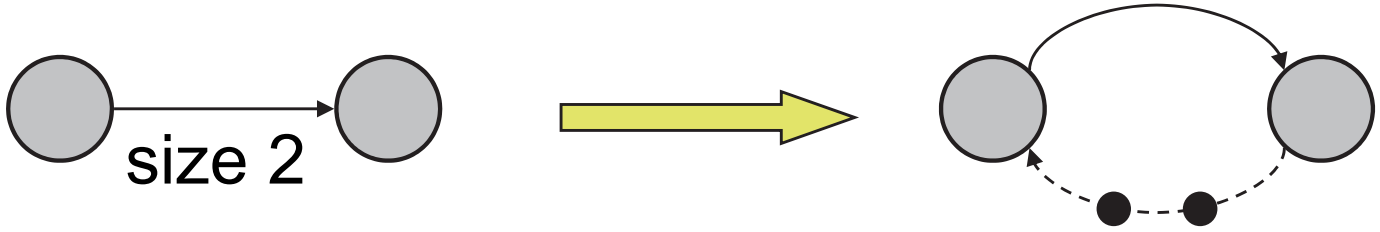
Tom Parks' Algorithm

- Schedules a Kahn Process Network in ***bounded memory*** if it is possible:
  - *Start* with a network *with bounded buffer sizes* and blocking write (can be transformed into a conventional KPN).
  - Use *any scheduling technique* to schedule the execution of processes that does not stall if there is still a process that is not blocked.
  - As long as the process networks runs *without deadlock* caused by blocking write -> *continue*.
  - If system *deadlock*s because of blocking write, *increase size* of smallest buffer and continue.

[*deadlock*: no process executes any instruction]

From Infinite to Finite Buffer Size

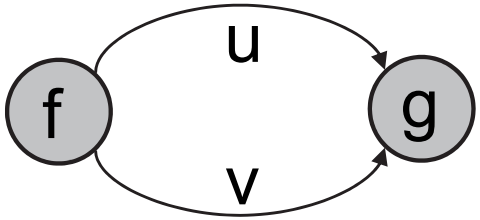
- A given KPN can be changed such that buffer sizes are finite:
  - To every channel between two processes an additional (virtual) channel in the reverse direction is added.
  - If the buffer size is restricted to *n*, then this new virtual channel has *n* initial data.
  - Every write (read) of the original channel leads to a read (write) of the virtual channel.
  - Invariant: The sum of token in both channels is constant.



- The resulting KPN has the same functional behavior but it may deadlock because of the finite buffer sizes

Deadlock Example

- An example of a KPN that produces a deadlock caused by a finite buffer size, for example if size(u) = 1:

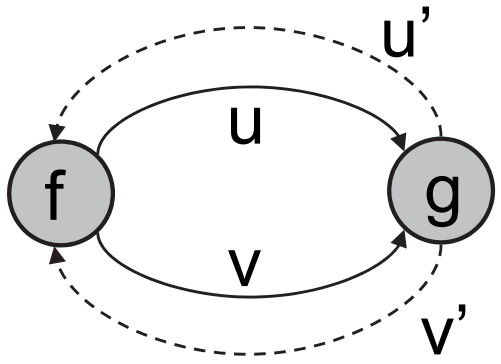



```
process f(out int a, out int b) {
  for (;;) {
    send(1, u);
    send(1, u);
    send(1, v);
  }
}
```

```
process g(in int a, in int b) {
  for (;;) {
    wait(v);
    wait(u);
    wait(u);
  }
}
```

Example: Finite Size Buffers in KPN

- The previous process network with finite buffer size can be converted into a KPN:



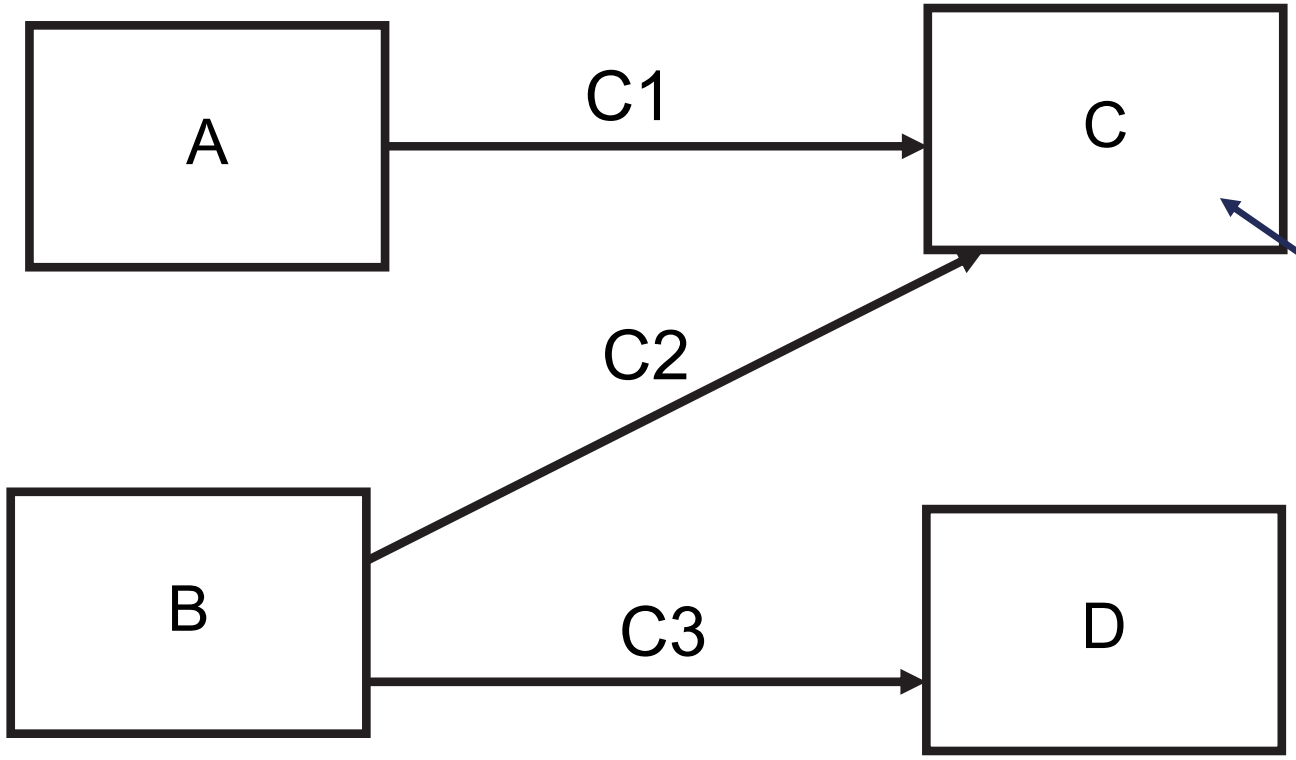
size(u) = 2
size(v) = 1

```
process f(...) {
  for (;;) {
    wait(u'); send(1, u);
    wait(u'); send(1, u);
    wait(v'); send(1, v);
  }
}
```

```
process g(...) {
  send(1,u'); send(1, u'); send(1, v');
  for (;;) {
    wait(v); send(1, v');
    wait(u); send(1, u');
    wait(u); send(1, u'); }
}
```

Parks' Algorithm in Action

- Start with buffers of size 1.
- Run A, B, C, D.



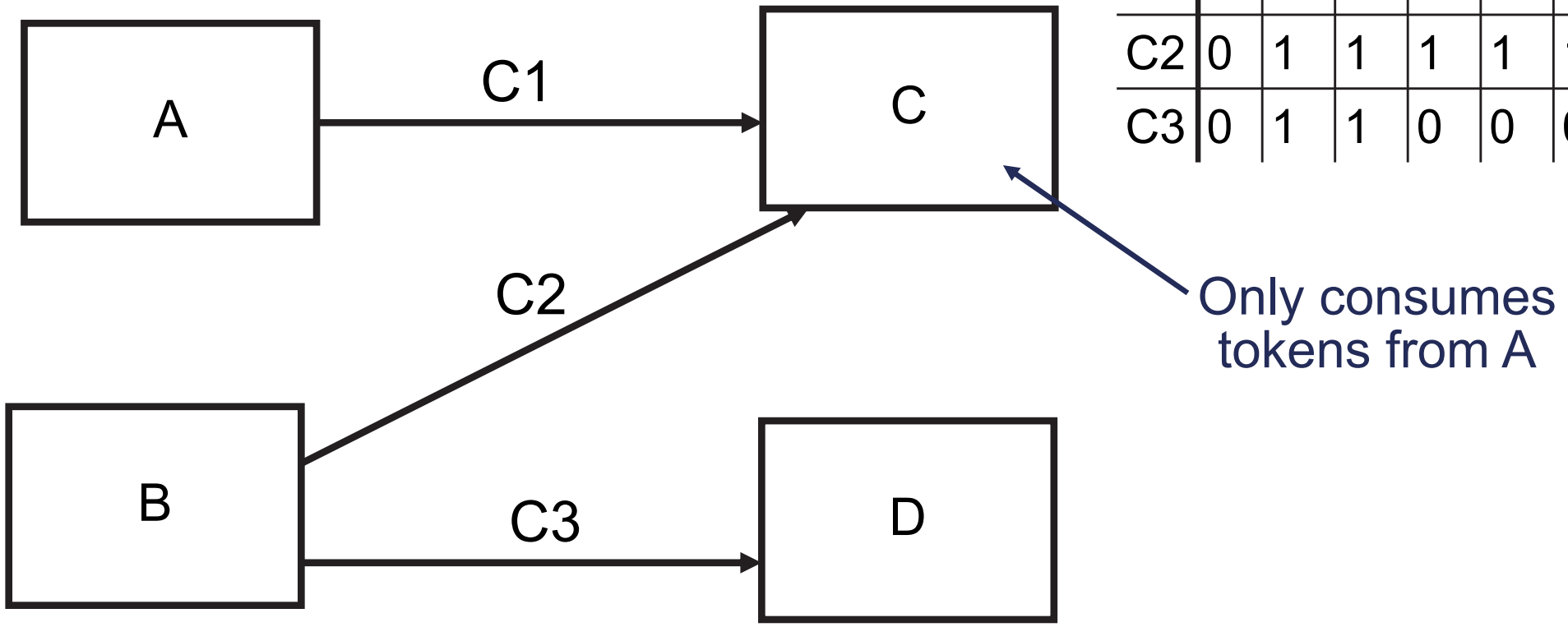
	A	B	C	D
C1	1	1	0	0
C2	0	1	1	1
C3	0	1	1	0

Only consumes tokens from A

Parks' Algorithm in Action

- Start with buffers of size 1.
- Execute processes A, B, C, D.
- B blocked waiting for space in B->C buffer.
- System will run indefinitely.

	A	B	C	D	A	C	A
C1	1	1	0	0	1	0	...
C2	0	1	1	1	1	1	...
C3	0	1	1	0	0	0	...



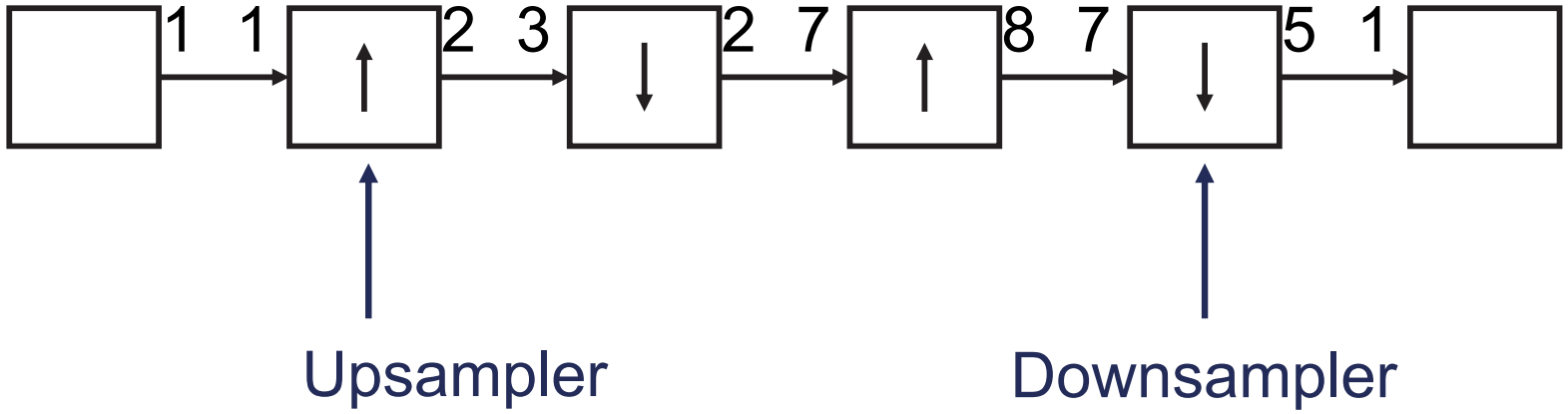
Evaluation of Kahn Process Networks

- ***Pro***:
  - Their advantage is that the scheduling algorithm does not affect the functional behavior
  - Matches stream-based processing
  - Makes coarse grained parallelism explicit
  - Favors mapping to multi-processor and distributed platforms

- ***Con***:
  - Difficult to schedule because of need to balance relative process rates
  - System inherently gives the scheduler few hints about appropriate rates
  - Parks' algorithm expensive and fussy to implement

Synchronous Dataflow (SDF)

- Edward Lee and David Messerschmitt, Berkeley, 1987:
  - *Restriction* of Kahn Networks to allow compile-time scheduling.
  - Each process reads and writes a ***fixed number of tokens*** each time it fires; firing is an atomic process.
- ***Example***: DAT-to-CD rate converter (converts a 44.1 kHz sampling rate to 48 kHz)



SDF Scheduling

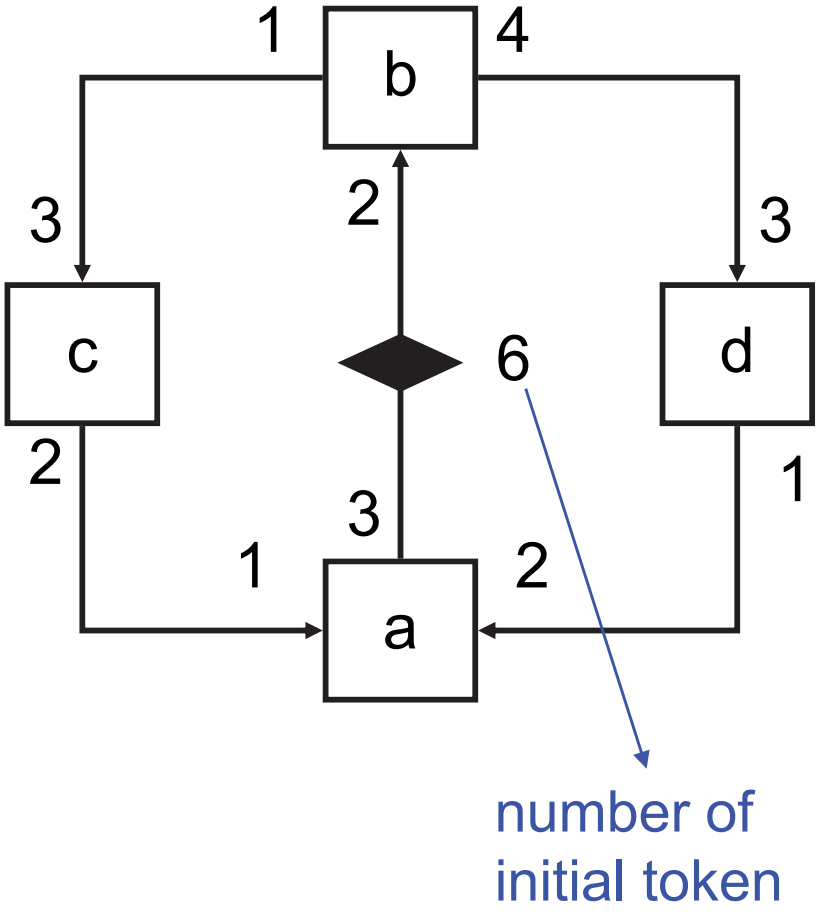
- ***Schedule*** can be determined completely ***at compile time*** (before the system runs).
- Two steps:
  1. *Establish relative execution rates* by solving a system of linear equations (balancing equations).
  2. *Determine periodic schedule* by simulating system for a single round (returns the number of tokens in each buffer to their initial state).

- ***Result***: the schedule can be executed repeatedly without accumulating tokens in buffers

Balancing Equations

- Each arc imposes a constraint:

$$3a - 2b = 0$$
$$4b - 3d = 0$$
$$b - 3c = 0$$
$$2c - a = 0$$
$$d - 2a = 0$$



M q

$$\begin{bmatrix} 3 & -2 & 0 & 0 \\ 0 & 4 & 0 & -3 \\ 0 & 1 & -3 & 0 \\ -1 & 0 & 2 & 0 \\ -2 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

Solving the Balancing Equation

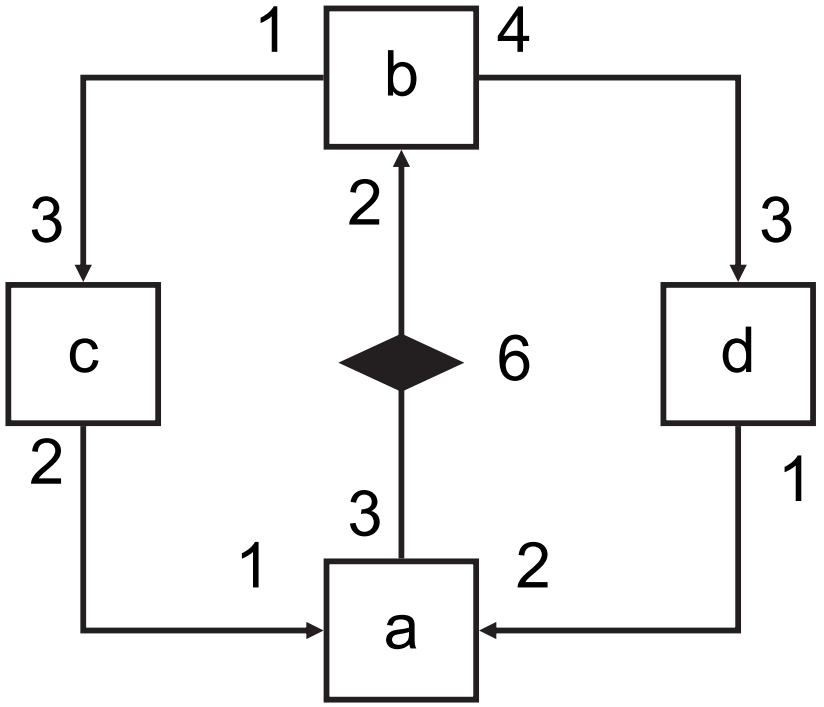
- ***Main SDF scheduling theorem*** (Lee '87):
  - A connected SDF graph with *n* actors has a periodic schedule iff its topology matrix *M* has rank *n*-1
  - If M has rank *n*-1 then there exists a unique smallest positive integer solution *q* to *M q* = 0 if there are sufficiently many initial token available
  - Inconsistent systems have only the all-zeros solution
  - Disconnected systems have two- or higher-dimensional solutions
- ***Example***:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & -2 & 0 & 0 \\ 0 & 4 & 0 & -3 \\ 0 & 1 & -3 & 0 \\ -1 & 0 & 2 & 0 \\ -2 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

Determine Periodic Schedule

- ***Possible schedules***:
  - (BBBCDDDDAA)*
  - (BDBDBCADDA)*
  - (BBDDBDDCAA)*
  - …

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 4 \end{bmatrix}$$



- Systems often have many possible schedules. How can we use this ***flexibility***?
  - Reduced code size (loop structure, hierarchy)
  - Reduced buffer sizes