Embedded Systems:
Specification and Modeling (part I)

Todor Stefanov
Leiden Embedded Research Center,
Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands

Outline

- Why considering modeling and specification?
- Requirements for Specification Techniques
- Models of Computation
  - State-based models (not considered in this course!)
    - FSM (classical automata)
    - Timed automata
    - StateCharts
  - Petri Nets (not considered in this course!)
    - Condition/Event Nets
    - Predicate/Transition Nets
    - Place/Transition Nets
  - Actor-based Dataflow models
    - SDF, CSDF, PPN, PSDF, PCSDF, PPPN, BDF, DDF, KPN
- Specification Languages
  - VHDL, SystemC, SpecC, Others

Why considering specifications?

- The first step in designing Embedded System is to ***precisely*** tell *what the system behavior should be*

Specification: correct, clear and unambiguous description of the required system behavior

- This can be extremely difficult
  - Increasing complexity of ES
  - Desired behavior often not fully understood in the beginning
- However, if something is wrong with the specification
  - difficult to get the design right
  - potentially wasting a lot of time
- How can we (correctly and precisely) capture systems behavior?

Use Model-based Specifications!

- We use ***models*** of the system under design at ***different levels of abstraction (LoA)***
    - LoA alleviate the complexity problem of specification
    - LoA will be discussed later
- Models allow to reason about the system under design
    - identifying flaws in the specification
    - correcting flaws in the specification

- What is a model anyway?

Model

Definition [Jantsch, 2004]**:** A model is a simplification of an entity, which can be a physical thing or another model:

1. Contains exactly those characteristics and properties of the entity that are relevant for a given task
2. Is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task

Quote [George Box, 1987]**:**
Essentially, all models are wrong, but some are useful!

-- Wrong: models are simplification of an entity!
-- Useful: models help to explain, predict, and understand **some aspects** of the entity!

NOTE: Engineers use models differently to scientists!

-- Scientists: use models to describe what the physical world is doing!
-- Engineers: use models to construct a physical system that behaves like the model!

Requirements for Model-based Specification Techniques (1)

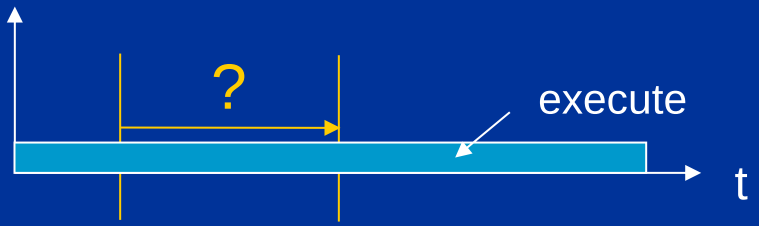
- Modularity
  - Systems specified as **composition of objects**
    - Most humans not capable to understand systems containing more than ~5 objects
    - BUT most actual systems require more objects!
  - **Hierarchical composition** of objects
    - Example for SW: statements -> procedures -> programs
    - Example for HW: transistors -> gates -> functional blocks
  - It must be “easy” to derive system behavior from behavior of subsystems
- Concurrency, synchronization and communication

Requirements for Model-based Specification Techniques (2)

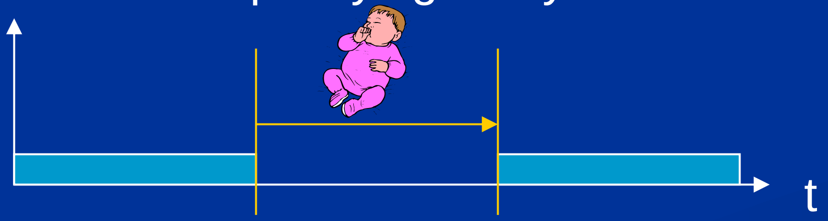
- Timing behavior
  - **Essential for embedded systems!**
  - Four types of timing specs required, according to Burns, 1990:

1. Techniques to measure elapsed time

   Check, how much time has elapsed since some computation has happened

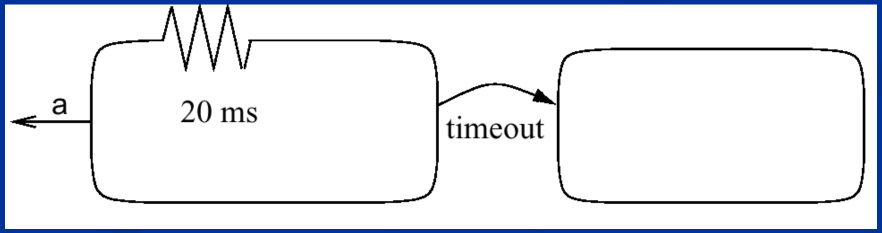


2. Means for specifying delays

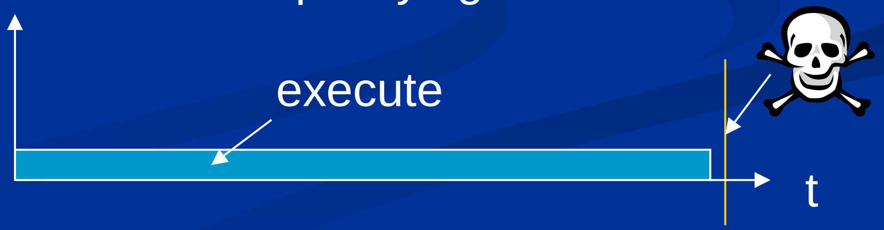


3. Possibility to specify timeouts

   Stay in a certain state a maximum time

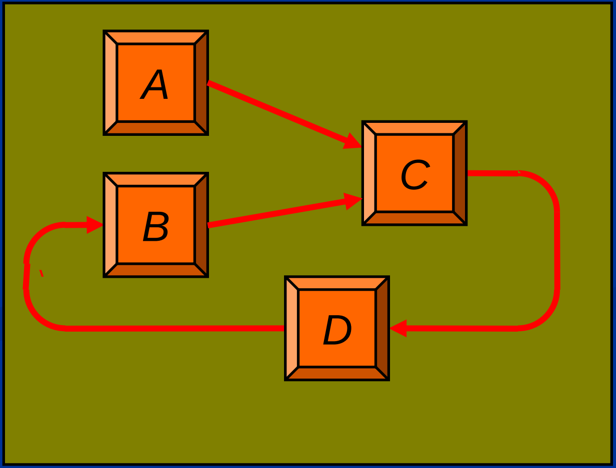


4. Methods for specifying deadlines



Models of Computation for Specs

- **Models of Computation (MoC) define**:
  - Components and execution model for computations for each component
  - Communication model for exchange of information between components



There is *NO* model of computation that meets all specification requirements previously discussed!

- Thus, we must
  - select appropriate MoC for specifying a system
  - this is key to successful and efficient design of ES

Is Van Neumann MoC appropriate?

- An instruction set, a memory, and a program counter, is all we need to execute whatever application we can dream of

- NOT appropriate for ES design!
  - Timing cannot be described
    - instructions cannot be delayed
    - instruction cannot be forced to execute at a specific time
  - Timing deadlines cannot be specified for instructions or sequence of instructions
  - Timeouts cannot be specified for sequence of instructions

Another Inappropriate MoC: Thread-based concurrency model

*"… **threads as a concurrency model are a poor match for embedded systems**. … they work well only … where best-effort scheduling policies are sufficient."*

Edward Lee: Absolutely Positively on Time, *IEEE Computer*, July, 2005

- Threads may access global variables
  - May lead to race conditions!
- To avoid races, we use mutual exclusion
  - May lead to deadlocks!

Other problems with thread-based concurrency

- Threads are nondeterministic!
- Programmers try to prune away the non determinism by imposing constraints on execution order (e.g., mutexes, locks, etc…)
- *Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to many humans*
- *Thus,*

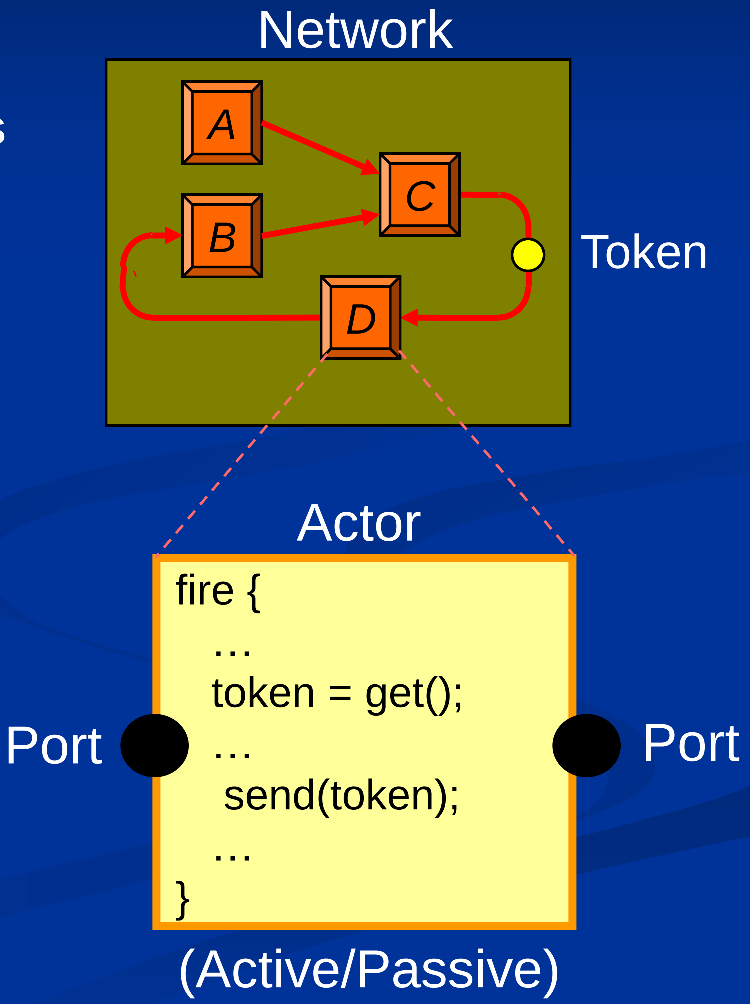
The bottom line is

When specifying and designing Embedded Systems we should search for and use NON-thread-based, NON-von-Neumann Models of Computation!

- Finding appropriate model to capture ES behavior is an important step!
- For control-dominated and reactive systems
  - State-based models are appropriate
  - Monitor control inputs and set control outputs
- For data-dominated systems
  - Actor-based dataflow models are appropriate
  - Transform input data streams to output data streams

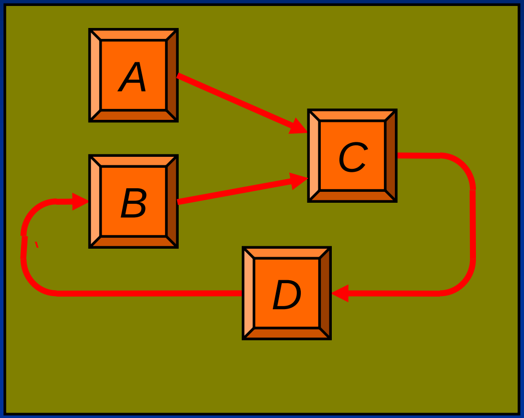
Actor-based Models of Computation: Terminology

- Actor-based MoC
  - Formal description of the operational semantics of a network of functional blocks
- Actor
  - Functional block representing some computation
- Relation
  - Describes the communication between actors
- Token
  - Quantum of information that is exchanged between actors
- Firing of actor
  - Quantum of computation
  - Moment of interaction with other actors



```
fire {
   …
   token = get();
   …
    send(token);
   …
}
```

Active/Passive Actors



```
fire {
  token = get();
  …
   send(token);
  …
}
```

Exit

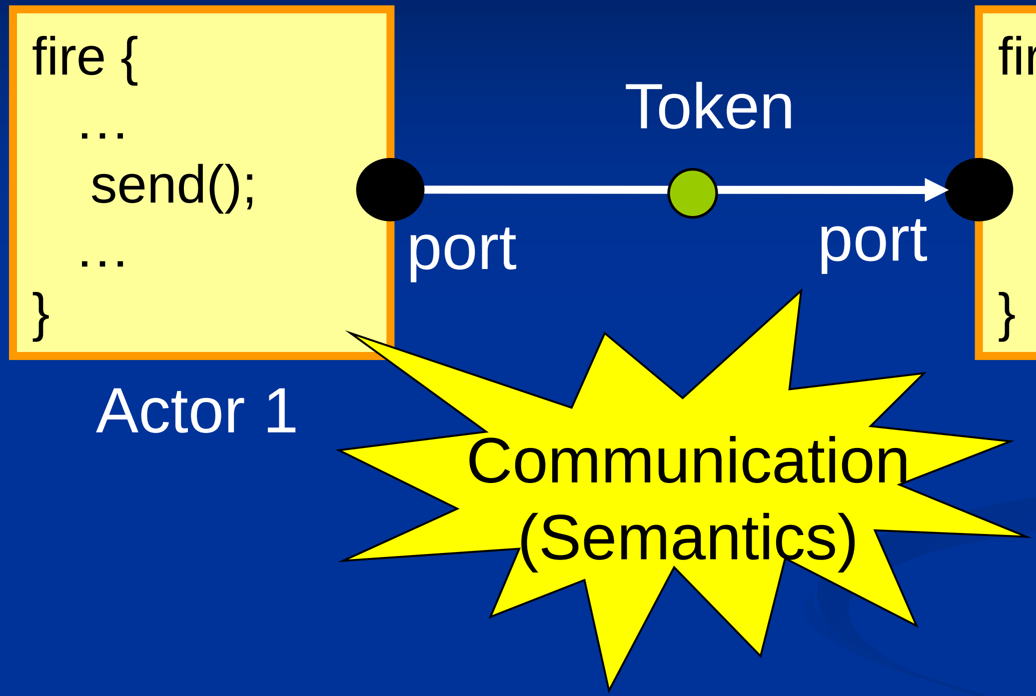
Two kinds of Actors:

```
fire {
  while(1) {
    token = get();
    …
    send(token);
  }
}
```

- **Passive Actors:**
  - Scheduler needed to activate the firing
    - Schedule ABBCD
  - A firing needs to terminate
  - Fire-and-exit behavior

- **Active Actors:**
  - Schedule themselves
  - A firing typically does not terminate
    - Endless while loop
  - Process behavior

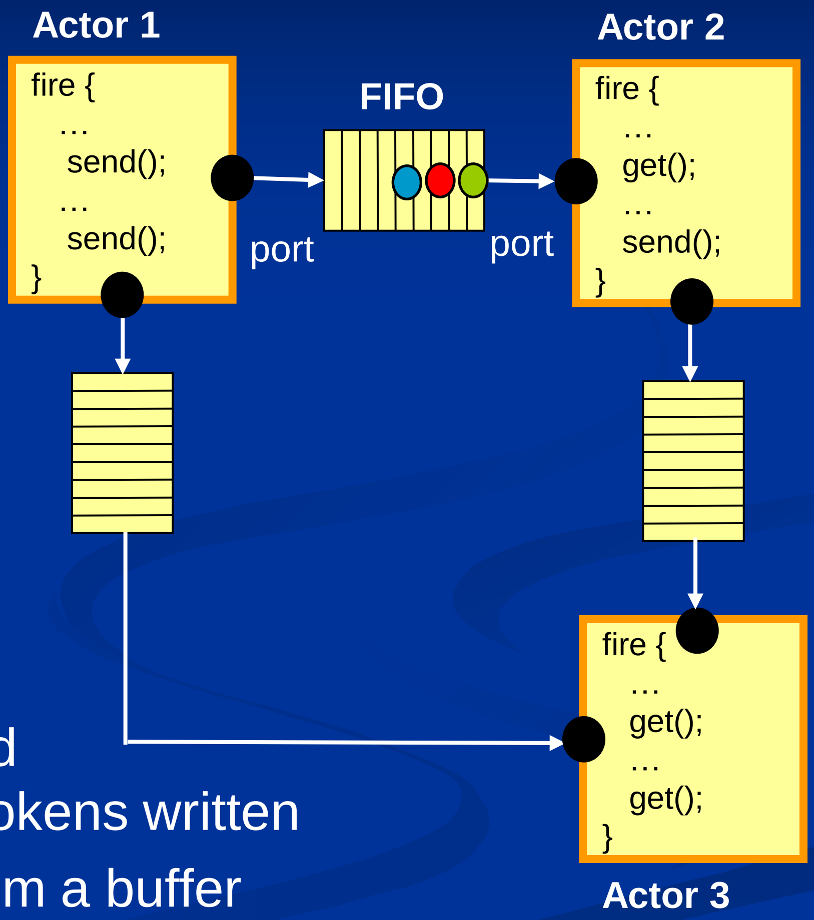
Communication Between Actors



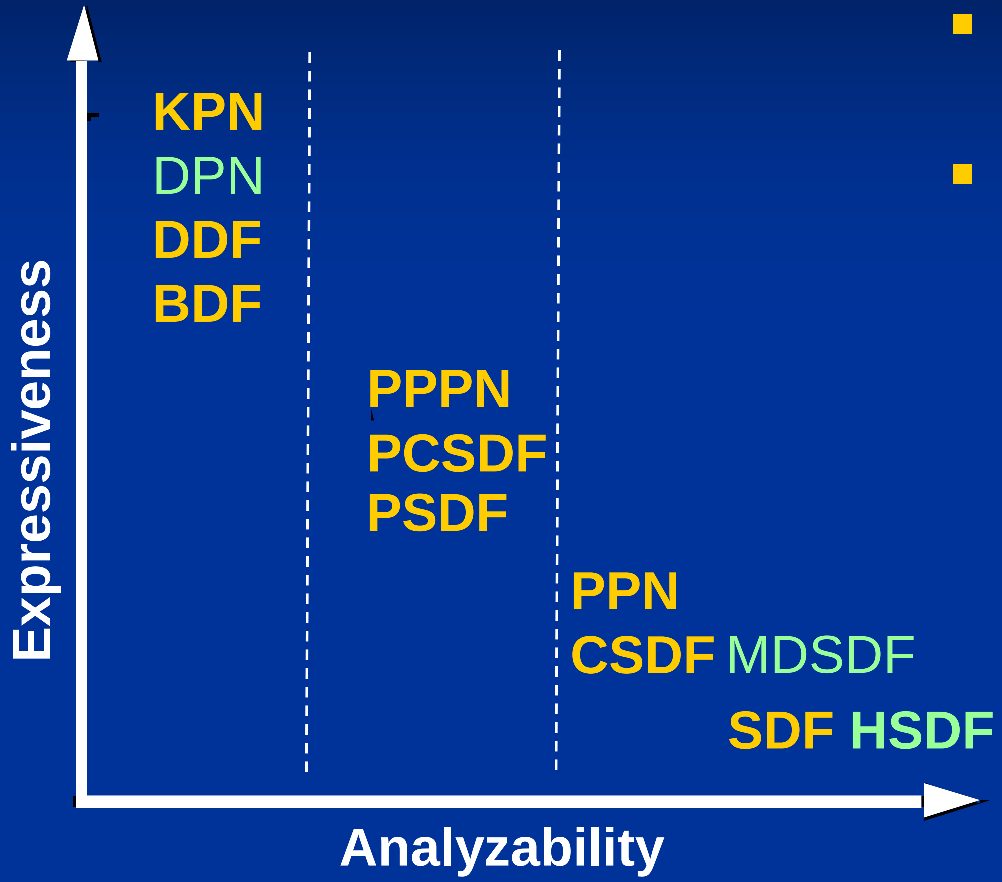
- Data Type of Tokens
  - Integer, Double
  - Complex
  - Matrix, Vector
  - Record
- Way exchange takes place
  - Buffered
  - Timed
  - Synchronized

Actor-based *Dataflow* Models (1)

- Network of concurrently executing actors
- Dataflow Actors
  - Can be Passive or Active
  - Can be described with imperative code
- Dataflow Communication
  - *Only* through FIFO buffers
  - Buffers usually treated as *unbounded* for flexibility
  - Sequence of tokens read guaranteed to be the same as the sequence of tokens written
  - *Destructive read*: reading a token from a buffer removes the token
  - Much more predictable than shared memory



Dataflow Modeling Space



Expressiveness

KPN
DPN
DDF
BDF

PPPN
PCSDF
PSDF

PPN
CSDF MDSDF

SDF HSDF

Analyzability

- **Expressiveness:**
  - Indicate what type of systems can be modeled and how compact the model is
- **Analyzability:**
  - Indicate the degree of possibility for compile-time analysis (scheduling, buffer sizes, etc.)

Decidable Models:

- Synchronous Data Flow (SDF)
- Homogeneous SDF (HSDF)
- Multi-Dimensional SDF (MDSDF)
- Cyclo-Static Data Flow (CSDF)
- Polyhedral Process Network (PPN)

Partly-Decidable Models:

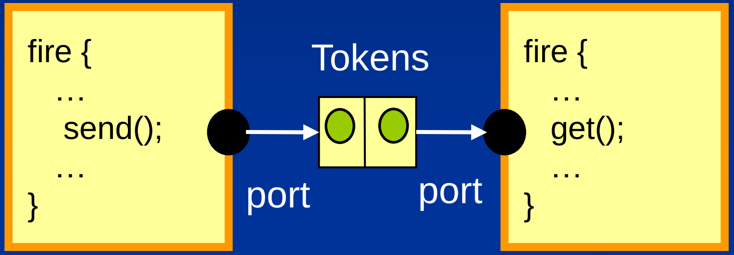
- Parameterized [SDF, CSDF, PPN]

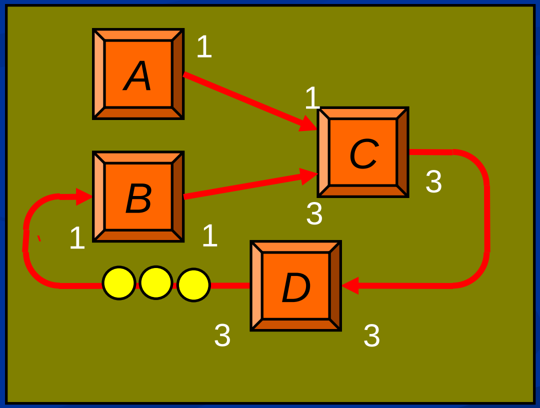
Undecidable Models:

- [Boolean, Dynamic] Data Flow
- [Dataflow, Kahn] Process Network

Synchronous Data Flow (SDF)

- Introduced by Lee and Messerchmitt, UC Berkeley, 1987
- Network of concurrent executing actors
  - Passive actors
  - Communication is buffered
- The model progresses as a sequence of “iterations”
- A “firing rule” determines the firing condition of an actor
- At each firing, a fixed number of tokens is consumed and produced
- Characteristics of SDF
  - Compile time analyzable
  - Static schedule
  - Optimization for memory/throughput/latency

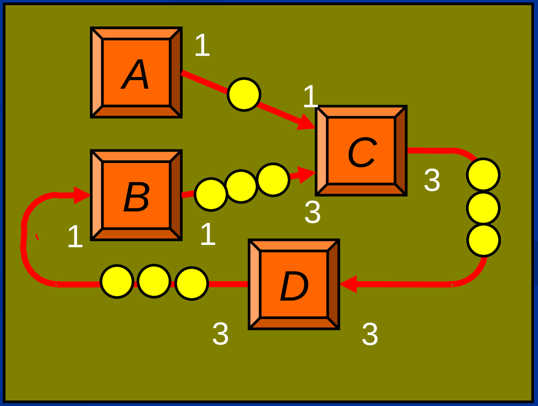




Iteration: **ABBBCD**

SDF Operational Semantics: Firing Rule

- An actor of SDF is ***enabled*** if there is a certain number of tokens on each of its input arcs
- An ***enabled actor is fired*** by removing a number of tokens from each of its input arcs and placing tokens on each of its output arcs
- Iteration: a sequence of actors' firings that brings the SDF network to its initial state
  - Many possible sequences as long as firing rules are obeyed



Iteration: **ABBBCD**

SDF: Fixed Production and Consumption Rate

fire A {
  …
  produce N
  …
}

channel

N M

fire B {
  …
  consume M
  …
}

- Balance equations (one for each channel):

$$f_A N = f_B M$$

number of tokens consumed

number of firings per “iteration”

number of tokens produced

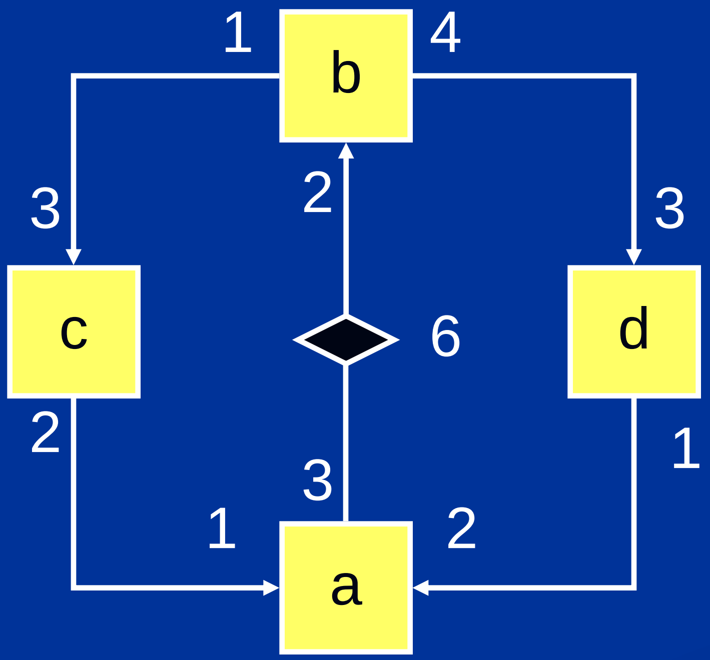
- Schedulable statically
- Decidable:
  - buffer memory requirements
  - deadlock

SDF: Scheduling

- Goal: Find a sequence of actor firings that
  - Runs each actor at least once
  - Avoids underflow
    - no actor fired unless all tokens it consumes are available
  - Returns the number of tokens in each buffer to their initial state
- Result: Schedule can be executed repeatedly without accumulating tokens in buffers
- Schedule can be determined completely at compile-time, i.e., before the system runs
  - Two steps:
    1. Establish relative firing rates of actors by using the balance equations
    2. Determine periodic sequence of actor firings by simulating the model for a single iteration

Step 1: Calculating Rates (1)

- Each channel imposes a constraint
  - The number of tokens produced should be equal to the number of the tokens consumed
  - The balance equation guarantees this for each channel
- Example:



$3a - 2b = 0$ (for ch. ab)

$4b - 3d = 0$ (for ch. bd)

$b - 3c = 0$ (for ch. bc)

$2c - a = 0$ (for ch. ca)

$d - 2a = 0$ (for ch. da)

Solution:

$a = 2c$ (a should fire twice more than c)

$b = 3c$

$d = 4c$

Step 1: Calculating Rates (2)

- **The modeled embedded system is <u>Consistent</u>!**
  - Has more than one solution (all-zeros solution + other solutions)
  - Usually we want the smallest integer non-all-zeros solution
- **Inconsistent systems:**
  - Have only the all-zeros solution
- **Disconnected systems:**
  - Relative rates between some actors undefined
- Example: Consistent Systems

$3a - 2b = 0$ (for ch. ab)

$4b - 3d = 0$ (for ch. bd)

$b - 3c = 0$ (for ch. bc)

$2c - a = 0$ (for ch. ca)

$d - 2a = 0$ (for ch. da)

<u>Solution:</u>

$a = 2c$ (**a** should fire twice more than **c**)

$b = 3c$

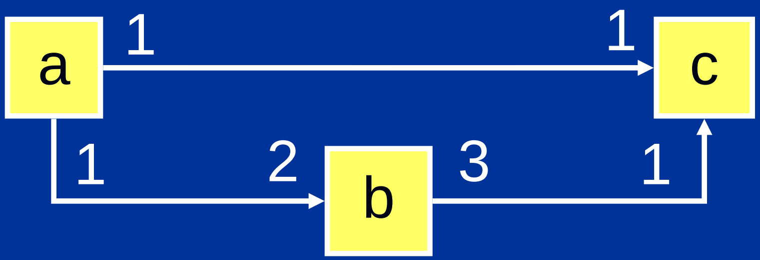
$d = 4c$

This is the smallest integer solution which is non-zero

a=2 b=3 c=1 d=4

Inconsistent and Disconnected Systems

- Inconsistent system
  - Only solution is “do nothing”, i.e.,
    - The only integer solution is **a=0 b=0 c=0**
  - No way to execute it without an unbounded accumulation of tokens on the channels



$a - c = 0$

$a - 2b = 0$

$3b - c = 0$

Or

$2b - c = 0$

$3b - c = 0$

- Disconnected system (under-constrained system)
  - Two or more unconnected pieces
  - Relative rates between pieces undefined

$a - b = 0$

$3c - 2d = 0$

Consistent Rates Not Enough!

- A consistent system may NOT have schedule
- Rates do not avoid deadlock
- Example: deadlock in consistent system

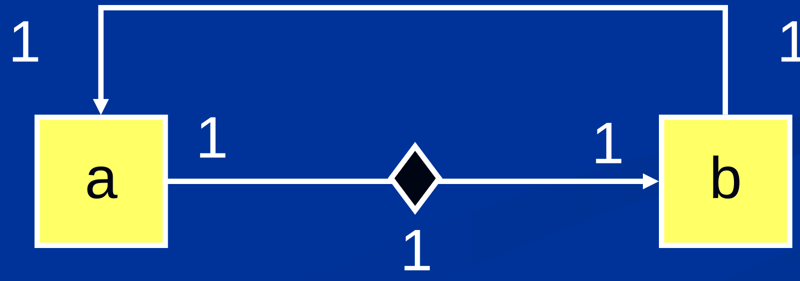
Initially No Tokens

a waits for b

b waits for a

Deadlock!!!

- Solution here: add an initial token on one of the channels



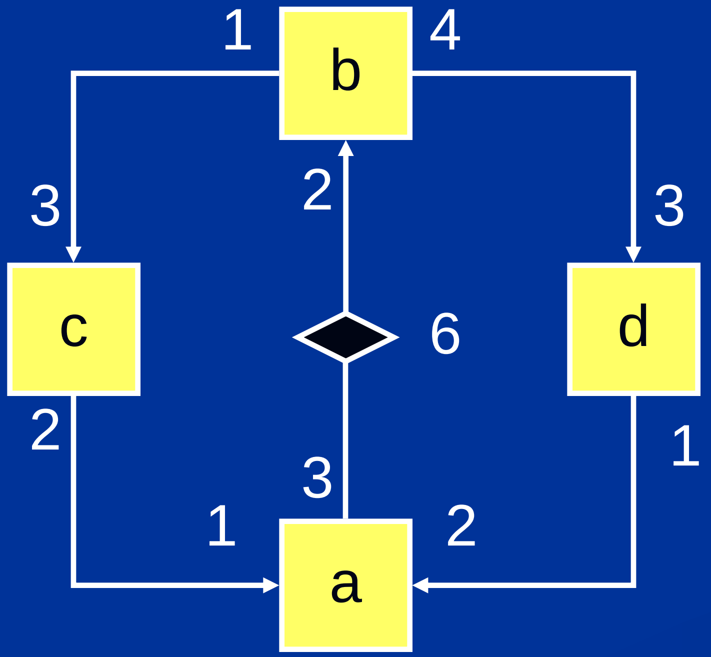
Initially 1 Token on arc ab

b can fire

Step 2: Fundamental SDF Scheduling Theorem

If rates can be established, any scheduling algorithm that avoids buffer underflow will produce a correct schedule if it exists

- Theorem guarantees that any valid model simulation will produce a schedule
- Example:



Rates: a=2 b=3 c=1 d=4

Possible schedules:

BBBCDDDDAA

BDBDBCADDA

BBDDBDDCAA

… many more

BC … is not valid

SDF: Scheduling Choices

- The SDF Scheduling Theorem guarantees that a schedule will be found if it exists
- A SDF system often has many possible schedules
- How can we use this flexibility?
    - Reduce size of code
    - Reduce sizes of buffers

SDF: Code Generation

- Consider schedule

  BBBCDDDDAA
- Rewrite schedule in "looped" form:

  (3 B) C (4 D) (2 A)
- Generated inline code becomes

```
for ( i = 0 ; i < 3; i++) B;
C;
for ( i = 0 ; i < 4 ; i++) D;
for ( i = 0 ; i < 2 ; i++) A;
```

- Consider schedule

  BDBDBCADDA
- Rewrite schedule in "looped" form:

  (2 BD) BCA (2 D) A
- Generated inline code becomes

```
for ( i = 0 ; i < 2; i++) {B;D;}
B;C;A;
for ( i = 0 ; i < 2 ; i++) D;
A;
```

Which code is smaller?

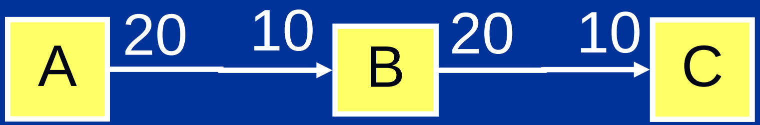
SDF: Code Size optimization

- Goal: Find ***Single Appearance Schedule***:
  - (3 B) C (4 D) (2 A)
  - a looped schedule in which each block appears exactly once
- Leads to efficient block-structured code
  - Only requires one copy of each block’s code
- Does not always exist!
- Often requires more buffer space than other schedules!
- Generated program with efficient code size

```
for ( i = 0 ; i < 3; i++) B;
          C;
for ( i = 0 ; i < 4 ; i++) D;
for ( i = 0 ; i < 2 ; i++) A;
```

SDF: Buffer Size optimization

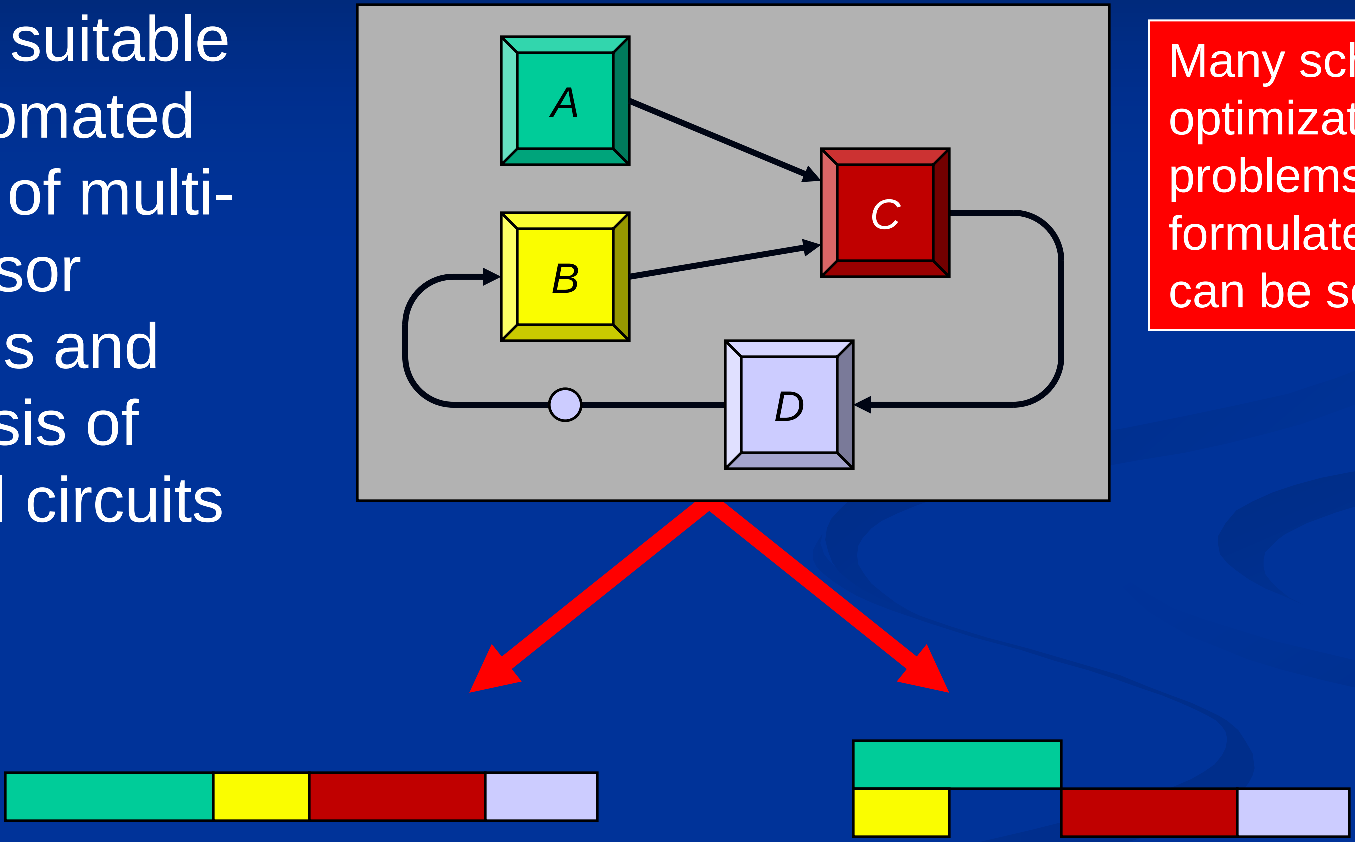
- Goal: Find ***Minimum Memory Schedules***
- Often increases code size (block-generated code)
- Static scheduling makes it possible to exactly predict memory requirements
- Example:



Schedule	Total buffer sizes
(1) ABCBCCC	50 tokens
(2) A(2B)(4 C)	60 tokens
(3) A(2(B (2C)))	**40 tokens**
(4) A(2(BC))(2 C)	50 tokens

SDF: Parallel Scheduling

SDF is suitable for automated design of multi-processor systems and synthesis of parallel circuits



Many scheduling optimization problems can be formulated. Some can be solved, too!

Sequential

Parallel

To be continued