# Announcement

- Anonymous evaluation requested
  - TAs: 1 (worst) – 10 (best)
    - 서영석 (Young-seok Seo)
    - 임형인 (Hyungin Im)
  - CS550 (by June 11th)
    - http://portal.kaist.ac.kr → Webcais→ 학사 → 성적 → 각 수강과목에 대하여 클릭 → 설문항 체크 → 확인
    - Note that you cannot see your final grade if you do not finish course evaluation, which is a policy of KAIST
- Safehome project 3rd part
  - Due is June 7th
  - Demonstration is scheduled on June 8th afternoon in the class room
    - You should bring your own notebook to demonstrate your safehome system due to various execution environments
    - You must submit all your project material both hardcopy and softcopy by the 8:00 AM June 8. 10% penalty will be applied to late submission.
- Final exam on 5:00 – 6:10 PM, June 12 (next Tuesday)

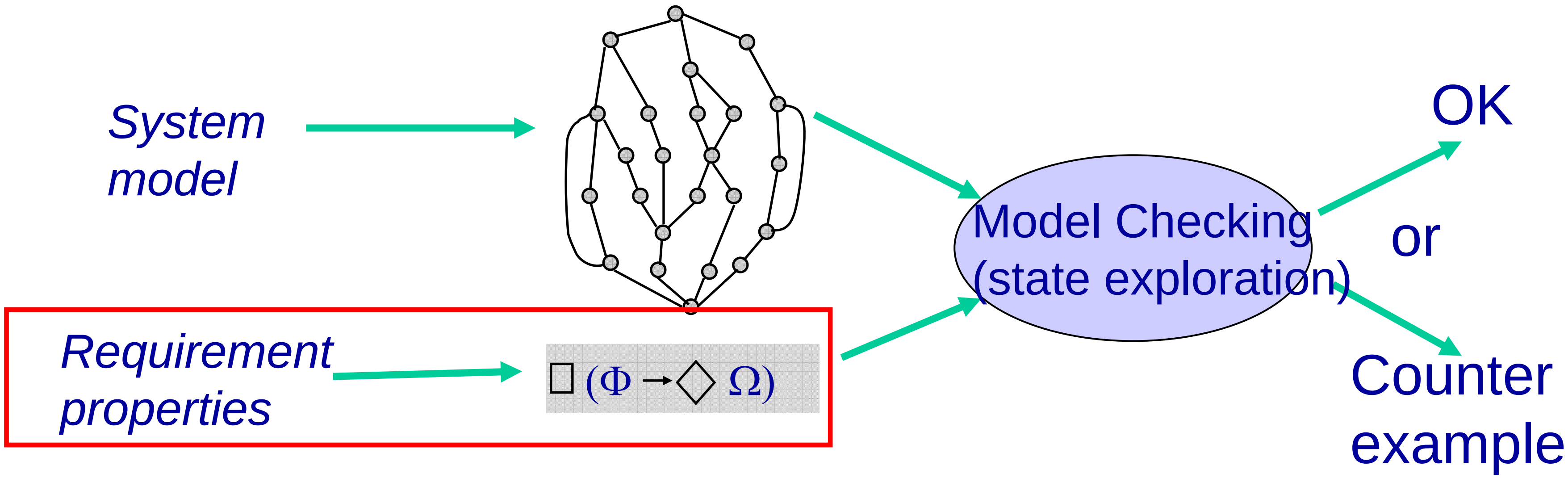
# The Spin Model Checker : Part II/II

*Moonzoo Kim*

*CS Dept. KAIST*

# Model Checking

- Specify requirement properties and build system model
- Generate possible states from the model and then check exhaustively whether given requirement properties are satisfied within the state space



# Specification of Requirement Properties

- `assert()` is simple, but powerful enough to verify many practical properties
  - Ex. `assert()` can express properties on invariance, critical section, mutual exclusion, etc
- However, `assert()` cannot express some popular properties such as eventuality
  - Ex. Whenever a server receives a read request from a client, the server should send back an acknowledge message to the client eventually
- We need more powerful mechanism to describe requirement properties
  - Linear temporal logic (LTL) can be a good solution

Slides from ["Logic Model Checking"]() taught by Dr.G.Holzmann at Caltech Spring 2005

## semantics

given a state sequence (from a run $\sigma$):

$s_0, s_1, s_2, s_3$ …

and a set of propositional symbols: p,q,… such that

$\forall i, (i \geq 0)$ and $\forall p$, $s_i \models p$ is defined

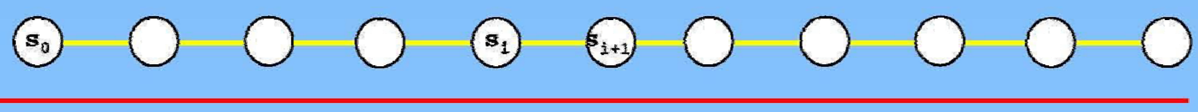
we can define the semantics of the temporal logic formulae:

[]f, <>f, Xf, and e U f

| | | |
|---|---|---|
| $\sigma \models f$ | iff | $s_0 \models f$ |

i.e., the property holds for the remainder of run $\sigma$, starting at position $s_0$

| | | |
|---|---|---|
| $s_i \models$ []f | iff | $\forall j, (j >= i): s_j \models f$ |
| $s_i \models$ <>f | iff | $\exists j, (j >= i): s_j \models f$ |
| $s_i \models$ Xf | iff | $s_{i+1} \models f$ |



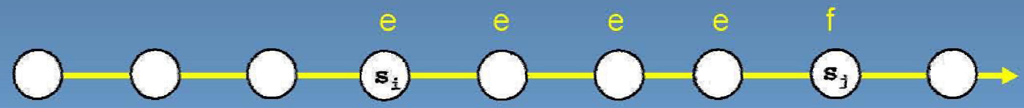
## weak and strong until

(cf. book p. 135-136)

**weak until**

$s_i \models e \; U \; f$ iff

$s_i \models f \lor (s_i \models e \land s_{i+1} \models (e \; U \; f))$




**strong until (Spin)**

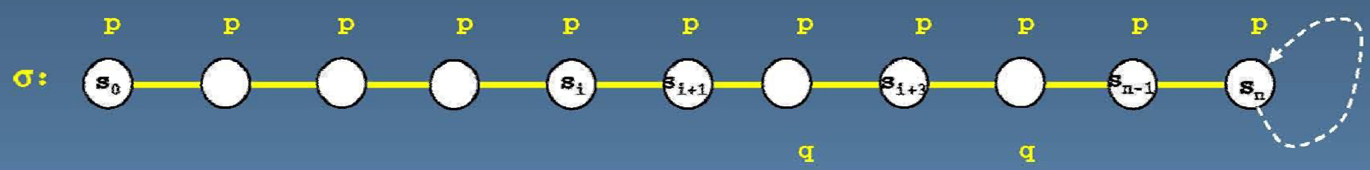
$s_i \models e \; U \; f$ iff

$\exists j, (j >= i): s_j \models f$ and

$\forall k, (i <= k < j): s_k \models e$

```
equivalences:
        (e U f) == (e U f) ∧ (<> f)
        (e U f) == (e U f) ∨ ([] e)
```

# examples




[]p is satified at all locations in σ

<>p is satisfied at all locations in σ

[]<>p is satisfied at all locations in σ

<>q is satisfied at all locations except $s_{n-1}$ and $s_n$

Xq is satisfied at $s_{i+1}$ and at $s_{i+3}$

pUq (strong until) is satisfied at all locations except $s_{n-1}$ and $s_n$

<>(pUq) (strong until) is satisfied at all locations except $s_{n-1}$ and $s_n$

<>(pUq) (weak until) is satisfied at all locations

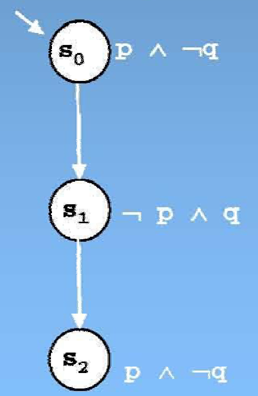
[]<>(pUq) (weak until) is satisfied at all locations

in model checking we are typically only interested in whether a temporal logic formula is satisfied for all runs of the system, starting in the initial system state (that is: at $s_0$)

# equivalences

(cf. book p. 137)

- `[]` p $\leftrightarrow$ (p U false) — weak until
- <>p $\leftrightarrow$ (true U p) — strong until
- ! `[]` p $\leftrightarrow$ <>!p
  - if p is not invariantly true, then eventually p becomes false
- !<>p $\leftrightarrow$ `[]` !p
  - if p does not eventually become true, it is invariantly false
- `[]` p && `[]` q $\leftrightarrow$ `[]` (p && q)
  - note though: (`[]` p || `[]` q) $\rightarrow$ `[]` (p || q)
  - but: (`[]` p || `[]` q) $\cancel{\leftarrow}$ `[]` (p || q)
- <>p || <> q $\leftrightarrow$ <> (p || q)
  - note though: (<> p && <> q) $\leftarrow$ <> (p && q)
  - but: (<> p && <> q) $\cancel{\rightarrow}$ <> (p && q)



## some standard LTL formulae

| | | |
|---|---|---|
| [] p | always p | invariance |
| <> p | eventually p | guarantee |
| p -> (<> q) | p implies eventually q | response |
| p -> (q U r) | p implies q until r | precedence |
| [] <> p | always, eventually p | recurrence (progress) |
| <> [] p | eventually, always p | stability (non-progress) |
| (<> p) -> (<> q) | eventually p implies eventually q | correlation |

non-progress
acceptance

dual types of properties

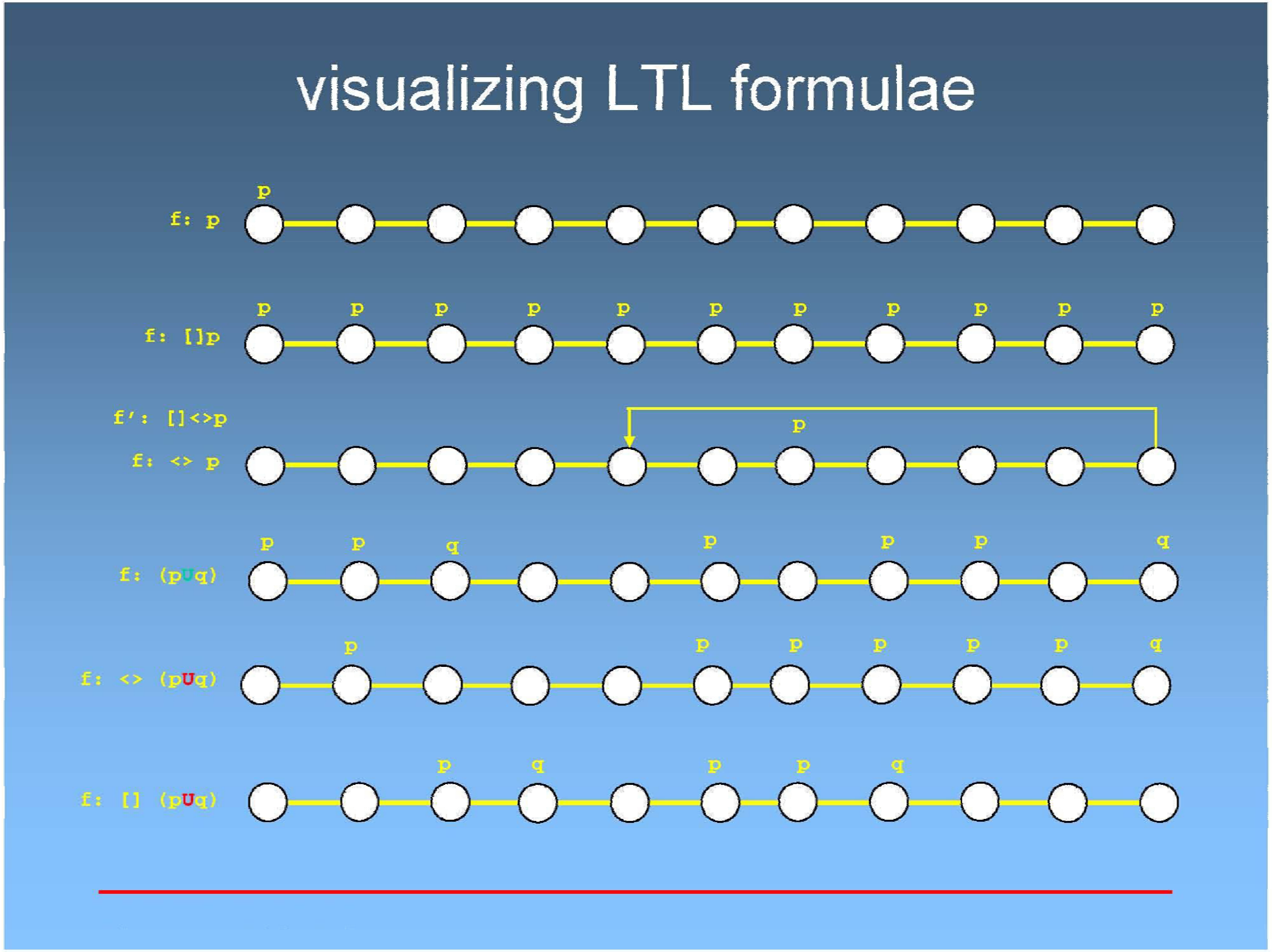
in every run where p eventually becomes true q also eventually becomes true (though not necessarily in that order)

## the earlier informally stated sample properties

(vugraph 12 lecture 11)

- p is invariantly true
  [] p
- p *eventually* becomes invariantly true
  <>[] p
- p *always eventually* becomes false at least once more
  []<>!p
- p *always* implies ¬q
  [] (p -> !q)
- p *always* implies *eventually* q
  [] (p -> <> q)

# Linear Temporal Logic



# Promela

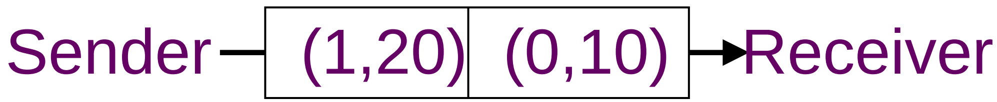
- The system specification language of the Spin model checker
- Syntax is similar to that of C, but simplified
  - No float type, no functions, no pointers etc
- Characteristics
  - Communication and concurrency
  - Formal operational semantics
  - Interleaved semantics
  - Asynchronous process execution
  - Two-way communication
- Unique features not found in programming languages
  - Non-determinism (process level and statement level)
  - Executability

# 6 Types of Basic Statements

- Assignment:  always executable
  - Ex. **`x=3+x`**, **`x=run A()`**
- Print: always executable
  - Ex. **`printf("Process %d is created.\n",_pid);`**
- Assertion: always executable
  - Ex. **`assert( x + y == z)`**
- Expression: depends on its value
  - Ex. **`x+3>0`**, **`0`**, **`1`**, **`2`**
  - Ex. **`skip, true`**
- Send: depends on buffer status
  - Ex. **`ch1!m`** is executable only if **`ch1`** is not full
- Receive: depends on buffer status
  - Ex. **`ch1?m`** is executable only if **`ch1`** is not empty

# Communication Using Message Channels

- Spin provides communications through various types of message channels
  - Buffered or non-buffered (rendezvous comm.)
  - Various message types
  - Various message handling operators
- Syntax
  - chan ch1 = [2] of { bit, byte};
    - ch1!0,10;ch1!1,20
    - ch1?b,bt;ch1?1,bt



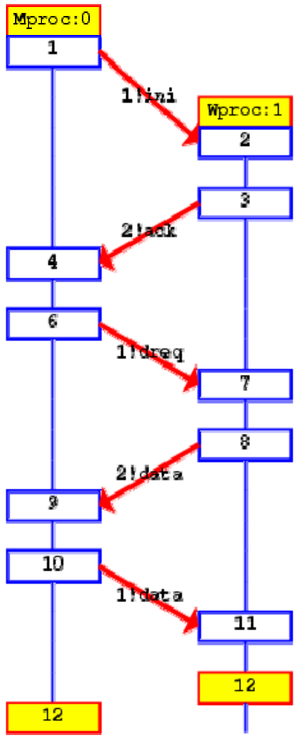

  - chan ch2= [0] of {bit, byte}

# Operations on Channels

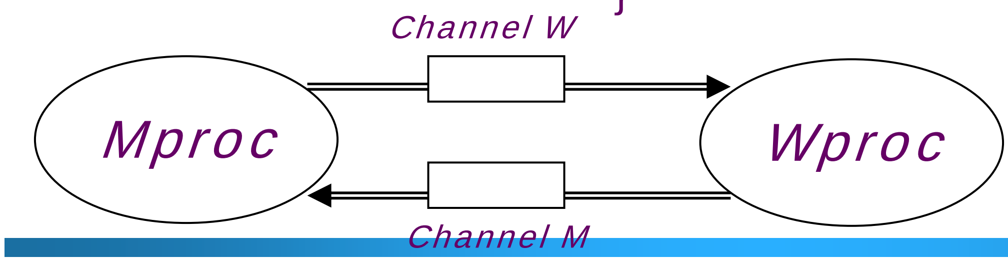
- **Basic channel inquiry**
  - `len(ch)`
  - `empty(ch)`
  - `full(ch)`
  - `nempty(ch)`
  - `nfull(ch)`
- **Additional message passing operators**
  - `ch?[x,y]:` polling only
  - `ch?<x,y>:` copy a message without removing it
  - `ch?x(y) == ch?x,y` (for user's understandability)
- Be careful to use these operators inside of expressions
  - They have side-effects, which spin may not allow

# Faulty Data Transfer Protocol
## (pg 27, data switch model proposed at 1981 at Bell labs)

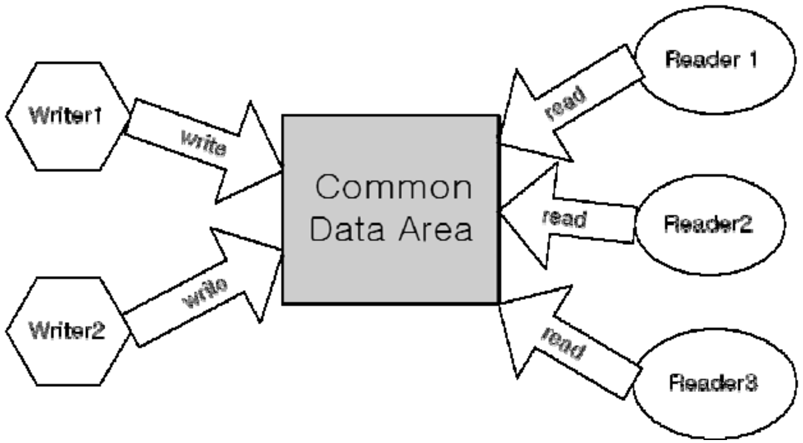
```
mtype={ini,ack, dreq,data, shutup,quiet, dead}
chan M = [1] of {mtype};
chan W = [1] of {mtype};

active proctype Mproc()
{
        W!ini;     /* connection */
        M?ack;  /* handshake */

        timeout ->   /* wait */
        if              /* two options: */
        :: W!shutup; /* start shutdown */
        :: W!dreq; /* or request data */
            do
           :: M?data -> W!data
           :: M?data-> W!shutup;
              break
          od
        fi;
        M?shutup;
        W!quiet;
        M?dead;
}
```

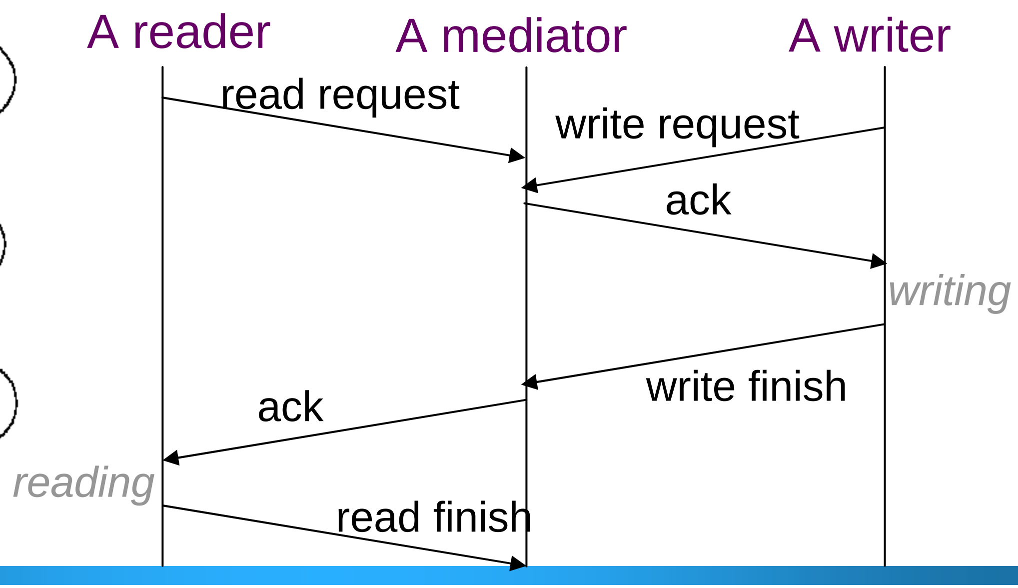



```
active proctype Wproc() {
        W?ini;               /* wait for ini*/
        M!ack;               /* acknowledge */

        do                   /* 3 options: */
        :: W?dreq->          /* data requested */
                   M!data    /* send data */
        :: W?data->          /* receive data   */
                   skip      /* no response */
        :: W?shutup->
                   M!shutup; /* start shutdown*/
                   break
        od;

        W?quiet;
        M!dead;
}
```



- Multiple reader/writer problems
  - Goal: model a system containing 3 readers and 2 writers sharing common data area
    - Common data area has a mediator which receives a read or write request message from a reader or a writer through a request channel
    - The mediator sends back an acknowledge message if a request can be allowed; it a request is not allowable now, the mediator waits to send the ack until the request becomes allowable
      - Hint: a mediator may have a queue to hold requests
    - Once allowed, a reader starts its operation and sends a finish message to the mediator when it finishes



# HW due June 19 (cont.)

- System requirements
  - Concurrency (CON)
    - Multiple readers can read data concurrently
  - Exclusive writing (EW)
    - Only one writer can write into the data area at an instant with no readers
  - High priority of a writer (HPW)
    - A writer's request should have a higher priority than that of a reader
- Prove that your system design satisfies all the system requirements
  - Submit your Promela code and your verification result screenshots