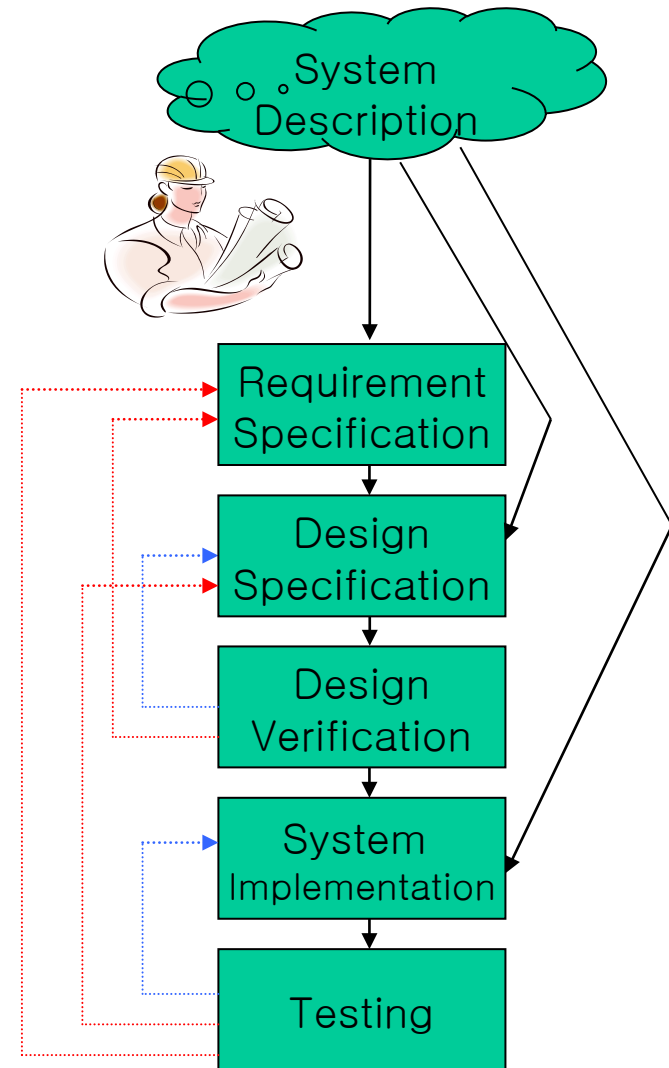


Case Study of Reader/Writer System

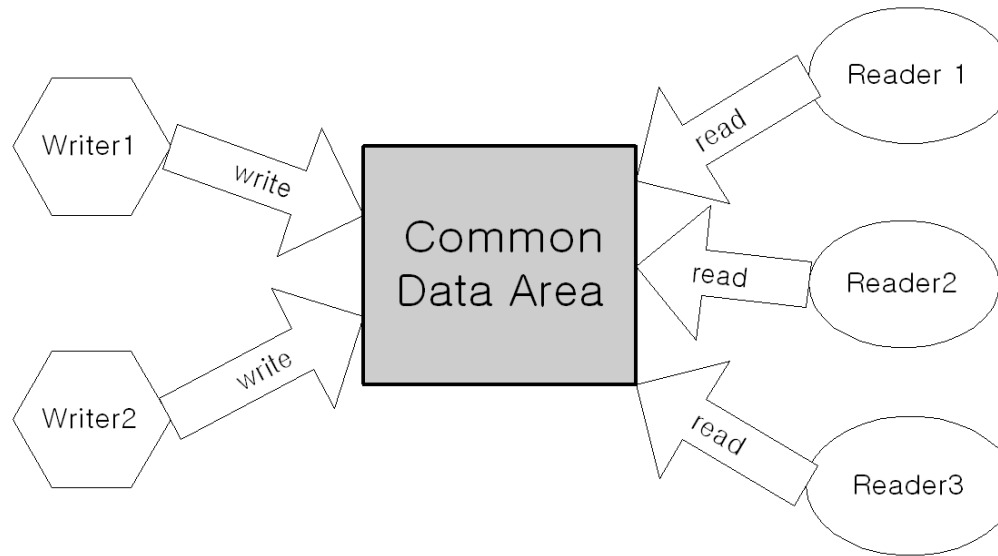
Moonzoo Kim
School of Computing
KAIST



- System Description
- Formal Requirement Specification
- Formal Design Specification
- Formal Verification
- Testing



Multiple Reader/Writer System



■ System requirement

- ✚ Concurrency (CON)
- ✚ Exclusive writing (EW)
- ✚ High priority of writer (HPW)



■ 2 versions of HPW

✚ HPW#1

- While no reader is reading the common data area (CDA), if a writer has tried to write to CDA at the time instance T , no reader should read CDA after T until the writer completes writing.

✚ HPW#2

- While no reader is reading CDA, if a writer has tried to write to CDA at the time instance T and no reader is waiting to read CDA before T , no reader should read CDA after T until the writer completes writing.
(i.e., respecting first-come-first-serve)



■ 1 writer and 2 readers system

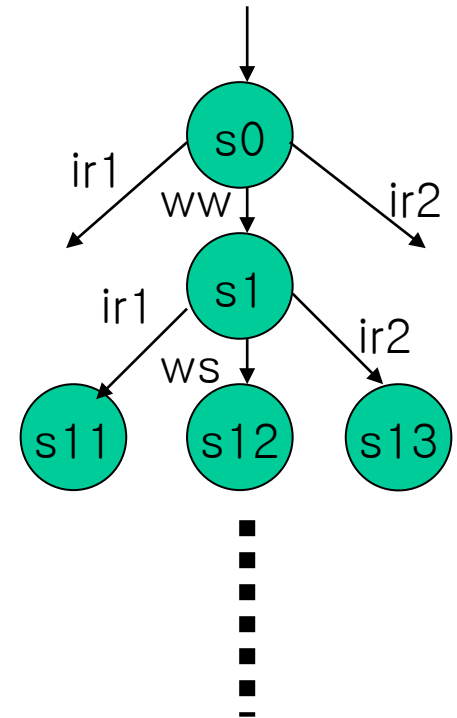
✚ Execution tree

✚ RW system has 9 events

- $\{ir1, rs1, re1, ir2, rs2, re2, ww, ws, we\}$

✚ A state $s = (n_{ir1}, n_{rs1}, n_{ir2}, n_{rs2}, n_{ww}, n_{ws})$

- $s0 = (0, 0, 0, 0, 0, 0)$
- $s1 = (0, 0, 0, 0, 1, 0)$
- $s11 = (1, 0, 0, 0, 1, 0)$
- $s12 = (0, 0, 0, 0, 0, 1)$



Valid execution paths

Defn 1 (An execution path) An execution tree is a labeled transition system (S, T_Σ) where S is a set of states and $T_\Sigma : S \times \Sigma \times S$ is a set of transition over S with a set of label Σ . A state s consists of the following 6 integer variables

$$s \stackrel{def}{=} (n_{ir1}, n_{rs1}, n_{ir2}, n_{rs2}, n_{ww}, n_{ws})$$

An execution path $\sigma = s_0 s_1 \dots s_n$ is a sequence of states in an execution tree. σ_{s_i} denotes the i th state of σ .

Defn 2 (Definition of a state) $\#ir1(\sigma_{s_0}) \stackrel{def}{=} 0$. $\#ir1(\sigma_{s_i}) \stackrel{def}{=} a$ a number of event $ir1$ in an event trace $\rho = l_0 \dots l_{i-1}$ such that $\sigma_{s_i} \xrightarrow{l_i} \sigma_{s_{i+1}}$ where $i > 0$. Similarly defined are $\#rs1, \#re1, \#ir2, \#rs2, \#re2, \#ww, \#ws$, and $\#we$.

Defn 2 (Definition of a state) $\#ir1(\sigma_{s_0}) \stackrel{def}{=} 0$. $\#ir1(\sigma_{s_i}) \stackrel{def}{=} a$ a number of event $ir1$ in an event trace $\rho = l_0 \dots l_{i-1}$ such that $\sigma_{s_i} \xrightarrow{l_i} \sigma_{s_{i+1}}$ where $i > 0$. Similarly defined are $\#rs1, \#re1, \#ir2, \#rs2, \#re2, \#ww, \#ws$, and $\#we$.

State σ_s of an execution path σ consists of the following 6 variables

$$n_{ir1}(\sigma_s) \stackrel{def}{=} \#ir1(\sigma_s) - \#rs1(\sigma_s)$$

$$n_{rs1}(\sigma_s) \stackrel{def}{=} \#rs1(\sigma_s) - \#re1(\sigma_s)$$

$$n_{ir2}(\sigma_s) \stackrel{def}{=} \#ir2(\sigma_s) - \#rs2(\sigma_s)$$

$$n_{rs2}(\sigma_s) \stackrel{def}{=} \#rs2(\sigma_s) - \#re2(\sigma_s)$$

$$n_{ww}(\sigma_s) \stackrel{def}{=} \#ww(\sigma_s) - \#ws(\sigma_s)$$

$$n_{ws}(\sigma_s) \stackrel{def}{=} \#ws(\sigma_s) - \#we(\sigma_s)$$

$$\text{Initial state } \sigma_{s_0} \stackrel{def}{=} (0, 0, 0, 0, 0, 0)$$

$n_{ir1}(\sigma_s)$ indicates whether there is “active” $ir1$ in an execution path $s_0 \dots s$. We can think that i th occurrence of $rs1$ “cancels” the i th occurrence of $ir1$. $n_{ir1}(\sigma_s) = 1$ means that $ir1$ occurs i times and $rs1$ occurs $(i - 1)$ times upto state σ_s , which means that $ir1$ is “active”.



Valid execution path σ

- ✦ An execution path $\sigma = s_0. s_1 \dots s_n$
 - σ_{s_i} denotes the i th state of σ

✦ Definition of a state s_i .

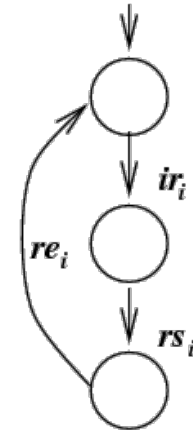
- $\#ir1(\sigma_{s_0}) = 0$
- $\#ir1(\sigma_{s_i}) = \# \text{ of } ir1 \text{ in a trace } l_0 \dots l_{i-1} \text{ s.t. } \sigma_{s_i} - l_i \rightarrow \sigma_{s_{i+1}}$
- $n_{ir1}(\sigma_s) = \#ir1(\sigma_s) - \#rs1(\sigma_s)$
- $n_{rs1}(\sigma_s) = \#rs1(\sigma_s) - \#re1(\sigma_s)$
- $n_{ir2}(\sigma_s), n_{rs2}(\sigma_s), n_{ww}(\sigma_s), n_{ws}(\sigma_s)$ are defined similarly



Valid execution path σ

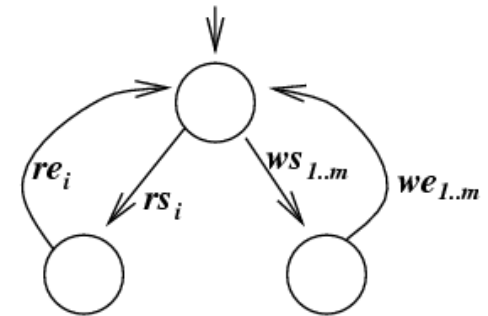
+ Correct Event Ordering

- For all state s_i in σ
 - $n_{ir1}(s_i) \geq 0 \wedge n_{rs1}(s_i) \geq 0 \wedge n_{ir1}(s_i) + n_{rs1}(s_i) \leq 1$
 - $n_{ir2}(s_i) \geq 0 \wedge n_{rs2}(s_i) \geq 0 \wedge n_{ir2}(s_i) + n_{rs2}(s_i) \leq 1$
 - $n_{ww}(s_i) \geq 0 \wedge n_{ws}(s_i) \geq 0 \wedge n_{ww}(s_i) + n_{ws}(s_i) \leq 1$



+ Exclusive Writing

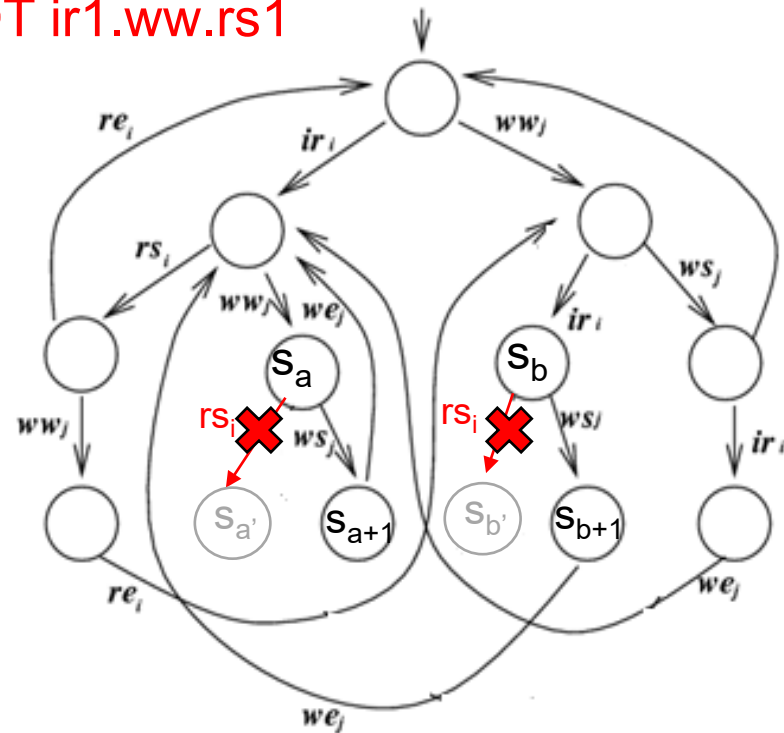
- For all state s_i in σ
 - $n_{ws}(s_i) = 1 \rightarrow (n_{rs1}(s_i) = 0 \wedge n_{rs2}(s_i) = 0)$



Valid execution paths

+ High Priority of Writer #1 (HPW#1)

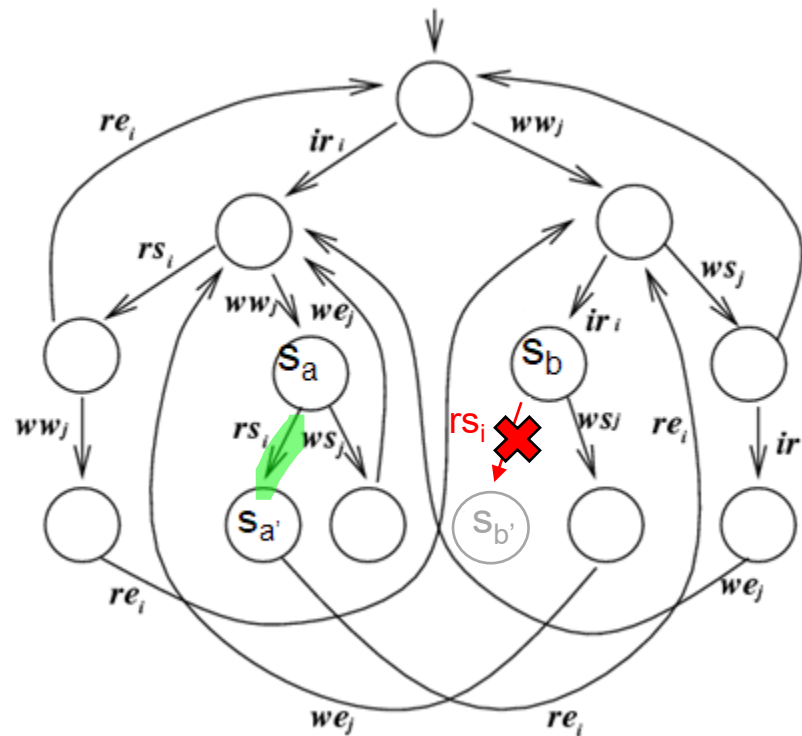
- $(n_{ww}(s_i)=1 \wedge n_{rs1}(s_i)=0 \wedge n_{rs2}(s_i)=0)$
 $\rightarrow (n_{rs1}(s_{i+1})=0 \wedge n_{rs2}(s_{i+1})=0)$
- Ex. HPW#1 allow $ir1.ww.ws$, but **NOT** $ir1.ww.rs1$
- Ex1. For a state s_a in the right LTS,
 - $s_a -ws_j \rightarrow s_{a+1}$ is valid, since
 $n_{ww}(s_a)=1, n_{rs1}(s_a)=0, n_{rs2}(s_i)=0,$
 $n_{rs1}(s_{a+1})=0, n_{rs2}(s_{a+1})=0$
- Ex2. For a state s_a in the right LTS,
 - $s_a -rs_i \rightarrow s_{a'}$ is **NOT** valid, since
 $n_{ww}(s_a)=1, n_{rs1}(s_a)=0, n_{rs2}(s_i)=0,$
 $n_{rs1}(s_{a'})=1, n_{rs2}(s_{a'})=0$



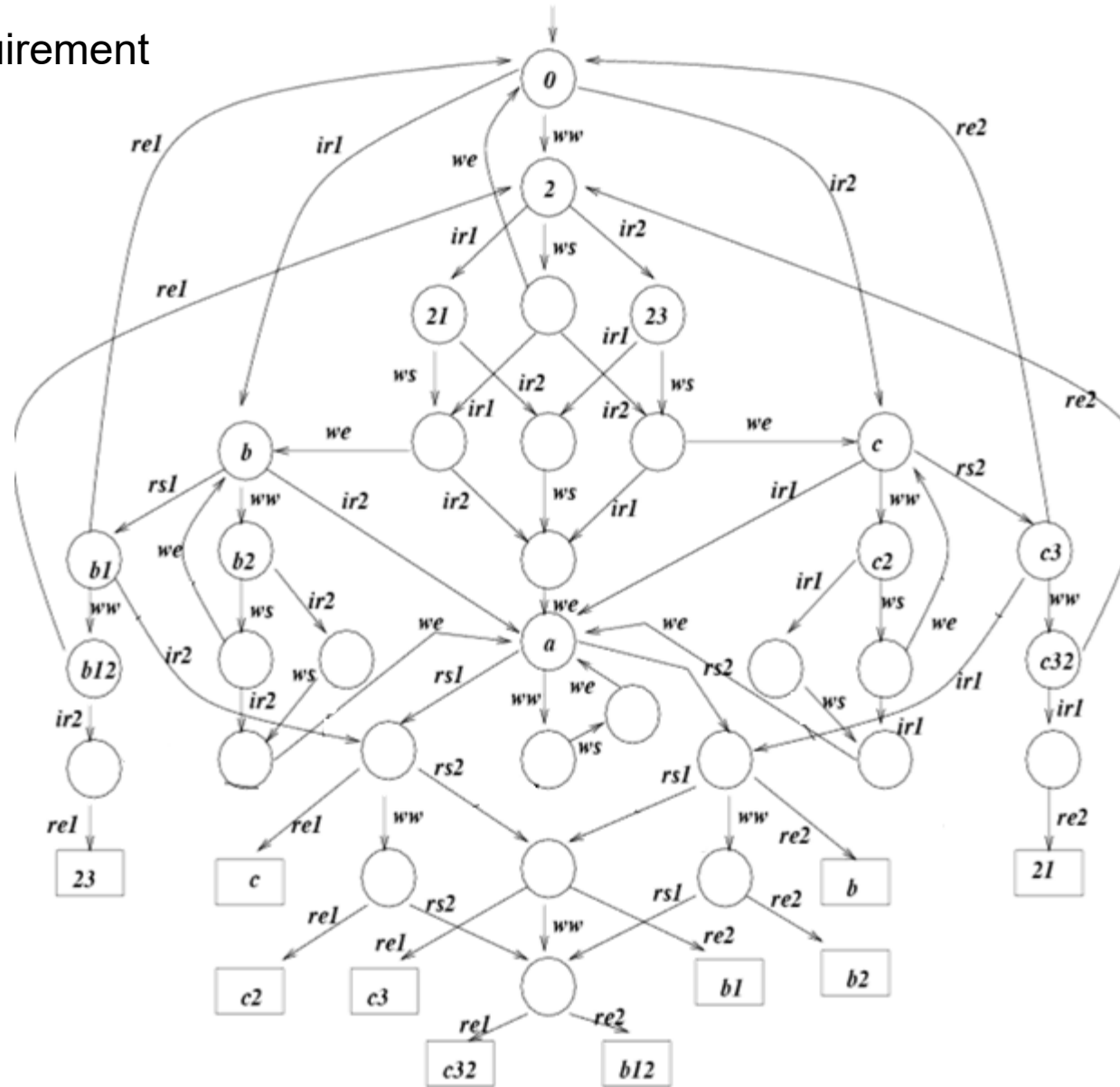
Valid execution paths

High Priority of Writer #2 (HPW#2)

- Difficult to formally specify HPW#2 in the given formal framework since we need to specify an order of events in a trace
 - Ex. we need to distinguish **ir1.ww.rs1** and **ww.ir1.rs1**



LTS for the Requirement
w/ HPW#1



* Requirement Specification w/ HPW#1 *

```
proc ReqRW_HPW1 = ir1.B + ww.S2 + ir2.C
proc S2 = ir1.S21 + ws.S22 + ir2.S23
proc S21 = ws.S212 + ir2.S213
proc S22 = ir1.S212 + we.ReqRW_HPW1 + ir2.S232
proc S23 = ir1.S213 + ws.S232
proc S212 = we.B + ir2.S2123
proc S213 = ws.S2123
proc S232 = ir1.S2123 + we.C
proc S2123 = we.A

proc A = rs1.A1 + ww.A2 + rs2.A3
proc A1 = re1.C + ww.A12 + rs2.A13
proc A2 = ws.we.A
proc A3 = rs1.A13 + ww.A32 + re2.B
proc A12 = re1.C2 + rs2.A123
proc A13 = re1.C3 + ww.A123 + re2.B1
proc A32 = rs1.A123 + re2.B2
proc A123 = re1.C32 + re2.B12

proc B = rs1.B1 + ww.B2 + ir2.A
proc B1 = re1.ReqRW_HPW1 + ww.B12 + ir2.A1
proc B2 = ws.B22 + ir2.B23
proc B12 = re1.S2 + ir2.B123
proc B22 = we.B + ir2.B223
proc B23 = ws.B223
proc B123 = re1.S23
proc B223 = we.A

proc C = ir1.A + ww.C2 + rs2.C3
proc C2 = ir1.C21 + ws.C22
proc C3 = ir1.A3 + ww.C32 + re2.ReqRW_HPW1
proc C21 = ws.C221
proc C22 = ir1.C221 + we.C
proc C32 = ir1.C321 + re2.S2
proc C221 = we.A
proc C321 = re2.S21
```



Formal Design Specification

- RW system designed in “Concurrent Programming in Java[Lea99]”
- `proc S =`
`(R1|R2|W|AR0|WW0|AW0|`
`LOCK|SLEEP0)\`
`{ dec_WW, inc_WW, dec_AW, inc_AW, ... }`
`proc R1 = ...`
 - Processes (R1, R2, W, Lock, etc)
communicate each other through signals
(`dec_WW`, `inc_WW`, etc)
 - variables in RW code are represented as
processes (AR0, AW0, etc)

```
class RW {  
    int activeReaders_ = 0;  
    int activeWriters_ = 0;  
    int waitingReaders_ = 0;  
    int waitingWriters_ = 0;  
  
    void read() {  
        beforeRead();  
        read_();  
        afterRead();  
    }  
  
    void beforeRead()    {...}  
    void read_()         {...}  
    void afterRead()     {...}  
    ...  
}
```



Testing using Formal Specification

- Insert probe into the RW source code
 - ✚ Probe generates event signal
- Testing RW code utilizing formal requirement spec as a test oracle
 - ✚ Use CWB-NC based simulation feature
 - ✚ Inappropriate event signal means violation

```
public abstract class RW{
    protected int activeReaders_ = 0;
    protected int activeWriters_ = 0;
    protected int waitingReaders_ = 0;
    protected int waitingWriters_ = 0;

    public void read(String id) {
        beforeRead();
        read_(id);
        afterRead();
    }

    protected synchronized void beforeRead(){
        Event("ir" + pid);
        ... }

    public void read_() {
        Event("rs" + pid);
        ... }

    protected synchronized void afterRead(){
        Event("re" + pid);
        ... }

    ... }
```



```
public abstract class RW2 {
    protected int activeReaders_ = 0; //threads executing read_
    protected int activeWriters_ = 0; //always 0 or 1
    protected int waitingReaders_ = 0; //threads not yet in read_
    protected int waitingWriters_ = 0; //same for write_

    protected abstract void read_(String id);
    protected abstract void write_(String id);

    void Event(String s){ //System.out.println(s);}

    public void read(String id) {
        beforeRead();
        read_(id); // Event("rs" + pid);
        afterRead();
    }

    public void write(String id) {
        beforeWrite();
        write_(id); // Event("ws"+ pid);
        afterWrite();
    }

    protected boolean allowReader() {
        if (waitingWriters_ == 0 && activeWriters_ == 0) {
            return true;
        }
        else
            return false;
    }
}
```

```
protected boolean allowWriter() {
    if(activeReaders_ == 0 && activeWriters_ == 0) {
        return true;
    } else return false; }

protected synchronized void beforeRead() {
    Event("ir" + pid);
    ++waitingReaders_;
    while(!allowReader())
        try{ wait();}
        catch(InterruptedException ex){}
    --waitingReaders_;
    ++activeReaders_;}

protected synchronized void afterRead() {
    Event("re" + pid);
    --activeReaders_;
    notifyAll();}

protected synchronized void beforeWrite() {
    Event("ww" + pid);
    ++waitingWriters_;
    while(!allowWriter())
        try{wait();}
        catch(InterruptedException ex){}
    --waitingWriters_;
    ++activeWriters_;}

protected synchronized void afterWrite() {
    Event("we" + pid);
    --activeWriters_;
    notifyAll(); }
```



RW System Design

* RW system design of 2 Readers and 1 Writer *

* matching to the Java Implementation *

```
proc DesignRW = (R1|R2|W|AR0|WW0|AW0|LOCK|SLEEP0)\
  {dec_WW, inc_WW, dec_AW, inc_AW, dec_AR, inc_AR,
   zero_WW, zero_AW, zero_AR, non_zero_WW, non_zero_AW,
   non_zero_AR, lock, unlock,
   zero_sleep, one_sleep, two_sleep, dec_sleep, inc_sleep,
   wake_up}
```

```
proc WW0 = zero_WW.WW0 + inc_WW.WW1
proc WW1 = dec_WW.WW0 + non_zero_WW.WW1
```

```
proc AW0 = zero_AW.AW0 + inc_AW.AW1
proc AW1 = dec_AW.AW0 + non_zero_AW.AW1
```

```
proc AR0 = zero_AR.AR0 + inc_AR.AR1
proc AR1 = dec_AR.AR0 + inc_AR.AR2
  + non_zero_AR.AR1
proc AR2 = dec_AR.AR1 + non_zero_AR.AR2
```

```
proc SLEEP0 = zero_sleep.SLEEP0 + inc_sleep.SLEEP1
proc SLEEP1 = one_sleep.SLEEP1 + inc_sleep.SLEEP2 + dec_sleep.SLEEP0
proc SLEEP2 = two_sleep.SLEEP2 + dec_sleep.SLEEP1
```

```
proc R1 = 'lock.ir1.
  ( 'zero_WW.
    ('zero_AW.'inc_AR.'unlock.READ1
     + 'non_zero_AW.'inc_sleep.'unlock.R1')
    + 'non_zero_WW.'inc_sleep.'unlock.R1')
proc R1' = wake_up.'lock.
  ( 'zero_WW.
    ('zero_AW.'inc_AR.'unlock.READ1
     + 'non_zero_AW.'inc_sleep.'unlock.R1')
    + 'non_zero_WW.'inc_sleep.'unlock.R1')
```

```
proc R2 = 'lock.ir2.
  ( 'zero_WW.
    ('zero_AW.'inc_AR.'unlock.READ2
     + 'non_zero_AW.'inc_sleep.'unlock.R2')
    + 'non_zero_WW.'inc_sleep.'unlock.R2')
proc R2' = wake_up.'lock.
  ( 'zero_WW.
    ('zero_AW.'inc_AR.'unlock.READ2
     + 'non_zero_AW.'inc_sleep.'unlock.R2')
    + 'non_zero_WW.'inc_sleep.'unlock.R2')
```

```
proc W = 'lock.ww.'inc_WW.
  ( 'zero_AR.
    ('zero_AW.'dec_WW.'inc_AW.'unlock.WRITE
     + 'non_zero_AW.'inc_sleep.'unlock.W')
    + 'non_zero_AR.'inc_sleep.'unlock.W')
proc W' = wake_up.'lock.
  ( 'zero_AR.
    ('zero_AW.'dec_WW.'inc_AW.'unlock.WRITE
     + 'non_zero_AW.'inc_sleep.'unlock.W')
    + 'non_zero_AR.'inc_sleep.'unlock.W')
```

```
proc READ1 = rs1.re1.'lock.'dec_AR.
  ('zero_sleep.'unlock.R1 +
   'one_sleep.'wake_up.'dec_sleep.'unlock.R1 +
   'two_sleep.'wake_up.'dec_sleep.'wake_up.'dec_sleep.'unlock.R1)
proc READ2 = rs2.re2.'lock.'dec_AR.
  ('zero_sleep.'unlock.R2+
   'one_sleep.'wake_up.'dec_sleep.'unlock.R2+
   'two_sleep.'wake_up.'dec_sleep.'wake_up.'dec_sleep.'unlock.R2)
proc WRITE = ws.we.'lock.'dec_AW.
  ('zero_sleep.'unlock.W +
   'one_sleep.'wake_up.'dec_sleep.'unlock.W +
   'two_sleep.'wake_up.'dec_sleep.'wake_up.'dec_sleep.'unlock.W)
```

```
proc LOCK = lock.unlock.LOCK
```



■ May preorder (classical trace inclusion)

■ $P \cdot_{\text{may}} Q$ iff on $T'(P) \mu T'(Q)$

- Ex. $\text{le } -S \text{ may "a.nil" "a.b.nil"}$
 - Since $T'(\text{a.nil}) = \{a\}$, $T'(\text{a.b.nil}) = \{a, a.b\}$
 - But **not** $\text{le } -S \text{ may "a.b.nil" "a.nil"}$



Formal Verification Result

```
cwb-nc>
le -S may DesignRW ReqRW_HP W1
Building automaton...
.....
States: 620
Transitions: 1016
Done building automaton.
Building automaton...
States: 34
Transitions: 69
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
DesignRW has trace:
      ir1 ww rs1
ReqRW_HP W1 does not.
```

Does ReqRW_HP W2 allow ir1.ww.rs1?

Can DesignRW perform ww.ir1.rs1?
Does ReqRW_HP W1 allow it?
Does ReqRW_HP W2 allow it?

DesignRW violates ReqRW_HP W1, because,
(as a counter example **ir1.ww.rs1** indicates)
R1 can read before W writes
if R1 performs ir1 before W performs ww.

See the following Java code for which
DesignRW was specified:

```
protected synchronized void beforeRead() {
    Event("ir" + pid);
    ++waitingReaders_;
    while(!allowReader())
        try{ wait();}
        catch(InterruptedException ex){}
    --waitingReaders_;
    ++activeReaders_;
}

protected boolean allowWriter() {
    if(activeReaders_ == 0 && activeWriters_ == 0) {
        return true;
    } else return false;
}
```

