Formal Semantics of CCS

Moonzoo Kim

School of Computing, KAIST

Review of the Previous Class

- Sequential system v.s. Reactive system
  - Ex1. Mathematical functions with given inputs generate outputs
    - Usually no environment consideration and timing consideration.
  - Ex2. Ad-hoc On-Demand Vector routing protocol
    - Should model multiple concurrent nodes (environment)
    - Should model communication among the nodes
    - Should model timely behavior (e.g. time-out, etc)
- Modeling of a complex system
  - Concurrency => interleaving semantics
  - Communication => synchronization
  - Hierarchy => refinement

Process Algebra

- A process algebra consists of
  - a set of operators and syntactic rules for constructing processes
  - a semantic mapping which assigns meaning or interpretation to every process
  - a notion of equivalence or partial order between processes
- Advantages: A large system can be broken into simpler subsystems and then proved correct in a modular fashion. Also, correctness can be checked
  - A hiding or restriction operator allows one to abstract away unnecessary details.
  - Equality for the process algebra is also a congruence relation; and thus, allows the substitution of one component with another equal component in large systems.

Notations (1/2)

- A system is described as a set of communicating processes
  - Each process executes a sequence of actions
  - Actions represents either inputs/outputs or internal computation steps

- A set of actions/events $Act = L \cup L' \cup \{\tau\}$
  - L ={a,b,…} is a set of *names* and L' ={a',b',…} is a set of *co-names*
    - a $\in L$ can be considered as the act of receiving a signal
    - a' $\in L'$ can be considered as the act of emitting a signal
    - τ is a special action to represent internal hidden action
  - $Act - \{\tau\}$ represents the set of externally visible actions:

- Operational (transitional) semantics of CCS process
  - Define the "execution steps" that processes may engaged in
  - P –a-> P' holds if a process P is capable of engaging in action a and then behaving like P'
  - Define –a-> inductively using inference rules for operators
    - premises

      -------------- *(side condition)*

      conclusion

Example 1:

$$\text{Choice}_R \quad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

Example 2:

$$\text{Prefix} \quad \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

Operators for Sequential Process

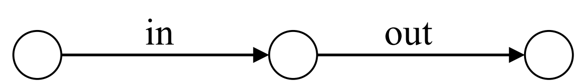
The idea: 7 elementary ways of producing or putting together labelled transition systems

1.Nil 0 No transitions (deadlock)

Prefix Prefix

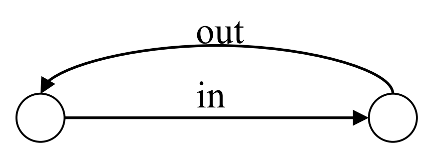
2.Prefix $\alpha.P$ ($\alpha \in Act$) in.out'.0 –*in*-> out'.0 –*out'*-> 0

$$\text{Prefix}\ \frac{\text{(empty)}}{\alpha.P \text{ –}\alpha\text{-> } P}$$



3.Defn $A = P$ Buffer = in.out'.Buffer

Buffer-*in*->out'.Buffer-*out'*->Buffer



Operators for Sequential Process (cont.)

4.Choice *P* + *Q*

BadBuf = in.(τ.0 + out.BadBuf)

Prefix

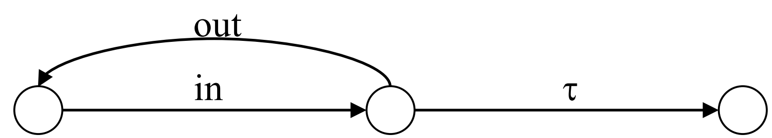
BadBuf –*in*-> τ.0 + out.BadBuf

Choice$_L$ Choice$_R$

-*τ*-> 0 **or** –*out*-> BadBuf

$$\text{Choice}_L \quad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$$

$$\text{Choice}_R \quad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$



Obs: No priorities between τ's, a's or a's !

May use Σ notation to comactly represent sequential process

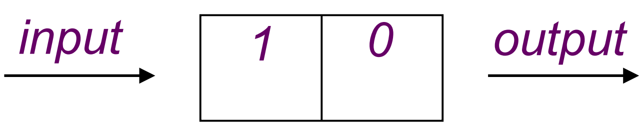
$$P = \sum_{i \in I} \alpha_i . P_i$$

Example: Boolean Buffer of Size 2

Action and Process Def.

in_0 :0 is coming as input
in_1 :1 is coming as input
out_0 :0 is going out as output
out_1 :1 is going out as output

Buf^2 : Empty 2-place buffer
Buf^2_0 : 2-place buffer holding 0
Buf^2_{01}: 2-place buffer holding 0 at head and 1 at tail



Buf^2_{01}

$$Buf^2 = in_0.Buf^2_0 + in_1.Buf^2_1$$

$$Buf^2_0 = out_0.Buf^2 + in_0.Buf^2_{00} + in_1.Buf^2_{01}$$

$$Buf^2_1 = out_1.Buf^2 + in_0.Buf^2_{10} + in_1.Buf^2_{11}$$

$$Buf^2_{00} = out_0.Buf^2_0$$

$$Buf^2_{01} = out_0.Buf^2_1$$

$$Buf^2_{10} = out_1.Buf^2_0$$

$$Buf^2_{11} = out_1.Buf^2_1$$

Operators for Concurrent Process

5. Composition

$$\mathrm{Par}_L \quad \frac{P \text{ -}\alpha\text{-> } P'}{P|Q \text{ -}\alpha\text{-> } P'|Q}$$

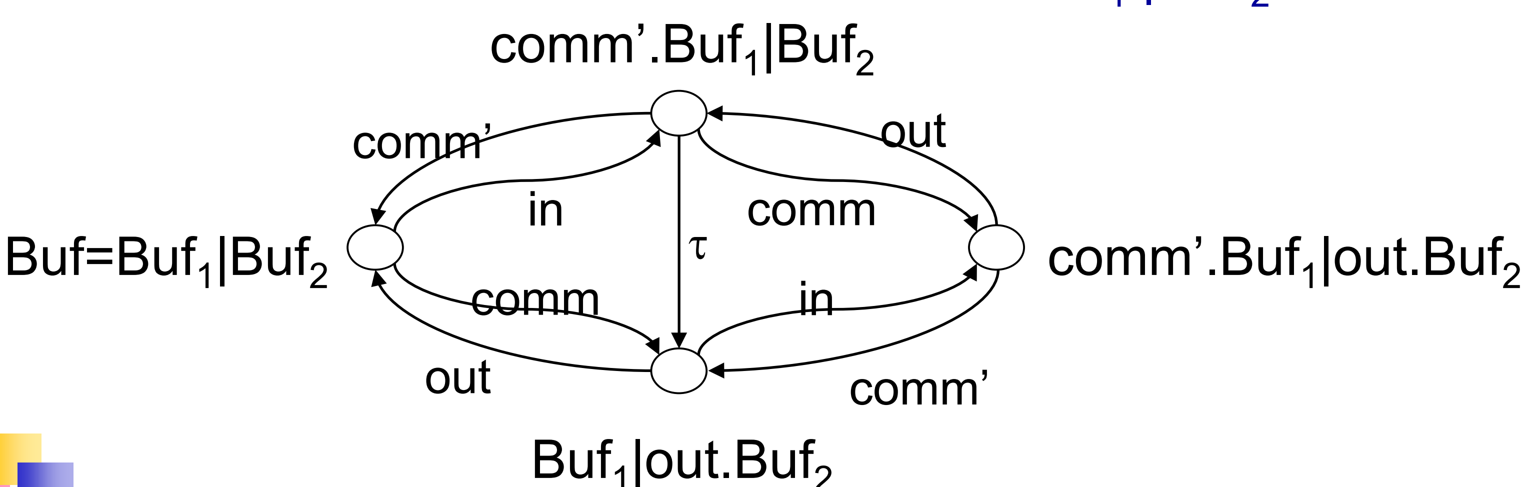
$$\mathrm{Par}_R \quad \frac{Q \text{ -}\alpha\text{-> } Q'}{P|Q \text{ -}\alpha\text{-> } P|Q'}$$

$$\mathrm{Par}\tau \quad \frac{P\text{-}a\text{->}P',\ Q\text{–}a'\text{->}Q'}{P|Q \text{ -}\tau\text{-> } P'|Q'}$$

Buf_1 = in.comm'.Buf_1
Buf_2 = comm.out.Buf_2
Buf = Buf_1 | Buf_2

Buf
Par_L: -*in*-> comm'.Buf_1 | Buf_2
Par_τ: -τ> Buf_1 | out.Buf_2
Par_R: -*out*-> Buf_1 | Buf_2

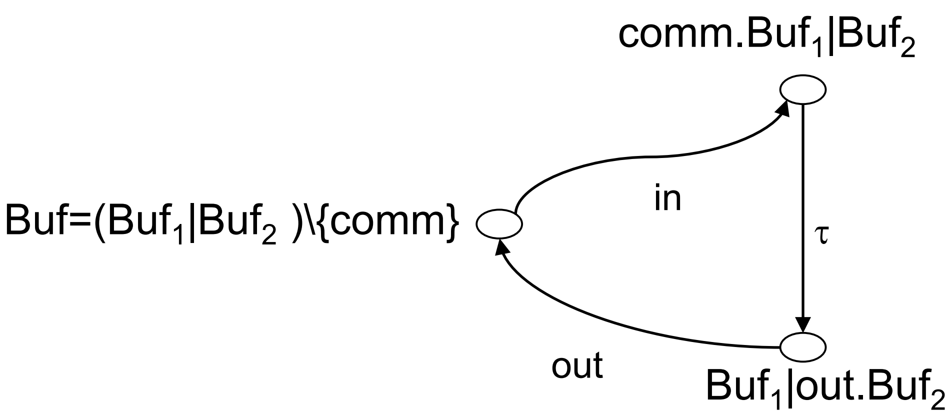
Buf
Par_R: -*comm*-> Buf_1 | out.Buf_2
Par_R: -*out*-> Buf_1 | Buf_2



6. Restriction *P\L*

$$\text{Res} \quad \frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L} \quad \alpha \notin L \cup L'$$

$Buf_1 = in.comm.Buf_1$
$Buf_2 = comm'.out.Buf_2$
$Buf = (Buf_1 \mid Buf_2)\backslash\{comm\}$



Buf
-*in*-> (comm.Buf$_1$ | Buf$_2$)\{comm}
-*τ*-> (Buf$_1$ | out.Buf$_2$)\{comm}
-*out*-> (Buf$_1$ | Buf$_2$)\{comm}

Buf
~~-*comm'*-> Buf$_1$ | out.Buf$_2$~~

(Buf1 | Buf2)\{comm} : a design for buffer with separated input/output ports
ReqBuf = in.out.ReqBuf : a requirement for buffer design
(Buf1 | Buf2)\{comm} == ReqBuf means that buffer design satisfies the requirement

Operators for Concurrent Process (cont.)

7. Relabelling ***P*[*f*]**

$$\text{Rel}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$Buf = in.out.Buf$

$Buf_1 = Buf[comm/out]$

$\quad = in.comm.Buf_1$

$Buf_2 = Buf[comm'/in]$

$\quad = comm'.out.Buf_2$

Relabelling function *f* must preserve complements:

$f(a') = f(a)'$

Relabelling function often given by name substitution as above

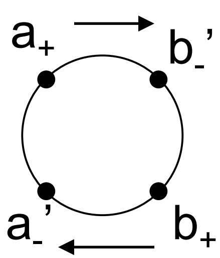
Example: 2-way Buffers

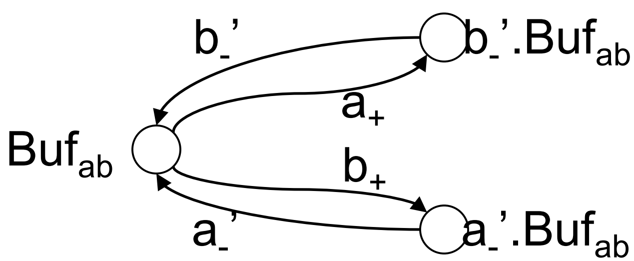
1-place 2-way buffer:

$Buf_{ab} = a_+.b_-'.Buf_{ab} + b_+.a_-'.Buf_{ab}$

$Buf_{bc} = b_-.c_-'.Buf_{bc} + c_+.b_+'.Buf_{bc}$

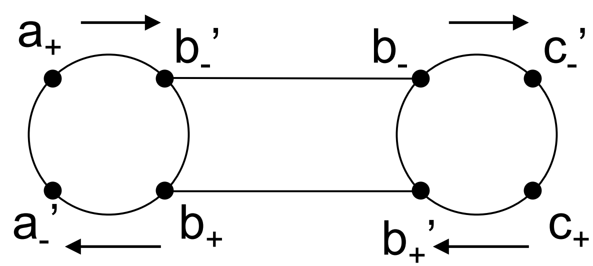
LTS:





$Buf_{bc} =$
$Buf_{ab}[c_+/b_+, c_-/b_-, b_-/a_+, b_+/a_-]$
(Obs:simultaneous substitution!)

$Sys = (Buf_{ab} \mid Buf_{bc})\backslash\{b_+, b_-\}$



But what's wrong? Deadlock occurs

In other words, $Sys == Buf_{ac}$?

Summary of CCS Semantics

$$\textit{Act}\quad \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

in.P -in-> P

$$\textit{Choice}_L\quad \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \textit{Choice}_R\quad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

in.P + out.Q -in-> P *or* –out-> Q

$$\textit{Par}_L\quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \textit{Par}_R\quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

in.P|in'.Q -in->P|in'.Q *or* –in'-> in.P|Q

$$\textit{Par}\tau\quad \frac{P \xrightarrow{a} P',\ Q \xrightarrow{a'} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

in.P | in'.Q -τ-> P|Q

$$\text{Res}\quad \frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\ \alpha \notin L \cup L'$$

(in.P | in'.Q)\{in} -τ-> (P|Q)\{in} only

$$\text{Rel}\quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

in.P [out/in] -out-> P[out/in]

Inference of Process Execution

Proof of ((a.E + b.0)| a'.F)\{a} -τ-> (E|F)\{a}

Act ------------------

a.E –a-> E

Choice_L -------------------------- Act ------------------

(a.E + b.0) -a-> E a'.F –a'-> F

Parτ ---

(a.E + b.0)| a'.F -τ-> (E|F)

Res ---

((a.E + b.0)| a'.F)\{a} -τ-> (E|F)\{a}

- Derive following process execution from the inference rules
  - (a.E + b.0) | a'.F –a-> E | a'.F
  - (a.E + b.0) | a'.F –a'-> (a.E + b.0) | F
  - (a.E + b.0) | a'.F –b-> 0 | a'.F
  - ((a.E + b.0) | a'.F)\{a} –b-> (0 |a'.F)\{a}
- Draw corresponding labeled transition diagrams
  - (a.E + b.0) | a'.F
  - ((a.E + b.0) | a'.F)\{a}
  - A = a.c'.A, B = c.b'.B
    - A|B, (A|B)\{c}

Proof 1

$$\text{Prefix}\;\frac{}{a.E \;\text{–a->}\; E}$$

$$\text{Choice}_L\;\frac{}{(a.E + b.0))\;\text{–a->}\;E}$$

$$\text{Par}_L\;\frac{}{\mathbf{(a.E + b.0) \mid a'.F \;\text{–a->}\; E \mid a'.F}}$$

Proof 2

$$\text{Prefix}\;\frac{}{a'.F \;\text{–a'->}\; F}$$

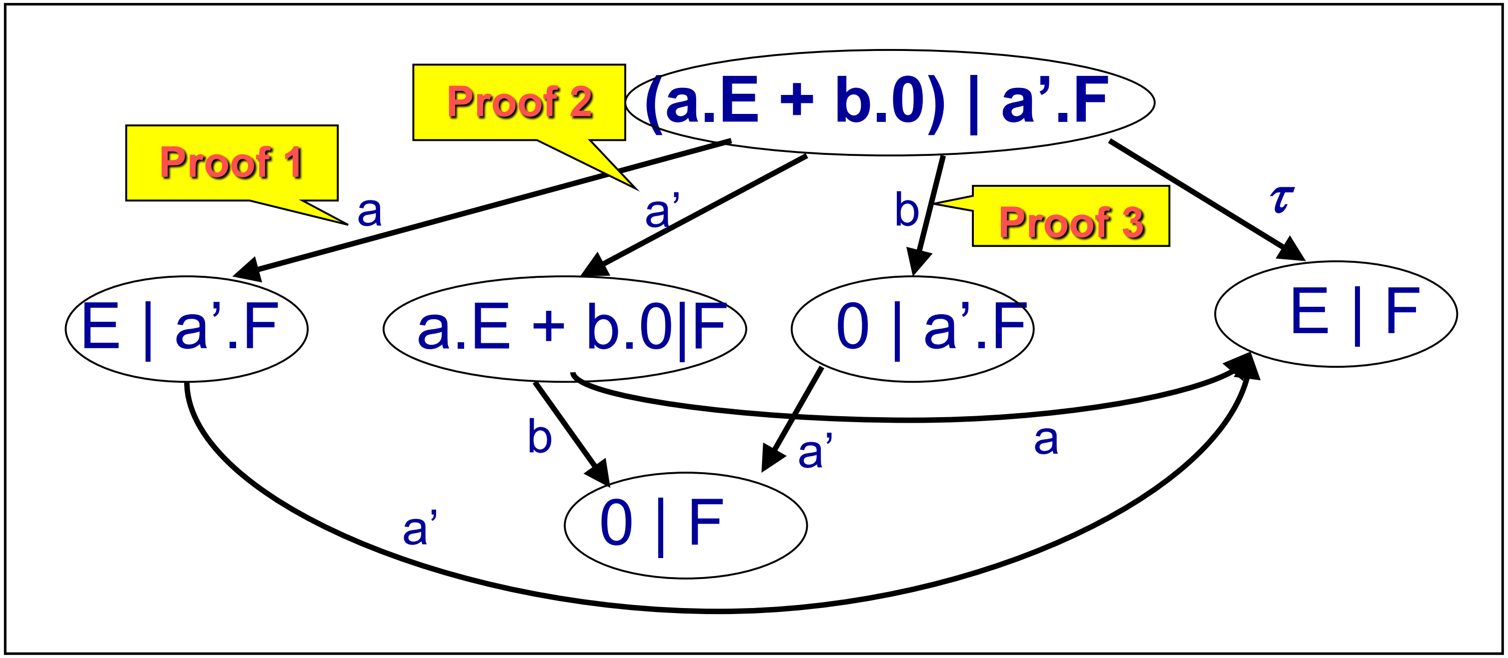
$$\text{Par}_R\;\frac{}{\mathbf{(a.E + b.0) \mid a'.F \;\text{–a'->}\; (a.E + b.0) \mid F}}$$

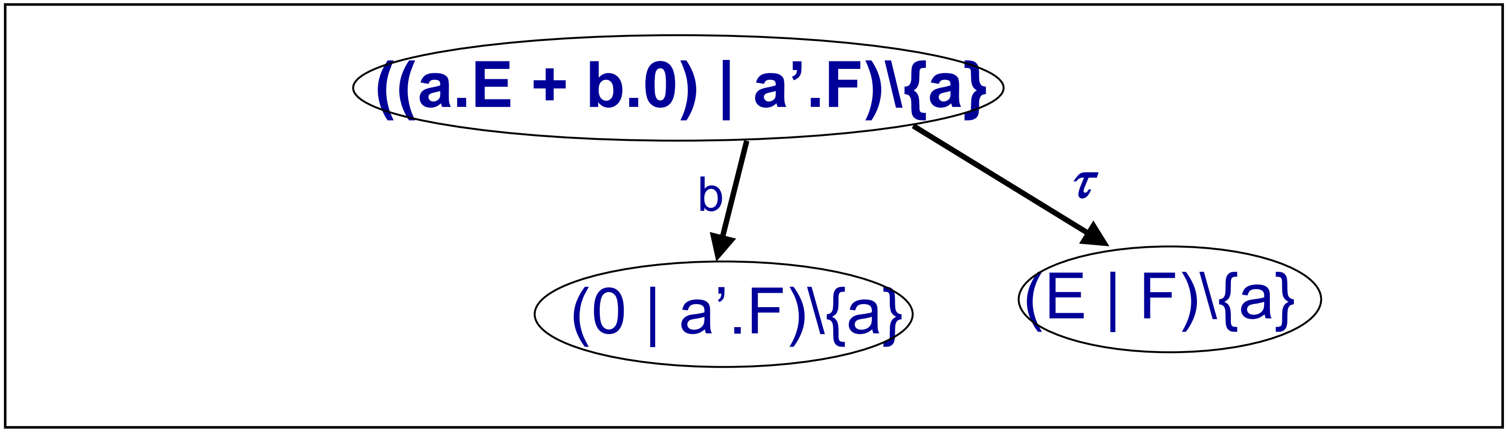
Proof 3

$$\text{Prefix}\;\frac{}{b.0 \;\text{–b->}\; 0}$$

$$\text{Choice}_R\;\frac{}{(a.E + b.0)\;\text{–b->}\;0}$$

$$\text{Par}_L\;\frac{}{\mathbf{(a.E + b.0) \mid a'.F \;\text{–b->}\; 0 \mid a'.F}}$$


(a.E + b.0) | a'.F
Proof 1
Proof 2
Proof 3
a
a'
b
τ
E | a'.F
a.E + b.0|F
0 | a'.F
E | F
b
a'
a
a'
0 | F


((a.E + b.0) | a'.F)\{a}
b
τ
(0 | a'.F)\{a}
(E | F)\{a}

Example: Faulty Mutual Exclusion Protocol

```
byte cnt, byte x,y,z;
active[2] proctype user()
{       byte me = _pid +1; /* me is 1 or 2*/
again:
        x = me;
        If
        :: (y ==0 || y== me) -> skip
        :: else -> goto again
        fi;

        z =me;
        If
        :: (x == me) -> skip
        :: else -> goto again
        fi;

        y=me;
        If
        :: (z==me) -> skip
        :: else -> goto again
        fi;

        /* enter critical section */
        cnt++
        assert( cnt ==1);
        cnt --;
        goto again
}
```

```
proc Sys = (P1|P2|X0|Y0|Z0|CNT0)\{x_[0-2],y_[0-2],z_[0-2],
test_x_[0-2],test_y_[0-2],test_z_[0-2], inc_cnt,dec_cnt}

proc P1    = x_1.(test_y_0.P1' + test_y_1.P1' + test_y_2.P1)
proc P1'   = z_1.(test_x_0.P1  + test_x_1.P1'' + test_x_2.P1)
proc P1''  = y_1.(test_z_0.P1  + test_z_1.P1''' + test_z_2.P1)
proc P1''' = inc_cnt.dec_cnt.P1

proc P2    = x_2.(test_y_0.P2' + test_y_1.P2 + test_y_2.P2')
proc P2'   = z_2.(test_x_0.P2  + test_x_1.P2 + test_x_2.P2'')
proc P2''  = y_2.(test_z_0.P2  + test_z_1.P2 + test_z_2.P2''')
proc P2''' = inc_cnt.dec_cnt.P2

* Variable x, y,z, and cnt
proc UpdateX = 'x_0.X0 + 'x_1.X1 + 'x_2.X2
proc X0 = 'test_x_0.X0 + UpdateX
proc X1 = 'test_x_1.X1 + UpdateX
proc X2 = 'test_x_2.X2 + UpdateX

proc UpdateY = 'y_0.Y0 + 'y_1.Y1 + 'y_2.Y2
proc Y0 = 'test_y_0.Y0 + UpdateY
proc Y1 = 'test_y_1.Y1 + UpdateY
proc Y2 = 'test_y_2.Y2 + UpdateY

proc UpdateZ = 'z_0.Z0 + 'z_1.Z1 + 'z_2.Z2
proc Z0 = 'test_z_0.Z0 + UpdateZ
proc Z1 = 'test_z_1.Z1 + UpdateZ
proc Z2 = 'test_z_2.Z2 + UpdateZ

proc CNT0 = 'inc_cnt.cnt_1.CNT1
proc CNT1 = 'inc_cnt.cnt_2.CNT2 + 'dec_cnt.cnt_0.CNT0
proc CNT2 = 'dec_cnt.cnt_1.CNT1

proc Spec = cnt_1.cnt_0.Spec
```

Homework #1: Due Sep 21

- Draw LTS diagrams
  - Buf^2 in the slide 6
  - Sys in the slide 10 (specify which two actions make τ if any)
- Minimize Sys of slide 14 by using relabelling functions
- Specify Peterson's mutual exclusion protocol for 2 processes

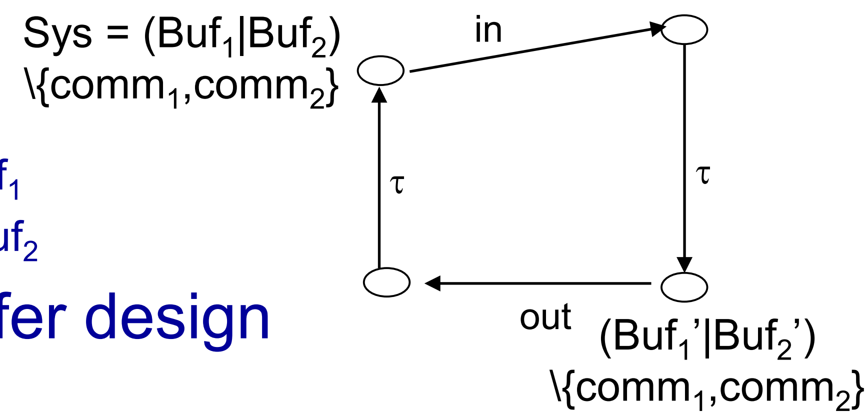
```
/* Peterson's solution to the mutual exclusion problem - 1981 */
boolean turn, flag[2];
byte ncrit;
active [2] proctype user(){
again:    flag[_pid] = 1;
          turn = _pid;
          while(!(flag[1 - _pid] == 0 || turn == 1 - _pid));

          ncrit++;
          assert(ncrit == 1);          /* critical section */
          ncrit--;

          flag[_pid] = 0;
          goto again;
}
```

CWB-NC Commands

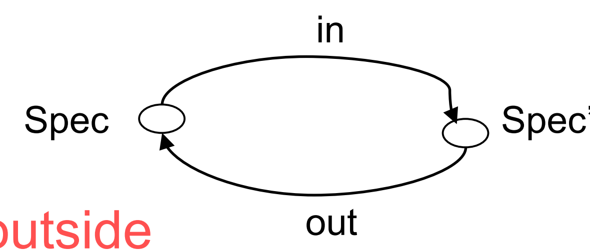
- help <command>
- load <ccs filename>
- cat <process>
- compile <process>
- es <script file> <output file>
- eq –S <trace|bisim|obseq> <proc1> <proc2>
- le –S may <proc1> <proc2> /* Trace subset relation */
- quit
- sim <process>
  - semantics <bisim|obseq>
  - random <n>
  - back <n>
  - break <act list>
  - history
  - quit

Observational Trace Equivalence

- Sys is a design for buffer with separated input/output ports
  - Sys= $(Buf_1 \mid Buf_2)\backslash\{comm_1,comm_2\}$
    - $Buf_1 = in.comm_1.Buf_1'$, $Buf_1' = comm_2.Buf_1$
    - $Buf_2 = comm_1'.Buf_2$,$Buf_2 = out.comm_2'.Buf_2$



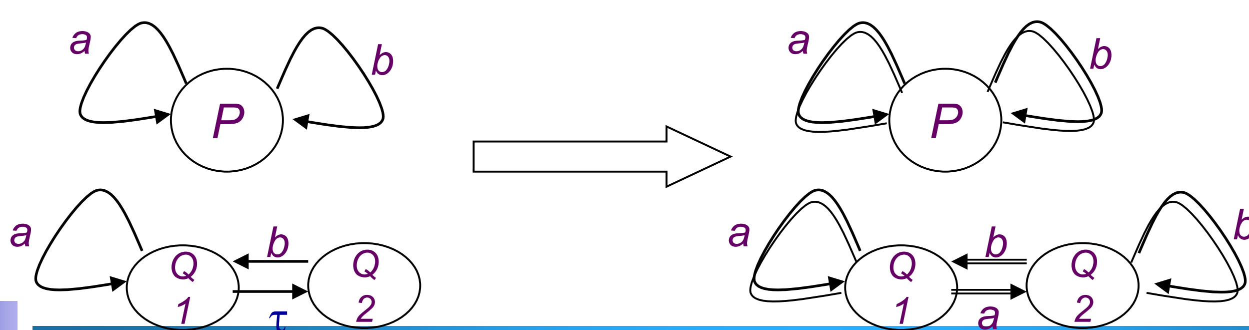
- Spec is a requirement for the buffer design
- Sys $=_{TR}$ Spec?
  - No. Sys has τ which Spec does not
    - Exec(Sys) = {in,in.τ, in.τ.out, in.τ.out.τ,…}
    - Exec(Spec) = {in, in.out, …}



  - Yes. τ is an internal hidden action not visible outside (not observable). Thus, τ is not inc_cntluded in an execution
    - If $s \in Act^*$, then $\hat{s} \in (Act - \{\tau\})^*$ is the action sequence obtained by deleting all occurrences of τ from s.
      - Ex> s = a.τ.b.τ.c, then $\hat{s}$ = a.b.c
    - A set of observable executions: Exec'(P) = $\{\hat{s} \mid s \in Exec(P)\}$
      - Exec'(Sys) = {in, in.out,…}
      - Exec'(Spec) = {in, in.out, …}

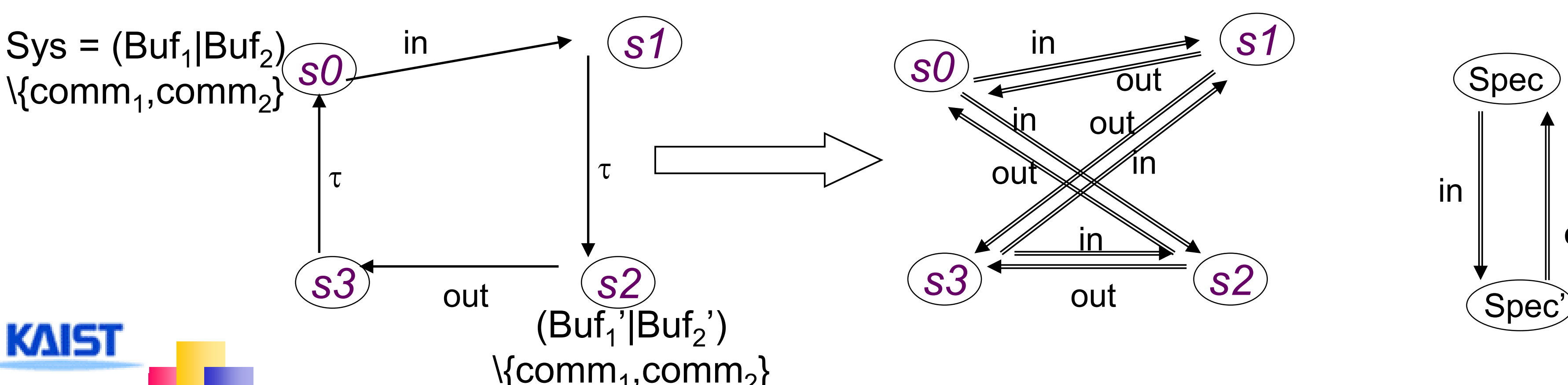
Observational Bisimulation Equivalence

- P =α=> Q iff P(-τ->)*P'-α->Q'(-τ->)*Q where $\alpha \in Act$-{τ}
  - Let s$\in$(Act-{τ})*. Then q =s=> q' if there exists s' s.t. q-s'->q' and s=ŝ'
- τ is an internal hidden action which affects internal behaviors, although itself is not visible outside.
  - *P = a.P + b.P, Q1=a.Q1* + τ.b.Q1
    - Suppose that 'a' means pushing button 'a'. Similarly for 'b'
      - P always allows a user to push any buttons.
      - Q1 allows a user to push button 'a' sometimes, button 'b' sometimes.
    - Thus, we need to distinguish P from Q1 (P and Q1 are not observationally bisimilar), which can be done using =α=> instead of -α->
      - *Q1-a->Q1 implies Q1=a=>Q1. Similary Q2-b->Q1 implies Q2=b=>Q1*
      - *Q1-a->Q1-*τ->Q2 implies Q1=a=>Q2. Q2-b->Q1- τ->Q2 implies Q2=b=>Q2



Observational Bisimulation Equivalence

- Sys $=_{BS}$ Spec? (see slide 8)
  - No. Sys has τ which Spec does not (i.e. not strongly bisimilar)
  - Yes. Sys is observationally bismilar to Spec
    - BS = { (s0,Spec), (s1,Spec'),(s3,Spec),(s2,Spec')}
      - s0 –in->s1 implies s0=in=> s1. Similarly, s2-out->s3 implies s2=out=>s3
      - s0 -in->s1 -τ->s2 implies s0=in=>s2.
      - s2-out->s3-τ-> s0 implies s2=out=>s0



Example: Scheduler

Action and Process Def.

a_i: start task$_i$

b_i: stop task$_i$

Requirements:

- $a_1,\ldots,a_n$ to occur cyclically
- a_i/b_i to occur alternately beginning with a_i

$\mathrm{Sched}_{i,X}$ for $X \subseteq \{1,\ldots,n\}$

- i to be scheduled
- X pending completion

$\mathrm{Scheduler} = \mathrm{Sched}_{i,\emptyset}$

$$\begin{aligned}
&\mathrm{Sched}_{i,X} \\
&\quad = \Sigma_{j\in X}\, b_j.\mathrm{Sched}_{i,X-\{j\}},\ \text{if } i \in X \\
&\quad = \Sigma_{j\in X}\, b_j.\mathrm{Sched}_{i,X-\{j\}} \\
&\qquad + a_i.\mathrm{Sched}_{i+1,X\cup\{i\}},\ \text{if } i \notin X
\end{aligned}$$