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.

KAIST

- 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:

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

Example 2:

$$\text{Prefix} \quad \frac{}{\alpha.P \text{ –}\alpha\text{-> } P}$$

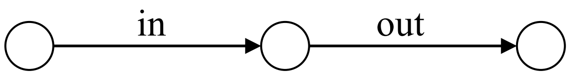
Operators for Sequential Process

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

1.Nil 0 No transitions (deadlock)

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

Prefix Prefix

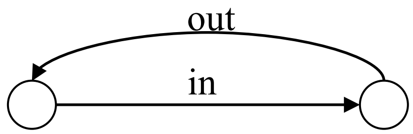


$$\text{Prefix}\ \frac{\text{(empty)}}{\alpha.P \xrightarrow{\alpha} P}$$

3.Defn $A = P$

Buffer = in.out'.Buffer

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



4.Choice $P + Q$

BadBuf = in.(τ.0 + out.BadBuf)

Prefix

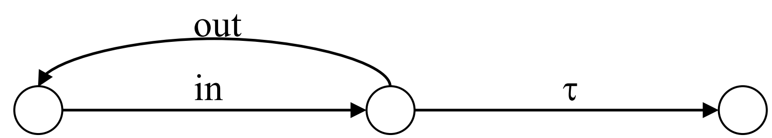
BadBuf –*in*-> τ.0 + out.BadBuf

$$\text{Choice}_L \quad \frac{P \;\text{-}\alpha\text{->}\; P'}{P+Q \;\text{-}\alpha\text{->}\; P'}$$

$$\text{Choice}_R \quad \frac{Q \;\text{-}\alpha\text{->}\; Q'}{P+Q \;\text{-}\alpha\text{->}\; Q'}$$

Choice_L Choice_R

-τ-> 0 **or** –*out*-> BadBuf



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

May use Σ notation to comactly represent sequential process

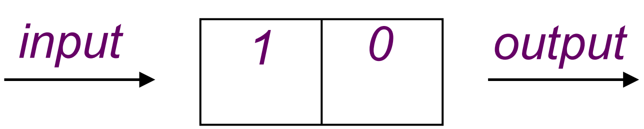
$$P = \sum_{i \in I} \alpha_i . P_i$$

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}

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

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

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

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

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

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

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

5. Parallel composition

$$\mathrm{Par}_L \quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

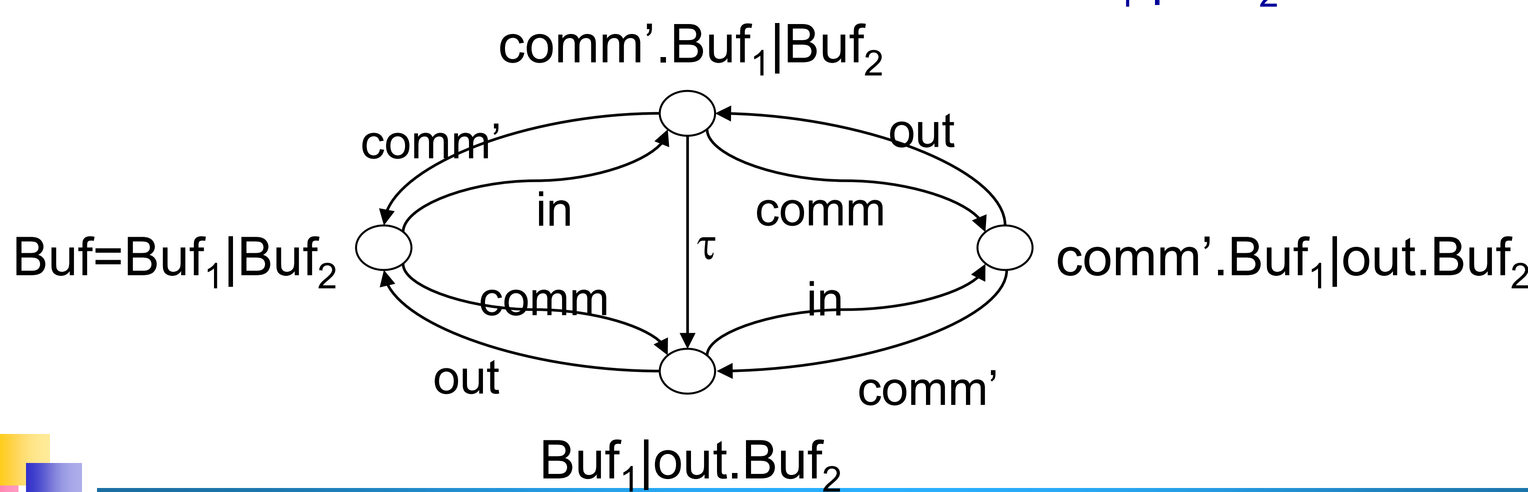
$$\mathrm{Par}_R \quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

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

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

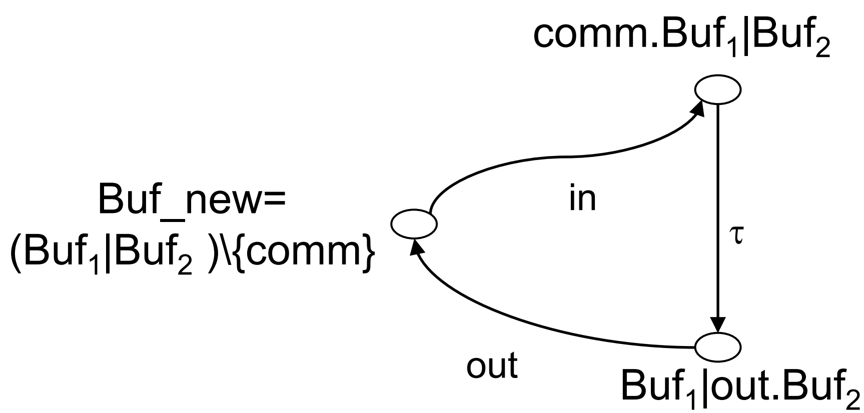
Buf
Par_R -*comm*-> Buf_1 | out.Buf_2
Par_R -*out*-> Buf_1 | Buf_2



6. Restriction *P\L*

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

Buf_1 = in.comm'.Buf_1
Buf_2 = comm.out.Buf_2
Buf_new=(Buf_1 | Buf_2)\{comm}



Buf_new
-*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

KAIST

Operators for Concurrent Process (cont.)

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

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

$Buf_0 = in.out.Buf_0$

$Buf_1 = Buf[comm'/out]$

$= in.comm'.Buf_1$

$Buf_2 = Buf[comm/in]$

$= comm.out.Buf_2$

Relabelling function *f* must preserve complements:

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

Relabelling function often given by name substitution as above

Summary of CCS Semantics

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

in.P -in-> P

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

in.P + out.Q -in-> P *or* –out-> Q

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

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

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

in.P | in'.Q -τ-> P|Q

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

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

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

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

KAIST

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

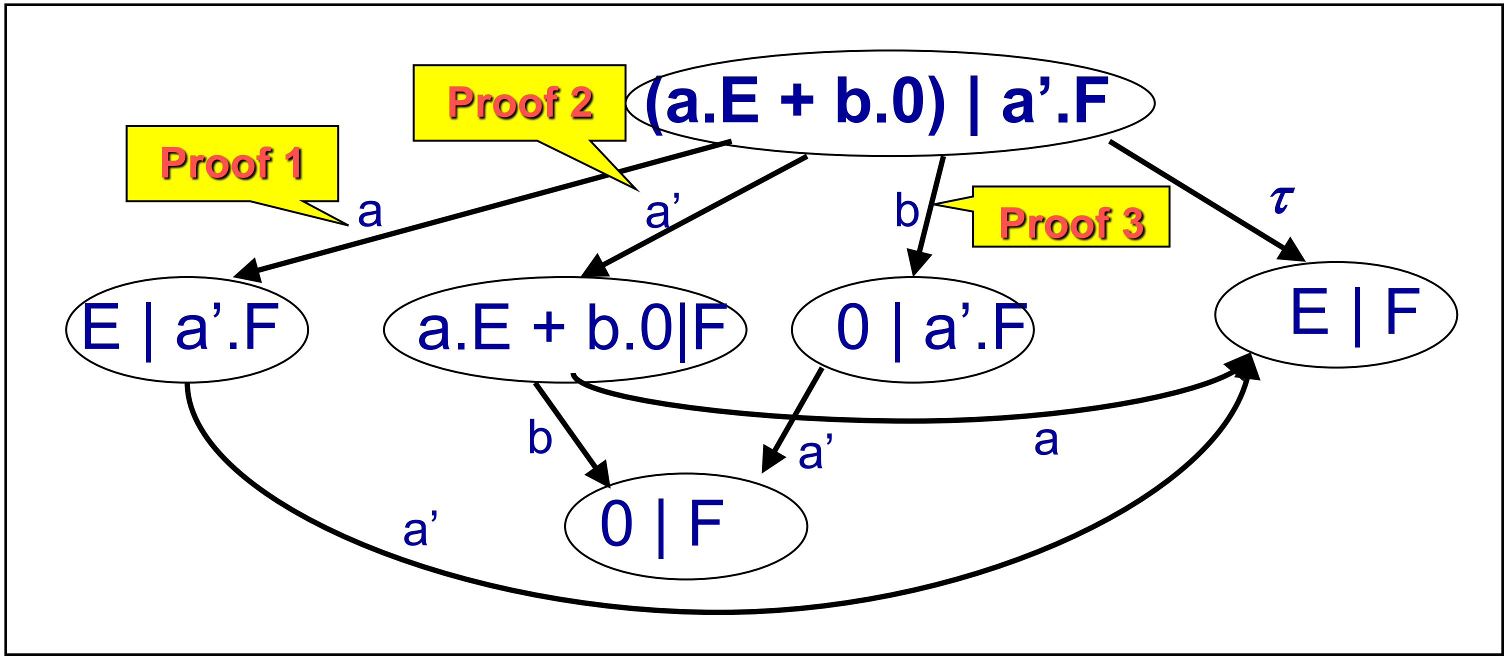
$$\dfrac{\dfrac{\dfrac{}{a.E \ \text{–a->}\ E}\ \text{Prefix}}{(a.E + b.0))\ \text{–a->}\ E}\ \text{Choice}_L}{\mathbf{(a.E + b.0) \mid a'.F\ \text{–a->}\ E \mid a'.F}}\ \text{Par}_L$$

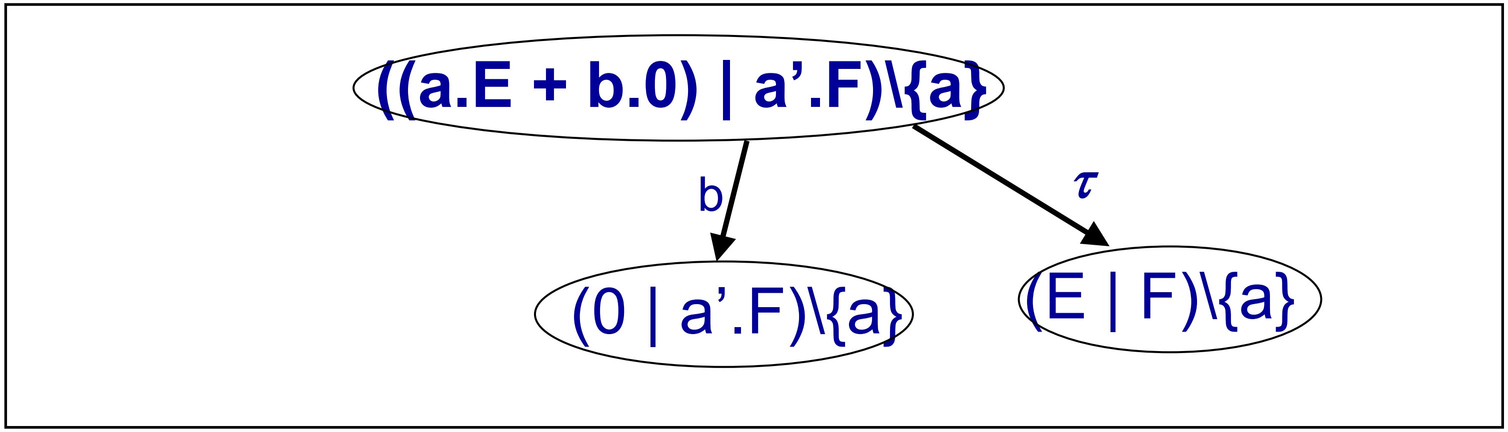
Proof 2

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

Proof 3

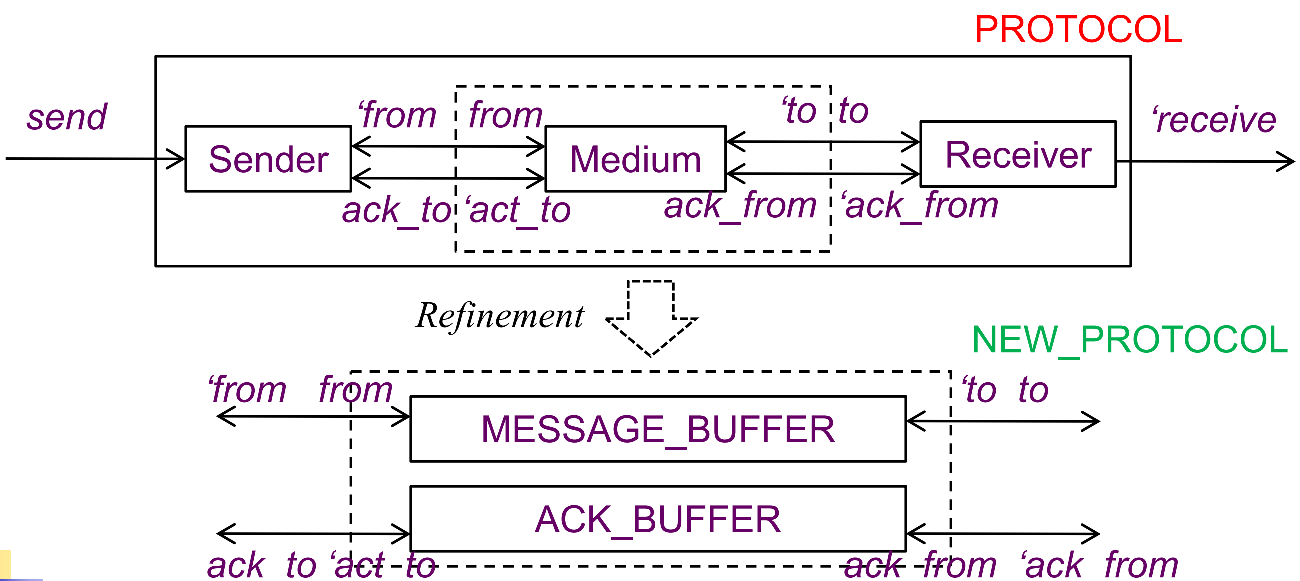
$$\dfrac{\dfrac{\dfrac{}{b.0\ \text{–b->}\ 0}\ \text{Prefix}}{(a.E + b.0)\ \text{–b->}\ 0}\ \text{Choice}_R}{\mathbf{(a.E + b.0) \mid a'.F\ \text{–b->}\ 0 \mid a'.F}}\ \text{Par}_L$$


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

Simple Protocol Example

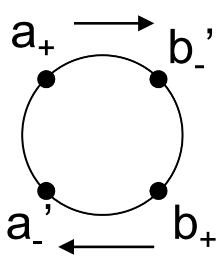
```
proc PROTOCOL =
  (SENDER | MEDIUM | RECEIVER) \ {from,to,ack_from,ack_to}
proc SENDER = send.'from.ack_to.SENDER
proc MEDIUM = from.'to.MEDIUM + ack_from.'ack_to.MEDIUM
proc RECEIVER = to.'receive.'ack_from.RECEIVER

proc NEW_PROTOCOL =
  (SENDER | NEW_MEDIUM | RECEIVER) \ {to, from, ack_to, ack_from}
proc NEW_MEDIUM         =  MESSAGE_BUFFER | ACK_BUFFER
proc MESSAGE_BUFFER =   from.'to.MESSAGE_BUFFER
proc ACK_BUFFER          =   ack_from.'ack_to.ACK_BUFFER
```



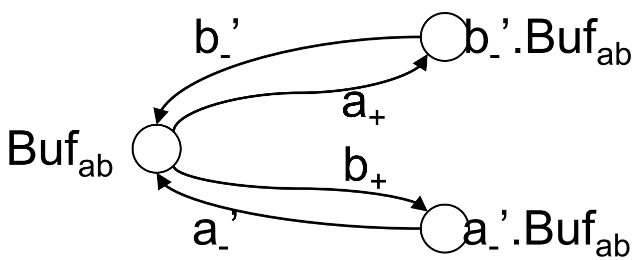
Example: 2-way Buffers

1-place 2-way buffer:
$Buf_{ab} = a_+.b_-'.Buf_{ab} + b_+.a_-'.Buf_{ab}$

Interface/architecture of Buf_{ab}

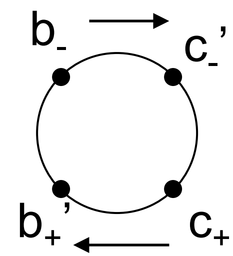


LTS of Buf_{ab} :



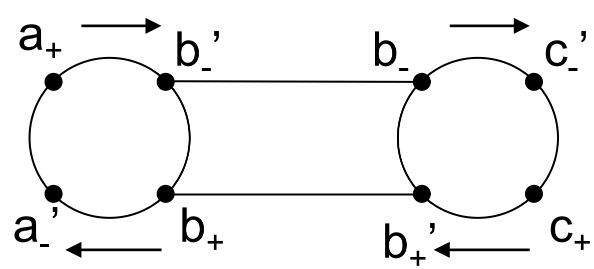
$Buf_{bc} = Buf_{ab}[c_+/b_+, c_-/b_-, b_-/a_+, b_+/a_-]$
$\quad = b_-.c_-'.Buf_{bc} + c_+.b_+'.Buf_{bc}$
(note. simultaneous substitution)

Interface/architecture of Buf_{bc}



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

Interface/architecture of Sys



But what's wrong w/ Sys?

Deadlock occurs

Example: Faulty Mutual Exclusion Protocol (1/2)

```
char cnt=0,x=0,y=0,z=0;

void enter_crit_sect() {
        char me = _pid +1; /* me is 1 or 2*/
again:
        x = me;
        If (y ==0 || y== me) ;
        else goto again;

        z =me;
        If (x == me) ;
        else goto again;

        y=me;
        If(z==me);
        else goto again;

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

Software locks

Critical section

Mutual Exclusion Algorithm

Process 0	Process 1

	x = 2
	if(y==0 || y ==2)
	z = 2
	if(x==2)

x = 1 if(y==0 || y == 1)	
	y=2 if(z==2) cnt++

z = 1
if(x==1)
y = 1
if(z == 1)
cnt++	

Violation detected !!!

Counter Example

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:
	// P1 or P2
	x = me;
	If
	:: (y ==0 || y== me) -> skip
	:: else -> goto again
	fi;

	// P1' or P2'
	z =me;
	If
	:: (x == me) -> skip
	:: else -> goto again
	fi;

	// P1'' or P2''
	y=me;
	If
	:: (z==me) -> skip
	:: else -> goto again
	fi;

	// P1''' or P2'''
	/* 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 Req = cnt_1.cnt_0.Req
```

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

KAIST