Report - CSE 586 Project: Termination Detection

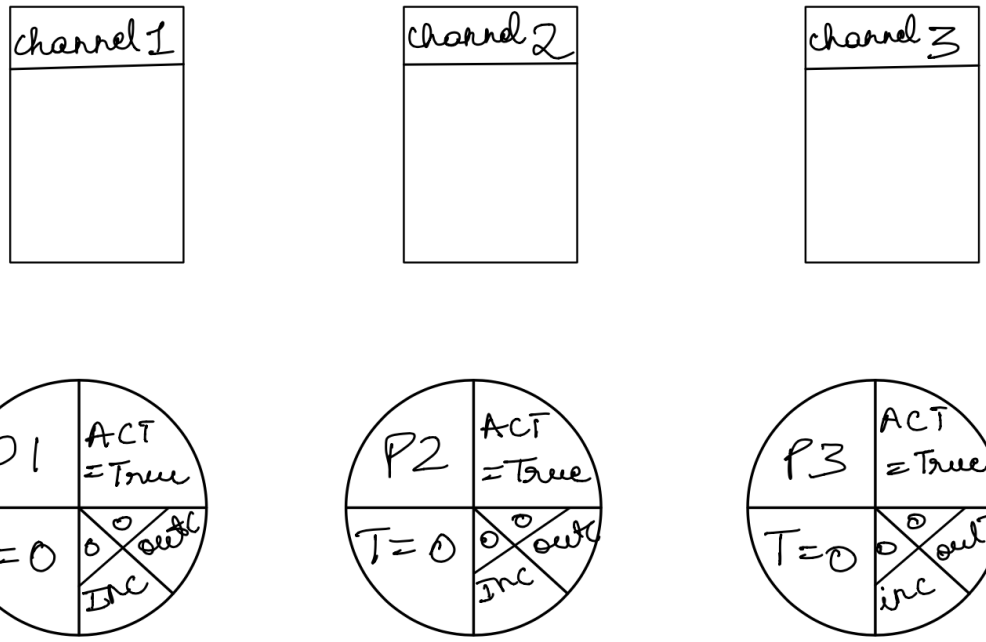
Vinay Krishna Sudarshana
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
vinaykri@buffalo.com
50321417

Aniruddha Karlekar
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
akarleka@buffalo.edu
50314912

OBSERVATIONS ABOUT HOW THE SAFETY PROPERTY IS VIOLATED

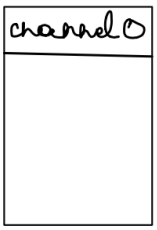
In the first part, the processes do not keep track of how many messages are sent to every other process, Instead, each process only keeps a single integer as total messages sent (inC) and another integer for the total messages received (outC). When a given process becomes idle, it sends a tuple (Process id, inC, outC) to the detector. The detector compares the total number of received(inC) and sent messages(outC) from each process, i.e. inS and outS(inS=outS) to determine if termination has been reached, by adding the inC and outC received from the idle messages of the processes. However, we observe that this violates the safety property in some cases. Below is an example of such a condition:

(1)

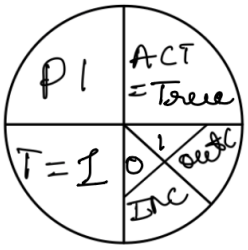


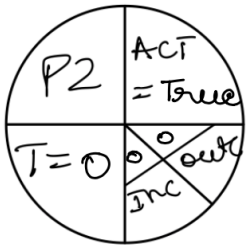
Notified	

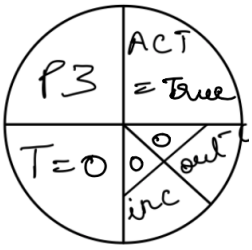
Ins	Outs	Done
0	0	False



(2) Action - Process 1 performs a send action to process 2.

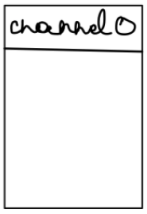




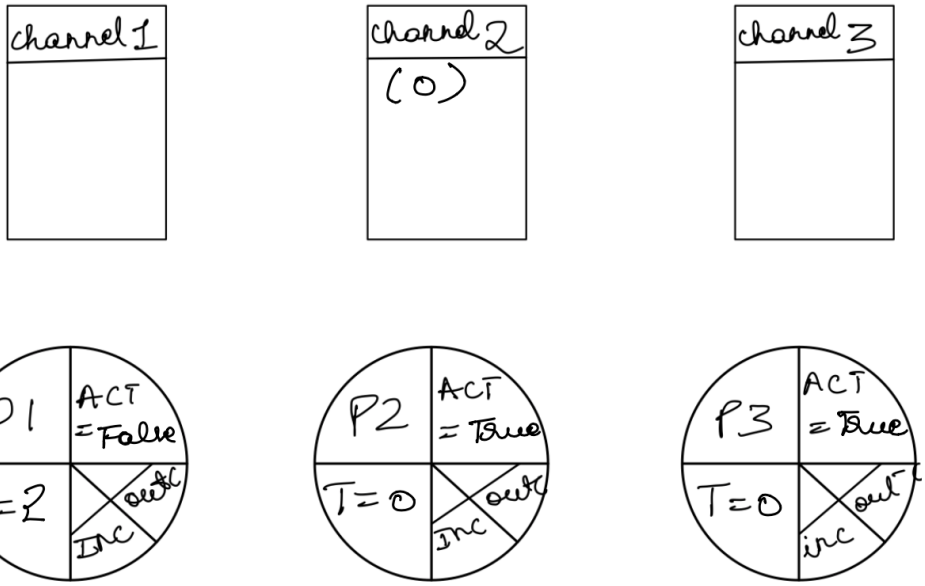


Notified	

Ins	Outs	Done
0	0	False

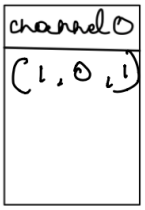


(3) **Process 1 goes to idle and sends detector a message.**

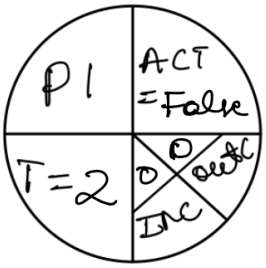


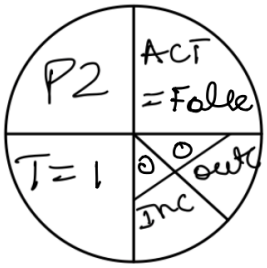
Notified	

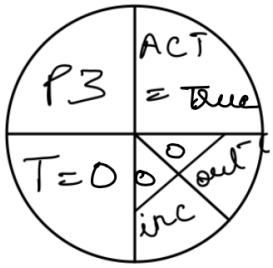
Ins	Outs	Done
0	0	False



(4) **Process 2 goes idle and notifies detector.**

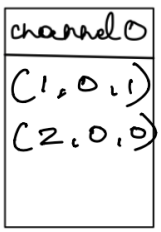




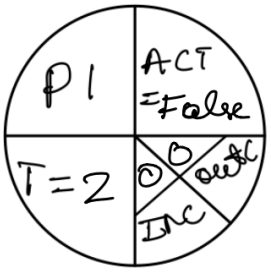


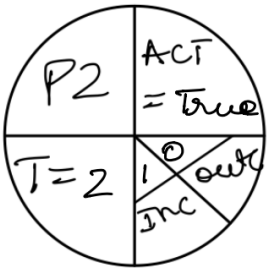
Notified	

Ins	Outs	Done
0	0	False



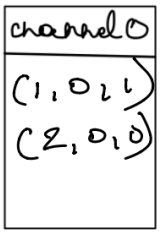
5 **Process 2 received the message and became active.**



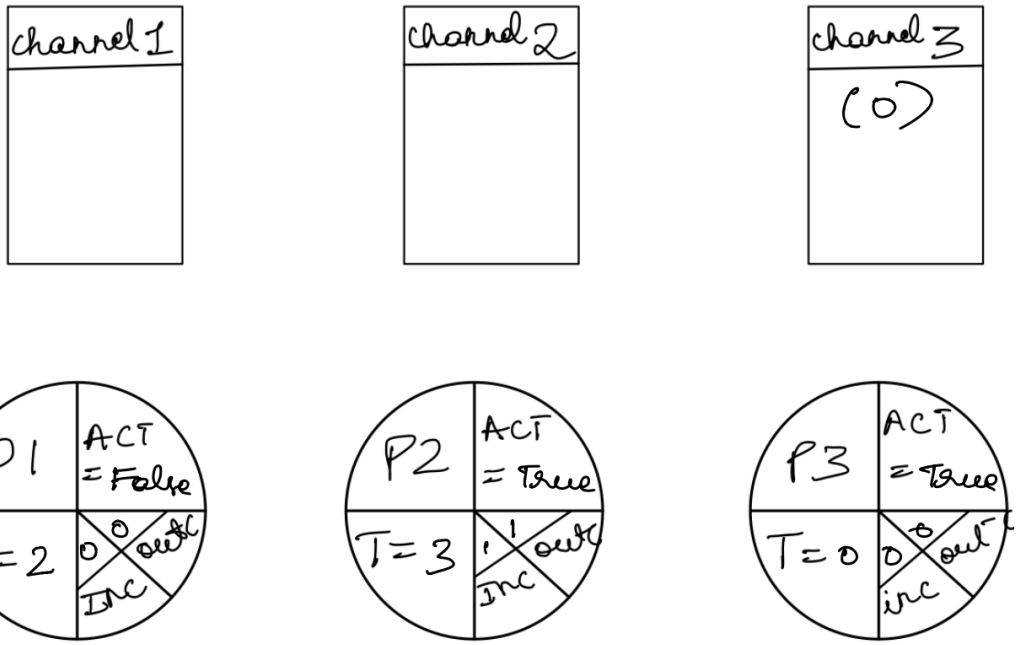


Notified	

Ins	Outs	Done
0	0	False



(6) **Process 2 performed a send action to process 3.**

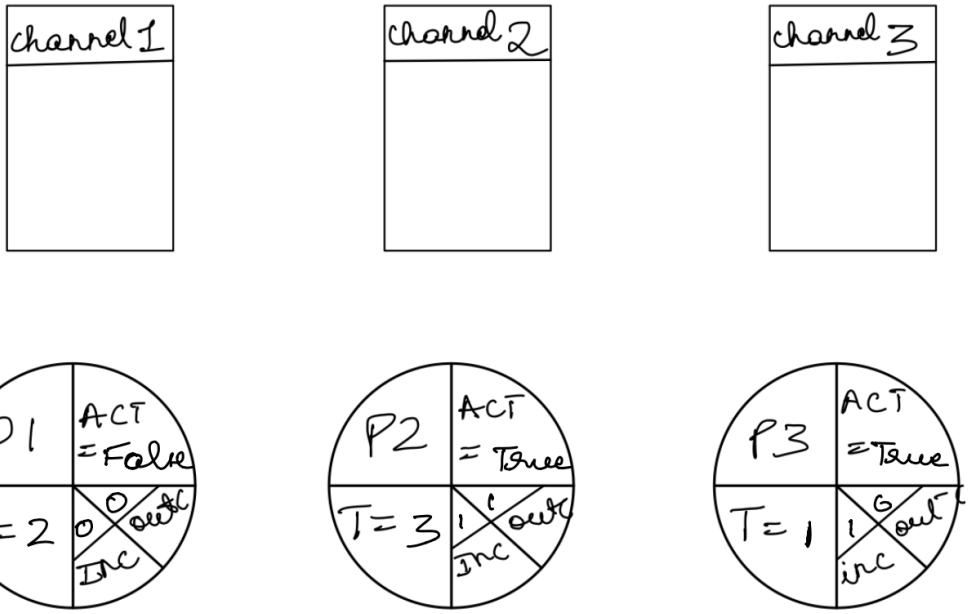


Notified	

Ins	Outs	Done
0	0	False

channel 0
(1, 0, 1)
(2, 0, 0)

7 **Process 3 received a message.**

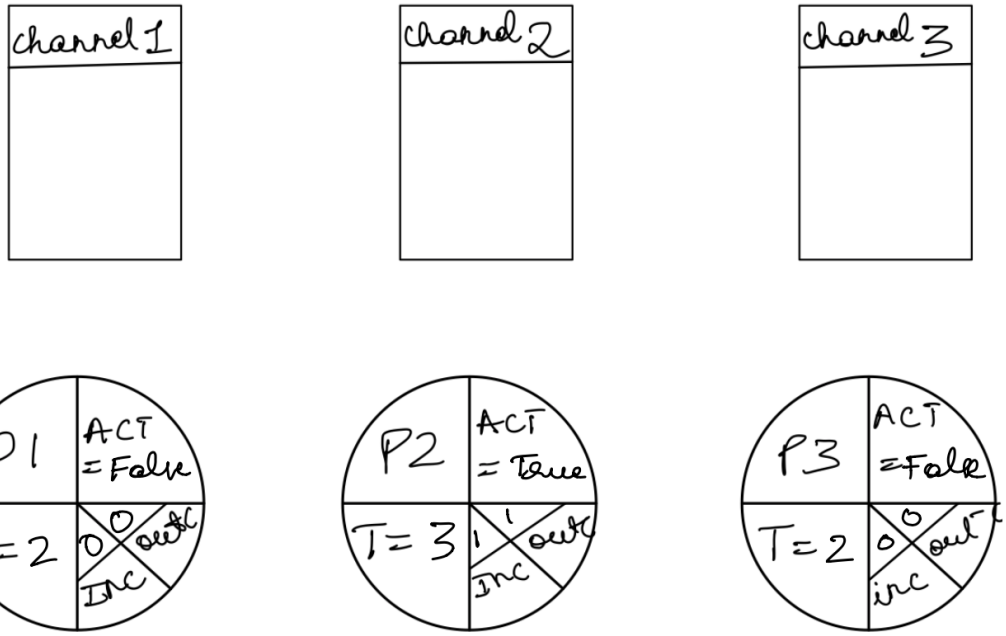


Notified	

Ins	Outs	Done
0	0	False

channel 0
(1, 0, 1)
(2, 0, 0)

8 **Process 3 goes idle and notifies detector.**

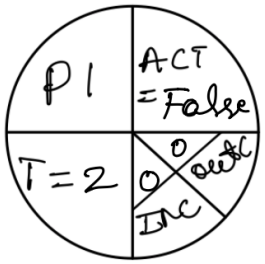


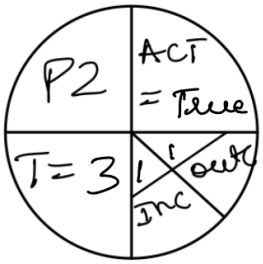
Notified	

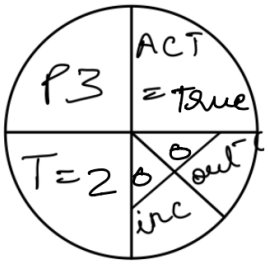
Ins	Outs	Done
0	0	False

channel 0
(1, 0, 1)
(2, 0, 0)
(3, 1, 0)

(9) **Detector process reads the message sent by process 1.**





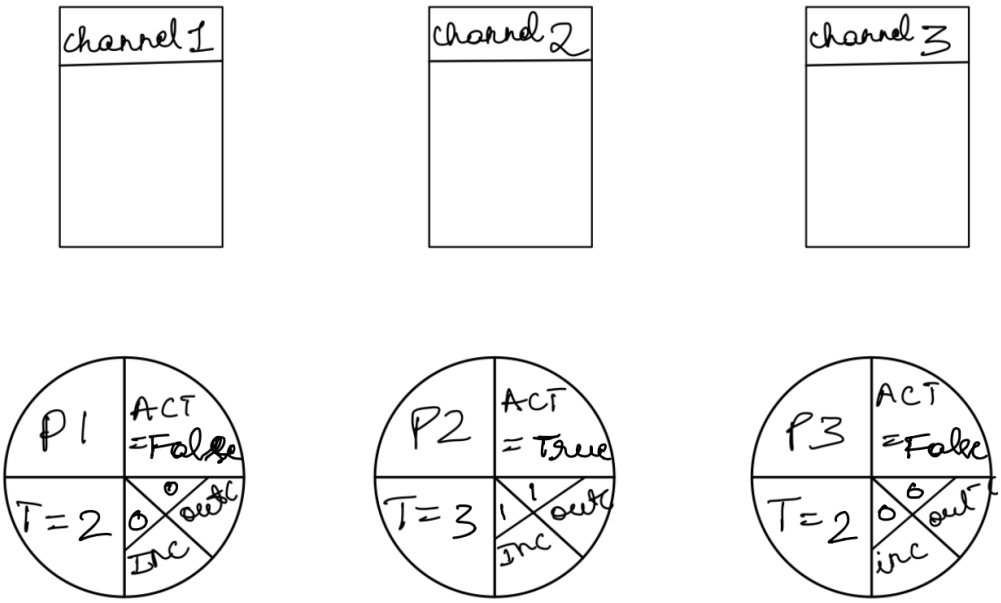


Notified	1

Ins	Outs	Done
0	1	False

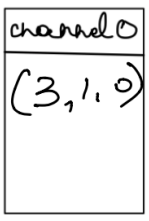
channel 0
(2, 0, 0)
(3, 1, 0)

(1b) **Detector process reads the message sent by process 2.**

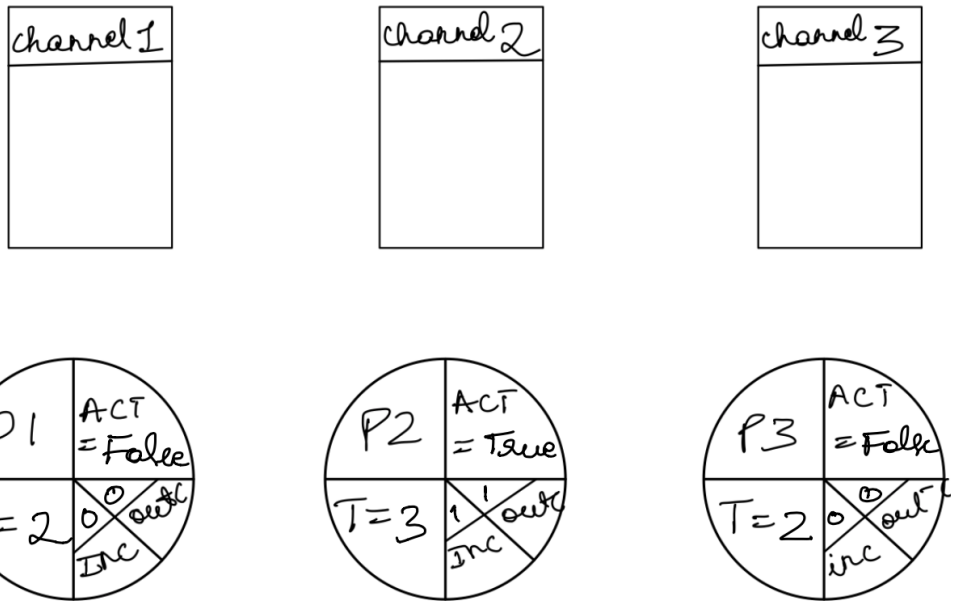


Notified	1, 2

Ins	Outs	Done
0	1	False



11 **Detector process reads the message sent by process 3.**



Notified	1, 2, 3

Ins	Outs	Done
1	1	True

Below, is the error trace for the same behaviour(counter example) that we found:

Name	Value
<Initial predicate>	State (num = 1)
T	<<0, 0, 0>>
active	<<TRUE, TRUE, TRUE>>
chan	(0 :> << >> @@ 1 :> << >> @@ 2 :> << >> @@ 3 :> << >>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 0, 0>>
m	<<0, 0, 0>>
msg	(0 :> << >>)
notified	(0 :> { })
outS	(0 :> 0)
outc	<<0, 0, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")
<P line 128, col 12 to line 151, col 60 of module td2>	State (num = 2)
T	<<1, 0, 0>>
active	<<TRUE, TRUE, TRUE>>
chan	(0 :> << >> @@ 1 :> << >> @@ 2 :> <<0>> @@ 3 :> << >>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 0, 0>>
m	<<0, 0, 0>>
msg	(0 :> << >>)
notified	(0 :> { })
outS	(0 :> 0)
outc	<<1, 0, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")

Name	Value
<P line 128, col 12 to line 151, col 60 of module td2>	State (num = 3)
T	<<2, 0, 0>>
active	<<FALSE, TRUE, TRUE>>
chan	(0 :> <<<<1, 0, 1>>>> @@ 1 :> << >> @@ 2 :> <<0>> @@ 3 :> << >>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 0, 0>>
m	<<0, 0, 0>>
msg	(0 :> << >>)
notified	(0 :> { })
outS	(0 :> 0)
outc	<<0, 0, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")
<P line 128, col 12 to line 151, col 60 of module td2>	State (num = 4)
T	<<2, 1, 0>>
active	<<FALSE, FALSE, TRUE>>
chan	(0 :> <<<<1, 0, 1>>, <<2, 0, 0>>>> @@ 1 :> << >> @@ 2 :> <<0>> @@ 3 :> << >>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 0, 0>>
m	<<0, 0, 0>>
msg	(0 :> << >>)
notified	(0 :> { })
outS	(0 :> 0)
outc	<<0, 0, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")

Name	Value
<P line 128, col 12 to line 151, col 60 of module td2>	State (num = 5)
T	<<2, 2, 0>>
active	<<FALSE, TRUE, TRUE>>
chan	(0 :> <<<<1, 0, 1>>, <<2, 0, 0>>>> @@ 1 :> << >> @@ 2 :> << >> @@ 3 :> << >>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 1, 0>>
m	<<0, 0, 0>>
msg	(0 :> << >>)
notified	(0 :> { })
outS	(0 :> 0)
outc	<<0, 0, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")
<P line 128, col 12 to line 151, col 60 of module td2>	State (num = 6)
T	<<2, 3, 0>>
active	<<FALSE, TRUE, TRUE>>
chan	(0 :> <<<<1, 0, 1>>, <<2, 0, 0>>>> @@ 1 :> << >> @@ 2 :> << >> @@ 3 :> <<0>>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 1, 0>>
m	<<0, 0, 0>>
msg	(0 :> << >>)
notified	(0 :> { })
outS	(0 :> 0)
outc	<<0, 1, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")

Name	Value
<P line 128, col 12 to line 151, col 60 of module td2>	State (num = 7)
T	<<2, 3, 1>>
active	<<FALSE, TRUE, TRUE>>
chan	(0 :> <<<<1, 0, 1>>, <<2, 0, 0>>>> @@ 1 :> << >> @@ 2 :> << >> @@ 3 :> << >>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 1, 1>>
m	<<0, 0, 0>>
msg	(0 :> << >>)
notified	(0 :> { })
outS	(0 :> 0)
outc	<<0, 1, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")
<P line 128, col 12 to line 151, col 60 of module td2>	State (num = 8)
T	<<2, 3, 2>>
active	<<FALSE, TRUE, FALSE>>
chan	(0 :> <<<<1, 0, 1>>, <<2, 0, 0>>, <<3, 1, 0>>>> @@ 1 :> << >> @@ 2 :> << >> @@ 3 :...
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 1, 0>>
m	<<0, 0, 0>>
msg	(0 :> << >>)
notified	(0 :> { })
outS	(0 :> 0)
outc	<<0, 1, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")

Name	Value
<D line 155, col 12 to line 170, col 53 of module td2>	State (num = 9)
T	<<2, 3, 2>>
active	<<FALSE, TRUE, FALSE>>
chan	(0 :> <<<<2, 0, 0>>, <<3, 1, 0>>>> @@ 1 :> << >> @@ 2 :> << >> @@ 3 :> << >>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 1, 0>>
m	<<0, 0, 0>>
msg	(0 :> <<1, 0, 1>>)
notified	(0 :> {1})
outS	(0 :> 1)
outc	<<0, 1, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")
<D line 155, col 12 to line 170, col 53 of module td2>	State (num = 10)
T	<<2, 3, 2>>
active	<<FALSE, TRUE, FALSE>>
chan	(0 :> <<<<3, 1, 0>>>> @@ 1 :> << >> @@ 2 :> << >> @@ 3 :> << >>)
done	(0 :> FALSE)
inS	(0 :> 0)
inc	<<0, 1, 0>>
m	<<0, 0, 0>>
msg	(0 :> <<2, 0, 0>>)
notified	(0 :> {1, 2})
outS	(0 :> 1)
outc	<<0, 1, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")

<D line 155, col 12 to line 170, col 53 of module td2>	State (num = 11)
T	<<2, 3, 2>>
active	<<FALSE, TRUE, FALSE>>
chan	(0 :> << >> @@ 1 :> << >> @@ 2 :> << >> @@ 3 :> << >>)
done	(0 :> TRUE)
inS	(0 :> 1)
inc	<<0, 1, 0>>
m	<<0, 0, 0>>
msg	(0 :> <<3, 1, 0>>)
notified	(0 :> {1, 2, 3})
outS	(0 :> 1)
outc	<<0, 1, 0>>
pc	(0 :> "D" @@ 1 :> "P" @@ 2 :> "P" @@ 3 :> "P")

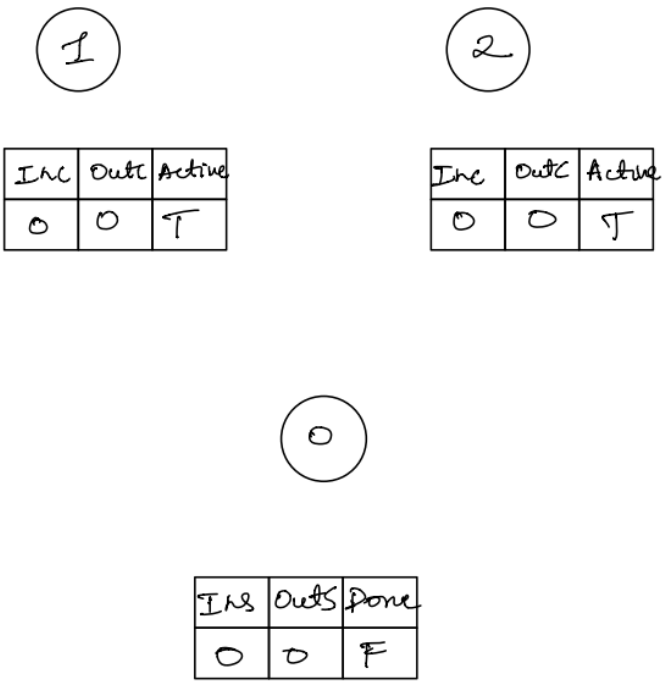
There were messages that were recorded as received by the detector but were never recorded as being sent and there were also messages that were recorded as sent by the detector but not recorded as being received. So if we use a single integer we wrongly conclude that the channels are being empty just by looking at the total number of messages being sent (ins) and checking it's equality with the total number of messages recorded as being received (outs) even if some/all of the actual messages that were recorded as being sent to particular channels were different from the messages that were recorded as received by different channels.

For example **A**, if you have only two messages - one (let's call it alpha) that was sent by process A to B and another message that was sent by process B to A (let's call it Beta). In the case where message alpha was being recorded as sent (ins=1) and beta is being recorded as being received (outs=1), in this case ins=outs, so we set done and assume the channels are

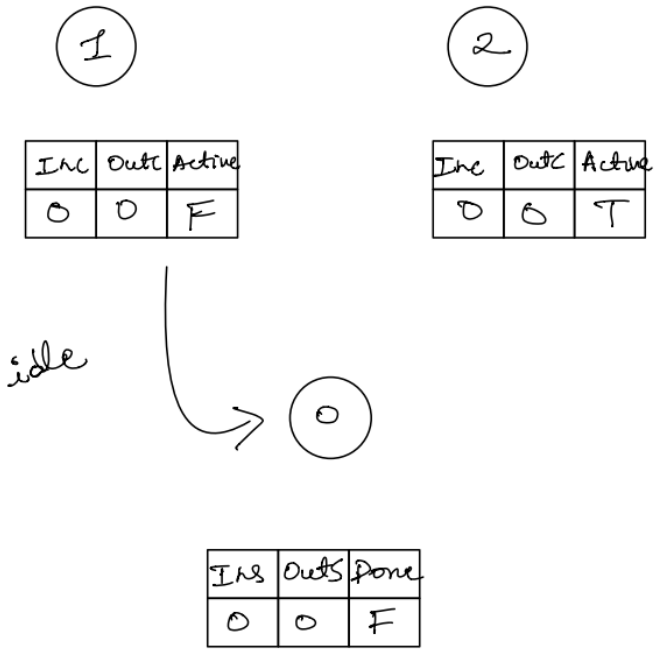
empty wherein actually there is a message alpha that was recorded as being sent but never recorded as being received and there is also a message beta which was recorded as being received but was never recorded as being sent. It would be wrong to assume channels are empty and set DONE in this case.

Here is the error trace of an execution that replicates behaviour referred to in example **A**.

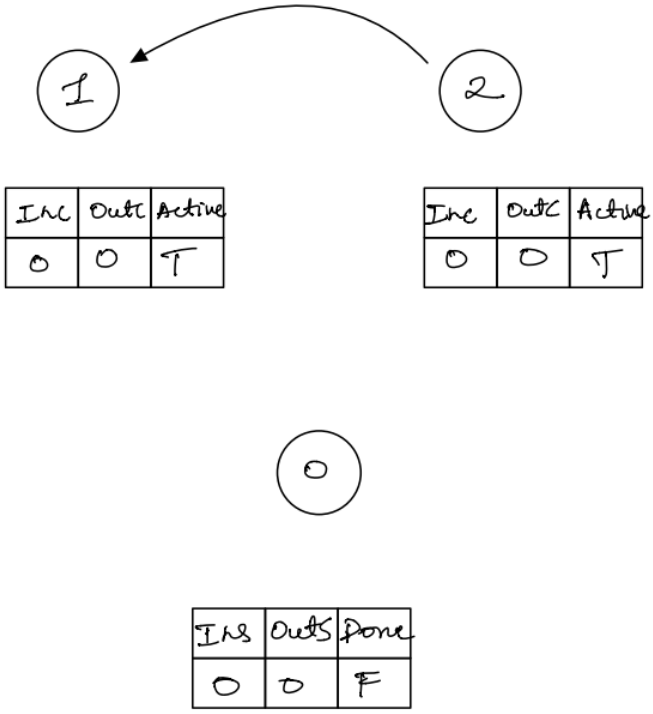
Initial



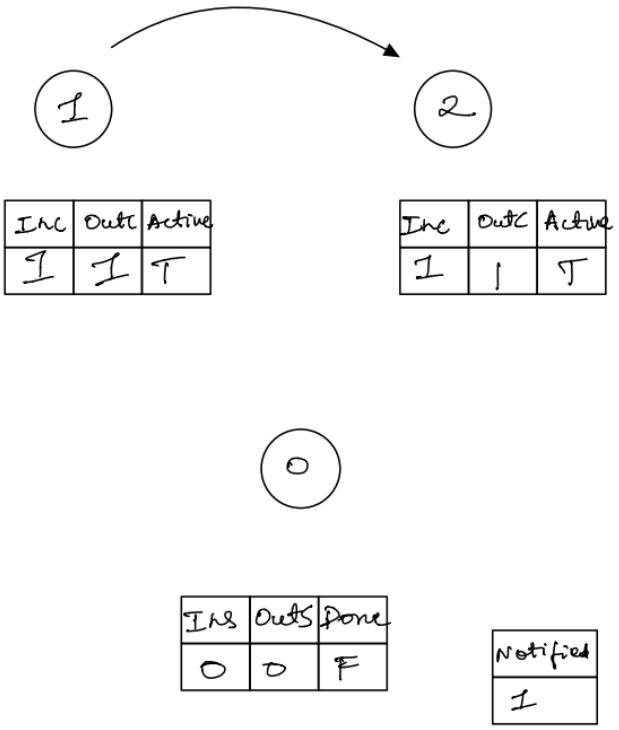
Action: Process 1 becomes idle and notifies detector



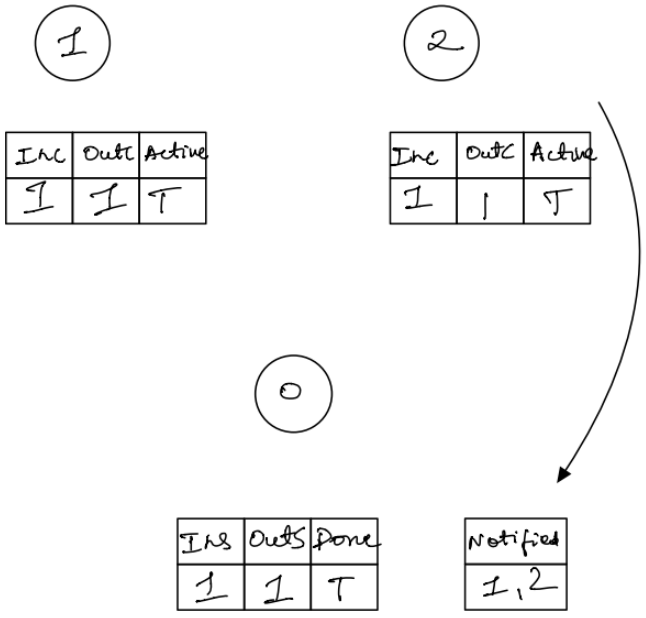
Action: Process 2 sends a message to process 1.



Process 1 sends a message to process 2



Action: process 2 becomes idle and notifies detector



HOW OUR MODEL SOLVES TERMINATION DETECTION PROBLEM CORRECTLY

The termination detection problem involves N processes that can be in 2 states: active and idle. Each process can perform 3 actions: send, receive and become idle. Only an active process can do the send and become idle action. An idle process however can become active by performing a receive action. We need to determine when the system terminates, i.e. all the processes are idle and the channels are empty.

The problem with using a single integer for inC and outC as explained in the last part of the previous section is that we wrongly conclude that the channels are empty.

We can solve this problem by making each process maintain

1. A tuple of N integers to keep track of how many messages are sent to every other process.
2. Like the previous case each process also maintains a 1 integer (**inC**) to denote how many messages it received in total.

We implement this by making every process maintain a function **outC** with domain 1..N and initialize each value to 0.

Whenever a process **A** chooses to perform a send action to a destination **D** that is chosen non deterministically. We increment the **D**'th value of the function **outC** in the process **A**.

```
with ( des ∈ Procs )
{

send(des, m);
outc[des] := outc[des] + 1;

 }
```

Whenever a process chooses to perform it's receive action, the message is read and it's inC value is incremented. If the process was initially IDLE it becomes ACTIVE upon performing the receive action when it's channel is non-empty.

```
receive(m);
active := TRUE;
inc := inc + 1;
```

Whenever a process(ACTIVE) chooses to do the idle action, it sends to the detector a message with its process id, integer **inC** and the function **outC** as a tuple. It also resets all the values of **outC** to 0.

$$\begin{aligned}
&\textbf{await } (active = \text{TRUE})\,; \\
&active := \text{FALSE}\,; \\
&send(0, \langle self, inc, outc \rangle)\,; \\
\\
&outc := [n \in 0 \,..\, N \mapsto 0]\,;
\end{aligned}$$

In this scenario, the detector has the following information:

1. Total number of messages that were recorded as being sent TO EACH CHANNEL (Vector **inS**). We implement this by maintaining a function **inS** with domain 1..N and initialize each value to 0.
2. Total number of messages that were recorded as being received BY EACH CHANNEL. (Vector **outS).** We implement this by maintaining a function **outS** with domain 1..N and initialize each value to 0.
3. Set of processes that were notified as idle to the detector. This is implemented as a set **notified**, that maintains the set of processes that have notified the detector of being idle.

$$\begin{aligned}
&notified = \{\}\,; \\
&outS = [n \in 0 \,..\, N \mapsto 0]\,; \\
&inS \;\; = [n \in 0 \,..\, N \mapsto 0]\,;
\end{aligned}$$

The detector upon performing a receive action will read a message that contains three fields :

1. Process ID **i**
2. **inC** of that process - which is the number of messages it read.
3. **outC** of that process - which is a function with domain 1..N where N is the number of Processes. **outC** is the number of messages it sent to each process or more accurately the number of messages that it recorded as being sent by putting it into the destination's channel.

The detector then :

1. Adds process with process ID **i** to the set **notified**.
2. Updates the **inS** of that process **i** which is the i'th value in it's function inS, by simply updating the value with the value it received.
3. Updates the outS of that process **i** which is the i'th value in it's function outS, by adding the received value to the old value for each entry in the function.

```
receive(msg);

notified := notified ∪ {msg[1]};

inS[msg[1]] := msg[2];

c := 0;
while ( c < N )
{
    c := c + 1;
    outS[c] := outS[c] + msg[3][c];
} ;
```

We set the done variable only when each channel notifies the detector of being idle and each channel's inS is equal to it's outS.

```
if ( (Cardinality(notified) = N) ∧ (∀ h ∈ Procs : inS[h] = outS[h]) )
{

    done := TRUE;
}
```

Therefore, if we go back to example **A** in the last paragraph of previous section, i.e if you have two messages - one (let's call it alpha) that was sent by process A to B and another message that was sent by process B to A (let's call it Beta). In the case where message alpha was being recorded as sent, the detector's **outS** vector would be [0,1] and beta is being recorded as received the detector's **inS** vector would be [1,0]. So we capture enough information here to say the detector will not set the variable **DONE** and claim termination has reached. So we can say the detector has enough information to overcome the problem discussed in example **A.**

SAFETY AND PROGRESS PROPERTIES

The safety property that we have considered is :

Safety == done[0] => ((\A s \in Procs: active[s]=FALSE) /\ (\A f \in Procs: chan[f]= <<>>))

$$Safety \triangleq done[0] \Rightarrow ((\forall s \in Procs : active[s] = \text{FALSE}) \land (\forall f \in Procs : chan[f] = \langle\rangle))$$

The safety property states that if **done** is true implies that all the processes are idle (i.e their ACTIVE=FALSE) and channels are empty. (All the channels are empty tuples which means every message that was put in it has been picked up making them empty.)

The progress property that we considered is :

Progress == ((\A s \in Procs: active[s]=FALSE) /\ (\A f \in Procs: chan[f]= <<>>)) ~> done[0]

$$Progress \triangleq ((\forall s \in Procs : active[s] = \text{FALSE}) \land (\forall f \in Procs : chan[f] = \langle\rangle)) \leadsto done[0]$$

The progress property states that if at any state if all the processes are idle and all the channels are empty, then at a later state done is set. This means that a state where all the processes are idle and all the channels are empty, leads to a state where done is set to TRUE.