CSE 586 Project: Termination detection

Rohit Lalchand Vishwakarma
Abhishek Dalvi

1.1 When considering the cumulative sum of the incoming and outgoing messages of all the processes

For the program to terminate with invariant-safety, all states of the processes must be inactive and no messages should be on the fly. The first implementation violates safety as there are state/s which is active (some processes are still active i.e. the channels are not empty) even though the sum of all incoming messages and sum of all outgoing messages of all processes taken into consideration are equal.

Counter example: -

Let's, consider (msg)m=99

- P-1 sends m to P-2; then P-1 becomes idle; P-2 becomes idle before receiving. The state looks like this.

chan = (0 :> <<<<1, 1, 0>>, <<2, 0, 0>>>> @@ 1 :> <<>> @@ 2 :> <<99>> @@ 3 :> <<>>)

active = <<FALSE, FALSE, TRUE>>

- P-2 receives and becomes active again and then P-2 sends 'm' to P-3

chan = (0 :> <<<<1, 1, 0>>, <<2, 0, 0>>>> @@ 1 :> <<>> @@ 2 :> <<>> @@ 3 :> <<99>>)

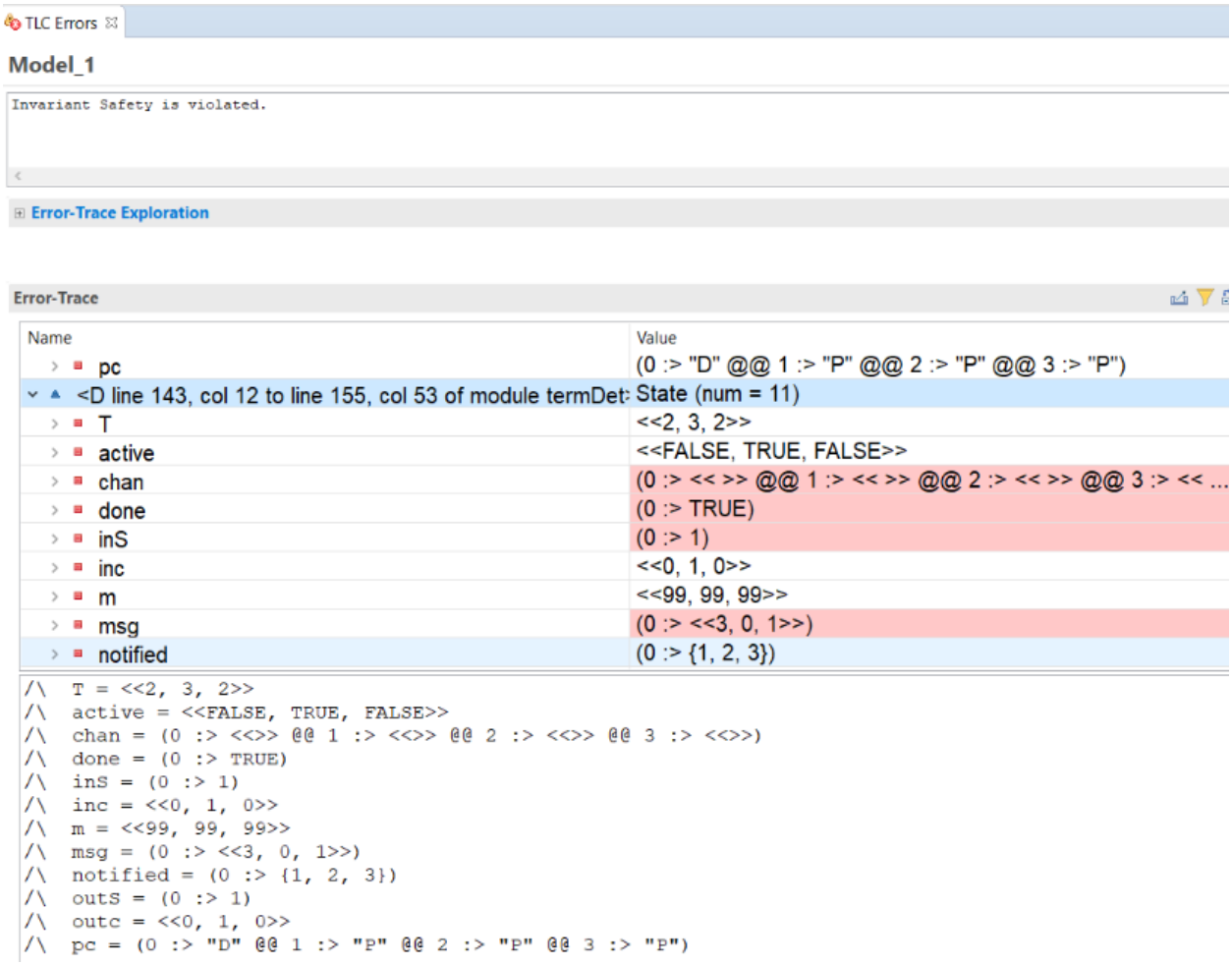
/\ active = <<FALSE, TRUE, TRUE>>; inS = 0; outS=0

- P-3 receives and then become idle.

chan = (0 :> <<<<1, 1, 0>>, <<2, 0, 0>>, <<3, 0, 1>>>> @@ 1 :> <<>> @@ 2 :> <<>> @@ 3 :> <<>>)

/\ active = <<FALSE, TRUE, FALSE>>; inS = 0; outS=0

- After this the detector kicks in and adds up all appropriate values to get inc and outc to get inS and outS
- inS = 0+0+1 ; outS = 1+0+0 i.e inS=outS BUT P-2 is active; hence safety is violated.



We can see from above though the channels are empty i.e. outS==inS. The processes are still active i.e the program has not been terminated yet and our safety has been violated.

1.2 When considering the vector count for each process

For the correct implementation we take a vector of outc (outgoing channel) i.e. each process will maintain how many messages it has sent to each process. The inc (incoming channel) is still an integer.

Each process can have 3 actions: -

1) **A process can send a message 'm' to another random process.**
   - A random process is picked using with (I \in Procs) statement which selects a number between 1 and N.
   - This can happen only if the sending process is in ACTIVE state
   - We increase outc[destination process] for the sending process as mentioned before.
2) **A process can receives a message.**

- When a process receives a message, it turns back to active state if it has idle. An active receiving process stays active, no changes in this case.
- We increase the inc counter by 1 as the process receives this message.

3) **A process turns to idle state from active state and sends a tuple to the detector channel.**
   - The process sends its (id, outc, inc).
   - After sending this packet to the detector, the process resets its outc counter as the detector will already have the previous outc values stored.
   - Not resetting these values will lead to the detector cumulatively adding the outc value when the process becomes idle multiple times.
   - Counter E.g.: - Process 1 becomes idle outc =[3,3] and sends these values to the detector. The detector adds up these values in its counter. Let's say process 1 wakes up as it receives a message, and then it sends a mssg to P-2; the outc changes to [3,4]. P-1 becomes idle again and sends inc and outc to the detector again. The detector already has outS counter with previous idle state of P-1 which is [3,3]. Now the outS for P-1 will become [3+3, 3+4]; which is WRONG. Hence, we have to reset the outc.

```
P:    while (T<MaxT)
        {
            T := T+1;
            \* Write the code for the actions using either or
            {
                either
                    {
                        await (active=TRUE);

                        with (i \in Procs)
                        {
                            send(i,m);
                            outc[i] := outc[i]+1;
                        };

                    };
                or
                    {
                        receive(m);
                        active := TRUE;
                        inc := inc+1;
                    };
                or
                    {
                        await(active=TRUE);
                        active := FALSE;
                        send(0,<<self,outc,inc>>);
                        outc := [n \in 1..N |-> 0];
                    };
            }


        }
```

<u>There is one detector which can do the following actions:</u>

- So, in the detector part we have the notified set which checks on each process's id whether it has heard from all the process or not.

- It also has its own vectors which keeps the count of total number of “sent and receive messages count” it has received from various different processes.
- outS: keeps the total count of each process’s “sent msgs” from which it has heard uptill now.
- inS: keeps the total count of each process’s “received msgs” from which it has heard uptill now.

Working of the detector

- We add each process which has gone idle to the set notified using UNION operation.
  `notified: = UNION {{msg[1]}, notified};`
- We maintain a variable “done” which is initially false.
- We increase ins[i] i.e. number of messages received on the incoming channels ‘I’.
- The detector receives msg which has the following structure `<<self,outc,inc>>` of a particular process.
- We update outS by using `msg[2],`which is the number of messages sent on each of its outgoing channels. functional i.e
  `outS := [i \in 1..N |-> outS[i] + msg[2][i]];`
- for each i in range 1 to N, outS[i] becomes outS[i] + msg[2][i]
- inS is also updated `inS[msg[1]] := msg[3];`
- The detector changes its done value to true if all process ids are in notified and if inS is equal to outS.
- To check if all process ids are in notified set is done by using the following expression
- `{x : x \in Procs} \subseteq notified.` Which means that if the set of all process ids are a subset of the set notified.
- The above steps keep on repeating until “done” becomes true.

1.3 Safety

- In our implementation using vector count for sent messages we are checking if the sent messages count of the particular process id is equals to its received messages i.e. in short, we are checking if the channels are empty and all the state has become inactive.
- Here, we know that **done => (∀p : : notified.p) /\ (∀c : : in.c = out.c),**

  ```
  done := {x : x \in Procs} \subseteq notified /\ (inS = outS);
  ```

  we have got notified by each process i.e. it has some information from the past cut of the computation. So, here we can tell that it has the valid snapshot (The notified processes may still be active i.e. they may have some messages in transit). (∀c : : s.c >= r.c)
- However, we still cannot conclude that our program has been terminated. So, for that we need to prove that the process that has been notified should also be idle (active=false) i.e there should not be any messages on the fly when the program has been terminated. For that, we have to check for the safety invariant to hold if the program has been terminated.

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

- Since, done is true and for this particular cut we have ($\forall c :: s.c = r.c$) i.e. `(\A self \in Procs:chan[self] = <<>>))` we can say that we got a valid snapshot as the channels are empty i.e incoming messages count is equal to the outgoing messages count.
- Along with this, we can also see that all the processes have become idle when taking this valid snapshot.

```
 (\A self \in Procs:active[self] =FALSE)
```

- So, its conjunction states the program has been terminated and our **safety invariant holds**.
- In short,
  done ⇒ (∀p :: p.idl) ∧ (∀c :: r.c = s.c))
  (∀p :: notified.p) ∧ (∀c :: in.c = out.c) ⇒ (∀p :: p.idl) ∧ (∀c :: r.c = s.c))

General

Start: 16:48:37 (Nov 12) End: 16:51:57 (Nov 12) Not running

Fingerprint collision probability: calculated: 5.8E-7 observed: 1.2E-7

Statistics

State space progress (click column header for graph)

Time	Diameter	States Found	Distinct States	Queue Size
00:03:20	19	7,212,289	2,098,431	0
00:01:35	12	4,619,206	1,403,065	466,419
00:00:04	7	70,655	26,099	19,763
00:00:00	0	1	1	1

Sub-actions of next-state (at 00:03:20)

Module	Action	Location	States Found	Distinct States
termDet	Terminating	line 160, col 1 to line 160, col 11	48	0
termDet	Init	line 99, col 1 to line 99, col 4	1	1
termDet	P	line 116, col 1 to line 116, col 7	5,789,571	941,622
termDet	D	line 143, col 1 to line 143, col 7	1,424,763	1,156,808

1.4 Progress

- For the progress property to satisfy i.e. to achieve the liveness property of the program we must see that our program in execution eventually terminates in the end.
- As the program proceeds the processes will become from active to inactive and vice versa but there will come a point when there are no processes in the execution.

```
((\A self \in Procs:active[self] = FALSE)
```

- Also, along with this we have to make sure that the channels are eventually becoming empty (total incoming messages==total outgoing messages for each process)

```
(\A self \in Procs:chan[self] = <<>>)
```

- And when the conjunction of both of these holds true the program will lead to its termination

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

- We can see that in our implementation the **progress property holds true** for each state in execution.
- In short,
  (∀p :: p.idl) ∧ (∀c :: r.c = s.c)); When this state is reached, this leads to termination i.e done[0] is TRUE.