Murphi-Class2

March 6, 2019

DeadLock

No next state

- no rule to execute
- have one rule

Aim of the research classes

- add research background
- have sth to say when being interviewed
- improve your ability and skills

Running Example: Mutual Exclusion Protocol

N symmetric processors, behaviour of processor i is described by:

- $try(i) := a[i] = I \rightarrow a[i]' = T$
- $crit(i) := (a[i] = T \wedge x = true \rightarrow a[i]' = C \wedge x' = false)$
- $exit(i) := a[i] = C \rightarrow a[i]' = E$
- $idle(i) := a[i] = E \rightarrow a[i]' = I \wedge x' = true$

Initial states: $x = true$ and $a[i] = I$ for all i

Invariant property (where we assume parameters are pairwise disjoint):
$\neg(a[i] = C \wedge a[j] = C)$

Reachable state set

T

he set of reachable states for a protocol $\mathcal{P} = (I, R)$, denoted as $\mathrm{RS}(\mathcal{P})$, can be defined inductively:

- a state s is in $\mathrm{RS}(\mathcal{P})$ if there exists a formula $f \in I$ such that $s \models f$;
- a state s' is in $\mathrm{RS}(\mathcal{P})$ if there exists a state s and a guarded command $r \in R$ such that $s \in \mathrm{RS}(\mathcal{P})$ and $s \xrightarrow{r} s'$.

Important properties–Safety Properties

- Bad things never happen $\Box P$.
- Invariants properties of a protocol: mutual exclusion
  $\neg(a[i] = C \wedge a[j] = C)$
- Data Coherence: $(ExGntd = false \rightarrow MemData = AuxData)$
  $\forall i \in NODE.Cache[i].State! = I \rightarrow Cache[i].Data = AuxDataend;$
- No deadLock.

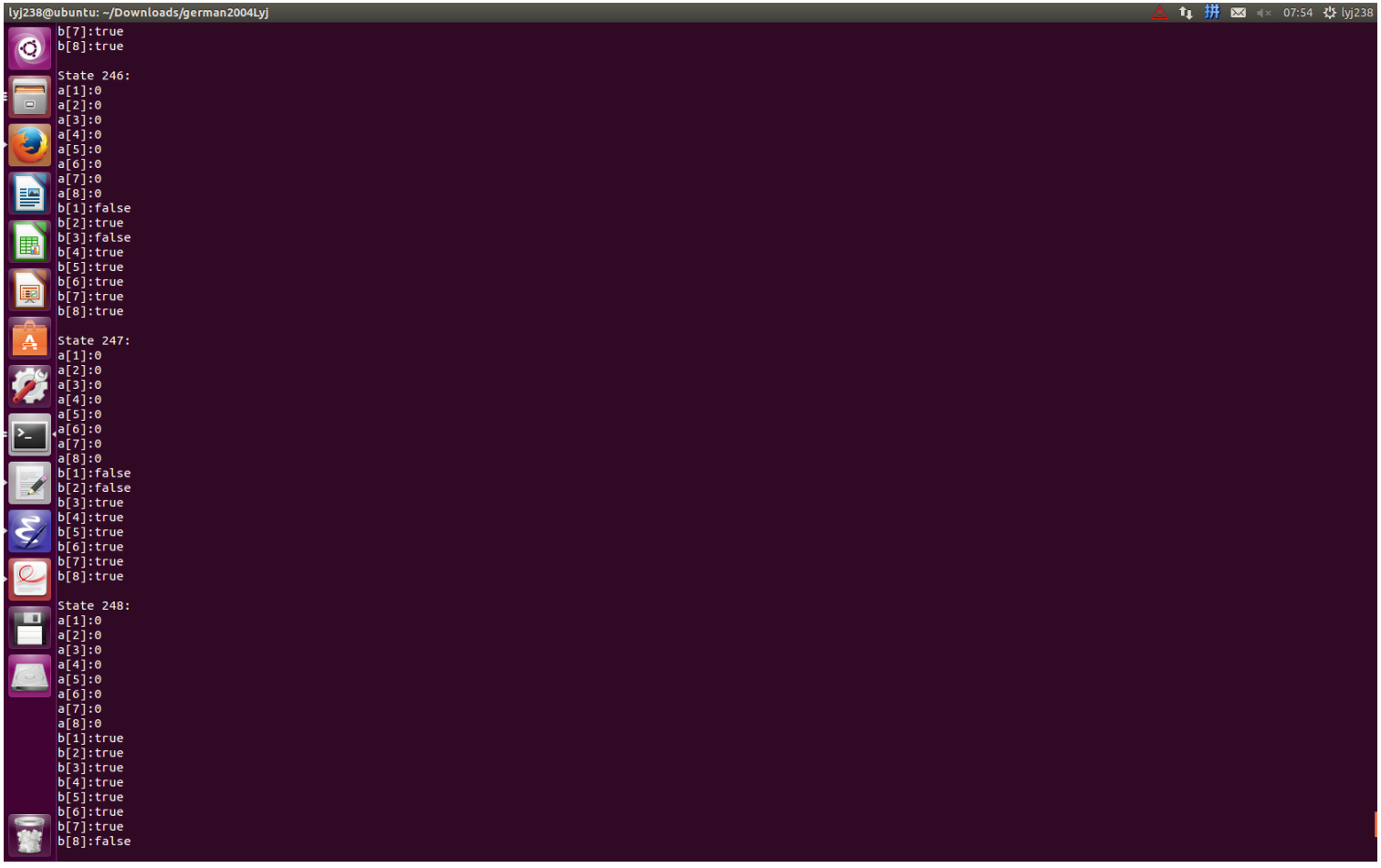
Important properties–Liveness Properties

- Good things eventually happen
- A request eventually is served $\Box(P \rightarrow \Diamond Q)$
- A process is eventually scheduled
- A Loop is terminated

Use Murphi to Compute Reachable state set

./mutualEx -ta -d ./
Use python to create the table from the trace

Output Result



A Table to illustrate a reachable state set

Table: a data table transformed from reachable state set

n[1]	n[2]	x
I	I	TRUE
T	I	TRUE
I	T	TRUE
C	I	FALSE
T	T	TRUE
I	C	FALSE
E	I	FALSE
C	T	FALSE
T	C	FALSE
I	E	FALSE
E	T	FALSE
T	E	FALSE

```
const
  SIZE: 5;

type
  foreground: 1..SIZE;
  background: -SIZE..2*SIZE;
  status: enum {Occupied, Empty};
  ar: array [foreground] of status;
  arr: array [foreground] of ar;
var
  board: arr;
  numOfQueens: 0..SIZE;

ruleset i: foreground; j: foreground do
  rule "Place Queen" board[i][j] = Empty &
   numOfQueens <SIZE==>
  begin
    board[i][j] := Occupied;
```

```
    numOfQueens := numOfQueens + 1;

    if numOfQueens >= SIZE then
      printBoard();
    endif;
  endrule;
endruleset;

rule "skip" numOfQueens =SIZE ==>
begin
    for i: foreground; j: foreground do
      board[i][j] := Empty;
    endfor;

    numOfQueens := 0;
end;

---function (i: foreground; j: foreground;
```

```
i1: foreground; j1: foreground



startstate
  begin
    for i: foreground; j: foreground do
      board[i][j] := Empty;
    endfor;

    numOfQueens := 0;
  endstartstate;

invariant "conflict"
numOfQueens <SIZE |
(exists i: foreground; j: foreground; k:foreground do
        (board[i][j]=Occupied & board[i][k]=Occupied &
        (board[j][i]=Occupied & board[k][i]=Occupied &
end
```

```
) |

(exists i: foreground; j: foreground; i1: foreground;
 j1: foreground do
          i !=i1 & j!=j1 &
          (board[i][j]=Occupied & board[i1][j1]=Occupied
     &((absDiff(i ,i1)=absDiff(j , j1)) ))
end
)
```