MRS Assignment, general comments

Mostly the assessment was done very well, and some high marks achieved. Some of the modelling issues I noticed when looking at your models were fairly common, so I address these in this document. You can also pick up an individual mark sheet from the teaching office (where I have added comments about your own submission).

Question 1: What I was looking for:

- Simple solution
- No invalid endstates (i.e. program does not deadlock)
- Evidence of correct implementation via MSC
- (would have been nice..) explanation of any behaviour that was unrealistic (e.g. in many cases the train could shoot round after the gate had opened and get through again before it closed. This is fine, but worthwhile noting in the report).

When presented with a specification (in this case described in textual form plus LTSs), stick to the terminology used as much as possible. E.g. don't change the names of the processes!

Most people achieved all, or almost all of the marks for this question.

Question 2:
It was quite possible to model this problem so that it worked but was almost impossible to understand. Indeed this was often the case. Marks were awarded for correct behaviour (as demonstrated by the LTL property, and error trace) but they were also awarded for clarity. As I've repeatedly stated throughout the course, Promela is a very simple language and isn't designed for clever tricks. It is designed to clearly model communication and simple behaviour that looks like a finite state machine (so devious programming tricks are wasted in this context). However, there were some things that would have helped reduce the amount of repetition involved, and made your programs easier to read:

-clear commenting. Even in the cases where none of the devices below were used, and programs consisted of lines of very similar code, clear commenting as to what part of the strategy was being implemented at which point really helped in the clarity of the code. You would be surprised how many people didn't do this.

- use of (sensible) constant names. At the very least you should have used constant names for 'nought', 'cross' and 'blank'. Somehow I was supposed to be able to work out that (for example) -7, 3 and 1 stood for 'nought' and 'cross' and 'blank' respectively.

-Mtype declaration. Remember that this is equivalent to declaring integer constants from right to left. For example, the mtype declaration

```
mtype = {cross,blank,nought}
```

is equivalent to

```
#define nought 1
#define blank 2
#define cross 3
```

(Apologies, in some of the mark sheets I wrote 0,1,2 rather than 1,2,3 which is wrong. This shouldn't affect your mark).

- Arithmetic logic. I can use my knowledge of the value of the constants to determine some general properties. If I have an mtype declaration as above and the sum of the elements in a row (column, diagonal) is 3 then player 1 has won. Similarly if the sum is 9, then player 2 has won. Similarly if the sum is 4 then player 1 can win in the next move.

- LTL property. We are using SPIN to generate paths of particular interest (i.e. ones in which player 2 wins). As LTL has no existential operator (we can't say "there is a path for which X holds") all of our properties have to be of the form "for all paths X holds". However, by disproving this property we ARE proving an existential property – the error path proves the existence of a path for which !X holds. So, if we define p to be winner==player2, finding an error path for the property []!p shows that there is a path in which player 2 wins.

  Some people got this the wrong way round, or only checked to see if there were paths for which player 1 didn't win (which could just generate a path for which there was a draw, not what we were looking for). There are other possibilities for the LTL property. For example, if q is defined to be winner==1 or draw, then a counterexample for the property <>q would also give a path for which player 2 wins.

Question 3 (Masters students only)

You are all clearly very well trained in summary writing! This question was almost universally done very well, with either all or most marks awarded. I added some comments to the mark sheet, but didn't necessarily take of marks for minor grammatical issues (unless it obscured the meaning of the text).

Example solutions
Please note I haven't spent time trying to line all of this code up nicely. It is just here for reference**.**

Question 1:

```
/*model of a railway system*/
/* simple version with 1 train and 1 controller for assessment
2016*/
/*1 gate */

mtype={leaves,approaches,raise,lower,up,down,far_away,near};
mtype bar=down;
mtype train_position=far_away;


#define N_trains 1 /*number of trains*/
```

```
chan train_chan=[0] of {mtype};
chan gate_chan=[0] of {mtype};

proctype controller(chan in,out)
{mtype msg;

idle:
 if
 ::in?approaches->goto Raise_bars
 ::in?leaves->goto Lower_bars
 fi;

Lower_bars: out!lower;goto idle;

Raise_bars: out!raise;goto idle

}

proctype train(chan out)
{mtype position=far_away;
do
::position==far_away->position=near;out!approaches
::((position==near) && (bar==up))->position=far_away;out!leaves
od}

proctype gate(chan in)
{do
  :: in?raise -> bar = up
  :: in?lower -> bar = down
od}



init
{atomic{run
controller(train_chan,gate_chan);run train(train_chan);
run gate(gate_chan)
}}
```

Question2(a)

```
/*To see if player 2 can win/draw noughts and crosses when*/
/* player 1 always follows the strategy below */
/* and player 2 non-deterministically chooses an available blank
square */

/*Player 1 strategy:
/* If a row/col/diagonal contains two 0s and a blank square, place
final 0 and
win */
/* else place 0 in available corner square */
/* else place 0  in any available square */

mtype = {cross,blank,nought}
```

```
/* if player 1 has won, a row, col or diag will sum to 3 */
/* if player 2 has won, a row, col or diag will sum to 9 */

/* if player 1 can win in next move, a row, col or diag will sum to
4
*/

/* if player 2 can win in next move, a row, col or diag will sum to
8
*/


typedef array{
     mtype col[3]
}

array row[3];

inline rowSum(r,sum){
   sum = row[r].col[0]+row[r].col[1] + row[r].col[2]
}

inline colSum(c,sum){
   sum = row[0].col[c]+row[1].col[c] + row[2].col[c]
}

inline diag1Sum(sum){
   sum = row[0].col[0]+row[1].col[1] + row[2].col[2]
}

inline diag2Sum(sum){
   sum = row[2].col[0]+row[1].col[1] + row[0].col[2]
}

int winner=0;
int turn = 0;

proctype player1(){

int rsum0,rsum1,rsum2,csum0,csum1,csum2,diagsum1,diagsum2 = 0;


do
:: !(turn%2);
   if
   :: winner>0 -> goto end
   :: else ->

      /*first check if this player can win*/

rowSum(0,rsum0);rowSum(1,rsum1);rowSum(2,rsum2);colSum(0,csum0);colS
um(1,csum1);
      colSum(2,csum2);diag1Sum(diagsum1);diag2Sum(diagsum2);
      if
      :: rsum0==4 ->
```

```
         if
         ::row[0].col[0] == blank -> row[0].col[0]=nought
         ::row[0].col[1] == blank -> row[0].col[1]=nought
         ::row[0].col[2] == blank -> row[0].col[2]=nought
         fi;
         atomic{rowSum(0,rsum0); assert (rsum0==3); winner =1;
turn++; goto end}
      :: rsum1==4 ->
         if
         ::row[1].col[0] == blank -> row[1].col[0]=nought
         ::row[1].col[1] == blank -> row[1].col[1]=nought
         ::row[1].col[2] == blank -> row[1].col[2]=nought
         fi;
         atomic{rowSum(1,rsum1); assert (rsum1==3); winner =1;
turn++;goto end}
      :: rsum2==4 ->
         if
         ::row[2].col[0] == blank -> row[2].col[0]=nought
         ::row[2].col[1] == blank -> row[2].col[1]=nought
         ::row[2].col[2] == blank -> row[2].col[2]=nought
         fi;
         atomic{rowSum(2,rsum2); assert (rsum2==3); winner =1;
turn++;goto end}
      :: csum0==4 ->
         if
         ::row[0].col[0] == blank -> row[0].col[0]=nought
         ::row[1].col[0] == blank -> row[1].col[0]=nought
         ::row[2].col[0] == blank -> row[2].col[0]=nought
         fi;
         colSum(0,csum0); assert (csum0==3); winner =1; goto end
      :: csum1==4 ->
         if
         ::row[0].col[1] == blank -> row[0].col[1]=nought
         ::row[1].col[1] == blank -> row[1].col[1]=nought
         ::row[2].col[1] == blank -> row[2].col[1]=nought
         fi;
         atomic{colSum(1,csum1); assert (csum1==3); winner =1;
turn++; goto end}
      :: csum2==4 ->
         if
         ::row[0].col[2] == blank -> row[0].col[2]=nought
         ::row[1].col[2] == blank -> row[1].col[2]=nought
         ::row[2].col[2] == blank -> row[2].col[2]=nought
         fi;
         atomic{colSum(2,csum2); assert (csum2==3); winner =1;
turn++; goto end}

      :: diagsum1==4 ->
         if
         ::row[0].col[0] == blank -> row[0].col[0]=nought
         ::row[1].col[1] == blank -> row[1].col[1]=nought
         ::row[2].col[2] == blank -> row[2].col[2]=nought
         fi;
         atomic{diag1Sum(diagsum1); assert (diagsum1==3); winner =1;
turn++; goto end}
```

```
     :: diagsum2==4 ->
        if
        ::row[0].col[2] == blank -> row[0].col[2]=nought
        ::row[1].col[1] == blank -> row[1].col[1]=nought
        ::row[2].col[0] == blank -> row[2].col[0]=nought
        fi;
        atomic{diag2Sum(diagsum2); assert (diagsum2==3); winner =1;
turn++; goto end}

     ::  else ->
         /* place a 0 in an available corner, if there is one */
         /* if not, place in an available blank square */
         if
         :: row[0].col[0] == blank -> row[0].col[0] = nought
         :: row[0].col[2] == blank -> row[0].col[2] = nought
         :: row[2].col[0] == blank -> row[2].col[0] = nought
         :: row[2].col[2] == blank -> row[2].col[2] = nought
         :: else ->
            if
            :: row[0].col[1] == blank -> row[0].col[1] = nought
            :: row[1].col[0] == blank -> row[1].col[0] = nought
            :: row[1].col[1] == blank -> row[1].col[1] = nought
            :: row[1].col[2] == blank -> row[1].col[2] = nought
            :: row[2].col[1] == blank -> row[2].col[1] = nought
            fi
         fi
     fi
   fi;
   atomic{turn=turn+1;
   if
   ::turn==9 -> goto end
   ::else -> skip
   fi;
   }
od;


end: skip
}

proctype player2(){
int rsum0, rsum1,rsum2,csum0,csum1,csum2,diagsum1,diagsum2 = 0;


do
:: turn%2;
   if
   ::winner > 0 || turn == 9  -> goto end
   :: else ->
   /* find a blank square and fill it, non-deterministically */
      if
      :: row[0].col[0] == blank -> row[0].col[0] = cross;
      :: row[0].col[1] == blank -> row[0].col[1] = cross;
      :: row[0].col[2] == blank -> row[0].col[2] = cross;
      :: row[1].col[0] == blank -> row[1].col[0] = cross;
      :: row[1].col[1] == blank -> row[1].col[1] = cross;
```

```
      :: row[1].col[2] == blank -> row[1].col[2] = cross;
      :: row[2].col[0] == blank -> row[2].col[0] = cross;
      :: row[2].col[1] == blank -> row[2].col[1] = cross;
      :: row[2].col[2] == blank -> row[2].col[2] = cross;
      fi;

rowSum(0,rsum0);rowSum(1,rsum1);rowSum(2,rsum2);colSum(0,csum0);

colSum(1,csum1);colSum(2,csum2);diag1Sum(diagsum1);diag2Sum(diagsum2
);
     if
     :: (rsum0==0) || (rsum1==9) || (rsum2==9) || (csum0==0) ||
        (csum1==9) || (csum2==9) || (diagsum1 ==9) || (diagsum2==9)
->
        winner =2; goto end
     :: else -> turn ++
     fi
   fi
od;

end: skip
}


init{
  row[0].col[0]=blank;
  row[0].col[1]=blank;
  row[0].col[2]=blank;
  row[1].col[0]=blank;
  row[1].col[1]=blank;
  row[1].col[2]=blank;
  row[2].col[0]=blank;
  row[2].col[1]=blank;
  row[2].col[2]=blank;

  atomic{
    run player1();
    run player2();
  }
}

#define p (winner==2)

ltl prop1 {[]!p}
```

question 2(b)

```
/*To see if player 2 can win/draw noughts and crosses when*/
/* player 1 always follows the IMPROVED strategy below */
/* and player 2 non-deterministically chooses an available blank
square */

/*Player 1 (IMPROVED) strategy:
/* If a row/col/diagonal contains two 0s and a blank square, place
final 0 and
```

```
win */
/* else, if a row/col/diagonal contains two crosses and a blank,
place 0 in
final position to prevent player 2 winning */
/* else place 0 in available corner square */
/* else place 0  in any available square */

mtype = {cross,blank,nought}

/* if player 1 has won, a row, col or diag will sum to 3 */
/* if player 2 has won, a row, col or diag will sum to 9 */

/* if player 1 can win in next move, a row, col or diag will sum to
4
*/

/* if player 2 can win in next move, a row, col or diag will sum to
8
*/


typedef array{
     mtype col[3]
}

array row[3];

inline rowSum(r,sum){
   sum = row[r].col[0]+row[r].col[1] + row[r].col[2]
}

inline colSum(c,sum){
   sum = row[0].col[c]+row[1].col[c] + row[2].col[c]
}

inline diag1Sum(sum){
   sum = row[0].col[0]+row[1].col[1] + row[2].col[2]
}

inline diag2Sum(sum){
   sum = row[2].col[0]+row[1].col[1] + row[0].col[2]
}

int winner=0;
int turn = 0;

proctype player1(){

int rsum0,rsum1,rsum2,csum0,csum1,csum2,diagsum1,diagsum2 = 0;


do
:: !(turn%2);
   if
   :: winner>0 -> goto end
   :: else ->
```

```
      /*first check if this player can win*/

rowSum(0,rsum0);rowSum(1,rsum1);rowSum(2,rsum2);colSum(0,csum0);colS
um(1,csum1);
      colSum(2,csum2);diag1Sum(diagsum1);diag2Sum(diagsum2);
      if
      :: rsum0==4 || rsum0==8 ->
         if
         ::row[0].col[0] == blank -> row[0].col[0]=nought
         ::row[0].col[1] == blank -> row[0].col[1]=nought
         ::row[0].col[2] == blank -> row[0].col[2]=nought
         fi;
         rowSum(0,rsum0);
         if
         :: rsum0==3-> atomic{winner =1; turn++; goto end}
         :: else -> assert(rsum0==7)
         fi
      :: rsum1==4 || rsum1==8->
         if
         ::row[1].col[0] == blank -> row[1].col[0]=nought
         ::row[1].col[1] == blank -> row[1].col[1]=nought
         ::row[1].col[2] == blank -> row[1].col[2]=nought
         fi;
         rowSum(1,rsum1);
         if
         :: rsum1==3-> atomic{winner =1; turn++; goto end}
         :: else -> assert(rsum1==7)
         fi
      :: rsum2==4 || rsum2==8 ->
         if
         ::row[2].col[0] == blank -> row[2].col[0]=nought
         ::row[2].col[1] == blank -> row[2].col[1]=nought
         ::row[2].col[2] == blank -> row[2].col[2]=nought
         fi;
         rowSum(2,rsum2);
         if
         :: rsum2==3-> atomic{winner =1; turn++; goto end}
         :: else -> assert(rsum2==7)
         fi
      :: csum0==4 || csum0==8 ->
         if
         ::row[0].col[0] == blank -> row[0].col[0]=nought
         ::row[1].col[0] == blank -> row[1].col[0]=nought
         ::row[2].col[0] == blank -> row[2].col[0]=nought
         fi;
         colSum(0,csum0);
         if
         :: csum0==3-> atomic{winner =1; turn++; goto end}
         :: else -> assert(csum0==7)
         fi
      :: csum1==4 || csum1==8 ->
         if
         ::row[0].col[1] == blank -> row[0].col[1]=nought
         ::row[1].col[1] == blank -> row[1].col[1]=nought
         ::row[2].col[1] == blank -> row[2].col[1]=nought
```

```
    fi;
    colSum(1,csum1);
    if
    :: csum1==3-> atomic{winner =1; turn++; goto end}
    :: else -> assert(csum1==7)
    fi
 :: csum2==4  || csum0==8->
    if
    ::row[0].col[2] == blank -> row[0].col[2]=nought
    ::row[1].col[2] == blank -> row[1].col[2]=nought
    ::row[2].col[2] == blank -> row[2].col[2]=nought
    fi;
    colSum(2,csum2);
    if
    :: csum2==3-> atomic{winner =1; turn++; goto end}
    :: else -> assert(csum2==7)
    fi
 :: diagsum1==4  || diagsum1==8->
    if
    ::row[0].col[0] == blank -> row[0].col[0]=nought
    ::row[1].col[1] == blank -> row[1].col[1]=nought
    ::row[2].col[2] == blank -> row[2].col[2]=nought
    fi;
    diag1Sum(diagsum1);
    if
    :: diagsum1==3-> atomic{winner =1; turn++; goto end}
    :: else -> assert(diagsum1==7)
    fi

:: diagsum2==4 || diagsum2==8 ->
   if
   ::row[0].col[2] == blank -> row[0].col[2]=nought
   ::row[1].col[1] == blank -> row[1].col[1]=nought
   ::row[2].col[0] == blank -> row[2].col[0]=nought
   fi;
   diag2Sum(diagsum2);
    if
    :: diagsum2==3-> atomic{winner =1; turn++; goto end}
    :: else -> assert(diagsum2==7)
    fi
::  else ->
    /* place a 0 in an available corner, if there is one */
    /* if not, place in an available blank square */
    if
    :: row[0].col[0] == blank -> row[0].col[0] = nought
    :: row[0].col[2] == blank -> row[0].col[2] = nought
    :: row[2].col[0] == blank -> row[2].col[0] = nought
    :: row[2].col[2] == blank -> row[2].col[2] = nought
    :: else ->
       if
       :: row[0].col[1] == blank -> row[0].col[1] = nought
       :: row[1].col[0] == blank -> row[1].col[0] = nought
       :: row[1].col[1] == blank -> row[1].col[1] = nought
       :: row[1].col[2] == blank -> row[1].col[2] = nought
       :: row[2].col[1] == blank -> row[2].col[1] = nought
       fi
```

```
         fi
      fi
   fi;
   atomic{turn=turn+1;
   if
   ::turn==9 -> goto end
   ::else -> skip
   fi;
   }
od;


end: skip
}

proctype player2(){
int rsum0, rsum1,rsum2,csum0,csum1,csum2,diagsum1,diagsum2 = 0;


do
:: turn%2;
   if
   ::winner > 0 || turn == 9  -> goto end
   :: else ->
   /* find a blank square and fill it, non-deterministically */
      if
      :: row[0].col[0] == blank -> row[0].col[0] = cross;
      :: row[0].col[1] == blank -> row[0].col[1] = cross;
      :: row[0].col[2] == blank -> row[0].col[2] = cross;
      :: row[1].col[0] == blank -> row[1].col[0] = cross;
      :: row[1].col[1] == blank -> row[1].col[1] = cross;
      :: row[1].col[2] == blank -> row[1].col[2] = cross;
      :: row[2].col[0] == blank -> row[2].col[0] = cross;
      :: row[2].col[1] == blank -> row[2].col[1] = cross;
      :: row[2].col[2] == blank -> row[2].col[2] = cross;
      fi;

rowSum(0,rsum0);rowSum(1,rsum1);rowSum(2,rsum2);colSum(0,csum0);

colSum(1,csum1);colSum(2,csum2);diag1Sum(diagsum1);diag2Sum(diagsum2
);
     if
     :: (rsum0==0) || (rsum1==9) || (rsum2==9) || (csum0==0) ||
        (csum1==9) || (csum2==9) || (diagsum1 ==9) || (diagsum2==9)
->
        winner =2; goto end
     :: else -> turn ++
     fi
   fi
od;

end: skip
}


init{
```

```
  row[0].col[0]=blank;
  row[0].col[1]=blank;
  row[0].col[2]=blank;
  row[1].col[0]=blank;
  row[1].col[1]=blank;
  row[1].col[2]=blank;
  row[2].col[0]=blank;
  row[2].col[1]=blank;
  row[2].col[2]=blank;

  atomic{
    run player1();
    run player2();
  }
}

#define p (winner==2)

ltl prop1 {[]!p}
```