Soutenance mini-projet ASTRE

Présenté par :

Smail AIDER, Sami SERBEY

Encadré par :

Dr. Tali Sznajder

Dr. Emmanuelle Encrenaz

Année académique 2019-2020

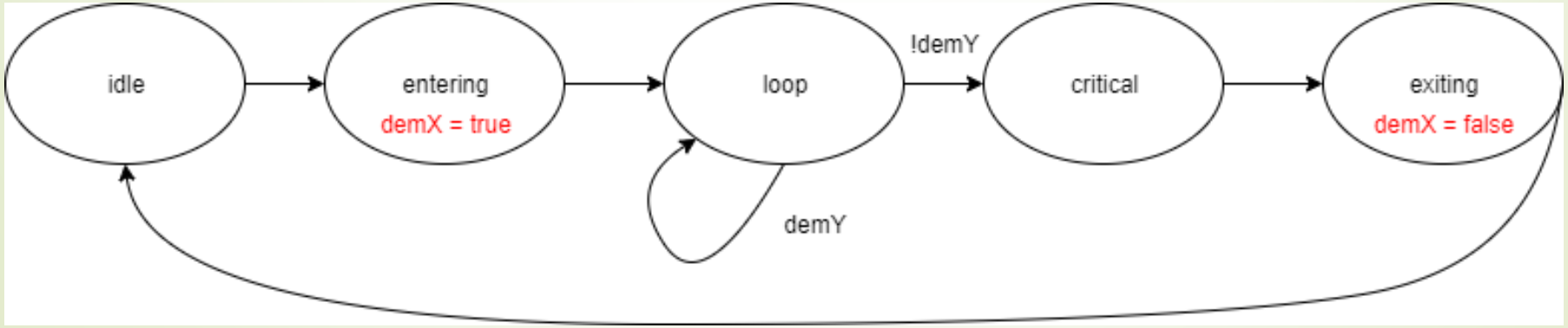
Computation tree logic (CTL)

Exercice 2 - CTL:

- L'algorithm proposé peut être traduit par le schema ci-dessous:

```
processus P
   tant que VRAI faire
1      demP ← VRAI
2      tant que demQ=VRAI faire
3      finTq
4      <<SECTION CRITIQUE>>
5      demP ← FAUX
   finTq
```

```
processus Q
       tant que VRAI faire
1'             demQ ← VRAI
2'             tant que demP=VRAI faire
3'             finTq
4'             <<SECTION CRITIQUE>>
5'             demQ ← FAUX
       finTq
```



Exercice 2 - CTL:

- Traduction du schéma en smv.

```
MODULE processus(demX, demY)
VAR
        state : {idle, entering, loop, critical, exiting};

ASSIGN
        init(state) := idle;
        next(state) :=
                case
                        state = idle : entering;
                        state = entering : loop;
                        state = loop & demY: loop;
                        state = loop & !demY: critical;
                        state = critical : exiting;
                        state = exiting : idle;
                esac;
        next(demX) :=
                case
                        state = entering : TRUE;
                        state = exiting : FALSE;
                        TRUE : demX;
                esac;
FAIRNESS
        running
```

Exercice 2 - CTL:

- Initialisation du module avec la description des propriétés P1.. P4 en CTL.

```
MODULE main
VAR
demP : boolean;
demQ : boolean;
P : process processus(demP, demQ);
Q : process processus(demQ, demP);

ASSIGN
init(demP) := FALSE;
init(demQ) := FALSE;

--P1: exclusion mutuelle
SPEC AG !(P.state = critical & Q.state = critical);

--P3: tout processus demandant l'accès à l'état critique pourra l'obtenir
SPEC AG (P.state = entering -> EF P.state = critical) & AG(Q.state = entering -> EF Q.state = critical);

--P4: l'ordre d'accès à la section critique respecte l'ordre des demandes d'accès
SPEC A[P.state != critical U P.state = entering] & A[Q.state != critical U Q.state = entering];

--P2: tout processus demandant l'accès à la section critique finira (forcemant) par l'obtenir
SPEC AG (P.state = entering -> AF P.state = critical);
```

Exercice 2 - CTL:

- Résultat:

```
-- specification AG !(P.state = critical & Q.state = critical)  is true
-- specification (AG (P.state = entering -> EF P.state = critical) & AG (Q.state = entering -> EF Q.state = critical))  is true
-- specification (A [ P.state != critical U P.state = entering ]  & A [ Q.state != critical U Q.state = entering ] )  is true
-- specification AG (P.state = entering -> AF P.state = critical)  is false
```

P2 est fausse

- Conclusion: l'algorithm proposé peut faire un blocage (en cas où les variables **demP** et **demQ** sont varies à la fois)

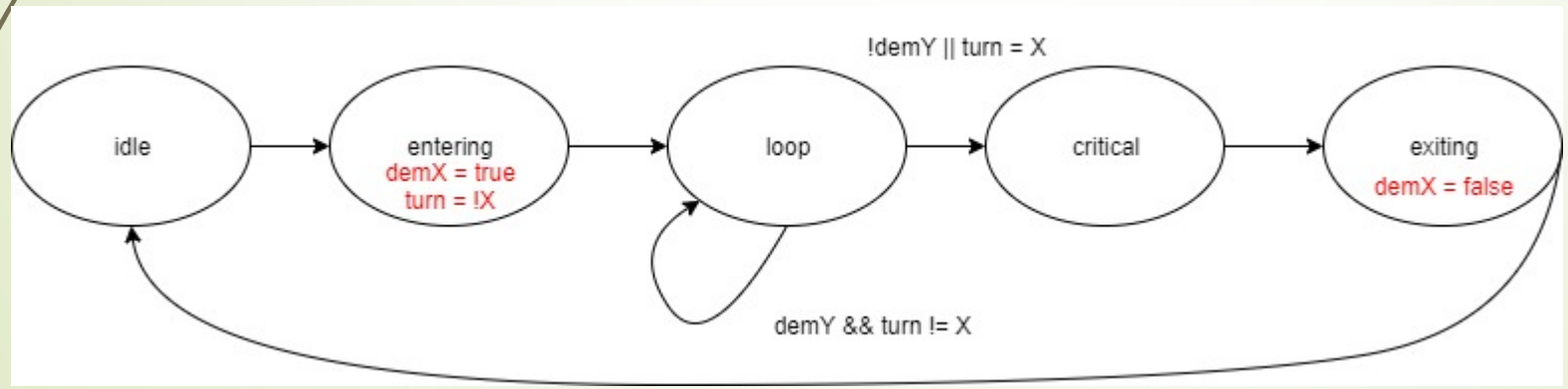
```
-> State: 1.4 <-
  demP = TRUE
  P.state = loop
-> Input: 1.5 <-
  _process_selector_ = Q
  Q.running = TRUE
  P.running = FALSE
-- Loop starts here
-> State: 1.5 <-
  demQ = TRUE
  Q.state = loop
-> Input: 1.6 <-
-- Loop starts here
```

Exercice 2 - CTL:

- Pour éviter le blocage, **l'algorithm de Peterson** peut être adapté:

```
do {
    demX = TRUE;
    turn = Y; //turn prends la value de l'autre processus
    while(!demY && turn == Y);
    /**critical section**/
    demX = FALSE;
} while(TRUE);
```

- Cela peut être modéliser par le schema suivant:



Exercice 2 - CTL:

- Fichier smv corrigé:

```
MODULE processus(demX, demY, turn, i)
VAR
        state : {idle, entering, loop, critical, exiting};

ASSIGN
        init(state) := idle;
        next(state) :=
                case
                        state = idle : entering;
                        state = entering : loop;
                        state = loop & (!demY | turn = i): critical;
                        state = critical : exiting;
                        state = exiting : idle;
                        TRUE : state;
                esac;
        next(demX) :=
                case
                        state = entering : TRUE;
                        state = exiting : FALSE;
                        TRUE : demX;
                esac;
        next(turn) :=
                case
                        state = entering : !i;
                        TRUE : turn;
                esac;
FAIRNESS
        running
```

Exercice 2 - CTL:

- Après l'adaptation de l'algorithm de Peterson, toutes les propriétés P1.. P4 sont maintenant vraies.

- P1: Exclusion mutuelle
- P2: Demande d'accès faible
- P3: l'algorithm respecte l'ordre d'accès à la section critique (sureté)
- P4: Demande d'accès fort (absence de famine)

```
-- specification AG !(P.state = critical & Q.state = critical)  is true
-- specification (AG (P.state = entering -> EF P.state = critical) & AG (Q.state = entering -> EF Q.state = critical))  is true
-- specification (A [ P.state != critical U P.state = entering ]  & A [ Q.state != critical U Q.state = entering ] )  is true
-- specification (AG (P.state = entering -> AF P.state = critical) & AG (Q.state = entering -> AF Q.state = critical))  is true
```

Exercice 3 - CTL:

- Capteur en SMV

```
MODULE capteur(C_A, C_P, peut_passer)
 VAR
   state : {absent, present};
 ASSIGN
   init(state) := absent;
   next(state) :=
     case
       state = absent   : {absent, present};
       state = present & peut_passer : {absent, present};
        state = present & !peut_passer : present;
     esac;
   next(C_A) :=
     case
       next(state) = absent : TRUE;
       next(state) = present : FALSE;
     esac;
   next(C_P) :=
     case
       next(state) = present : TRUE;
       next(state) = absent : FALSE;
     esac;
   next(peut_passer) :=
        case
                 state = present & next(state) = absent : FALSE;
                 TRUE : peut_passer;
        esac;
```

Le flag **peut_passer** s'est introduit pour avoir une modélisation plus réaliste

Exercice 3 - CTL:

- Temporisateur en SMV

```
MODULE temporisateur(restart, short, long)
 VAR
   state : {st_start, st_short, st_long};
 ASSIGN
  init(state) := st_start;
  next(state) :=
        case
                restart : st_start;
                state = st_start : {st_start, st_short};
                state = st_short : {st_short, st_long};
                state = st_long : st_long;
        esac;
  next(restart) :=
        case
                state = st_start : FALSE;
                TRUE : restart;
        esac;
  next(short) :=
        case
                next(state) = st_short : TRUE;
                next(state) = st_long : TRUE;
                TRUE : FALSE;
        esac;
  next(long) :=
        case
                next(state) = st_long : TRUE;
                TRUE : FALSE;
        esac;
FAIRNESS state = st_start; FAIRNESS state = st_short; FAIRNESS state = st_long;
```

FAIRNESS est adapté pour assurer le bon fonctionnement du temporisateur

Exercice 3 - CTL:

- Feu secondaire en SMV

```
MODULE feu_secondaire(enable_small, short, long, C_A, restart, enable_hwy, peut_passer)
  VAR
        state : {green, yellow, red};
  ASSIGN
    init(state) := red;
    next(state) :=
        case
                state = red & enable_small : green;
                state = red & !enable_small : red;
                state = yellow & short : red;
                state = yellow & !short : yellow;
                state = green & (C_A | long) : yellow;
                state = green & !(C_A | long) : green;
                TRUE : state;
        esac;
   next(enable_small) :=
        case
                state = red & enable_small : FALSE;
                TRUE : enable_small;
        esac;
   next(enable_hwy) :=
        case
                state = yellow & short : TRUE;
                TRUE : enable_hwy;
        esac;
  next(restart) :=
        case
                state = red & enable_small : TRUE;
                state = green & next(state) = yellow : TRUE;
                TRUE : restart;
        esac;
  next(peut_passer) :=
        case
                next(state) = green : TRUE;
                TRUE : peut_passer;
        esac;
```

Exercice 3 - CTL:

- Feu primaire en SMV

```
MODULE feu_principale(enable_small, short, long, C_P, restart, enable_hwy)
 VAR
       state: {green, yellow, red};
 ASSIGN
    init(state) := green;
    next(state) :=
        case
                state = green & C_P & long : yellow;
                state = green & !(C_P & long) : green;
                state = yellow & !short : yellow;
                state = yellow & short : red;
                state = red & !enable_hwy : red;
                state = red & enable_hwy : green;
        esac;
    next(enable_small) :=
        case
                state = yellow & next(state) = red : TRUE;
                TRUE : enable_small; --test
        esac;
    next(enable_hwy) :=
        case
                state = red & enable_hwy : FALSE;
                TRUE: enable_hwy;
        esac;
    next(restart) :=
        case
                state = red & enable_hwy : TRUE;
                state = green & next(state) = yellow : TRUE;
                TRUE : restart;
        esac;
```

Exercice 3 - CTL:

- Module MAIN

```
MODULE main
 VAR
   C_A : boolean;
   C_P       : boolean;
   short : boolean;
   long : boolean;
   restart : boolean;
   enable_small : boolean;
   enable_hwy: boolean;
   peut_passer : boolean;

   capteur   : process capteur(C_A, C_P, peut_passer);
   temporisateur : process temporisateur(restart, short, long);
   feu_secondaire : process feu_secondaire(enable_small, short, long, C_A, restart, enable_hwy, peut_passer);
   feu_principale : process feu_principale(enable_small, short, long, C_P, restart, enable_hwy);
ASSIGN
   init(C_A) := TRUE;
   init(C_P) := FALSE;
   init(restart) := FALSE;
   init(short) := FALSE;
   init(long) := FALSE;
   init(enable_small) := FALSE;
   init(enable_hwy) := FALSE;
   init(peut_passer) := FALSE;
```

Exercice 3 - CTL:

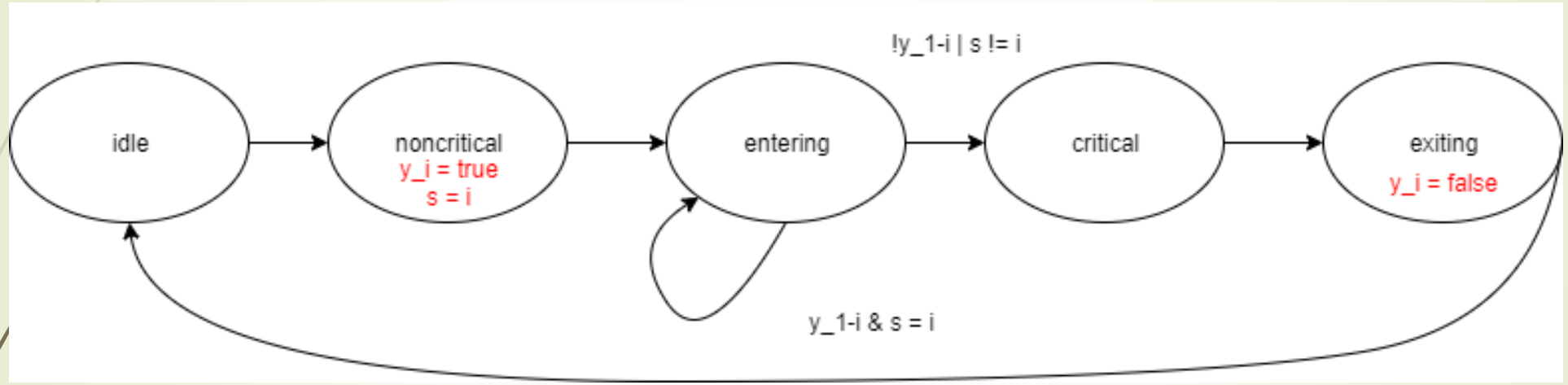
- Verification des propriétés:
- Sureté
- Vivacité
- Absence de famine

```
-- si un des feux est à GREEN alors l'autre est forcement RED ou YELLOW
 SPEC AG (feu_secondaire.state = green -> (feu_principale.state = red | feu_principale.state = yellow))
 SPEC AG (feu_principale.state = green -> (feu_secondaire.state = red | feu_secondaire.state = yellow))

-- chaque feux repassera dans l'état GREEN
SPEC (AG AF feu_principale.state = green) & (AG AF feu_principale.state = green)

-- une voiture restant suffisamment longtemps en attente d'accès au carrefour finira par y avoir accès
SPEC AG (feu_secondaire.state = red -> AF feu_secondaire.state = green)
SPEC AG(feu_principale.state = red -> AF feu_principale.state = green)
```

Linear temporal logic (LTL)

Exercice 2 - LTL:

- Le schema de l'algorithm de Pi est le suivant:



Exercice 2 - LTL:

- Modelisation de processus Pi :

```
MODULE user(s, y_i, y_1i, i)
 VAR
   state : {idle, noncritical, entering, critical, exiting};
 ASSIGN
   init(state) := idle;
   next(state) :=
     case
       state = idle                    : noncritical;
       state = noncritical                          : entering;
       state = entering & (!y_1i | s !=i) : critical;
       state = critical                             : exiting;
       state = exiting                              : idle;
       TRUE                                         : state;
     esac;
   next(s) :=
     case
       state = noncritical : i;
       TRUE                : s;
     esac;
   next(y_i) :=
     case
       state = noncritical : TRUE;
       state = exiting     : FALSE;
       TRUE                : y_i;
     esac;

FAIRNESS
        running
```

Exercice 2 - LTL:

- Main module + verification:

```
MODULE main
 VAR
   s : boolean;
   y0        : boolean;
   y1        : boolean;

   proc1     : process user(s, y0, y1, FALSE);
   proc2     : process user(s, y1, y0, TRUE);
 ASSIGN
   init(s) := FALSE;
   init(y0) := FALSE;
   init(y1) := FALSE;

 LTLSPEC G ! (proc1.state = critical & proc2.state = critical)
 LTLSPEC G (proc1.state = entering -> F proc1.state = critical)
       & G (proc2.state = entering -> F proc2.state = critical)
 LTLSPEC  G F (proc1.state = critical) & G F (proc2.state = critical)
```

Exercice 2 - LTL:

- Propriété d'exclusion mutuelle:

```
-- specification  G !(proc1.state = critical & proc2.state = critical)  is true
```

- L'absence de famine (marche pas sans **FAIRNESS running**):

```
-- specification ( G (proc1.state = entering ->  F proc1.state = critical)
 &  G (proc2.state = entering ->  F proc2.state = critical))  is true
```

- Chaque processus va entrer en section critique infiniment souvent (**FAIRNESS** requis):

```
-- specification ( G ( F proc1.state = critical) &  G ( F proc2.state = critical))  is true
```

Questions?