Université de Lorraine: Diplôme Telecom Nancy 2A IL
Epreuve: ASPD
Horaires et lieu : 8 h 00 à 10 h 00 le vendredi 16 avril 2021
Durée de l'épreuve : 2 h 00
Nom du rédacteur : Dominique Méry
Tous les documents personnels sont autorisés

Epreuve Écrite

La clarté de la rédaction et la justification des réponses sont des éléments essentiels de l'appréciation. Les exercices sont indépendants. Le nombre d'exercices est cinq (5) qui sont tous à faire.

Ecrit

On rappelle pour les trois prochains exercices qu'un programme PlusCal peut contenir plusieurs processus et que chaque processus est traduit par une relation next spécifique. Le système global est alors défini par la disjunction des nexts composant le programme. Ensuite, on traduit automatiquement le programme et ses processus en une suite d'actions et de relations next. Une des relations next est utilisée pour le modèle qui produit toutes les exécutions possibles selon la relation next à partir d'une assertion définissant les conditions initiales. Vous avez utilisé l'outil PlusCal et vous avez testé des programme avec plusieurs processus partageant des variables communes. Une instruction particulière est l'instruction assert A qui revient à tester à l'exécution si les valeurs courantes des variables satisfont A. Comme l'outil engendre toutes les exécutions possibles, si pour une exécution A est faux, l'outil stoppera et vous informera de cet état. Dans les exercices 1, 2 et 3 vous devez trouver un condition pour que le assert ne soit pas faux pour une exécution possible. Pour cela, vous devez simuler à la main les différents cas d'exécution et proposer une assertion qui convienne. Evidemment si vous utilisez TRUE, cela va fonctionner mais cette réponse n'est pas celle attendue et votre assertion doit contenir au moins une occurence de variables du programme.

Exercice 1 *Soit le petit module pluscalaspd1.tla.*

Listing 1: petit programme pluscalaspd1

```
1  ---------------------------- MODULE pluscalaspd1 ----------------------------
2  EXTENDS Integers, Sequences, TLC, FiniteSets
3  (*
4  --wf
5  --algorithm ex1{
6  variables x = 0;
7
8
9  process (one = 1)
10 {
11   A:
12     x := x - 1;
13 };
14
15 process (two = 2)
16 {
17   C:
18     x := x + 1;
19   D:
20     assert P1;
21 };
22
23 }
24 end algorithm;
25
```

```
*)
\* BEGIN TRANSLATION
VARIABLES x, pc

vars == << x, pc >>

ProcSet == {1} \cup {2}

Init == (* Global variables *)
        /\ x = 0
        /\ pc = [self \in ProcSet |-> CASE self = 1 -> "A"
                                      [] self = 2 -> "C"]

A == /\ pc[1] = "A"
     /\ x' = x - 1
     /\ pc' = [pc EXCEPT ![1] = "Done"]

one == A

C == /\ pc[2] = "C"
     /\ x' = x + 1
     /\ pc' = [pc EXCEPT ![2] = "D"]

D == /\ pc[2] = "D"
     /\ Assert(....)
     /\ pc' = [pc EXCEPT ![2] = "Done"]
     /\ x' = x

two == C \/ D

(* Allow infinite stuttering to prevent deadlock on termination. *)
Terminating == /\ \A self \in ProcSet: pc[self] = "Done"
               /\ UNCHANGED vars

Next == one \/ two
        \/ Terminating

Spec == Init /\ [][Next]_vars

Termination == <>(\A self \in ProcSet: pc[self] = "Done")

\* END TRANSLATION
```

Donner une expression P1 à placer dans la partie `assert` *afin que la vérification ne détecte pas d'erreurs dans cette assertion. Par exemple, on pourrait proposer* $x = 1 \vee x = 2$ *mais il vous appartient de simuler le programme pluscal pour vérifier que jamais l'assertion que vous proposerez ne soit fausse. La solution TRUE fonctionne mais n'est pas autorisée et l'expression demandée doit contenir une occurence de* x *au moins.*

Exercice 2 *Soit le petit module pluscalaspd2.tla.*

Listing 2: petit programme pluscalaspd2

```
------------------------------ MODULE pluscalaspd2 ------------------------------
EXTENDS Integers, Sequences, TLC, FiniteSets
(*
--wf
--algorithm ex1{
variables x = 0;


process (one = 1)
{
  A:
    x := x + 1;
  B:
    x := x +1;
};

```

```
process (two = 2)
{
  C:
    x := x - 1;
  D:
    assert \ldots;
};

}
end algorithm;

*)
\* BEGIN TRANSLATION
VARIABLES x, pc

vars == << x, pc >>

ProcSet == {1} \cup {2}

Init == (* Global variables *)
        /\ x = 0
        /\ pc = [self \in ProcSet |-> CASE self = 1 -> "A"
                                      [] self = 2 -> "C"]

A == /\ pc[1] = "A"
     /\ x' = x + 1
     /\ pc' = [pc EXCEPT ![1] = "B"]

B == /\ pc[1] = "B"
     /\ x' = x+1
     /\ pc' = [pc EXCEPT ![1] = "Done"]

one == A \/ B

C == /\ pc[2] = "C"
     /\ x' = x - 1
     /\ pc' = [pc EXCEPT ![2] = "D"]

D == /\ pc[2] = "D"
     /\ Assert P2
     /\ pc' = [pc EXCEPT ![2] = "Done"]
     /\ x' = x

two == C \/ D

(* Allow infinite stuttering to prevent deadlock on termination. *)
Terminating == /\ \A self \in ProcSet: pc[self] = "Done"
               /\ UNCHANGED vars

Next == one \/ two
        \/ Terminating

Spec == Init /\ [][Next]_vars

Termination == <>(\A self \in ProcSet: pc[self] = "Done")

\* END TRANSLATION
```

Donner une expression P2 à placer dans la partie `assert` *afin que la vérification ne détecte pas d'erreurs dans cette assertion. Par exemple, on pourrait proposer* $x = 1 \vee x = 2$ *mais il vous appartient de simuler le programme pluscal pour vérifier que jamais l'assertion que vous proposerez ne soit fausse. La solution TRUE fonctionne mais n'est pas autorisée et l'expression demandée doit contenir une occurence de x au moins.*

Exercice 3 *Soit le petit module pluscalaspd3.tla.*

Listing 3: petit programme pluscalaspd3

```
---------- MODULE pluscalaspd3 ----------
```

```
EXTENDS Integers, Sequences, TLC, FiniteSets
(*
--wf
--algorithm ex3{
variables x = 0, y = 2;


process (one = 1)
{
  A:
    x := x + 1;
  B:
    y := y -1;
  C:
    assert Q1;
};

process (two = 2)
{
  D:
    x := x - 1;
  E:
    y:=y+2;
   F:
    x:= x+2;
  G:
    assert Q2;
};

}
end algorithm;

*)
\* BEGIN TRANSLATION
VARIABLES x, y, pc

vars == << x, y, pc >>

ProcSet == {1} \cup {2}

Init == (* Global variables *)
        /\ x = 0
        /\ y = 2
        /\ pc = [self \in ProcSet |-> CASE self = 1 -> "A"
                                      [] self = 2 -> "D"]

A == /\ pc[1] = "A"
     /\ x' = x + 1
     /\ pc' = [pc EXCEPT ![1] = "B"]
     /\ y' = y

B == /\ pc[1] = "B"
     /\ y' = y -1
     /\ pc' = [pc EXCEPT ![1] = "C"]
     /\ x' = x

C == /\ pc[1] = "C"
     /\ Assert(...)
     /\ pc' = [pc EXCEPT ![1] = "Done"]
     /\ UNCHANGED <<x, y>>

one == A \/ B \/ C

D == /\ pc[2] = "D"
     /\ x' = x - 1
     /\ pc' = [pc EXCEPT ![2] = "E"]
     /\ y' = y

E == /\ pc[2] = "E"
     /\ y' = y+2
     /\ pc' = [pc EXCEPT ![2] = "F"]
     /\ x' = x

F == /\ pc[2] = "F"
     /\ x' = x+2
     /\ pc' = [pc EXCEPT ![2] = "G"]
     /\ y' = y

G == /\ pc[2] = "G"
```

```
        /\  Assert((...)
        /\  pc' = [pc EXCEPT ![2] = "Done"]
     /\ UNCHANGED <<x, y>>

two == D \/ E \/ F \/ G

(* Allow infinite stuttering to prevent deadlock on termination. *)
Terminating == /\ \A self \in ProcSet: pc[self] = "Done"
               /\ UNCHANGED vars

Next == one \/ two
        \/ Terminating

Spec == Init /\ [][Next]_vars

Termination == <>(\A self \in ProcSet: pc[self] = "Done")

\* END TRANSLATION









====
```

Donner les deux expressions Q1 et Q2 à placer dans les parties `assert` *afin que la vérification ne détecte pas d'erreurs dans cette assertion. Par exemple, on pourrait proposer* $(x = 1 = 2) \wedge (y = 0 \vee y = 5)$ *mais il vous appartient de simuler le programme pluscal pour vérifier que jamais l'assertion que vous proposerez ne soit fausse. La solution TRUE fonctionne mais n'est pas autorisée et les expressions demandées doivent contenir une occurence de* x *au moins et une occurence de* y*.*

Exercice 4 *(Exclusion mutuelle)*

Vous avez eu les présentations des protocoles de demande de la section critique notamment Lamport et Ricart-Agrawala. Expliquez simplement les différences entre les deux protocoles notamment le nombre de messages nécessaires pour obtenir une section critique.

Exercice 5 *(protocoles de communication)*

L'Internet des Objets repose sur des protocoles de communication qui doivent obéir à des contraintes de fiabilité dans un milieu bruité, voire hostile. L'un des protocole utilisé dans les solutions pratiques est appelé Lorawan et utilise des protocoles plus élémentaires de communication entre les différentes entités. Le schéma ci-après décrit les flux de communication:

- *entre sensors et Base station*
- *entre Base station et Gateways*
- *entre Gateways et Network server*
- *entre Network et server application servers*

On demande de proposer des protocoles de communication entre Base station et Gateways et entre Gateways et Network server. Vous devez proposer une ou plusieurs solutions en expliquant que vos solutions permettent de ne pas perdre de messages.