E.N.S.I.M.A.G.
3ème année
Février 2012

Examen

Modélisation et Vérification de Systèmes Concurrents et Temps-Réel

Durée 2 heures
Tous documents autorisés

Avertissement : Il vous est demandé d'apporter le plus grand soin dans la rédaction. Vous serez jugés plus sur la QUALITÉ que sur la QUANTITÉ de vos réponses.

Le barême est donné à titre purement indicatif.

Partie I Modélisation en LNT (10 points)

L'algorithme "*MCS queue lock*" (proposé par Mellor-Crummey et Scott en 1991) est un protocole d'exclusion mutuelle permettant à N agents[1] s'exécutant en parallèle d'accéder à une ressource critique. Une particularité de ce protocole est de satisfaire les demandes d'accès à la ressource critique en respectant l'ordre dans lequel les demandes des agents ont été enregistrées (premier arrivé = premier servi), tout en étant efficace sur des machines à mémoire partagée avec caches cohérents, même pour un grand nombre d'agents.

Comme tous les protocoles d'exclusion mutuelle, l'algorithme "*MCS queue lock*" repose sur deux procédures `acquire_lock()` et `release_lock()`, qui permettent à chaque agent de demander puis de libérer la ressource critique. Le pseudo-code ci-dessous (dans une syntaxe proche du langage C) décrit ces deux procédures :

```
typedef {0, ..., N} index; // intervalle des entiers de 0 a N
const index nil = N;
// les agents sont numerotes 0 a N-1, nil est un marqueur de fin de liste

// cellule memoire pour coder les elements d'une file d'attente
typedef struct node {
   index next = nil;    // numero de l'agent suivant dans la file ou nil si aucun
   bool locked = false; // vrai si un agent + prioritaire attend ou utilise la ressource
} qnode;

// variables partagees
qnode M[N-1];  // table des cellules indexee par les numeros d'agent
index L = nil; // numero de l'agent en queue de file ou nil si aucun

void acquire_lock (index pid) {
   // l'agent numero pid demande la ressource : ajout en fin de file
   index predecessor;
   M[pid].next = nil;
   predecessor = fetch_and_store (L, pid);
   if (predecessor != nil) {     // pid n'est pas le premier dans la file
      M[pid].locked = true;      // la ressource est indisponible
```

[1] On utilise ici le terme agent plutôt que processus pour éviter la confusion avec les processus du langage LNT.

```
      M[predecessor].next = pid; // le chainage est mis a jour
      while (M[pid].locked);     // attend la disponibilite de la resource
   }
}

void release_lock (index pid) {
   // l'agent pid libere la ressource
   if (M[pid].next == nil) {
      if (compare_and_swap (L, pid, nil))
         return; // personne d'autre n'attend la ressource
      while (M[pid].next == nil); // attend que l'agent suivant mette a jour le chainage
   }
   M[M[pid].next].locked = false; // libere la ressource pour l'agent suivant
}
```

Chaque agent est identifié par un index unique `pid` dans l'intervalle $0..N-1$ et à chacun est également associé une cellule mémoire `M[pid]`. La mémoire `M` permet ainsi de coder une file d'attente, sous la forme d'une liste chaînée de cellules (type `qnode`). La valeur d'index N permet de coder la fin de liste (`nil`). La variable partagée `L` mémorise l'index du dernier élément de la file d'attente si celle-ci n'est pas vide, ou `nil` sinon.

Les fonctions `fetch_and_store()` et `compare_and_swap()` sont des opérations atomiques définies comme suit :

- `fetch_and_store (X, E)` retourne la valeur courante de la variable partagée `X` et affecte à `X` la valeur de l'expression `E`;
- `compare_and_swap (X, E1, E2)` compare la valeur de `X` à la valeur de `E1` et, si les deux valeurs sont égales, affecte à `X` la valeur de `E2` et retourne `true`, sinon retourne `false`.

On se limite dans ce sujet au cas où deux agents essaient d'accéder à la ressource critique ($N = 2$).

Question I.1 Modélisation de la variable L

On dispose du type et de la fonction LNT suivants :

```
type Index is
  range 0 .. 2 of Nat -- intervalle des entiers de 0 a 2
  with "==", "!="
end type

function nil: Index is
  return Index (2)
end function
```

On modélise la variable partagée `L` sous la forme du processus LNT suivant :

```
process Lock [Fetch_and_Store, Compare_and_Swap : any] is
  var i, new_i, j: Index in
    i := nil;
```

```
    loop
      select
        Fetch_and_Store (i, ?new_i);
        i := new_i
      []
        Compare_and_Swap (?j, ?new_i, true) where (i == j);
        i := new_i
      []
        Compare_and_Swap (?j, ?new_i, false) where (i != j)
        -- on ignore new_i
      end select
    end loop
  end var
end process
```

Dessiner le système de transitions étiquetées correspondant au processus `Lock`.

Question I.2 Modélisation de la variable M

On modélise le tableau partagé `M`, constitué de deux cellules, comme l'entrelacement de quatre processus LNT, chacun représentant une variable partagée codant un champ d'une des cellules. En d'autres termes, `M` est modélisé par le processus LNT suivant :

```
process Memory [Read_next, Write_next, Read_locked, Write_locked : any] is
   par
      Next [Read_next, Write_next] (Index (0))
   || Locked [Read_locked, Write_locked] (Index (0))
   || Next [Read_next, Write_next] (Index (1))
   || Locked [Read_locked, Write_locked] (Index (1))
   end par
end process
```

Compléter la définition du processus `Locked` ci-dessous, en supposant que chaque variable a initialement la valeur `false` et que les actions de lecture et d'écriture ont respectivement la forme "`Read (pid, value)`" et `Write (pid, value)`", où `pid` a le type `index` et `value` a le type `Bool` :

```
process Locked [Read, Write : any] (pid : Index) is
   -- partie a completer
   ...
end process
```

Remarque : La définition du processus `Next` est similaire (modulo le type des variables locales). Elle n'est pas demandée.

Question I.3 Les procédures acquire_lock et release_lock

La procédure `release_lock` est modélisée par le processus LNT suivant :

```
process release_lock [Read_next, Write_next, Read_locked, Write_locked,
                      Fetch_and_Store, Compare_and_Swap : any] (pid: Index) is
  var next: Index, swap: Bool in
    Read_next (pid, ?next);
    if next == nil then
      Compare_and_Swap (pid, nil, ?swap);
      if swap == false then
        loop L in
          Read_next (pid, ?next);
          if next != nil then break L end if
        end loop;
        Write_locked (next, false)
      end if
    else
      Write_locked (next, false)
    end if
  end var
end process
```

En vous aidant de cette définition de `release_lock`, et en tenant compte des choix de représentation des actions du processus `Lock`, compléter la définition du processus LNT `acquire_lock` ci-dessous :

```
process acquire_lock [Read_next, Write_next, Read_locked, Write_locked,
                      Fetch_and_Store, Compare_and_Swap : any] (pid: Index) is
   -- partie a completer
   ...
end process
```

Question I.4 Modélisation du système complet

On suppose que chaque agent est modélisé comme une instance du processus `Agent` défini comme suit :

```
process Agent [NCS, CS, Read_next, Write_next, Read_locked, Write_locked,
               Fetch_and_Store, Compare_and_Swap : any] (pid: Index) is
  loop
    NCS (pid);
    acquire_lock [Read_next, Write_next, Read_locked, Write_locked,
                  Fetch_and_Store, Compare_and_Swap] (pid);
    CS (pid);
    release_lock [Read_next, Write_next, Read_locked, Write_locked,
                  Fetch_and_Store, Compare_and_Swap] (pid)
  end loop
end process
```

Compléter la définition du processus LNT `Main` ci-dessous, modélisant l'architecture du système constitué de deux agents d'index respectifs 0 et 1 et des processus `Lock` et `Memory` modélisant les variables partagées, en considérant les opérations d'accès aux variables partagées (portes `Read_*`, `Write_*`, `Fetch_and_store` et `Compare_and_swap`) comme des opérations internes.

```
process Main [ ... (* a completer *) ] is
```

```
  -- partie a completer
  ...
end process
```

Partie II Automates temporisés (5 points)

On considère le système formé par les trois automates temporisés (Agent_1, Agent_2 et Mutex) de la figure 1. Chaque transition est étiquetée par un triplet de la forme "g / a ; r", où g est la garde (**tt** dénote la garde qui est toujours satisfaite), a est la synchronisation (en syntaxe LNT), et r est l'instruction à exécuter si la synchronisation a lieu (**null** représente l'instruction vide). Les invariants des états sont écrits à l'intérieur des états. Les états initiaux sont marqués par un trait double.

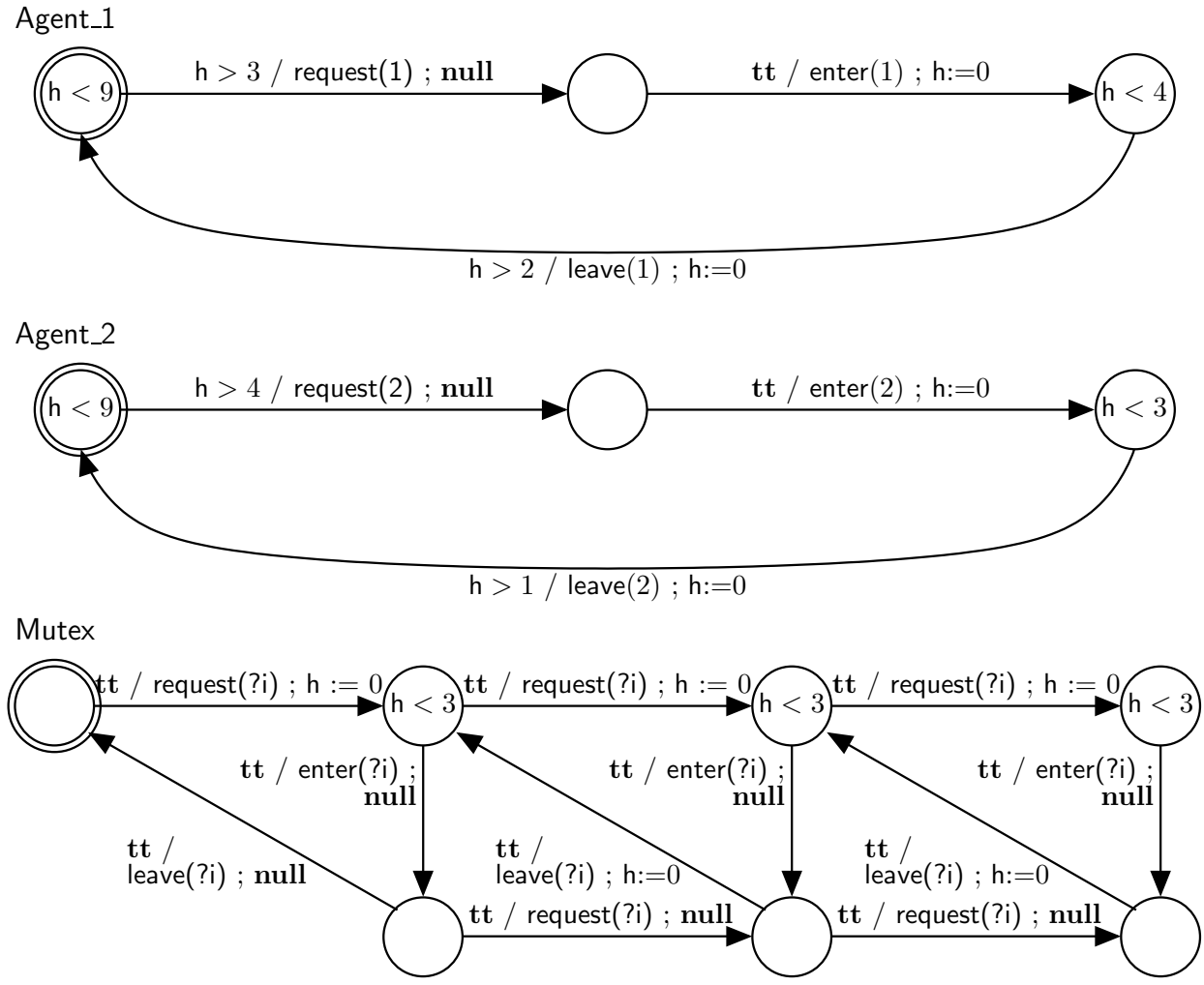


Figure 1: Réseau d'automates temporisés

Chacun de ces automates dispose d'une horloge locale h. L'automate Mutex dispose d'une variable locale i. Les trois automates se synchronisent deux-à-deux sur les portes request, enter et leave (qui attendent toutes un nombre naturel en tant qu'offre), c'est-à-dire la composition parallèle pourrait être décrite par l'expression de comportement LNT suivante :

```
par request, enter, leave in
   par
      Agent_1 [request, enter, leave]
    ||
      Agent_2 [request, enter, leave]
   end par
 ||
   Mutex [request, enter, leave]
end par
```

Est-ce que la séquence de synchronisations suivante est une trace possible de ce système ?

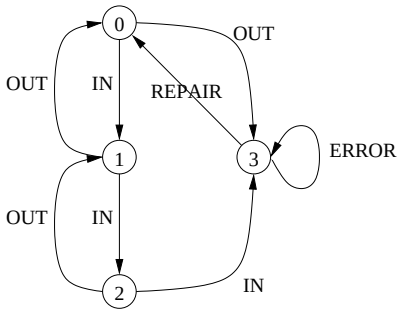
request(2), enter(2), leave(2), request(1), request(2), enter(2)

Le cas échéant, quelles sont les valeurs possibles des horloges Mutex.h, Agent_1.h et Agent_2.h à la fin de cette trace ?

Partie III Logique temporelle (5 points)

On considère l'automate et les huit formules de μ-calcul de la Figure 2.

1. Indiquer pour chaque formule les états de l'automate qui la satisfont.
2. La ou lesquelles de ces formules ont une valeur constante (vrai ou faux), indépendante de l'automate sur lequel elles sont évaluées ?



1. $\langle \text{IN} \rangle$ tt
2. [OUT] ff
3. [IN][IN][ERROR] ff
4. [ERROR] tt
5. $\langle \text{IN} \rangle [\neg \text{ERROR}] \langle \text{OUT} \rangle$ tt
6. $\nu X. \langle \text{IN} \rangle\ X$
7. $[\neg \text{ERROR}] \nu X. \langle \text{IN} \rangle \langle \text{OUT} \rangle\ X$
8. $\mu X. \langle \text{IN} \rangle\ X \vee \langle \text{ERROR} \rangle$ tt

Figure 2: Automate et formules de μ-calcul