Modélisation des systèmes avec NuSMV

Cătălin Dima

LACL, Université Paris-Est Créteil

Éléments clés à prendre en compte en modélisation

- Existence des processus/modules/fonctions indépendent(e)s :
  - Processus tournant en (pséudo)parallélisme.
  - Systèmes de type client-serveur.
  - Systèmes interactifs.
  - Modules d'une architecture matérielle (contrôleur, capteur...).
- Existence des variables (ressources) partagées :
  - Variables possédées par un module et accessibles seulement en lecture par les autres.
  - Variables que plusieurs modules peuvent mettre à jour.
- Situations d'attente bloquante :
  - Instructions de type `wait` : attente (processus n'avance pas) tant qu'une condition n'est pas remplie.
  - Lecture dans "tubes" de communication vides, écriture dans tubes pleins.
  - Instructions de synchronisation type `waitpid`.
- Scalabilité :
  - Nombre de processus pouvant varier.
  - Valeurs de variables non contraintes, mais pouvant influencer le comportement du système.
- Temporisation ou délais.

Modélisation NuSMV avec `MODULE`

Reprenons la solution de Peterson au problème d'exclusion mutuelle :

```
while (true) {
   flag1 := true ;
   turn := 2 ;
   wait ( !flag2 || (turn=1))
      section critique 1
   flag1 := false  ;
}
```

```
while (true) {
   flag2 := true ;
   turn := 1 ;
   wait ( !flag1 || (turn=2))
      section critique 2
   flag2 := false  ;
}
```

- C'est un exemple de système qui pourrait se généraliser (par exemple, nombre de processus $>$ 2).
- On prend la variante *monoprocesseur*, ou exécution par entrelacement.

Modélisation NuSMV avec `MODULE`

- Variables possedées par chaque processus :
  - `flag`i, `pc`i.
- Variables partagées, accessibles par chaque processus à la fois en lecture et en écriture :
  - `turn`.
- Variables “cachées”, nécessaires à la modélisation
  - `tourderole` : indique à qui le tour d’essayer d’exécuter une instruction.
  - `id` : identité du processus, servira à savoir, à l’intérieur de chaque processus, si son tour est venu.

Module processus *i* (avec seulement les variables possédées)

```
MODULE proc(id,trderole,turn,flagautr)
VAR
pc : 0..4;
flag : boolean;
ASSIGN
init(pc) := 0;
init(flag) := TRUE;
next(pc) :=
  case
    (trderole=id) & ((pc!=2) | (pc=2 & ((!flagautr) | turn=id))):
      (pc+1) mod 5;
    TRUE : pc;
  esac;
next(flag) :=
  case
    (trderole=id) & (pc=0) : TRUE;
    (trderole=id) & (pc=4) : FALSE;
    TRUE : flag;
  esac;
```

Module processus *i* : explications

- `flagautr` est la variable qui permet d'avoir accès au drapeau de l'autre processus.
- L'utilité de `id` apparaît lorsqu'on se rend compte qu'il faut avoir un moyen *symétrique* de décider à qui est le tour.
- Pas (encore) de modélisation de l'instruction *turn := i*.

Module `main`

```
MODULE main
VAR
turn : 0..1;
tourderole : 0..1;
flg0 : boolean;
flg1 : boolean;
proc0 : proc(0,tourderole,turn,flg1);
proc1 : proc(1,tourderole,turn,flg0);

ASSIGN
init(turn) := 0;
next(turn) :=
  case
    (proc0.pc=1 & tourderole=0) : 1;
    (proc1.pc=1 & tourderole=1) : 0;
    TRUE : turn;
  esac;

INVAR
(flg0 = proc0.flag) & (flg1 = proc1.flag);
```

Module `main`, explications

- On a déclaré le drapeau de chaque processus à l'intérieur du module, mais chaque processus utilise le drapeau de l'autre...
- On devrait déclarer que `proc1` utilise `proc0.flag`...
- ... mais `proc0` utilise lui aussi `proc1.flag`...
- On est dans une dépendance circulaire...
  - C'est un problème syntaxique (comment déclarer) pas un problème sémantique !
- ... et on la résout en déclarant les deux variables supplémentaires `flg0,flg1`,
- .. qui sont contraintes, par la déclaration `INVAR`, d'avoir toujours les valeurs `proc0.flag`, resp. `proc1.flag`.

Modélisation des situations d'interblocage

Reprenons la solution de Peterson, en éliminant `turn` :

```
while (true) {
   flag1 := true ;
   wait ( !flag2)
      section critique 1
   flag1 := false  ;
}
```

```
while (true) {
   flag2 := true ;
   wait ( !flag1)
      section critique 2
   flag2 := false  ;
}
```

- Peut-on modéliser l'interblocage aussi à l'aide de `ASSIGN` ?

Modélisation des situations d'interblocage

ASSIGN peut permettre de modéliser des situations de "blocage en vie" (angl. *livelock*) : un processus ne fait plus que des affectations de type `next(v) := v` pour toutes ses variables.

```
MODULE proc(id,trderole,flagautr)
VAR
pc : 0..4;
flag : boolean;
ASSIGN
init(pc) := 0;
init(flag) := TRUE;
next(pc) :=
  case
    (trderole=id) & ((pc!=2) | (pc=2 & (!flagautr)):
      (pc+1) mod 5;
    TRUE : pc;
  esac;
next(flag) :=
  case
    (trderole=id) & (pc=0) : TRUE;
    (trderole=id) & (pc=4) : FALSE;
    TRUE : flag;
  esac;
```

Modélisation des situations d'interblocage

```
MODULE main
VAR
turn : 0..1;
tourderole : 0..1;
flg0 : boolean;
flg1 : boolean;
proc0 : proc(0,tourderole,flg1);
proc1 : proc(1,tourderole,flg0);

INVAR
(flg0 = proc0.flag) & (flg1 = proc1.flag);
```

Protocôles de communication

- Découverte des bogues dans certaines spécifications/implémentations de protocôles (e.g. Kerberos).
- Permet surtout de retrouver les situations d'interblocage, grand casse-tête des méthodes de test.
- Motre la puissance du model-checking tant que les modèles restent limités en taille.

Protocôles de communication de bas niveau

- Communication “synchrone”, pas d’attente.
  - Exemples : ABP, CSMA/CD.
- Communication “synchrone”, attente bloquante.
  - Exemples : TCP, communication par tubes, producteur/consommateur.

Alternating Bit Protocol

:

- Un serveur veut envoyer une donnée à un client.
- Les médias de communication peut provoquer des pertes de messages.
- Le serveur commence par envoyer la donnée, suivie d'un bit (initialement 0), continuellement, jusqu'à ce qu'un accusé de réception soit reçu.
- Lorsque l'accusé de réception est reçu, le serveur commence à envoyer la donnée suivante avec le bit changé à 1.
- Le client, lors de la réception du premier message, envoie un accusé de réception (ACK0), et l'envoie continuellement jusqu'à la réception d'un nouveau message.
- Puis, après la prochaine réception de donnée, envoie l'accusé de réception ACK1, etc.

Exemple de propriété à vérifier :

- Si le client reçoit une donnée correcte avec le bit 0, c'est que le serveur l'a bien envoyée auparavant.

Alternating Bit Protocol

Exemple de programme expediteur en pseudocode :

```
                ab :=0
get       :     data := random ;
send      :     sndmedia1d := data ;
                sndmedia1b := ab ;
waitack   :     if (recmedia2a != ab) goto send ;
                ab := 1-ab ;
                goto get ;
```

Exemple de programme recepteur en pseudocode :

```
                abrec :=1
recv      :     if (recmedia1d != err)
                      abrec := 1-abrec ;
                sndmedia2a := abrec ;
                goto recv
```

Et les canaux de communication :

```
trsm      :     recmedia1d := { sndmedia1d, err }
          :     recmedia1b := sndmedia1b ;
                goto get ;
```

Et similaire pour *sndmedia2a/recmedia2a*.

Producteur/consommateur

- Un serveur produit (avec un délai aléatoire) des objets et les place dans une file.
- Un client récupère (avec un délai aléatoire) les objets dans l'ordre d'envoi.
- Le client est bloqué tant que la file est vide.
- La file a une dimension maximale, donc le serveur est bloqué lui aussi tant que la file est pleine.

Exemple de propriété à vérifier :

- Si une donnée d_1 est produite après une autre donnée d_2, d_1 sera consommée après d_2.

Producteur/consommateur

Producteur :

```
start   :   data := random ;
            wait (top <N) ;
            belt[top] := data ;
            top++ ;
            goto start
```

Consommateur :

```
start   :   wait (top >=0) ;
            consume := belt[0] ;
            for (i=1 ;i<top ;i++)
                belt[i-1] := belt[i] ;
            top−− ;
            goto start
```

Attention ! Les sections critiques (en rouge) doivent s'exécuter en exclusion mutuelle.

Modélisation des instructions s'exécutant en temps variable

- Variable supplémentaire `duree`, qui prédit la durée de l'instruction.
- À chaque début d'exécution d'une instruction, la durée est initialisée à une valeur aléatoire, entre 1 et le délai maximal d'exécution de l'instruction.
- Le compteur de programme `pc` avance seulement lorsque `duree` est 0.
- Cas particulier des instructions `pause` – durée non-aléatoire.

```
next(pc) :=
  case (pc = start) & (duree = 0) : at_wait ;
  case (pc = at_wait) & (top < N) & (duree = 0) : at_belt;
  ....
  TRUE : pc;
  esac;
next(duree) :=
  case (pc = at_delay3) & (duree = 0) : 3;
  case (duree = 0) : {1,2,3,4};
  TRUE : duree;
  esac;
```