Spécifications et vérifications formelles
Année 2000-2001 - Mardi 16 janvier 2001.
Documents autorisés.
(Alain Griffault)

SUJET + CORRIGE

Avertissements

1. Vous répondez sur ces feuilles.
2. Les deux exercices sont totalement indépendants. Le premier est un exercice de vérification (8 points) et le second est un exercice de modélisation (7 points). Le total des points possibles (15) est ainsi supérieur à la note maximale possible (10). Vous pouvez donc traiter les questions de votre choix.

1 Vérifications formelles

Soit le système de transition $S =< Q, I, A, T >$ avec :
- l'ensemble des états $Q = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
- l'ensemble des états initiaux $I = \{1\}$,
- l'alphabet des actions $A = \{t\}$ et
- $T = \{(1, t, 2), (2, t, 3), (3, t, 1), (3, t, 4), (4, t, 5), (5, t, 6), (6, t, 5), (6, t, 7), (7, t, 8), (8, t, 9)\}$

représenté par la figure 1.

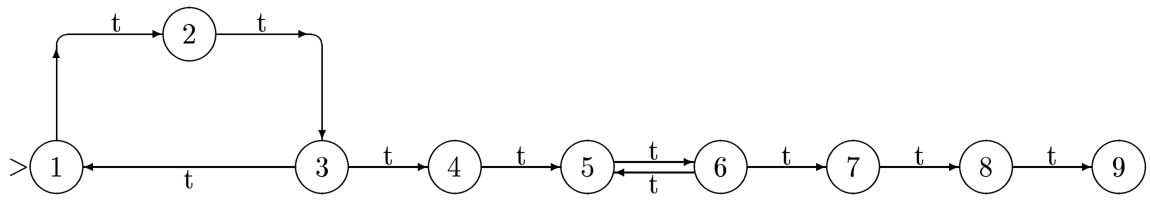

Figure 1: Un système

Avec le vérificateur de modèle Mec, nous effectuons les calculs suivants :

```
dead := *- src(*);
cfc := loop(*,*);
cfcInit := loop(rsrc(initial),*);
```

Question 1.1 (1 point) *Donnez en les résultats.*

Réponse à la question 1.1:

Calcul	Résultat
dead	$\{9\}$
cfc	$\{(1, t, 2), (2, t, 3), (3, t, 1), (5, t, 6), (6, t, 5)\}$
cfcInit	$\{(1, t, 2), (2, t, 3), (3, t, 1)\}$

Un utilisateur désire réécrire la fonction `loop` comme une fonction point fixe. Il charge pour cela la fonction suivante qui est sensée calculer les états appartenant aux composantes fortements connexes inclues dans l'ensemble d'états `dansS`.

```
function Scfc(dansS:state) return resScfc : state;
begin
resScfc = dansS /\ (src(rtgt(resScfc)) /\ tgt(rsrc(resScfc)))
end.
```

puis il effectue le calcul suivant :

```
etatBoucle := Scfc(*);
```

Question 1.2 (1 point) *Donnez le résultat du calcul précédent .*

Réponse à la question 1.2: `resScfc` est initialisée à $\{\}$, (plus petit point fixe). La première itération donne `resScfc` = dansS `/\` $\{\}$ `/\` $\{\}$, soit `resScfc` = $\{\}$. Le point fixe est donc atteint, et le résultat est $\{\}$.

Question 1.3 (1 point) *Modifiez légèrement la fonction précédente afin d'obtenir ce que l'utilisateur pense être la* bonne *fonction. Vérifiez que vous obtenez l'ensemble* $\{1, 2, 3, 4, 5, 6\}$.

Réponse à la question 1.3: Le calcul doit être initialiser avec tous les états. Il faut donc calculer un plus grand point fixe, d'ou le nouveau typage pour `resScfc`

```
function Scfc(dansS:state) return resScfc : _state ;
begin
resScfc = dansS /\ (src(rtgt(resScfc)) /\ tgt(rsrc(resScfc)))
end.
```

Question 1.4 (1 point) *Que calcule la fonction précédente ?*

Réponse à la question 1.4: Les états appartenant soient aux composantes fortements connexes, soient aux états intermédiaires entre elles, inclus dans l'ensemble de états `dansS`.

Question 1.5 (2 points) *L'utilisateur pense qu'avec les transitions, cela sera plus facile. Donnez la fonction* `Tcfc` *qui calcule maintenant les transitions appartenant soient aux composantes fortements connexes, soient aux transitions les reliant, inclues dans l'ensemble de transitions* `dansT`. *Vérifiez que vous obtenez* $\{(1,t,2),(2,t,3),(3,t,1),(3,t,4),(4,t,5),(5,t,6),(6,t,5)\}$

Réponse à la question 1.5:

```
function Tcfc(dansT:trans) return resTcfc : _trans ;
begin
resTcfc = dansT /\ (rtgt(src(resTcfc)) /\ rsrc(tgt(resTcfc)))
end.
```

Pour obtenir la *vraie* fonction `loop` de `Mec`, l'utilisateur à l'idée de calculer l'intersection de ce calcul avec les transitions *accessibles* à partir d'un second ensemble de transitions. Il ajoute donc deux équations à son système et obtient la fonction :

```
function MyLoop(dansT:trans; parT:trans) return resMyLoop :           ;
var resTcfc :
    accParT : trans
begin
resTcfc   =
accParT   = parT \/ rsrc(tgt(accParT));
resMyLoop = resTcfc /\ accParT
end.
```

Question 1.6 (2 points) *Expliquez pourquoi cette solution n'en est pas une.*

Réponse à la question 1.6: La variable `resTcfc` est de type `_trans` et calcule don un plus grand point fixe. La variable `accPart` est de type `trans` et calcule don un plus petit point fixe. La variable `resMyLoop` n'est donc pas d'un type cohérent. L'outil `Mec` refusera cette définition.

2 Modélisations formelles

Un utilisateur cherche à modéliser en AltaRica le problème de la communication par variables partagées. Plus précisément, il cherche à modéliser le système composé de deux processus identiques au suivant :

```
Processus
var aux : integer
begin
aux := compte;
aux := aux + 1;
compte := aux;
end.
```

```
Variable globale
compte : integer;
initialisée à 0
```

Le manuel AltaRica, contient le modèle *Les variables globales* dont voici le descriptif.

Intention Permet la modification et la consultation d'une entité par un ensemble de composants.

Motivation Les variables globales n'existent pas en AltaRica. Leur description cependant ne pose pas de réelle difficulté dès lors que l'on remarque :

- Par raison de symétrie, elle n'a pas de raison d'appartenir à un composant plus tôt qu'à un autre.
- D'un point de vue système, une variable globale est très certainement une variable de contrôle.

De fait, en AltaRica, une variable globale va être déclarée au niveau du contrôleur commun aux composants qui l'utilisent, c'est à dire au niveau du plus proche parent commun. La modification de cette variable sera donc du ressort du contrôleur, et les composants qui désireront la modifier devront déléguer cette tâche au contrôleur par synchronisation. La mise à disposition de sa valeur sera faîte par une variable de flux par composant.

Exemple

```
node A
  flow   ident, lireVG : [0,3];
  state  ecrireVG : [0,3];
  event  ecrire;
  trans  (ident!=lireVG) |- ecrire -> ecrireVG := ident;
edon

node Main
  sub    a1, a2, a3 : A;
  event  a1Ecrit, a2Ecrit, a3Ecrit;
  state  VG : [0,3];
  trans  true |- a1Ecrit -> VG := a1.ident;
         true |- a2Ecrit -> VG := a2.ident;
         true |- a3Ecrit -> VG := a3.ident;
  sync   <a1Ecrit, a1.ecrire>;
         <a2Ecrit, a2.ecrire>;
         <a3Ecrit, a3.ecrire>;
  assert a1.ident = 1;
         a2.ident = 2;
         a3.ident = 3;
         a1.lireVG = VG;
         a2.lireVG = VG;
         a3.lireVG = VG;
edon
```

L'utilisateur construit donc le premier modèle suivant :

```
domain donnee = [0,3] ;

node Processus
  flow   valeurLue, valeurEcrite : donnee;
  state  aux : donnee;
         compteurOrdinal : [1,3];
  event  lire, ecrire;
  trans  compteurOrdinal=1 |- lire ->
                            aux := valeurLue,
                            compteurOrdinal := compteurOrdinal + 1;
         compteurOrdinal=2 |- ecrire ->
                            aux := aux + 1,
                            compteurOrdinal := compteurOrdinal + 1;
  assert valeurEcrite = aux;
  extern initial_state = compteurOrdinal = 1;
edon

node Main
  sub    A, B : Processus;
  state  compte : donnee;
  event  AEcrit, BEcrit;
  trans  true |- AEcrit -> compte := A.valeurEcrite;
         true |- BEcrit -> compte := B.valeurEcrite;
  sync   <AEcrit, A.ecrire>;
         <BEcrit, B.ecrire>;
  assert compte = A.valeurLue;
         compte = B.valeurLue;
  extern initial_state = compte = 0;
edon
```

Question 2.1 (2 points) *Expliquez pourquoi toutes les exécutions de ce modèle AltaRica terminent avec la valeur 0 pour la variable globale* `compte`.

Réponse à la question 2.1: La sémantique AltaRica décrit l'effet d'une transition ainsi :

1. Tous les composants concernés modifient leurs variables d'éats,
2. Toutes les variables de flux sont ensuite (en un temps nul) caculées afin de satisfaire l'ensemble des assertions.

Sur notre exemple,

1. la première lecture copie la valeur de `compte` (0) dans la variable d'état `aux` (0), puis contraint la valeur de `valeurEcrite` à cette valeur (0).
2. la première ecriture copie les valeurs de (`valeurEcrite`,`aux`1)+ (0,1) dans `(compte,aux)` (0,1), puis contraint la valeur de `valeurEcrite` à la nouvelle valeur de `aux` (1).

L'action des processus est donc nulle sur la variable `compte`.

L'utilisateur construit donc le second modèle suivant :

```
domain donnee = [0,3] ;

node Processus
  flow   valeurLue, valeurEcrite : donnee;
  state  aux : donnee;
         compteurOrdinal : [1,4];
  event  lire, modifier, ecrire;
  trans  compteurOrdinal=1 |- lire ->
                             aux := valeurLue,
                             compteurOrdinal := compteurOrdinal + 1;
         compteurOrdinal=2 |- modifier ->
                             aux := aux + 1,
                             compteurOrdinal := compteurOrdinal + 1;
         compteurOrdinal=3 |- ecrire ->
                             compteurOrdinal := compteurOrdinal + 1;
  assert valeurEcrite = aux;
  extern initial_state = compteurOrdinal = 1;
edon

node Main
  sub    A, B : Processus;
  state  compte : donnee;
  event  AEcrit, BEcrit;
  trans  true |- AEcrit -> compte := A.valeurEcrite;
         true |- BEcrit -> compte := B.valeurEcrite;
  sync   <AEcrit, A.ecrire>;
         <BEcrit, B.ecrire>;
  assert compte = A.valeurLue;
         compte = B.valeurLue;
  extern initial_state = compte = 0;
edon
```

Question 2.2 (1 point) *Donnez toutes les valeurs possibles en fin d'exécution pour la variable globale* `compte`.

Réponse à la question 2.2:

Valeurs possibles en fin d'exécution	$\{1,2\}$

L'utilisateur découvre alors dans le manuel le modèle *L'exclusion mutuelle* dont voici le descriptif.

Intention Permet de garantir l'exclusivité (pour l'accès à une ressource, pour l'exécution d'une partie de code, ...) à un processus parmi N.

Motivation Il est très fréquent qu'un processus est besoin d'une garantie d'exclusivité lorsqu'il effectue une partie de son code. Les raisons sont diverses :

- il accède à une ressource non partageable (une imprimante, un fichier, une zone mémoire), il doit donc acquérir un droit d'utilisation exclusif à cet objet.
- il accède à plusieurs ressources liées (des variables reliées par une relation), il doit donc acquérir un droit d'utilisation exclusif afin de préserver cette relation de cohérence.

La solution qui consiste à décrire systématiquement en AltaRica un protocole d'exclusion mutuelle comme celui de Peterson étant évidemment non réaliste, il est nécessaire d'en avoir une abstraction.

Exemple 1 Le mécanisme suivant utilise la synchronisation par diffusion.

```
node Processus
  state  sectionCritique : bool;
  event  entrer, sortir,
         nonCritique1, nonCritique2, nonCritiqueN,
         critique1, critique2, critiqueN;
  trans  ~sectionCritique |- nonCritique1, nonCritique2, nonCritiqueN -> ;
         ~sectionCritique |- entrer -> sectionCritique := true;
         sectionCritique |- critique1, critique2, critiqueN -> ;
         sectionCritique |- sortir -> sectionCritique := false;
  extern initial_state =
         sectionCritique = false;
edon

node main
  sub    p1, p2, pN : Processus;
  state  sectionCritique : bool;
  event  entrer, sortir;
  trans  ~sectionCritique |- entrer -> sectionCritique := true;
         sectionCritique |- sortir -> sectionCritique := false;
  sync   <entrer,p1.entrer?,p2.entrer?,pN.entrer?> = 1;
         <sortir,p1.sortir?,p2.sortir?,pN.sortir?> >= 1;
  extern initial_state =
         sectionCritique = false;
edon
```

Exemple 2 Le mécanisme suivant utilise les flux pour un résultat identique.

```
node Processus
  flow   localFree, globalFree : bool;
  state  sectionCritique : bool;
  event  entrer, sortir,
         nonCritique1, nonCritique2, nonCritiqueN,
         critique1, critique2, critiqueN;
  trans  ~sectionCritique |- nonCritique1, nonCritique2, nonCritiqueN -> ;
         ~sectionCritique & globalFree |- entrer -> sectionCritique := true;
         sectionCritique |- critique1, critique2, critiqueN -> ;
         sectionCritique |- sortir -> sectionCritique := false;
  assert localFree = ~sectionCritique;
  extern initial_state =
         sectionCritique = false;
edon

node main
  sub    p1, p2, pN : Processus;
  flow   free : bool;
  assert free = (p1.localFree & p2.localFree & pN.localFree);
         free = p1.globalFree;
         free = p2.globalFree;
         free = pN.globalFree;
edon
```

Question 2.3 (4 points) *Intégrez un des deux mécanismes dans le second modèle afin d'obtenir un modèle AltaRica correct pour le problème initial.*

Réponse à la question 2.3: Une solution avec le premier mécanisme :

```
domain donnee = [0,3] ;

node Processus
  flow   valeurLue, valeurEcrite : donnee;
  state  aux : donnee;
         compteurOrdinal : [1,4];
         sectionCritique : bool;
  event  lire, modifier, ecrire, entrer, sortir;
  trans  ~sectionCritique & compteurOrdinal=1 |- entrer ->
                           sectionCritique := true;
         sectionCritique & compteurOrdinal=1 |- lire ->
                           aux := valeurLue,
                           compteurOrdinal := compteurOrdinal + 1;
         sectionCritique & compteurOrdinal=2 |- modifier ->
                           aux := aux + 1,
                           compteurOrdinal := compteurOrdinal + 1;
         sectionCritique & compteurOrdinal=3 |- ecrire ->
                           compteurOrdinal := compteurOrdinal + 1;
         sectionCritique & compteurOrdinal=4 |- sortir ->
                           sectionCritique := false;
  assert valeurEcrite = aux;
  extern initial_state = compteurOrdinal = 1,
                         sectionCritique = false;
edon

node Main
  sub    A, B : Processus;
  state  compte : donnee;
         sectionCritique : bool;
  event  AEcrit, BEcrit, entrer, sortir;
  trans  true |- AEcrit -> compte := A.valeurEcrite;
         true |- BEcrit -> compte := B.valeurEcrite;
         ~sectionCritique |- entrer -> sectionCritique := true;
         sectionCritique |- sortir -> sectionCritique := false;
  sync   <AEcrit, A.ecrire>;
         <BEcrit, B.ecrire>;
         <entrer, A.entrer?, B.entrer?> = 1;
         <sortir, A.sortir?, B.sortir?> >=1;
  assert compte = A.valeurLue;
         compte = B.valeurLue;
  extern initial_state = compte = 0,
                         sectionCritique = false;
edon
```

Une solution avec le second mécanisme :

```
domain donnee = [0,3] ;

node Processus
  flow   valeurLue, valeurEcrite : donnee;
         localFree, globalFree : bool;
  state  aux : donnee;
         compteurOrdinal : [1,4];
         sectionCritique : bool;
  event  lire, modifier, ecrire, entrer, sortir;
  trans  ~sectionCritique & globalFree & compteurOrdinal=1 |- entrer ->
                           sectionCritique := true;
         sectionCritique & compteurOrdinal=1 |- lire ->
                           aux := valeurLue,
                           compteurOrdinal := compteurOrdinal + 1;
         sectionCritique & compteurOrdinal=2 |- modifier ->
                           aux := aux + 1,
                           compteurOrdinal := compteurOrdinal + 1;
         sectionCritique & compteurOrdinal=3 |- ecrire ->
                           compteurOrdinal := compteurOrdinal + 1;
         sectionCritique & compteurOrdinal=4 |- sortir ->
                           sectionCritique := false;
  assert valeurEcrite = aux;
         localFree = ~sectionCritique;
  extern initial_state = compteurOrdinal = 1,
                         sectionCritique = false;
edon

node Main
  sub    A, B : Processus;
  flow   free : bool;
  state  compte : donnee;
  event  AEcrit, BEcrit;
  trans  true |- AEcrit -> compte := A.valeurEcrite;
         true |- BEcrit -> compte := B.valeurEcrite;
  sync   <AEcrit, A.ecrire>;
         <BEcrit, B.ecrire>;
  assert compte = A.valeurLue;
         compte = B.valeurLue;
         free = (A.localFree & B.localFree);
         free = A.globalFree;
         free = B.globalFree;
  extern initial_state = compte = 0;
edon
```

Une solution optimisée avec le premier mécanisme : l'action lire (resp ecrire) est interprétée par l'environnement (main) comme l'action qui permet d'entrer dans (resp. de sortir de) la section critique.

```
domain donnee = [0,3] ;

node Processus
  flow   valeurLue, valeurEcrite : donnee;
  state  aux : donnee;
         compteurOrdinal : [1,4];
  event  lire, modifier, ecrire;
  trans  compteurOrdinal=1 |- lire ->
                            aux := valeurLue,
                            compteurOrdinal := compteurOrdinal + 1;
         compteurOrdinal=2 |- modifier ->
                            aux := aux + 1,
                            compteurOrdinal := compteurOrdinal + 1;
         compteurOrdinal=3 |- ecrire ->
                            compteurOrdinal := compteurOrdinal + 1;
  assert valeurEcrite = aux;
  extern initial_state = compteurOrdinal = 1;
edon

node Main
  sub    A, B : Processus;
  state  compte : donnee;
         sectionCritique : bool;
  event  AEcrit, BEcrit, entrer;
  trans  true |- AEcrit -> compte := A.valeurEcrite, sectionCritique := false;
         true |- BEcrit -> compte := B.valeurEcrite, sectionCritique := false;
         ~sectionCritique |- entrer -> sectionCritique := true;
  sync   <AEcrit, A.ecrire>;
         <BEcrit, B.ecrire>;
         <entrer, A.lire?, B.lire?> = 1;
  assert compte = A.valeurLue;
         compte = B.valeurLue;
  extern initial_state = compte = 0,
                         sectionCritique = false;
edon
```

Une solution optimisée avec le second mécanisme : l'action lire (resp ecrire) est interprétée par l'environnement (main) comme l'action qui permet d'entrer dans (resp. de sortir de) la section critique. Il est cependant nécessaire de converser les variables de flux, mais il est possible d'interpréter la variable d'état `sectionCritique` à partir de la variable `compteurOrdinal`.

```
domain donnee = [0,3] ;

node Processus
  flow   valeurLue, valeurEcrite : donnee;
         localFree, globalFree : bool;
  state  aux : donnee;
         compteurOrdinal : [1,4];
  event  lire, modifier, ecrire;
  trans  globalFree & compteurOrdinal=1 |- lire ->
                           aux := valeurLue,
                           compteurOrdinal := compteurOrdinal + 1;
         compteurOrdinal=2 |- modifier ->
                           aux := aux + 1,
                           compteurOrdinal := compteurOrdinal + 1;
         compteurOrdinal=3 |- ecrire ->
                           compteurOrdinal := compteurOrdinal + 1;
  assert valeurEcrite = aux;
         localFree = ~(compteurOrdinal=2 | compteurOrdinal=3);
  extern initial_state = compteurOrdinal = 1;
edon

node Main
  sub    A, B : Processus;
  state  compte : donnee;
  event  AEcrit, BEcrit;
  trans  true |- AEcrit -> compte := A.valeurEcrite;
         true |- BEcrit -> compte := B.valeurEcrite;
  sync   <AEcrit, A.ecrire>;
         <BEcrit, B.ecrire>;
  assert compte = A.valeurLue;
         compte = B.valeurLue;
         A.globalFree = (A.localFree & B.localFree);
         A.globalFree = B.globalFree;
  extern initial_state = compte = 0;
edon
```