Multiprogrammes Corrects par Construction
Exclusion mutuelle de sections critiques

Pierre-Edouard Portier

1 Safe Sluice, un essai pour l'exclusion mutuelle de sections critiques

```
Pre: ⊤

Comp.p:
  *[ snc.p
   ;sc.p
   ]

Comp.q:
  *[ snc.q
   ;sc.q
   ]
```

Objectif : `Comp.p` et `Comp.q` ne doivent pas être dans leurs sections critiques (viz. `sc.p` et `sc.q`) en même temps.

Par ailleurs, nous posons qu'**une section critique termine toujours** (contrairement à une section non critique qui peut ne pas terminer).

Nous introduisons deux variables pour pouvoir modéliser cet objectif sous la forme du maintien d'un invariant système :

```
y.p ≡ Comp.p est dans sc.p
y.q ≡ Comp.q est dans sc.q


Pre: ¬y.p ∧ ¬y.q

Inv: ? R: ¬y.p ∨ ¬y.q

Comp.p:
  *[ snc.p
   ;y.p := ⊤
   ;sc.p
   ;y.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

Pour que `y.p := ⊤` ne puisse pas falsifier `R`, il faut lui ajouter la précondition `¬y.q` qui ne peut être établie que par `Comp.q`. `Comp.p` doit donc attendre que cette assertion soit vérifiée :

```
Pre: ¬y.p ∧ ¬y.q

Inv: R: ¬y.p ∨ ¬y.q

Comp.p:
  *[ snc.p
    ;if ¬y.q -> y.p := ⊤ fi
    ;sc.p
    ;y.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

Interblocage :
Nous en profitons pour introduire le concept d'interblocage. Il y a interblocage si tous les composants sont en même temps en attente dans un if bloquant. Pour montrer l'absence d'interblocage, il faut montrer que la conjonction des conditions d'attente est fausse.

Sur notre exemple, l'interblocage est impossible :

```
  y.p ∧ y.q
= {logique}
  ¬(¬y.p ∨ ¬y.q)
= {déf. de R}
  ¬R
= {inv R}
  faux
```

Progrès :
Nous introduisons maintenant le concept de progrès individuel des composants. Sous l'hypothèse d'un comportement équitable de l'ordonnanceur (viz. tout composant sera toujours un jour ou l'autre élu par l'ordonnanceur), il y a progrès individuel des composants s'il n'existe pas de séquences d'exécutions pour lesquelles un composant serait pour toujours bloqué.

Sur notre exemple, le progrès n'est pas assuré car nous pouvons construire un ordonnancement tel qu'à chaque fois que `Comp.p` a la main il soit bloqué sur son *if-bloquant*, tandis que `Comp.q` exécute alternativement ses sections critiques et non-critiques sans être bloqué.

Atomicité :
Un fragment de programme est dit **atomique** si son exécution ne peut pas être interrompue par l'ordonnanceur. Nous considérerons le plus souvent comme atomique un fragment de code qui ne contient qu'**au plus une lecture ou une écriture sur une variable partagée**.

Sur notre exemple, le *if-bloquant* n'est pas atomique. Par exemple, pour `Comp.p` il contient une lecture sur la variable partagée `y.q` et une écriture sur la variable partagée `y.p`. Ainsi, sous cette définition de l'atomicité, le code proposé n'est pas correct : les deux composants peuvent exécuter leurs sections critiques en même temps...

Pour réduire la granularité de l'atomicité, il est souvent possible d'employer la stratégie : *renforcer une clause de garde d'un if-bloquant ne modifie pas la correction d'un*

multi-programme (bien que cela puisse mettre en danger la présence de progrès et l'absence d'interblocage...).

Pour employer cette dernière stratégie, nous introduisons les variables `x.p` et `x.q` telles que :

```
  ¬x.q ⇒ ¬y.q
= {logique}
  y.q ⇒ x.q
```

...soit établi en précondition de `if ¬y.q -> y.p := ⊤ fi`.

Nous proposons que `y.q ⇒ x.q` soit un invariant système.

Pour éviter que `y.q := ⊤` ne falsifie `y.q ⇒ x.q`, nous pourrions proposer :

```
  if ¬y.p -> y.q, x.q := ⊤, ⊤ fi
```

Mais ce serait s'éloigner un peu plus d'une atomicité à la granularité suffisamment fine...

Ainsi, nous proposons :

```
Pre: ¬y.p ∧ ¬y.q ∧ ¬x.p ∧ ¬x.q

Inv: R: ¬y.p ∨ ¬y.q ,
     y.p ⇒ x.p ∧ y.q ⇒ x.q

Comp.p:
  *[ snc.p
    ;x.p := ⊤
    ;if ¬y.q -> y.p := ⊤ fi
    ;sc.p
    ;y.p := ⊥
    ;x.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

La correction de cette solution est due à la **topologie du multiprogramme** : i.e., seul `Comp.p` (resp. `Comp.q`) peut modifier la variable `x.p` (resp. `x.q`).

Maintenant l'invariant établi, nous pouvons remplacer `¬y.q` par `¬x.q` dans la clause de garde de `Comp.p`. `x` ayant pris le rôle de `y`, cette dernière n'est plus que *figurante* et peut disparaître :

```
Pre: ¬x.p ∧ ¬x.q

Comp.p:
  *[ snc.p
    ;x.p := ⊤
    ;if ¬x.q -> skip fi
    ;sc.p
    ;x.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

Remarque : Avec la disparition des variables `y`, l'invariant $\neg$`y.p` $\vee$ $\neg$`y.q` n'est plus visible. Par ailleurs, nous n'avons pas : $\neg$`x.p` $\vee$ $\neg$`x.q`.

Enfin, si cette transformation maintient bien l'exclusion mutuelle des sections critiques, elle autorise cependant l'interblocage...

2 L'algorithme de Peterson

Pour le programme *Safe Sluice*, l'interblocage est possible car `x.p` $\wedge$ `x.q` peut être vrai. Nous adoptons la stratégie **d'affaiblir la clause de garde du if-bloquant pour éviter l'interblocage**. Cet affaiblissement peut mettre en danger la correction du programme, i.e. ne plus permettre l'exclusion mutuelle des sections critiques.

```
Pre: ¬x.p ∧ ¬x.q

Comp.p:
  *[ snc.p
    ;x.p := ⊤
    ;if ¬x.q ∨ H.p.q -> skip fi
    ; {? R.p.q} sc.p
    ;x.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

Il faut donc faire en sorte que :
— **(i)** L'assertion R.p.q (resp. R.q.p) soit correcte
— **(ii)** $R.p.q \wedge R.q.p \Rightarrow \bot$

Pour que (ii) soit vérifiée, nous cherchons une assertion `R.p.q` aussi forte que possible. En considérant la structure de `Comp.p`, l'assertion `R.p.q` la plus forte serait :

$$\texttt{R.p.q} \triangleq \texttt{x.p} \wedge (\neg\texttt{x.q} \vee \texttt{H.p.q})$$

Seule l'instruction `x.q :=` $\top$ pourrait mettre en danger la correction globale de cette assertion. Ainsi, il faut que :
— **(iii)** `x.q :=` $\top$ de `Comp.q` établisse `H.p.q`
Ainsi, en établissant (iii), (i) sera établi. Essayons d'établir (ii) :

```
  R.p.q ∧ R.q.p
= {déf. de R}
  x.p ∧ (¬x.q ∨ H.p.q) ∧ x.q ∧ (¬x.p ∨ H.q.p)
= {logique}
  x.p ∧ H.p.q ∧ x.q ∧ H.q.p
⇒ {On ne sait rien à propos des x}
  H.p.q ∧ H.q.p
```

Pour établir (ii), il faut donc établir `H.p.q ∧ H.q.p ⇒ ⊥`. Nous préférons cette forme à (ii) car R est une *assertion* dont la forme est fortement contrainte par la structure du programme, tandis que le choix de H semble plus libre.

H.p.q et H.q.p sont des formules inconnues qui font intervenir les variables p et q. Nous venons de découvrir que leur conjonction doit être plus forte que ⊥. Pour progresser dans la dérivation, nous cherchons à exprimer ⊥ en fonction des variables p et q afin d'avoir peut-être une idée sur la forme de H. Nous remarquons que `p≠q`. Ainsi nous avons :

```
H.p.q ∧ H.q.p ⇒ p=q
```

Pour résoudre cette équation en trouvant une forme convenable aux formules inconnues H, nous nous concentrons sur la seule relation connue apparaissant dans cette équation : `p=q`. En observant que l'égalité est transitive, nous proposons d'*introduire une variable v* telle que :

```
H.p.q ≙ v=q
```

Ainsi, (ii) est établie. Il reste à établir (iii) pour vérifier (i). Nous pouvons simplement remplacer `x.q := ⊤` par `x.q,v := ⊤,q`. Ainsi, nous obtenons :

```
Pre: ¬x.p ∧ ¬x.q

Comp.p:
  *[ snc.p
    ;x.p,v := ⊤, p
    ;if ¬x.q ∨ v=q -> skip fi
    ;sc.p
    ;x.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

L'affectation multiple ne peut pas être considérée atomique. Pour résoudre ce problème, nous adoptons la stratégie de renforcement des clauses de garde en introduisant les variables y telles que :

```
¬y.p ⇒ ¬x.p ∧ ¬y.q ⇒ ¬x.q
```

Nous obtenons :

```
Pre: ¬x.p ∧ ¬x.q ∧ ¬y.p ∧ ¬y.q

Comp.p:
  *[ snc.p
    ;y.p:=⊤
    ;x.p,v := ⊤, p
    ;if ¬x.q ∨ v=q -> skip fi
    ;sc.p
    ;x.p:= ⊥
    ;y.p:= ⊥
   ]

Comp.q: Comp.p[p\q]
```

Puis nous remplaçons la garde `¬x.q ∨ v=q` par `¬y.q ∨ v=q`, et les variables x deviennent seulement figurantes et peuvent être retirées du programme. Et nous avons dérivé l'algorithme de Peterson :

```
Pre: ¬y.p ∧ ¬y.q

Comp.p:
  *[ snc.p
    ;y.p := ⊤
    ;v := p
    ;if ¬y.q ∨ v=q -> skip fi
    ;sc.p
    ;y.p := ⊥
   ]

Comp.q: Comp.p[p\q]
```

On vérifie facilement que l'interblocage est impossible :

$$
\begin{array}{ll}
 & \neg(\neg y.q \vee v = q) \wedge \neg(\neg y.p \vee v = p) \\
= & \{\text{De Morgan}\} \\
 & y.q \wedge v \neq q \wedge y.p \wedge v \neq p \\
= & \{v \neq q \wedge v \neq p = \bot\} \\
 & \bot
\end{array}
$$

Montrons que le progrès est assuré : un composant ayant terminé l'exécution de sa section non critique entrera dans sa section critique après un nombre fini d'instructions exécutées par le reste du système. Pour ce faire, nous montrons que sa condition d'attente, si elle n'est pas vraie, sera rendue vraie par l'autre composant et le restera stablement.

Ainsi, considérons `Comp.q` bloqué sur sa condition de passage : `¬y.p ∨ v=p`. Montrons qu'alors `Comp.p` converge vers un état pour lequel cette condition est stablement vraie. Nous observons que la condition d'attente de `Comp.p` termine car il y a absence d'interblocage. Par ailleurs, `sc.p` termine par hypothèse. Ainsi `Comp.p` se réduit à :

```
*[ {¬y.p} snc.p
  ;y.p  := ⊤
  ;v    := p
  ;y.p  := ⊥
 ]
Comp.q: Comp.p[p\q]
```

Ainsi, en un nombre fini d'instructions, `Comp.p` rendra `v=p` invariablement vrai — donc également la condition de passage de `Comp.q` —, ou il sera dans une exécution de `snc.p` qui ne termine pas et `¬y.p` — et donc aussi la condition de passage de `Comp.q` — sera invariablement vraie.