Multiprogrammes Corrects par Construction
Théorie

Pierre-Edouard Portier

1 Correction locale et globale

Un multiprogramme est l'exécution concurrente de plusieurs programmes séquentiels qui peuvent partager des variables.

Une assertion (i.e., une formule de la logique des prédicats utilisée pour annoter un programme, et qui doit être vérifiée après que l'instruction précédent l'assertion ait été exécutée) doit être :

— **localement** correcte, et

— **globalement** correcte.

La **correction locale** considère le composant où apparait l'assertion isolé du reste du système (i.e., des autres composants).

Ainsi, une assertion `P` au tout début d'un composant, avant qu'aucune instruction n'ait été exécutée, pour être correcte doit être impliquée par l'initialisation du multi-programme. Sinon, `P` est la post-condition d'une instruction `S` de précondition `Q`, et il faut vérifier la correction du triplet de Hoare `{Q}S{P}` comme nous le ferions pour un programme séquentiel.

La **correction globale** d'une assertion `{P}` est établie en prouvant que toutes les instructions `{Q}S` des composants autres que celui où apparait `{P}` maintiennent `{P}` : `{P ∧ Q} S {P}`.

`P` est un **invariant système** s'il est impliqué par les assertions vérifiées à l'initialisation du multi-programme et s'il est maintenu par chaque instruction `{Q}S` de chaque composant : `{P ∧ Q} S {P}`.

2 Modification de la sémantique de l'alternative

Pour un multi-programme, la sémantique de l'alternative est modifiée par rapport à la définition que nous en avions donnée pour les programmes séquentiels. Il n'est plus nécessaire qu'au moins une des clauses de garde soit vérifiée. Si aucune clause de garde n'est vérifiée, le composant est bloqué et il sera éventuellement débloqué par les actions d'autres composants qui permettront à au moins une des clauses de garde d'être vérifiée. Ainsi, nous utiliserons la forme suivante de **"if bloquant"** comme mécanisme de synchronisation :

```
if B -> skip fi
```

Cette dernière forme est procéduralement équivalente à :

```
do ¬B -> skip od
```

En termes de triplets de Hoare, nous avons :

```
{B⇒R} if B -> skip fi {R}
```

Nous parlerons maintenant de **précondition libérale la plus faible (wlp)**. `wlp.S.R` est la précondition la plus faible pour que, *si le programme* `S` *termine*, il établisse `R`.

Ainsi, nous pouvons maintenant rencontrer des répétitions sans condition de sortie. Nous noterons `*[S]` pour `do ⊤ -> S od`.

3 Un exemple simple

```
Pre: x=0
Comp.0: *[ x:=x+1 ]
Comp.1: *[ print.x ]
```

Le premier composant, viz. `Comp.1`, doit afficher la suite des nombres entiers naturels. Nous pouvons le spécifier en introduisant une nouvelle variable :

```
Pre: x=0 ∧ y=0
Comp.0: *[ x:=x+1 ]
Comp.1: *[ {? x=y, voir Note.0}
           print.x
         ;y:=y+1
         ]
```

Note.0 : x=y

Correction locale (L) :
Préfixer `x=y` par `if x=y -> skip fi`. C'est-à-dire que la seule façon pour `Comp.1` d'établir `x=y` est d'attendre que `Comp.0` l'ait établie.

Correction globale (G) :
Il faut s'assurer que l'instruction `x:=x+1` de `Comp.0` ne puisse pas falsifier `x=y` :

```
  x=y ⇒ wlp.(x:=x+1).(x=y)
= {déf. :=}
  x=y ⇒ x+1=y
= {arithmétique}
  x≠y
```

Il faut donc ajouter `x≠y` en pré-assertion de `x:=x+1`.

```
Pre: x=0 ∧ y=0

Comp.0: *[ {? x≠y, voir Note.1}
           x:=x+1
         ]

Comp.1: *[ if x=y -> skip fi
           {x=y}
         ;print.x
         ;y:=y+1
         ]
```

Note.1 :

(L) :

Préfixer l'assertion x≠y avec `if x≠y -> skip fi`.

(G) :

L'instruction `{x=y}y:=y+1` de `Comp.1` doit maintenir x≠y :

```
  x≠y ∧ x=y ⇒  wlp.(y:=y+1).(x≠y)
= {logique}
  ⊤
```

```
Pre: x=0 ∧ y=0

Comp.0: *[ if x≠y -> skip fi
           {x≠y}
           x:=x+1
         ]

Comp.1: *[ if x=y -> skip fi
           {x=y}
          ;print.x
          ;y:=y+1
         ]
```