

2.7 Régions Critiques.

Le concept de régions critiques (Bri, Hoa), identique dans sa sémantique à celui des sections critiques (Dij), a été émis dans le souci de fournir au programmeur un outil capable de lui apporter une aide lors de la spécification de la synchronisation. Cette aide réside dans la garantie que toute variable déclarée explicitement partageable sera utilisée en exclusion mutuelle. Autrement dit, le mécanisme d'exécution des régions critiques veillerait à ce que ces variables soient manipulées correctement. Ce mécanisme, intégrable au compilateur, avvertirait donc le programmeur de toute utilisation d'un objet déclaré partageable en dehors d'une région critique.

Une région critique est donc une structure linguistique qui définit une séquence d'instructions à exécuter comme section critique dans laquelle sont manipulées des variables déclarées explicitement partageables.

Les variables partagées sont explicitement placées dans des groupes appelés ressources. Chaque variable partagée peut être au plus dans une ressource. Par exemple, une ressource R contenant les variables V1, V2, ..., Vn est déclarée par: *Ressource R: V1,V2, ..., Vn*. Les variables de R peuvent être accessibles seulement à partir des formes structurales de régions critiques désignant ces variables.

Structure des régions critiques (syntaxe et sémantique):

Pour déclarer une variable partagée V du type T on utilise la notation suivante:

Var V : *shared* T qui définit une variable V du type T;

par exemple: **var** compte : *shared real*.

Il est possible de considérer T comme une structure regroupant un ensemble de variables partagées distinctes (identiques relativement au partage) comme l'illustre la notation suivante:

Type direction = *record* % Cf. exercice passage de véhicules sur un pont %

 attente, passant, en_tete: *integer*

end

Var pont : *shared record*

 gauchedroit, droitgauche : direction;

end

On définit une région critique à l'aide des syntaxes suivantes:

2.7.1 Région critique inconditionnelle

Region V do Sc

dont la sémantique exprime que la séquence d'instructions Sc dans laquelle est (ou sont) manipulée(s) la (ou les) variable(s) partagée(s) V, constitue une section critique (cf. § 2.4).

La séquence d'instructions Sc contient des variables déclarées partageables et éventuellement des variables locales au processus exécutant Sc.

Plusieurs régions critiques peuvent se rapporter à une même variable partagée; leurs exécutions s'excluent mutuellement. Autrement dit, si plusieurs processus tentent de les exécuter simultanément, un seul d'entre eux est autorisé à le faire (évolue normalement); les autres processus seront automatiquement bloqués. Lorsque le processus actif termine l'exécution de la séquence critique *sc*, un des processus en attente est systématiquement réactivé.

2.7.2 Région critique conditionnelle

a) *Region V when cond do sc*

signifie: l'exécution de la région critique *sc* est conditionnée par la valeur de la condition *cond*. *Cond* représente une expression booléenne composée d'éléments de *V* et éventuellement de variables locales et constantes. Le scénario d'exécution de cette primitive par un processus appelant *P* est comme suit:

P accède à la région critique *sc* puis évalue *cond*, si *cond* est vraie, alors *P* exécute *sc* puis quitte *sc*.

sinon (*cond* est fausse) *P* sort de la région critique et se bloque jusqu'à ce qu'il soit réveillé lorsqu'un autre processus *Q* sort de sa propre région critique associée à *V*. *P* reprend alors l'évaluation de *cond* à son début.

b) *Region V do sc await cond ...*

Cette instruction combine les deux formes précédentes (2.7.1 et 2.7.2).

Le processus appelant exécute donc *sc* de façon inconditionnelle puis, pour poursuivre, se met en attente (se bloque) jusqu'à ce que la condition *cond* devienne vraie. Il est possible que *sc* soit une opération nulle auquel cas, l'évolution du processus appelant dépendra de la valeur de *cond* (attente si *cond* est fausse, poursuite en séquence sinon).

Remarque: Des régions critiques peuvent s'imbriquer à la manière des boucles DO des langages de programmation de haut niveau, toutefois une attention particulière peut leur être accordée afin d'éviter les interblocages éventuels. Par exemple, soient deux processus concurrents *P* et *Q* se partageant deux variables *v1* et *v2* tels que leurs codes soient:

P: Region *v1* do Region *v2* do *sc1*;

Q: Region *v2* do Region *v1* do *sc2*;

Dans pareil cas, il est évident de constater que si *P* et *Q* entrent simultanément dans leur propre région critique, un interblocage est inévitable. Il est possible de charger le compilateur qui gère les régions critiques de détecter des situations d'interblocage et d'en informer le programmeur pour agir en conséquence. Une action possible, et à titre indicatif, serait d'imposer un ordre d'utilisation des variables partagées (cf. exercice 14 a1)

Les programmes contenant des éléments de synchronisation spécifiés en termes de régions critiques sont plus lisibles. En effet, il est possible d'utiliser une approche axiomatique, basée sur la notion d'invariants, pour prouver la validité de l'exécution de la séquence d'instructions

Sc. Ainsi, on peut associer un invariant I_V à l'état de chaque ressource V , et un prédicat P . Le prédicat P ayant la valeur vraie à l'initialisation de V doit également avoir la valeur vraie à la fin de l'exécution de Sc .

Les régions critiques ont été implantées dans le langage Edison [Brin 1981] conçu spécialement pour des systèmes multiprocesseurs. Des variantes ont été également adaptées pour des environnements distribués [Bri, 1978; Lis 1982]

Exercice: Simuler un sémaphore à l'aide de régions critiques.

var sem: *shared integer* (sem ≥ 0);

P(sem): *region* sem **do** sem := sem - 1 *await* sem ≥ 0 ... (1) % incorrect %

Bien que la forme (1) précédente, traduisant textuellement la primitive P originelle, semblerait « correcte » indépendamment de $V(\text{sem})$, elle est à rejeter puisqu'elle ne répond pas à une association correcte avec la primitive de réveil V ci-après:

V(sem): *region* sem **do** sem := sem + 1;

En effet, l'exécution de V simulée précédemment a pour objectif de libérer un processus Q éventuellement en attente. D'après le principe des régions critiques, Q réveillé, doit évaluer sa condition d'entrer en section critique. Si celle-ci est vraie, il exécute effectivement sa section et poursuit en séquence; dans le cas contraire (condition fausse), il entre de nouveau dans la file d'attente.

Quand plusieurs processus se trouvent bloqués par P , impliquant $\text{sem} < -1$, l'exécution de V , à la suite de laquelle sem est incrémenté ($\text{sem} := \text{sem} + 1$) provoquerait des réévaluations des conditions de franchissement des points de synchronisation ineffectives, bien que logiquement, un franchissement au moins, devrait avoir lieu.

Par exemple, soient trois processus concurrents, P_1 , P_2 , P_3 , désireux d'entrer dans leur section critique respective (sem initialisé à 1), admettant que P_1 soit le premier à exécuter la forme (1) précédente, sem devient = 0 et P_1 accède à sa section critique (condition vraie). Si P_2 exécute la forme (1) pendant que P_1 est encore en section critique, sem devient = -1 et P_2 se bloque (condition fausse). Si P_3 exécute également (1), il se bloque car $\text{sem} = -2$. Lorsque P_1 quitte sa section critique, il doit exécuter $V(\text{sem})$ qui incrémente sem (sem devient égal à -1). Le mécanisme d'exécution des régions critiques va réveiller P_2 qui évalue sa condition de franchissement ($\text{sem} \geq 0$) et trouve $\text{sem} = -1$ (condition fausse), P_2 libère sa section critique et se bloque une nouvelle fois. P_3 fait la même chose que P_2 , trouve $\text{sem} = -1$, et se bloque également. On voit bien que la section critique est libre et aucun des processus ne peut y accéder. P_1 , lui-même, ou tout autre nouveau processus concurrent ne peut accéder à sa section critique, il se produit donc un blocage. Ce problème est dû au fait que la décrémentation de sem s'est faite avant le test de la condition de franchissement. La solution correcte est donc la suivante:

Region sem **when** sem > 0 **do** sem := sem - 1 ...