Théorie et pratique de la concurrence

Mihaela Sighireanu

Mihaela.Sighireanu@liafa.jussieu.fr

http://www.liafa.jussieu.fr/~sighirea/cours/concur/

Bureau 6A7, Chevaleret

Introduction

Concurrence

Programme concurrent = deux ou plusieurs processus séquentiels qui coopèrent pour accomplir une tâche.

- en utilisant les mêmes variables (variables partagées)
- en échangeant des messages

Concurrence = abstraction du parallelisme

- parallelisme = exécution concomitante sur plusieurs processeurs
- concurrence = parallelisme potentiel, par entrelacement des exécutions

Exemples :

- SE : calcul et opérations d'entrées-sorties
- SE : multi-tâches, multi-thread
- multi-processeurs
- grappes d'ordinateurs
- réseau d'ordinateurs

Programmation concurrente = jeu

- règles = outils formels d'analyse
  - état du programme $\longrightarrow$ assertion (prédicat)
  - instruction $\longrightarrow$ transformateur de prédicat
- pieces = méchanismes d'un langage de programmation
  - sémaphore, region critique, moniteur, rendez-vous, réception/envoi message
- stratégies = paradigmes de programmation

Historique

Origines : années 1960 ⟶ révolution technologique du matériel ⟶ systèmes d'exploitations

Quelque noms :

- Edsger Dijkstra : 1965 exclusion mutuelle, 1968 sémaphores, 1972 class, ...
- C.A.R. Hoare : 1969 "Hoare logic", 1972 région critique, 1974 moniteur, 1978 CSP, ...
- Brinch Hansen : 1970 SE, 1972 région critique, 1978 moniteur, ...
- Leslie Lamport : 1974 exclusion mutuelle, 1977 logic and program properties, 1980 horloges logiques, ...
- Dekker, Petterson, Floyd, ...

Synchronisation

Coopération ⇒ synchronisation

Rôle de la synchronisation = restreindre les exécutions possibles d'un programme concurrent à ceux qui sont désirables.

Exemples :

- synchronisation par exclusion mutuelle = l'accès à des régions critiques d'instructions est fait de manière exclusive
- synchronisation sur condition = un processus est mis en attente jusqu'une condition devient vraie

Problème important de la programmation concurrente : implémenter correctement la synchronisation

Abstraction des programmes concurrents

Permet de raisonner sur le comportement des programmes.

Éléments de l'abstraction :

- Instruction atomique = instruction effectuée sans interruption.
- Processus séquentiel = séquence d'instructions atomiques.
- Programme concurrent = ensemble fini de processus séquentiels.
- Exécution d'un programme concurrent = entrelacement arbitraire des actions atomiques des processus séquentiels qui respecte l'ordre imposé par chaque processus
- Calcul ou scénario = une séquence d'exécution

Variables :

- globales (partagées)
- locale aux processus

Exemple :

- Processus `p = p1;p2`
- Processus `q = q1;q2`
- Programme concurrent = {`p,q`}

Questions :

- Quelles sont les exécutions possibles ?
- Est-ce que `p2;p1;q1;q2` est un scénario ?

Remarque : pour un programme avec n processus, chacun ayant m actions atomique, le nombre de scénarios du programme est $(n * m)!/(m!)^n$.

Définition formelle des éxécutions

Exécution d'un programme concurrent = états et transitions entre les états

Etat d'un programme concurrent = tuple dont les éléments sont

- compteur de programme pour chaque processus
- une valeur pour chaque variable globale
- une valeur pour chaque variable locale

Transition entre deux états s_1 et s_2 = éxécution d'une instruction atomique au compteur programme de s_1 qui change l'état s_1 en s_2

Diagramme d'états = graphe orienté défini par :

- noeud initial = état initial des processus et des variables;
- à chaque état lui correspond un noeud du graphe;
- si s_1 est un état d'un noeud du graphe et il existe une transition de s_1 vers s_2, alors il existe un arc orienté du noeud qui contient s_1 vers le noeud qui contient s_2.

Instructions atomiques

Par abstraction :

Toute instruction d'affectation et l'évaluation d'une expression booléenne.

Remarque importante : abstraction non réaliste mais utile.

Exemples qui contredisent cette abstraction :

- calculs complexes d'expressions
- variables non-atomiques = variables sur plusieurs mots de mémoire
- optimisation du compilateur

Pour obtenir l'atomicité :

- déclaration de type volatile pour une variable
- synchronisation par exclusion mutuelle sur un calcul

Propriétés des programmes

= attributs vrais pour toute trace possible du programme.

Deux classes spéciales :

- sûreté (*safety*) = “le programme ne passe jamais dans un état indésirable”
  - correction partielle = si un programme termine, alors il calcule le résultat attendu
  - exclusion mutuelle = un seul processus exécute une section critique à un moment donné
  - absence de blocage = dans tout état on peut exécuter une action
- vivacité (*liveness*) = “le programme passera forcement dans un état désirable”
  - terminaison = toute trace d’un programme est finie
  - absence de famine = tout processus peut entrer dans la section critique

Théorème : Toute propriété d’un programme peut être exprimée en termes de propriétés de sûreté ou de vivacité.

Démontrer des propriétés

Choix 1 : tester et déboguer

- usuel
- méthode partielle

Choix 2 : tester exhaustivement

- méthode automatisable = *model-cheking* (SPIN)
- peut exploser

Choix 3 : analyse par raisonnement mathématique

- représentation compacte, proportionnelle à la taille du programme
- méthode non-automatisable

Processus séquentiels
— construction et analyse —

Langage de programmation (1/2)

Types de base : **bit** ou **bool**, **byte**, **short**, **int**, **mtype**

```
mtype = { ack, nak, err };
```

Déclaration de constantes : macro-définition C

```
#define N 10
```

Déclaration de variables : C-like

```
bool in1, in2;
int turn;
byte ticket[N];
```

Déclaration de processus/procédures : **proctype**

```
proctype A (byte state; short foo)
{ (state == 1) -> state = foo }
```

avec instanciation dans le processus **init** :

```
init
{ run A (2, 3) /*; run A (1, 5) */ }
```

Langage de programmation (2/2)

Instructions :

- vide : **skip**
- affectation : **=**

```
ticket[i] = state + 1
```

- séquencement : **;** ($\neq$ avec C)

```
x = x+1; y=y+1
```

- sélection : **if**

```
if
:: x >= y -> m = x
:: y >= x -> m = y
fi
```

- répétition : **do**

```
do
:: x > y -> x = x-y
:: y > x -> y = y-x
/* :: x == y -> break */
od
```


Logique, propositions, prédicats

Une logique de programme est un système formel qui permet d'analyser et construire des programmes.

Système formel logique (SFL) =

- ensemble de symboles, $\mathcal{S}$
- ensemble de formules, $\mathcal{F}$, construites à partir de $\mathcal{S}$
- ensemble de formules spéciales, nommées axiomes, $\mathcal{A}$
- ensemble de règles d'inférence, $\mathcal{R}$, de la forme

$$\frac{H_1, H_2, \ldots, H_n}{C}$$

Une preuve dans un SFL = ensemble de lignes, chacune soit une axiome, soit étant dérivée des lignes précédentes par l'application d'une règle d'inférence.

Un théorème dans un SFL = ligne d'une preuve.

Une interprétation d'un SFL, $I : \mathcal{F} \rightarrow \{T, F\}$.

Pour une interprétation I, un SFL est :

- valide (*sound*) si $\mathcal{A}$ et $\mathcal{R}$ sont valides par I
- complet (*complete*) si $I(\phi) = T$ alors ϕ est prouvable

Logique propositionnelle

C'est un SFL où

$\mathcal{S} = \{T, F\} \cup Prop \cup \{\neg, \wedge, \vee, \Rightarrow, =\}$

L'interprétation d'une formule $P \in \mathcal{F}$ dans un état s = remplacer les variables par leurs valeurs dans s.

$P \in \mathcal{F}$ est satisfaite dans s état si elle est vrai dans s.

$P \in \mathcal{F}$ est satisfaisable s'il existe un état dans lequel est satisfaite.

$P \in \mathcal{F}$ est valide (tautologie) si P est satisfaite dans tous les états.

Rappel tautologies (lois d'équivalence)

Négation	$P = \neg(\neg P)$
Choix binaire	$P \vee \neg P = T$
Contradiction	$P \wedge \neg P = F$
Implication	$P \Rightarrow Q = \neg P \vee Q$
Egalité	$(P = Q) = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$
Simplification OU/ET	$P \vee P = P \quad P \wedge P = P$
	$P \vee T = T \quad P \wedge T = P$
	$P \vee F = P \quad P \wedge F = F$
	$P \vee (P \wedge Q) = P$
	$P \wedge (P \vee Q) = P$
Commutativité	$\vee, \wedge, =$
Associativité	$P \wedge (Q \wedge R) = (P \wedge Q) \wedge R$
	$P \vee (Q \vee R) = (P \vee Q) \vee R$
Distributivité	$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
	$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
De Morgan	$\neg(P \wedge Q) = \neg P \vee \neg Q$
	$\neg(P \vee Q) = \neg P \wedge \neg Q$
Elimination ET	$(P \wedge Q) \Rightarrow P$
Introduction OU	$P \Rightarrow (P \vee Q)$

Logique des prédicats

Prédicat = expression rélationnelle ou quantifié
$((\exists B :\ R :\ P), (\forall B :\ R :\ P))$.

Logique de prédicats = logique propositionnelle où les propositions sont remplacées par des prédicats.

Exemple de formule :
$(i = j)\ \wedge\ (\forall i, k :\ \ i \geq k \geq 0 :\ \ a[i] \geq a[k])$

Tautologies :

De Morgan	$(\exists B :\ R :\ P)\ =\ \neg(\forall B :\ R :\ \neg P)$
	$(\forall B :\ R :\ P)\ =\ \neg(\exists B :\ R :\ \neg P)$
Conjonction	$(\forall B :\ R :\ P \wedge Q)$
	$=\ (\forall B :\ R :\ P) \wedge (\forall B :\ R :\ Q)$
Disjonction	$(\exists B :\ R :\ P \vee Q)$
	$=\ (\exists B :\ R :\ P) \vee (\exists B :\ R :\ Q)$
Domain vide	$(\forall B :\ \emptyset :\ P)\ =\ T$
	$(\exists B :\ \emptyset :\ P)\ =\ F$
Dualité	$(\exists B :\ R :\ P)\ =\ \neg(\forall B :\ R :\ \neg P)$

P_e^x = substitution dans P d'une variable *libre* x par une expression e (ayant d'autre variables que celles *liées* par P).

Logique de programmation

SFL permettant de raisonner sur l'exécution des programmes.

- symboles : prédicats, $\{,\ \ \}$, instructions du langage
- formules : $\{P\}\ \ S\ \ \{Q\}$
  (P, Q prédicats, S instruction simple ou composée)
  dont l'interprétation est vraie si $s_b \models P$ et $s_b \xrightarrow{S} s_e$ alors $s_e \models Q$.
- axiomes :

  Skip : $\{P\}$ **skip** $\{P\}$

  Affectation : $\{P_e^x\}\ \ x = e\ \ \{P\}$
- règles d'inférence :

  Conséquence : $\dfrac{P' \Rightarrow P,\ \{P\}\ S\ \{Q\}\ ,Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$

  Composition : $\dfrac{\{P\}\ S_1\ \{Q\}\ ,\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;S_2\ \{R\}}$

  Choix : $\dfrac{P \wedge \neg(B_1 \vee \cdots \vee B_n) \Rightarrow Q\ ,\ \{P \wedge B_i\}\ S_i\ \{Q\},\ 1 \leq i \leq n}{\{P\}\ IF\ \{Q\}}$

  Itération : $\dfrac{\{I \wedge B_i\}\ S_i\ \{I\},\ 1 \leq i \leq n}{\{I\}\ DO\ \{I \wedge \neg(B_1 \vee \cdots \vee B_n)\}}$

Exemple de preuves dans la LP

On veut prouver $\{P\}\ S\ \{LS\}$ où

- P est $n > 0 \wedge (\exists j :\ 0 \leq j < n :\ a[j] = x)$
- S est

```
i=0;
do
:: a[i] != x -> i=i+1
od
```

- LS est $a[i] = x \wedge (\forall j :\ 0 \leq j < i :\ a[j] \neq x)$

Exemple d'esquise de preuves dans la LP

$\{\ P :\ n > 0\ \wedge\ (\exists\ j :\ 0 \leq\ j < n :\ a[j] = x)\ \}$
`i = 0;`
$\{\ P\ \wedge\ i = 0\ \}$
$\{\ I :\ P\ \wedge\ (\forall\ j :\ 0 \leq\ j < i :\ a[j] \neq x)\ \}$
do
`:: a[i] != x ->` $\{\ I\ \wedge\ a[i] \neq x\ \}$
`i=i+1`
$\{\ I\ \}$
od
$\{\ I\ \wedge\ a[i] = x\ \}$
$\{\ LS :\ a[i] = x\ \wedge\ (\forall\ j :\ 0 \leq\ j < i :\ a[j] \neq x)\ \}$