Notes de cours : théorie et pratique de la concurrence

François Laroussinie
francois.laroussinie@liafa.jussieu.fr

version provisoire : 28 février 2012

Table des matières

1 Introduction

- { notion de programmes concurrents
- { semantique d'entrelacement
- { adequation de cette semantique aux dierents types de systemes concurrents (systemes multi-tâches, systemes multi-processeurs, systemes distribues)
- { importance de la notion d'atomicite (de certaines instructions ou groupes d'instructions)
- { utilisation de la logique temporelle LTL (de temps lineaire) pour la speci cation { lenonce { de proprietes sur les executions des systemes

Dans la suite, on va distinguer les algorithmes d'exclusion mutuelle en fonction de la methode utilisee par les processus pour communiquer. Il y a :

- { les algorithmes utilisant des variables set : tous les processus ont acces en lecture comme en ecriture a ces variables partagees (on parle de variables \Multiple-Reader/Multiple-Writers"). C'est le cas des algorithmes de Dekker et Peterson.
- { les algorithmes ou les processus utilisent des variables p : tous les processus ont acces en lecture a ces variables mais seul un processus (le proprietaire) peut modi er une variable donnee (on parle de variables \Multiple-Reader/Single-Writer"). C'est le cas de l'algorithme de la Boulangerie.
- { les algorithmes distribues bases sur la communication par messages.

Dans le cours, nous verrons aussi d'autres approches basees sur l'utilisation de semaphores, de moniteurs ou d'instructions \Test-and-set".

2 Exemples de programme concurrent

La gure 1 decrit un programme concurrent P compose de deux processus, chacun contient des instructions qui manipulent une variable partagee x.

```
                int x := 2        // variable partagee

-- Proc. A                              -- Proc. B
a1: x := x+1                            b1: x := x*3
a2: x := x+2                            b2: end
a3: end
```

Figure 1 { Programme P

Quel est le comportement de P ? Quelle est la valeur nale de x apres l'execution de P ? Pour repondre a ces questions, il faut xer des conventions sur ce que signi e l'execution de ces processus. Ici on on supposera qu'une telle execution consiste en un entrelacement des actions de chaque processus : chaque processus execute une ou plusieurs instructions, puis c'est a un autre d'avancer de Les instructions sont donc executees les unes a la suite des autres et non pas en même temps... Cette semantique repose sur une notion importante : les instructions atomiques . Une instruction atomique est une instruction qui ne peut être interrompue : une fois commencee, elle est executee completement. Par exemple, on peut supposer que la lecture du contenu d'une variable ou son a ectation sont des operations

atomiques (on parle de registres atomiques). Mais qu'en est-il d'une instruction comme x=x+1 ou x:=x*3 ? Ici il y a, a la fois lecture de x, calcule d'une valeur puis a ectation... On peut donc supposer qu'une telle instruction se decompose en (au moins) deux etapes : la lecture de x et son a ectation dans une variable locale au processus, qui sera utilisee pour le calcul, puis l'a ectation du resultat dans x. On peut donc le representer par :

```
temp := x
x := temp + 1
```

Un tel choix est important sur le comportement global du prohramme. A titre d'exemple, on donne a la gure 3 le diagramme d'etat du programme precedent lorsque chaque instruction a1, a2 et b1 est consideree comme etant atomique. Et a la gure 4, on donne le diagramme d'etat correspondant a la semantique utilisant des temp, c'est-a-dire au programme P' de la gure 2.

```
int x := 2          // variable partagee

-- Proc. A                          -- Proc. B
a1: temp := x                       b1: temp := x
a1': x := temp +1                   b1': x := temp*3
a2: temp := x                       b2: end
a2: x := temp+2
a2: end
```

Figure 2 { Programme P

Figure 3 { Diagramme d'etats de P.

On voit donc que dans le premier cas (P sans variable temp), les valeurs nales possibles pour x sont 9, 11 ou 15. Mais pour P', elles sont :5, 6, 8, 9, 11 et 15 !

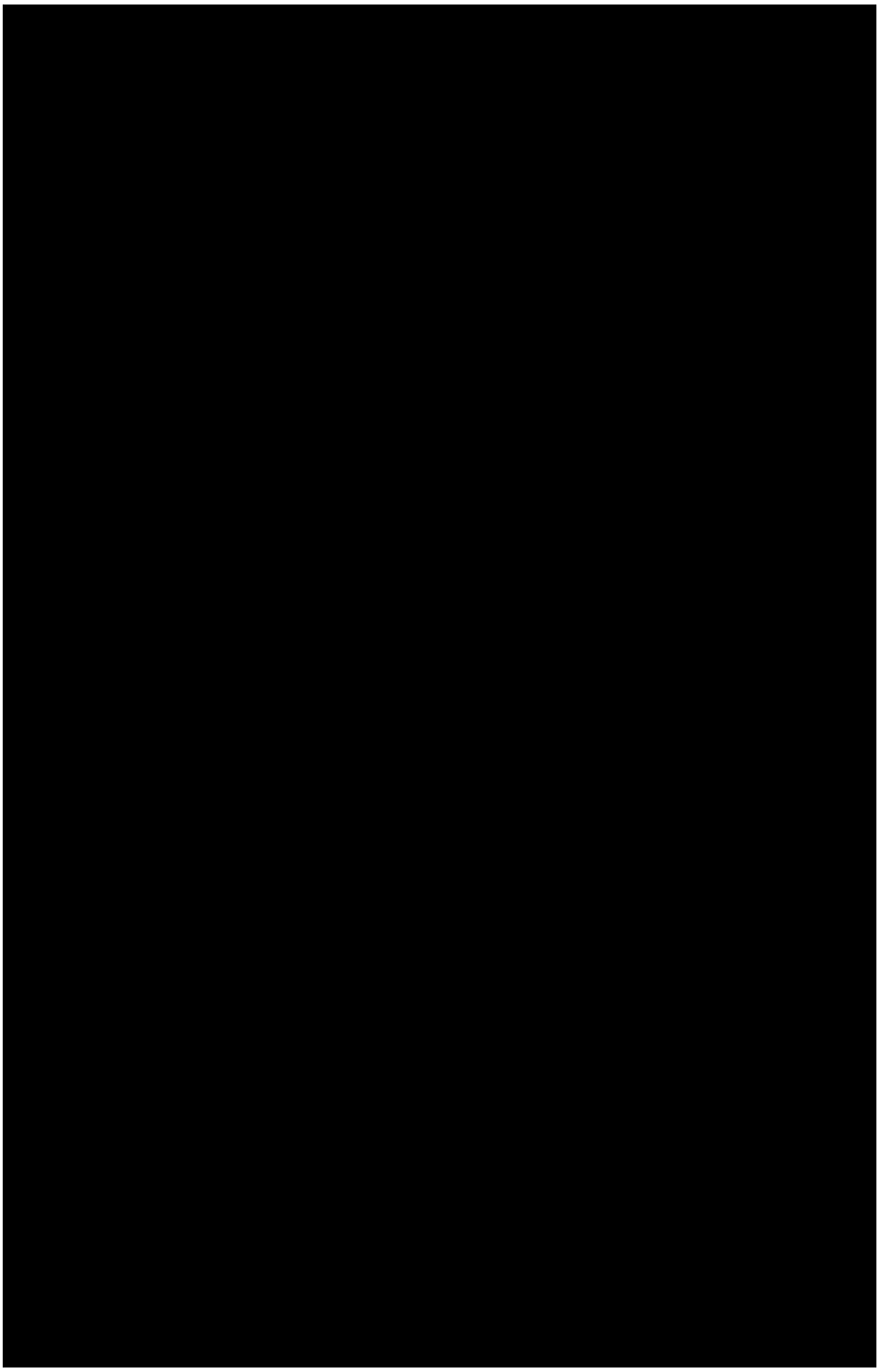

Figure 4 { Diagramme dètats de P^0.

Le choix des instructions atomiques est donc important, il in ue directement sur le com-

portement des programmes concurrents. Il n'y a pas de regle absolue. En fait, cela depend du langage, du compilateur et de la machine. On peut aussi faire des choix dierents pour avoir un programme plus ou moins abstrait. En general il est raisonnable deviter les instructions manipulant plusieurs variables (ou occurrence de variables) pouvant modiees et lues par plusieurs processus (par exemple, dans3, les instructions sont problematiques car x apparaît plusieurs fois dans chacune).

Dans la suite, on supposera { sauf indication contraire { que chaque instruction listee dans les processus est atomique.

3 De nitions

On a vu les points suivants :

- { Un programme concurrent est un ensemble ni de
- { Chaque processus est decrit par un ensemble ni d
- { Executer un programme concurrent consiste a entrelacer des instructions de chaque processus.
- { Un e (ou un \calcul") est une sequence d'instructions de chaque processus resultant d'un entrelacement des instructions de chaque processus.

Chaque processus dispose d'u (ou compteur de programme ou pointeur de contrôle) designant la prochaine instruction devant être executee. Au cours d'une execution d'un programme concurrent, la prochaine instruction est choisie parmi celles pointees par un des pointeurs de contrôleur.

On formalise ici quelques de nitions liees au diagramme d'etats d'un programme concurrent :

De nition 1

- { s_1 e s_2 P , s_2 s_1 e $s_1 \rightarrow_P s_2$) s_2 s_1 s_1 e s_2.
- { P $S_P = (S; A; s_0)$ uo $s_0 \in S$ P S e A
  - { s $s \in S$, S s' e $s \rightarrow_P s'$, e
  - { A $(s; s') \in$ e $(s; s') \in S$ e $s \rightarrow_P s'$.
- { S P .

Les chemins de S_P correspondent donc a d'actions de chaque processus. Ce point est tres important : cela signi e que tout entrelacement est possible... En pratique, ce n'est pas le cas (l'ordonnancement des actions des dierents processus repond a certaines regles) mais le choix de considerer tous les entrelacements possibles est robuste : une fois que nous avons prouve qu'ils veri ent, tous, les proprietes recherchees, nous savons que le programme reel les veri era aussi.

Comment décrire un programme concurrent ? On va utiliser un pseudo-code (comme en algorithmique) avec des primitives spéciales, notamment await (voir ci dessous) pour mettre le processus en attente d'une condition. Nous en verrons d'autres lorsque nous nous intéresserons aux algorithmes basés sur des échanges de messages entre processus. En TP, on utilisera Java. Nous verrons aussi le langage Promela utilisé dans l'outil jSPIN (basé sur l'outil SPIN) :

- jSPIN : http://stwww.weizmann.ac.il/g-cs/benari/jspin/
- SPIN : http://spinroot.com/

Les programmes Promela des algorithmes vus en cours seront disponibles sur la page web du cours : http://www.liafa.jussieu.fr/~francois l/m1tpc.html

Remarque : l'instruction await C est ici équivalente à while : C : skip; . Il s'agit d'une attente active où le processus testera régulièrement la valeur de C tant que celle-ci n'est pas égale à VRAI.

4 Problèmes de section critique

Cette partie s'appuie en grande partie sur [1].

Les algorithmes présentés ne sont pas utilisés en vrai (il existe des mécanismes de plus haut niveau pour gérer l'exclusion mutuelle) mais ils illustrent bien les mécanismes généraux de la concurrence.

Le problème. N ($N \geq 2$) processus exécutent continuellement (on utilise une boucle infinie) deux séquences d'instructions :

- la section non-critique (SNC)
- la section critique (SC),

On complète ces deux parties par un pré-protocole (à exécuter avant d'accéder à la SC) et un post-procole (à exécuter après l'accès à la SC).

Les propriétés de base s'énoncent (de manière informelle) comme suit :

- exclusion mutuelle : au plus un seul processus peut être dans sa section critique à un instant donné .
- absence d'inter-blocage : si plusieurs processus essaient en même temps d'accéder à leur section critique, alors un d'entre eux doit y parvenir : ils ne peuvent pas se bloquer mutuellement dans la phase de pré-protocole (voir ci-dessous).
- absence de famine : si un processus essaie d'entrer dans sa SC, alors il y parviendra .
- attente bornée : si un processus P attend pour accéder à sa SC, alors il y arrivera et on peut borner le nombre de fois où d'autres processus pourront accéder à leur SC avant P .
- absence de privilège : il n'y a pas de rôle privilégié pour un processus, chacun est traité à égalité. . .

On peut définir d'autres propriétés sur ces algorithmes. Toutes ces propriétés ne sont pas de même importance : la première { l'exclusion mutuelle { est la propriété fondamentale de ces algorithmes, et c'est une propriété de sûreté (on veut vérifier qu'une \mauvaise chose" n'arrive pas) . L'absence de famine est également une propriété clé et c'est une propriété de vie (\quelque chose de bien doit arriver") .

Notons aussi qu'il y a un lien entre l'absence d'interblocage, l'absence de famine et l'attente bornee :
att. bornee) abs. de famine) abs. d'interblocage

Nous allons voir plusieurs algorithmes et pour chacun d'entre eux, on verifiera quelles proprietes sont satisfaites.

Pour traiter ce probleme de section critique, nous allons faire des hypotheses importantes que nous utiliserons par la suite pour prouver la correction de nos algorithmes.

Les hypotheses. Nous faisons les hypotheses suivantes :

1. Nous nous interessons aux processus structures de la maniere suivante :
   Boucle infinie de :
   { section non-critique
   { pre-protocole (le processus veut acceder a la SC)
   { section critique (le processus y est arrive)
   { post-protocole (le processus a quitte sa SC)
2. On dispose d'un mecanisme de communication par variables partagees qui vont être utilisees dans le pre-protocole et le post-protocole. Ces variables ne sont pas manipulees en dehors (c'est-a-dire dans la SC et la SNC).
3. La section critique ne bloque jamais (on finit toujours par en sortir et par executer le post-protocole).
   NB : cette hypothese est assez naturelle et facile a garantir si on suppose que la section critique est une petite sequence d'instructions elementaires (contrairement a la SNC).
4. La section non-critique peut eventuellement bloquer (i.e. le processus peut s'arrêter ou boucler) et dans ce cas le pre-protocole ne sera plus execute.

5 Algorithmes avec variables partagees et pour deux processus

5.1 Preuve par analyse du graphe d'etats

On commence l'algorithme A_1 de la figure 5.

```
                int turn := 1        // variable partagee

-- Processus P1                      -- Processus P2
loop forever :                       loop forever :
p1: section NC                       q1: section NC
p2: await turn==1                    q2: await turn==2
p3: section critique                 q3: section critique
p4: turn:=2                          q4: turn:=1
```

Figure 5 { Algorithme A1

L'idee de l'algorithme est tres simple : a chaque instant la variable turn designe le numero du processus dont c'est le tour d'acceder a la section critique. Un processus souhaitant acceder a la SC doit attendre son tour.

L'instruction p2 de P_1 est son pre-protocole et p4 est son post-protocole (pour P_2, il s'agit de q2 et q4).

Comment prouver que cet algorithme verifie la propriete d'exclusion mutuelle ? On peut utiliser deux techniques : on peut montrer cette propriete sur le graphe d'etats de l'algorithme (une inspection de ce graphe montrera que la propriete est assuree) et on peut aussi utiliser une preuve classique sur l'algorithme. Pour cet algorithme, nous allons utiliser la premiere methode. Pour l'algorithme suivant (algorithme de Dekker), nous utiliserons la seconde.

Construction du diagramme d'etats. Un etat de l'algorithme A1 contient :

- un pointeur sur la prochaine instruction de P_1,
- un pointeur sur la prochaine instruction de P_2,
- la valeur de turn ,
- et la valeur des differentes variables internes de P_1 et P_2 utilisees dans les sections critiques et non-critiques.

Pour representer le comportement lie au protocole, on peut oublier les variables autres que turn (car elles n'ont pas d'effet sur les proprietes etudiees). On va donc representer un etat de l'algorithme par un triplet (p, q, t) ou p est un element de $\{p1, p2, p3, p4\}$, q un element de $\{q1, q2, q3, q4\}$ et t un element de $\{1, 2\}$.

Le graphe d'etats va contenir tous les etats de cette forme et des transitions representant l'execution de l'instruction p ou q et conduisant a des etats ou les pointeurs ont ete mis a jour ainsi que la variable turn . Pour verifier la propriete d'exclusion mutuelle, il suffit de verifier qu'aucun etat $(p3, q3, 1)$ ou $(p3, q3, 2)$ n'est atteignable depuis l'etat de depart $(p1, q1, 1)$.

Combien peut-il y avoir d'etats dans le graphe ? $4 \times 4 \times 2 = 32$. On va le construire de maniere incrementale en partant de l'etat de depart. Cela donne le graphe de la figure 6.

Dans ce diagramme d'etats, nous avons precise pour chaque transition, si elle correspondait au processus P_1 ou au processus P_2 en etiquetant la transition avec 1 ou 2.

On peut remarquer que chaque etat a toujours deux ou trois transitions sortantes : au moins une correspond a l'execution pointee dans P_1, et au moins une correspond a celle de P_2. Comme nous ne savons pas si l'execution d'une section non-critique termine ou boucle, c'est-a-dire si son execution conduit a l'instruction suivante (p2 ou q2), nous sommes obliges de considerer les deux possibilites : une boucle sur le meme etat (le pointeur restant en p1 ou q1) ou une transition vers p2 ou q2... Les transitions de boucle en SNC sont indiquees avec des pointilles.

Notons aussi qu'il n'y a que 16 etats car les 16 autres ne sont pas accessibles depuis l'etat de depart.

Equite entre processus. Une execution de l'algorithme A1 correspond a un chemin (infini) dans le graphe d'etats. Mais l'inverse n'est pas vrai : certains chemins infinis du graphe ne sont pas de bonnes executions de l'algorithme. Par exemple, les chemins qui bouclent infiniment dans un unique etat avec une transition pleine correspondent a une execution infinie soit du pre-protocole de P_1 (pour les etats de la forme $(p2, \_, 2)$), soit du pre-protocole de P_2 (pour les etats de la forme $(\_, q2, 1)$). Or dans tous ces cas, cela signifierait que l'un des deux processus est systematiquement empeche de progresser : ce serait toujours l'autre qui aurait la main. Ce genre de situation n'est pas acceptable : on souhaite autoriser tous les entrelacements possibles d'actions des deux processus mais pas l'exclusion d'un d'un des deux processus... Les executions de A1 seront donc representees par les chemins

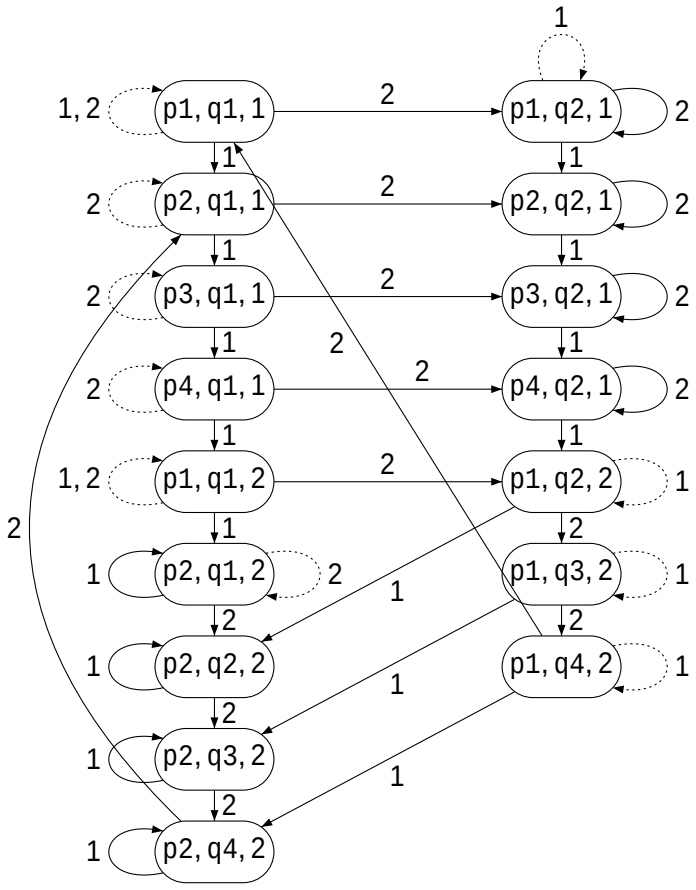

Figure 6 { Diagramme d'etats de A1.

ou un processus qui peut avancer, avancera un jour. Dans le diagramme d'etats de la gure 6, les deux processus peuvent toujours executer une action, et donc les chemins equitables sont les chemins in nis ou il y a une in nite de transitions etiquetees par 1 et une in nite de transitions etiquetees par 2.

Etude des proprietes de l'algorithme. L'analyse du graphe d'etats de A1 montre clairement que les etats $(p3; q3; 1)$ et $(p3; q3; 2)$ ne sont pas accessibles. La propriete d'exclusion mutuelle est donc veriee.

Qu'en est-il des autres proprietes ?

- absence d'inter-blocage : les processus ne peuvent pas se bloquer mutuellement dans la phase de pre-protocole.
  Cette propriete est aussi veriee : une analyse du graphe de la gure 6 montre que lorsque les deux processus sont ensemble dans leur phase de Pre-protocole, il y en a toujours un qui peut accedera la SC : P_1 si turn vaut 1, et P_2 si turn vaut 2, sachant que turn vaut toujours 1 ou 2.
- absence de famine : La question est de savoir si a tout moment lorsqu'un processus se trouve dans son pre-protocole (i.e. quand il souhaite accedera sa SC), alors il y arrivera

un jour.

On ne considère ici que des exécutions équitables entre processus. Par exemple, on exclut une exécution qui bouclerait sur l'état $(p2; q2; 1)$: car cette boucle consiste à exécuter infiniment le test `turn==2` du processus P_2. Dans une exécution équitable, après un certain nombre (arbitrairement grand) de tours de boucle, le processus P_1 finira toujours par avancer en passant `await turn==1` et accédera à $(p3; q2; 1)$.

Mais une analyse du graphe de A1 montre qu'il est possible de rester bloqué dans l'état $(p1; q2; 1)$ tout en suivant une exécution équitable (car contenant à la fois des transitions du processus P_1 en pointillées et des transitions du processus P_2...). Ce scénario correspond au cas où le processus P_1 ne sort pas de sa SNC et ne mettra donc plus `turn` à 2, ce qui empêche P_2 d'accéder à sa SC ! On a bien sûr le même phénomène pour P_1 depuis l'état $(p2; q1; 2)$. On voir ici l'importance de l'hypothèse sur la non-terminaison possible de la SNC.

Dans l'algorithme A1, il y a une famine est possible.

- { attente bornée : Cette propriété est clairement fausse puisqu'il peut y avoir famine...
- { absence de privilège : On peut considérer que le processus P_1 est avantagé car il est sûr de pouvoir accéder au moins une fois à sa SC contrairement à P_2.

Dans l'annexe A, on discute de plusieurs simplifications possibles pour le diagramme d'états de A1.

5.2 Algorithme de Dekker

On considère maintenant l'algorithme de Dekker (proposé en 1964) qui est décrit à la figure 7. C'est, semble-t-il, le premier algorithme correct pour résoudre le problème de section critique dans ce cadre.

```
boolean D1 := False        // variables partagées
boolean D2 := False
int turn := 1
```

```
-- Processus P1                    -- Processus P2
loop forever :                     loop forever :
p1: section NC                     q1: section NC
p2: D1 := True                     q2: D2 := True
p3: while (D2 == True) :           q3: while (D1 == True) :
p4:         if (turn == 2)         q4:         if (turn == 1)
p5:             D1 := False        q5:             D2 := False
p6:             await (turn == 1)  q6:             await (turn == 2)
p7:             D1 := True         q7:             D2 := True
p8: section critique               q8: section critique
p9: turn := 2                      q9: turn := 1
p10: D1:=False                     q10: D2:=False
```

Figure 7 { Algorithme de Dekker

Idee de l'algorithme. Lorsque P_1 veut acceder a sa SC, il commence par mettre D1 a VRAI (signifiant ainsi a P_2 son souhait d'y aller). Ensuite si P_1 constate que P_2 souhaite aussi acceder a sa SC (i.e. D2== True), alors il regarde qui est prioritaire (valeur de turn) : si c'est P_2, il se met en attente de son tour (turn == 1) apres avoir signifie a P_2 qu'il n'etait plus candidat a la SC (en mettant D1 a FAUX). Apres avoir accede a sa SC, P_1 mettra turn a 2 (donnant donc la priorite a P_2) et remettant D1 a FAUX.

Voici deux exemples de scenario pour l'algorithme de Dekker. Les etats successifs du systeme sont representes ligne par ligne mais on ne note que les changements (de pointeur d'instruction et/ou de variables) :

	P_1	P_2	D1	D2	turn
1	p1	q1	?	?	1
2	p2				
3	p3		>		
4		q2			
5	p8				
6		q3		>	
7		q4			
8		q5			
9		q6		?	
10	p9				
11	p10				2
12		q7			
13		q3		>	
14	p1		?		
15		q8			
			: : :		

	P_1	P_2	D1	D2	turn
1	p1	q1	?	?	1
2		q2			
3		q3		>	
4	p2				
5	p3		>		
6		q4			
7	p4				
8	p3				
9		q5			
10		q6		?	
11	p8				
			: : :		

Pour cet algorithme, nous allons d'abord prouver un premier lemme qui enonce plusieurs invariants dans lesquels on utilise les etiquettes des instructions (p- ou q-) pour indiquer la position du pointeur de contrôle du processus P_1 ou P_2.

Lemme 1 est un invariant de la boucle :

{ turn == 1 _ turn == 2

{ (p3 _ p4 _ p5 _ p8 _ p9 _ p10) , D1== >

{ (q3 _ q4 _ q5 _ q8 _ q9 _ q10) , D2== >

Preuve : On montre que la propriete est vraie au debut et se maintient tout au long de l'execution de l'algorithme.

Pour la premiere propriete, le resultat est direct : turn est initialise a 1, puis on ne lui affecte que les valeurs 1 ou 2...

Pour les deux autres proprietes, on peut examiner chaque processus separement car la premiere propriete ne porte que sur le processus P_1 et la seconde que sur P_2. En effet, la variable D1 n'est modifiee que par P_1 (et D2 que par P_2).

On construit donc le diagramme d'etats de la figure 8 qui decrit les comportements possibles de P_1 : chaque etat du graphe contient la prochaine instruction de P_1 a executer et la

valeur courante de D1. Notons qu'avec un tel graphe, nous ne pouvons pas savoir le résultat des tests sur turn ou D2 puisque leurs valeurs ne sont pas représentées dans les états. Nous sommes donc obligés de représenter les deux possibilités (test vrai ou test faux) et faire partir deux transitions différentes. Nous avons donc une approximation du comportement de P_1 dans la mesure où certaines exécutions de ce graphe ne seraient pas forcément possibles dans le graphe complet. Mais nous allons voir que malgré ces comportements supplémentaires, nous avons bien toujours la propriété recherchée sur la valeur de D1 et les états de P_1. Cela nous permettra de conclure que la propriété est vraie sur le vrai comportement de P_1 (qui est plus restreint).

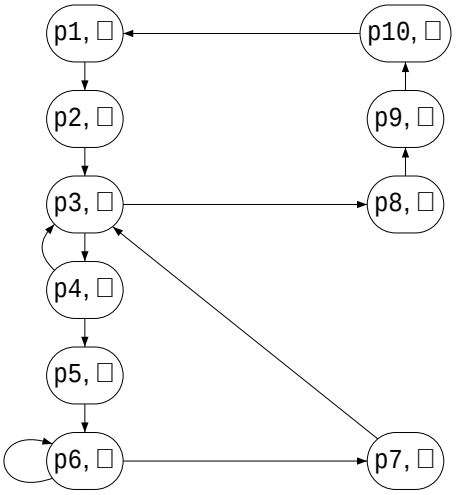

Figure 8 { Diagramme d'états simplifié de P_1 (avec la valeur de D1).

On fait la même chose pour D2 et P_2. 2

Ce lemme peut aussi s'énoncer en utilisant l'opérateur 2 de la logique temporelle (voir l'annexe B) pour énoncer il est toujours vrai... , on dira ainsi les propriétés suivantes sont vraies pour algorithme :

{ 2 turn == 1 _ turn == 2

{ 2 (p3 _ p4 _ p5 _ p8 _ p9 _ p10) , D1== >

{ 2 (q3 _ q4 _ q5 _ q8 _ q9 _ q10) , D2== >

On en déduit le théorème :

Théorème 1 (Exclusion mutuelle de l'algorithme de Dekker)
[illegible] 2 : (p8 ^ q8).

Preuve : Pour que le pointeur de P_1 passe en p8, il faut que le programme soit sorti de la boucle en p3, et donc que la variable D2 soit à False. Ce dernier point implique (par le D213D7(Cpe[illegible]

On fait le même raisonnement pour le cas où le pointeur de P_2 passe en q8 : on en déduit alors que celui de P_1 ne peut pas être en p8.

Il y a bien exclusion mutuelle dans la section critique. 2

Théorème 2 [illegible]

Preuve : Montrer l'absence d'interblocage consiste à montrer que si les deux processus sont dans leur pré-protocole, alors il ne peuvent pas y rester bloqués tous les deux. On fait une preuve par l'absurde. Supposons qu'il existe un interblocage, alors les deux processus P_1 et P_2 sont bloqués dans leur instructions p3,...,p7 et q3...,q7. Supposons que la variable turn vaille 1 (nous savons que cette valeur ne changera plus puisque les deux processus ne pourront plus la modifier avec leurs instructions p9 et q9). Dans ce cas, P_1 boucle sur le while et le if : P_1 exécutera pour toujours les instructions p3,p4,p3,... On a donc D1=True pour tous les états, et la variable D2 doit être vraie infini[illegible] (pour permettre à P_1 de tester correctement le while). Que peut faire P_2 ? Il ne peut pas être bloqué en q6 car sinon cela empêcherait D2 de prendre la valeur True infiniment souvent. P_2 doit donc boucler en q3, mais alors il doit exécuter q4 et comme turn==1 , alors il se retrouve bloquer en q6, ce qui est contradictoire avec la remarque précédente.
Il n'y donc pas d'interblocage : si les deux processus exécutent leur pré-protocole, alors un en sortira... 2

Nous allons maintenant montrer l'absence de famine. Il s'agit donc de montrer que tout processus qui entre dans son pré-protocole finira par accéder à sa section critique. Cette propriété s'énonce $\square(p2 \Rightarrow \diamond p8)$ en logique temporelle. On a le résultat suivant :

Théorème 3 [illegible]

- $\square(p2 \Rightarrow \diamond p8)$, et
- $\square(q2 \Rightarrow \diamond q8)$

Preuve : Supposons qu'il y ait une famine pour P_1. Alors cela signifie que P_1 va rester bloquer dans son pré-protocole. Que peut-il se passer pour P_2 ?

- Il ne peut pas bloquer dans le pré-protocole (car il n'y a pas d'interblocage, voir le théorème 2).
- Supposons qu'il bloque dans sa section non-critique. Soit t_4 un instant où P_2 est bloqué dans sa SNC et P_1 est bloqué dans son pré-protocole. Alors on a D2==False pour tous les instants futurs (grâce au Lemme 1) . Donc si P_1 boucle, ce n'est pas dû au while , mais au await (turn==1) (instruction p6). Donc turn==2 (pour toujours).
  Soit t_2 le dernier instant où P_1 a exécuté l'instruction p3 ($t_2 < t_4$) : en t_2, on avait D2==True et donc entre t_2 et t_4, le processus P_2 a exécuté D2:=False (q10), soit t_3 cet instant.
  Maintenant, nous savons qu'avant t_3, P_2 a effectué q9 mettant ainsi turn à 1 (soit $t_0 < t_3$ cet instant) : comme turn vaut 2 en t_4, il a fallu qu'après t_0, P_1 exécute p9 pour mettre turn à 2... soit t_1 cet instant : on a $t_0 < t_3$ et aussi $t_1 < t_2$. D'où la figure 9.
  En t_1, le pointeur de programme de P_1 est donc en p9 et celui de P_2 est en q10. On sait de plus que D1==True et D2==True Une telle configuration (p9; q10; 1; True; True) est

Figure 9 { Scenario conduisant au blocage de P_1 dans le pre-protocole avec P_2 en SNC.

inaccessible si l'exclusion mutuelle est assuree (voir le lemme 2 ci dessous qui montre ce point en detail). Ce cas est donc aussi impossible.

Le blocage de P_1 alors que P_2 reste dans sa section non-critique est donc impossible.

{ Supposons en n que P_2 ne bloque pas dans sa SNC : qu'il execute son protocole in ni-ment souvent. Mais alors apres avoir accede a sa SC, il mettra turn a 1, donnant ainsi la priorite a P_1. Cette valeur pour turn ne changera plus. Du coup, P_1 ne peut plus être bloque en p6 et bouclera en p3p4p3p4 . . . et D1 sera a True pour toujours. Mais alors P_2 testera son while et nira bloque en q6 ! Ce qui conduirait a un interblocage, ce qui n'est pas possible. . .

2

On utilise le lemme suivant dans la preuve d'absence de famine :

Lemme 2 [illegible] (p9; q10; turn = 1 ; D1 = > ; D2 = >) ou (p9; q10; turn = 2 ; D1 = > ; D2 = >) [illegible]

Preuve : Pour le montrer il su t de calculer les predecesseurs possibles pour ces con gurations (on omet la valeur de turn car elle n'est pas utile ici). Notons d'abord que les dernieres instructions e ectuees par P_1 sont p3 p8 p9, et celles de P_2 sont q3 q8 q9 q10. La gure 10 contient ce graphe d'etats construits en arriere a partir de (p9; q10; ; D1 = > ; D2 = >).

A chaque etat on associe les deux transitions entrantes correspondant a l'execution d'une instruction de P_1 ou P_2. Une transition en pointillee est une transition impossible a franchir.

Une analyse du graphe montre bien que de tous les chemins menant a une con guration de la forme (p9; q10; ; > ; >) passe par un con it en section critique (p8; q8; : : :), ce qui est exclu par le theoreme 1.

2

Il n'y a donc pas de famine, ni d'interblocage. Le protocole veri e-t-il l'attente bornee ? Oui car un processus P peut se faire doubler au plus une fois : il se fait doubler si la variable turn lui est defavorable mais alors l'autre processus Q modi e turn de maniere a avantager P. . .

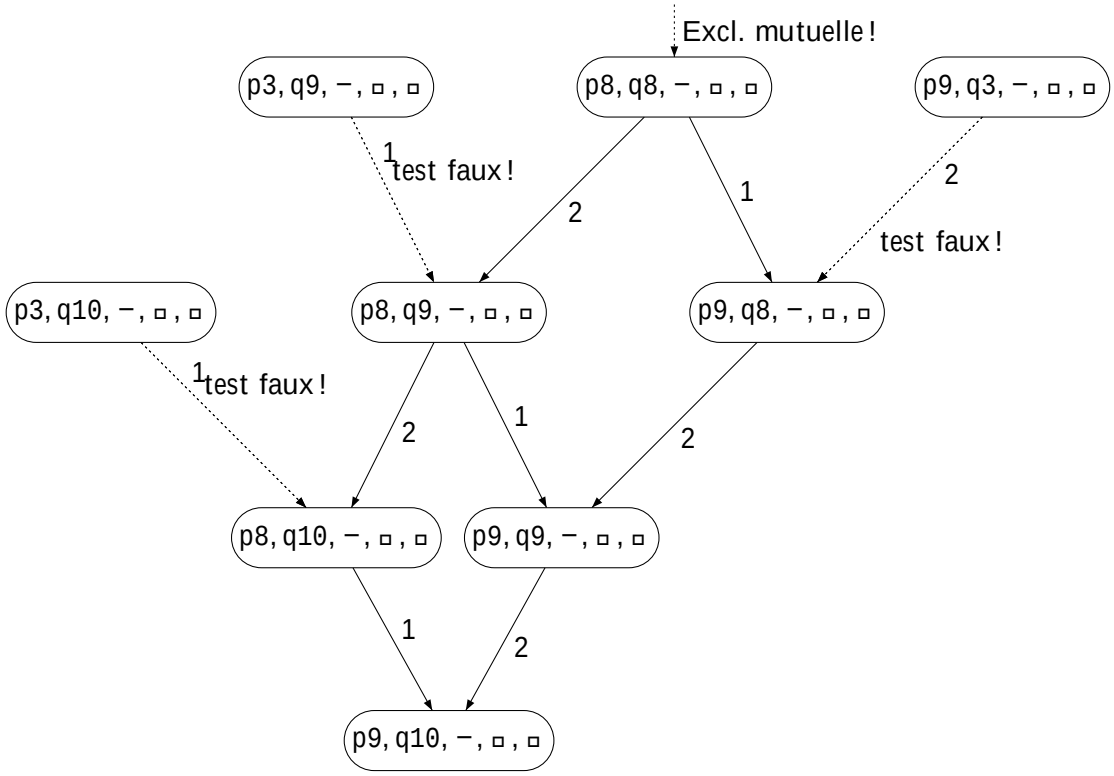

Figure 10 { Graphe arrière depuis (p9; q10; ; >; >) { algorithme de Dekker.

5.3 Algorithme de Peterson

L'algorithme de Peterson est plus simple et plus recent (1981) que l'algorithme de Dekker. Il est décrit dans la figure 11.

```
boolean D1 := False        // variables partagees
boolean D2 := False
int turn := 1
```

```
-- Processus P1                          -- Processus P2
loop forever :                           loop forever :
p1: section NC                           q1: section NC
p2: D1 := True                           q2: D2 := True
p3: turn := 2                            q3: turn := 1
p4: await (D2==False  OR turn==1)        q4: await (D1==False  OR turn==2)
p5: section critique                     q5: section critique
p6: D1:=False                            q6: D2:=False
```

Figure 11 { Algorithme de Peterson

Idee de l'algorithme. Lorsque le processus P_1 veut accéder à sa SC, il le signale en mettant D1 à VRAI puis donne la priorité à l'autre processus P_2, et ensuite vérifie avant d'accéder à sa SC, que P_2 ne veut pas y aller (D2== False) ou alors qu'il (P_1) est prioritaire.

On a :

Théorème 4 (Exclusion mutuelle de l'algorithme de Peterson)
[illegible] 2 : (p5 ∧ q5).

Preuve : Supposons que P_1 soit déjà dans sa SC lorsque P_2 accède à sa SC. On a alors D1==D2==True (NB : la valeur de D1 n'est modifiable que par P_1 et celle de D2 n'est modifiable que par P_2 : on peut donc connaître directement leur valeur en fonction du pointeur d'instruction des deux processus). Donc, si P_2 accède à sa SC, c'est que turn vaut 2. Quel test a pu permettre à P_1 d'accéder dans sa SC ?

- { D2 == False ? Mais alors P_2 a (depuis l'arrivée de P_1 en SC) fait D2:=True puis turn:=1 et personne n'a pu changer cette valeur et donc lui permettre d'accéder à la SC...
- { turn == 1 ? Là encore, P_2 a modifié turn après la modification (turn:=2) de P_1 et donc turn vaut toujours 1 lorsque P_2 veut accéder à sa SC et donc il ne peut y arriver.

Donc P_2 ne peut pas rejoindre P_1... 2

Il n'y a pas d'interblocage des processus lors du pré-protocole. En effet, si il y avait un tel blocage, cela signifierait que P_1 serait en attente de D2==False OR turn==1 et P_2 serait en attente de D1==False OR turn==2 , on en conclut que dans cette situation on aurait D1==True , D2==True et surtout turn!=1 et turn!=2 , ce qui n'est clairement pas possible !

On en déduit :

Théorème 5 [illegible]

- { 2 (p2 3 p5), et
- { 2 (q2 3 q5)

Preuve : Supposons que P_1 soit en situation de famine. Il a donc exécuté p2, puis p3 (l'équité en processus garantit sa progression) et arrivera en p4 où il restera bloqué sur le await . Si il est bloqué à ce stade, c'est que l'on a (D2==True ∧ turn==2) lors des tests du await . Cela signifie que la valeur de D1 restera à True pour toujours (P_1 est bloqué), tandis que turn devra valoir 2 et D2 True infiniment souvent (à chaque fois que P_1 testera la condition de son await). Mais si la variable D2 est à True, c'est que le processus P_2 se trouve en q3, q4, q5 ou q6 :

- { de q3, il irait en q4 (équité entre processus) et mettrait turn à 1 avant de bloquer en q4 et laisser passer P_1, ce qui est contradictoire avec l'hypothèse de départ... Ce n'est donc pas possible.
- { en q4, il franchirait le await si turn vaut 2, et passerait en q5 ;
- { de q5, il irait en q6 (car la SC termine) ; et
- { de q6, il finirait par mettre D2 à False et arriverait en q1. De là, P_2 ne peut pas bloquer en SNC car alors D2 resterait à False pour toujours et cela contredirait l'hypothèse du

blocage de P_1 en p4... Donc P_2 executera q2 puis q3 et on retombe sur le premier cas qui contredit l'hypothese.

On a donc vu que dans tous les cas, P_2 ne peut pas assurer le fait d'avoir in niment souvent (D2==True^ turn==2). Il n'y a donc pas de famine pour P_1. 2

Le protocole veri e-t-il l'attente born ee ? Oui : un processus P peut se faire doubler au plus une fois (il se fait doubler si il execute son pr e-protocole juste apr es le processus Q et modi e la variable turn pour lui donner l'avantage, mais alors l'autre processus Q, apr es avoir atteint sa SC, modi era turn de mani ere a avantager P...).

Dans l'algorithme de Dekker, il n'y a pas vraiment de r ôle privil egi e pour un processus, m ême si l'initialisation de la variable turn donne un l eger avantage a P_1. Et dans l'algorithme de Peterson, il n'y a strictement aucun privil ege : l'initialisation de la variable n'in ue pas sur le comportement de l'algorithme (pourquoi ?).

6 Algorithmes distribués pour n processus

6.1 De 2 à n : les tournois

Ici nous allons voir une idée proposée par G.L. Peterson et M.J. Fischer en 1977 pour passer d'un algorithme de section critique pour 2 processus à un algorithme pour n processus (voir [4]). L'idée repose sur une approche « diviser-pour-régner ».

Supposons que l'on dispose de $n = 2^k$ processus : $P_0, P_1, \ldots, P_{n-1}$. On va faire un tournoi à $\log(n)$ ($= k$) tours. Après le premier tour, il ne reste que $\frac{n}{2}$ processus (bien sûr chaque compétition entre deux processus est régie par l'algorithme de section critique pour deux processus dont on dispose : ici on va utiliser l'algorithme de Peterson), puis après le second tour il ne reste que $\frac{n}{4}$ processus, etc.

On numérote les tours de 0 à $\log(n) - 1$ selon la figure 12(a).

Figure 12 – Organisation du tournoi

Chaque compétition est repérée par un niveau ℓ et un numéro k : $\ell \in \{0, \ldots, \log(n) - 1\}$ et $k \in \{0, \ldots, 2^{\log(n)-1-\ell} - 1\}$. Pour chaque compétition, on va utiliser l'algorithme de Peterson et on doit donc disposer de l'équivalent des variables partagées turn, D1 et D2, pour cela on utilise :

- turn $[\ell, k]$: qui est prioritaire ?
- D$[\ell, 2k]$: est-ce que le processus de gauche veut accéder à la SC ?
- D$[\ell, 2k+1]$: est-ce que le processus de droite veut accéder à la SC ?

De plus, chaque processus P_i doit pouvoir se repérer et savoir à tout moment à quel niveau il se trouve et si il correspond au « processus gauche » ou au « processus droit » du tournoi en cours afin de pouvoir interpréter correctement la valeur de turn $[\ell, k]$. Pour chaque processus P_i, on a donc aussi les variables locales suivantes :

- ℓ : le niveau de la compétition en cours ;
- k : le noeud correspondant à la compétition en cours au niveau ℓ ; et
- id $\in \{1, 2\}$: indiquant si P_i est le processus de gauche (i.e. joue le rôle de P_1 dans l'algorithme de Peterson) ou si c'est le processus de droite (i.e. joue le rôle de P_2).

Algorithme P_i
loop forever :

```
1  Section NC
2  k := i
3  pour ‘ = 0 : : : log(n)  1 faire
4  |  id := ( k%2) + 1
5  |  k := b k/2 c
6  |  D[‘; 2k + ( id  1)] := true
7  |  turn [‘; k ] := 3  id
8  |  await (: D[‘; 2k + 2  id ] _ turn [‘; k ] == id )
9  Section critique
10 pour ‘ = log( n)  1 : : : 0 faire
11 |  D[‘; b i/2 c] := false
```

Preuve : Si un processus P commence a executer son pre-protocole et se retrouve bloque a un noeud (i,k), alors il deviendra prioritaire des que le processus responsable du blocage a ce noeud aura execute son post-protocole. L'equite entre processus lui assurera alors de franchir le noeud (i,k). 2

Mais l'attente n'est pas bornee : pendant qu'un processus est bloque en (i,k), d'autres processus peuvent acceder a leur SC un nombre arbitrairement grand de fois. Sur l'exemple precedent, on peut imaginer que P_0 est bloque en $(0,1)$, et pendant ce temps les processus P_2, P_3,..., P_7 peuvent acceder a leur SC un nombre de fois arbitrairement grand (le noeud $(0,1)$ n'est pas utilise pendant toute cette periode). Bien sûr cela suppose que P_0 est tres tres lent (arbitrairement lent !).

6.2 Avec des variables propres : l'algorithme de la boulangerie

On s'interesse a des problemes faisant intervenir n 2 processus.

A present nous n'utiliserons plus des variables partagees (comme la variable turn des algorithmes de Dekker ou Peterson, que chaque processus peut lire et modi er. Ici les dierents processus ne pourront communiquer que par des variables ou Multiple-Readers/Single-Writer : une telle variable peut être lue par tous les processus mais elle ne peut être modiee que par un seul processus.

Notons que ce mecanisme permet a chaque processus P d'obtenir des informations sur l'etat d'un autre processus Q (par la lecture des variables propres de Q) mais cela ne lui permet pas d'in uer directement sur ces etats (impossible de modi er ces variables). On est donc dans une con guration dierente des premiers algorithmes presentes ou les processus partageait reellement et completement des variables.

Nous allons etudier l'algorithme de la Boulangerie (Bakery algorithm) propose par Lamport en 1974. L'idee de cet algorithme est assez classique et on le retrouve dans les gares, les bureaux de poste, l'administration : Il su t de choisir un numero superieur a celui de toutes les personnes deja en attente, puis d'attendre su samment longtemps pour que ce numero devienne le plus petit des numeros des personnes en attente...

La gure 13 presente l'algorithme de la Boulangerie. Les tableaux Choosing et Nb sont des variables propres : chaque element est lisible par tous les processus mais seul le processus peut modi er les i-eme elements de ces tableaux (c.-a-d. Choosing[i] et Nb[i] appartiennent a P_i).

Dans cet algorithme on doit calculer le Max de tous les numeros utilises par les processus. Cette operation n'est pas consideree comme une operation atomique on peut donc avoir plusieurs max qui se calculent en même temps et obtenir le même numero pour chacun de ces calculs. Comme l'algorithme repose sur l'idee que le numero minimal est prioritaire, on resout les egalites en considerant les numeros de processus (on suppose que ces numeros sont uniques) dans la relation d'ordre, on de nit ainsi la relation $\ll$ sur les paires d'entiers :

$$(n,i) \ll (n',i') \quad , \quad (n < n') \_ (n == n' \wedge i < i')$$

Notons que si $i \neq i'$ on a toujours $(n,i) \ll (n',i')$ ou $(n',i') \ll (n,i)$.

On notera $p@X^{(i)}$ le fait que le processus P_i se trouve a la X-eme instruction.

L'algorithme de la Boulangerie garantit l'exclusion mutuelle :

```
Algorithme de la Boulangerie
boolean Choosing[1..n] := False          // variables propres
int Nb[1..n] := 0

-- Processus Pi
loop forever :
p1: section NC
p2: Choosing[i] := True
p3: Nb[i] := Max(Nb[1..n])+1
p4: Choosing[i] := False
p5: For j =1..n : if j != i :
p6:     await Choosing[j]==False
p7:     await Nb[j]==0   or   (Nb[i],i) << (Nb[j],j)
p8: section critique
p9: Nb[i] := 0
```

Figure 13 { Algorithme de la Boulangerie

Theoreme 8 [illegible]

$$\forall i \neq j : (p8^{(i)} \wedge p8^{(j)})$$

Preuve : On va montrer que lorsque P_i arrive en section critique, aucun autre processus ne peut deja s'y trouver.

Supposons que P_i atteigne sa SC au temps t_2. Soit P_k un autre processus ($k \neq i$). On veut montrer que P_k n'est pas en SC au temps t_2.

Puisque P_i est en SC au temps t_2, c'est qu'il a franchi, lors de l'iteration $j = k$ de la boucle for , son instruction p6 (soit t_0 cet instant ou Choosing[k]==False) et son instruction p7 (soit t_1 cet instant ou l'on a Nb[k]==0 or ((Nb[i],i) << (Nb[k],k))). On a bien sûr $t_0 < t_1 < t_2$.

Ou peut être P_k a l'instant t_0 ? Il y a deux cas :

- { en p1 ou en p2 : lorsque P_k essaiera d'entrer en SC, son numero sera forcement superieur strictement a Nb[i] car la valeur de Nb[i] sera prise en compte dans le calcul du max. Et donc P_k ne pourra ni doubler, ni rejoindre rejoindre P_i dans la SC.
- { en p5, p6, p7, p8 ou p9 : Dans ce cas, on distingue deux autres cas selon la valeur du test (Nb[k],k]) << (Nb[i],i) a l'instant t_0 :
  - { (Nb[k],k]) << (Nb[i],i) : alors Nb[k] sera obligatoirement remis a zero avant l'instant t_1 (ou P_i franchit p7 en examinant P_k) et on se ramene au cas precedent avec P_k en p1 ou p2, ou au cas suivant avec (Nb[i],i] << (Nb[k],k) ...
  - { (Nb[i],i]) << (Nb[k],k) : alors dans ce cas, P_k ne peut pas franchir son instruction p7 lors de son examen de P_i avant que P_i ne remette Nb[i] a zero, c'est a dire en sortant de la SC. Donc P_k ne peut acceder a sa SC entre t_0 et t_2.
    Il reste donc a verifier que P_k n'a pas pu acceder a sa SC avant t_0 (et y être toujours). Si cela etait le cas, alors P_k serait arrive dans la SC apres son choix de Nb[k] . Or au moment du choix de Nb[k] , le calcul de Nb[i] a deja commence (ou même termine)

car Nb[k] est superieur ou egal a Nb[i] et donc lorsque P_k examinera P_i lors de son instruction p7, il ne pourra pas la franchir avant P_i et donc il ne peut pas acceder a sa SC avant t_0.

Il n'y a donc pas de possibilite ou P_i rejoigne un autre processus en SC. 2

Il n'y a pas d'interblocage car le processus bloque avec le plus petit numero (et, en cas d'egalite, ayant le plus petit indice) passera en SC. L'absence de famine utilise le même argument :

Theoreme 9

$$\forall i \quad 2^{(i)} \Rightarrow (p2^{(i)} \quad 3\ p8^{(i)}).$$

Preuve : Considerons le cas du processus P_i. Lorsque P_i commence a executer son pre-protocole, il execute p2, puis p3 (ce calcul termine...), puis p4 et commence les iterations de p6 et p7. Il ne peut bloquer qu'en p7 (car tous les calculs de Max terminent). Supposons que P_i bloque en p7 a l'iteration j, donc sur l'instruction :

await(Nb[j]==0 or (Nb[i],i) << (Nb[j],j))

Comme il n'y a pas d'interblocage, les processus ayant des numeros plus prioritaires que P_i finiront par acceder puis quitter leur SC, et P_j accedera un jour a sa SC (avant P_i), puis :

- soit P_j restera en section non-critique et alors Nb[j]==0, ce qui liberera P_i de son blocage...
- soit P_j reexecutera son pre-protocole mais alors le numero Nb[j] qu'il choisira sera strictement superieur a Nb[i] et donc a aussi, P_i sera debloque...

2

L'algorithme de la Boulangerie assure aussi une attente bornee : le nombre de processus qui peuvent passer devant P_i apres que P_i ait demande a acceder a sa SC et calcule son Nb, est borne par n 1 : apres un eventuel acces a la SC, tout processus P_j se verra attribuer (si il demande a y retourner) un numero superieur a celui de P_i...

Un inconvenient de l'algorithme de la Boulangerie est souligne dans l'exercice ci-dessous : les numeros renvoyes par le Max peuvent croître indefiniment. Il existe des ameliorations de l'algorithme de la Boulangerie qui evitent ce probleme.

Exercice 1

7 Algorithmes distribués utilisant des messages

Maintenant nous allons considérer le cas des algorithmes distribués communiquant par envois/réceptions de messages. Nous considérons des communications asynchrones : un envoi de message est suivi (plus tard) par une réception.

Les différents processus sont distribués sur différentes machines (on parle de sites), chacun a sa mémoire locale qu'il ne partage pas avec les autres. Chaque processus peut envoyer des messages à n'importe quel autre processus : un message est typé (sa structure est fixée). L'envoi de messages de la part d'un processus P permet de signaler aux autres processus des changements dans l'état de P.

Sur chaque site, on va distinguer deux parties du processus : une sera chargée de représenter l'activité du processus (pour accéder à la SC) et l'autre sera chargée de la réception des messages des autres processus. Ces deux parties évoluent en parallèle (en partageant la mémoire du site).

Nous allons faire les hypothèses suivantes :

- un site ne peut pas s'arrêter complètement : il doit toujours être capable d'envoyer et recevoir des messages ;
- chaque site peut envoyer (et recevoir) des messages à (de) n'importe quel autre site ;
- les canaux sont fiables : aucun message n'est perdu ;
- le temps de communication (le délai séparant l'envoi de la réception d'un message) est arbitraire (mais fini !).

7.1 Algorithme de Lamport (1978)

Hypothèses. Dans cet algorithme (voir par exemple [2]), on ne connaît pas le temps nécessaire à la transmission d'un message (entre son envoi et sa réception), mais on suppose toujours qu'entre deux processus donnés, deux messages ne peuvent pas se doubler : si P_i envoie un message M_1 à P_j, puis un autre message M_2, alors on sait que P_j recevra d'abord M_1 puis M_2.

Idée de l'algorithme. Chaque processus P_i dispose d'une horloge locale h_i (ce mécanisme est expliqué ci-dessous) que l'on va utiliser pour dater les messages. Un processus qui veut accéder à sa section critique va signaler son intention à tous les autres processus en envoyant un message *request* accompagné de la valeur de son horloge locale. Chaque processus souhaitant accéder à la SC dispose donc de toutes les demandes en cours des autres processus et sait si sa propre demande est la plus ancienne ou non : pour accéder à la SC, il suffit à P_i d'attendre que sa demande soit la plus ancienne de toutes les demandes en cours. Et bien-sûr tout processus quittant la SC doit le signaler avec un message *release* adressé à tous les processus.
Comme dans cet algorithme, on ne connaît pas le temps nécessaire à la transmission d'un message, il faut vérifier, avant d'accéder à la SC, qu'aucun message "ancien" n'est en cours de transmission, pour cela on impose au processus accédant à la SC d'avoir reçu un message "récent" de la part de tous les autres processus. Afin de garantir l'émission de messages récents (autres que les demandes d'accès ou les libérations de SC), on demande à chaque processus d'envoyer un accusé de réception *ack* à chaque fois qu'il reçoit une demande *request* d'un autre processus.

Evolution des horloges locales. Un probleme courant dans les algorithmes distribues est d'ordonner les dierents evenements (envoi ou reception de messages) : chaque processus a une vision locale qu'il se construit a partir de son comportement et des messages recus. Chaque processus de l'algorithme va disposer d'une horloge locale qui est incrementee de 1 a chaque envoi de messages. Mais ce mecanisme peut engendrer un decalage tres grand entre les valeurs des horloges des dierents processus. Pour limiter ce decalage, on impose a chaque processus P_i qui recoit un message de P_j datee par la valeur t (t est la date locale a P_j au moment de l'envoi du message) de comparer la valeur de son horloge c_i a t : si $c_i < t$ alors P_i met a jour son horloge avec la valeur t. De plus, P_i incrementera son horloge apres chaque envoi (ou di usion groupee) et reception de messages.

Description de l'algorithme. Tout d'abord, on precise les objets manipules par les processus. Les horloges sont des entiers positifs. Les messages seront des triplets (type, date d'horloge, num. de proc.) ou le type peut etre request , release ou ack. Chaque processus dispose d'un tableau F[1..n] pour stocker les derniers messages importants (voir ci-dessous) arrives pour chaque processus. Dans son $F[i]$, le processus P_i stockera son dernier message de type request ou release .

Lorsque P_i recoit un message $(request; t; j)$ ou $(release; t; j)$, il le stocke en $F[j]$. Lorsqu'il recoit un message $(ack; t; j)$, il le stocke en $F[j]$ seulement si $F[j]$ ne contient pas de message de type request .

Notons que les dates des messages contenus dans $F[j]$ sont croissantes (pour tout j) : un message avec une date t est remplace par un message avec une date t' telle que $t < t'$.

La description de l'algorithme se trouve dans les gures 14 et 15. On distingue la partie gestion des receptions de messages (gure 15) du reste de l'algorithme (gure 14).

On note Send(request,c,i) -> j l'envoi du message (request,c,i) au processus j. La fonction type(m,t,k) retourne le type m du message et la fonction date(m,t,k) retourne la date t associee a un message.

Algorithme de Lamport (1978)

```
-- Processus Pi
Message F[1..n] := nil
int c := 0
loop forever :
p1: section NC
p2: { for all j != i : Send(request,c,i) -> j
p3:     F[i] := (request,c,i)
p4:     c := c+1 }
p5: await [ (date(F[i]) << date(F[j]) for all j != i ]
p6: section critique
p7:   { for all j!=i : Send(release,c,i) -> j
p8:       F[i] := (release,c,i)
p6:       c := c+1   }
```

Figure 14 { Algorithme de Lamport 1978

Algorithme de Lamport (1978) -- gestion des RECEPTIONS du processus Pi

```
case (request,t,j)   :
 { if (t > c) : c:=t
   c := c+1
   F[j] := (request,t,j)
   Send(ack,c,i) -> j   }

case (release,t,j)   :
 { if (t > c) : c:=t
   c := c+1
   F[j] := (rel,t,j)   }

case (ack,t,j)   :
 { if (t > c) : c:=t
   c := c+1
   if (type(F[j]) != request) : F[j] := (ack,t,j)   }
```

Figure 15 { Algorithme de Lamport 1978 { gestion des receptions

Remarque : Dans le code des gures 14 et 15, on utilise des f : : :g pour designer des instructions que l'on fait de maniere atomique localement : le code de la partie principale et celui de la gestion des receptions d'un meme processus ne peuvent pas alors s'entremeler (pour garantir cette exclusion mutuelle on peut utiliser les algorithmes vus precedemment base sur le partage de la memoire comme Peterson). On verra pourquoi cette hypothese est necessaire.

L'algorithme de Lamport-1978 assure l'exclusion mutuelle :

Theoreme 10 [illegible]

$$\forall i \neq j \quad : (p6^{(i)} \wedge p6^{(j)}) \text{ est faux}$$

Preuve : Supposons que les processus P_i et P_j soient au meme moment en p6. Supposons qu'a cet instant, on ait $date(F^i[i]) \ll date(F^j[j])$ ou F^k designe le tableau du processus k. Alors c'est le processus P_j qui n'a pas respecte le protocole. En e et, P_j a envoye une requete au temps t, et P_i a lui envoye une requete au temps t^0 avec $t^0 < t$ ou $(t = t^0 \wedge i < j)$ (i.e. $(t^0, i) \ll (t; j)$). Si P_j accede a sa SC, c'est qu'il a stocke un message recent de tous les processus, et donc en particulier de P_i, dans son $F^j[i]$, ce message est de la forme :

- { (request, t^{00},i) ou (release, t^{00},i) avec $(t; j) \ll (t^{00}, i)$: alors on a $t^0 < t^{00}$, le processus P_i a donc envoye le message (request, t^{00},i) ou (release, t^{00},i) apres le message (request, t^0,i) : d'apres le code du processus P_i, il n'est pas possible que (request, t^0,i) soit encore stocke dans $date(F^i[i])$ ...
- { (ack,t'',i) avec $(t; j) \ll (t^{00}, i)$: alors on sait que le message (request, t^0,i) a ete recu par P_j avant (ack,t'',i) (car par hypothese les messages ne se doublent pas). Or

si ce dernier message a été stocké dans date($F^j[i]$) c'est qu'il n'y avait pas de message de type request (voir la gestion des réceptions) dans F^j mais un message release signifiant que P_i avait quitté sa SC.

2

Exercice 2

Pour quelles propriétés, on utilise le mécanisme de synchronisation des horloges (mise à jour avec les valeurs d'horloges reçues dans les messages) ?

7.2 Algorithme de Ricart-Agrawala (1981)

L'idée de cet algorithme est assez proche de l'algorithme précédent mais on va essayer de réduire le nombre de messages nécessaires à l'accès de la section critique : lorsqu'un processus P_i communique à P_j son souhait d'accéder à la SC, P_j ne lui retourne un accusé de réception ok que si P_j ne souhaite pas y accéder ou si P_j n'est pas prioritaire (voir ci dessous). Dans le cas où P_j ne répond pas tout de suite, il le fera après avoir accédé à la SC (il utilise un tableau Waiting pour stocker la liste des processus à qui il doit envoyer un ok plus tard).

La priorité d'un processus est établie par un numéro que chaque processus choisit lorsqu'il souhaite accéder à sa SC. Plus ce numéro est petit, plus sa demande d'accès à la SC est prioritaire. Ce numéro est choisi supérieur à tous les autres numéros que le processus a vu pour le moment (on utilise pour cela une variable maxnb).

Un processus n'accède alors à sa SC que lorsqu'il a reçu un accord explicite de tous les autres processus ($N - 1$ messages).

Cet algorithme utilise deux types de message : request (comme précédemment) et ok qui servira à donner son accord à un processus pour qu'il accède à la SC. Notons enfin qu'ici nous ne faisons pas l'hypothèse que les messages arrivent dans l'ordre : si j'envoi un premier message à P_j, puis un second, alors l'ordre d'arrivée n'est pas fixé.

> Remarque : Dans le code des figures 16 et 17, on utilise des f : : :g pour désigner des instructions que l'on fait de manière atomique : le code de la partie principale et celui de la gestion des réceptions ne peuvent pas alors s'entremêler (pour garantir cette exclusion mutuelle on peut utiliser les algorithmes vus précédemment basé sur le partage de la mémoire comme Peterson). On verra pourquoi cette hypothèse est nécessaire.

Théorème 11

$$\exists i \neq j : (p7^{(i)} \wedge p7^{(j)})$$

Preuve : Supposons que P_i et P_j se trouvent ensemble dans leur section critique. Il se sont donc chacun envoyés un message ok. Ils ont aussi chacun choisit une priorité : pr_i pour P_i et pr_j pour P_j.

Pour P_i on distingue les dates suivantes :

- t_i^0 où P_i choisit son numéro pr ;

Algorithme de Ricart-Agrawala

```
-- Processus Pi
boolean Waiting[1..n] := False
boolean reqCS :=False
int pr := 0
int maxpr := 0          // plus grand numero recu
int nba:=0              // compte le nb de reponses attendues

loop forever :
p1: section NC
p2: { reqCS := True
P3:    nba := N-1
p4:    pr := maxpr +1 }
P5: for all j != i : Send(request,pr,i) -> j
p6: await [ nba == 0 ]
p7: section critique
p8: { reqCS := False
p9:    for all j: if (Waiting[j] == True) :
p10:              Waiting[j] := False
p11:              Send(ok,i) -> j }
```

Figure 16 { Algorithme de Ricart-Agrawala

```
case (request,k,j)  :
  { if (k > maxpr) :  maxpr := k
    if (!reqCS or (k,j)<<(pr,i)) : Send(ok,i) -> j
      else : Waiting[j] := True }

case (ok,j)  :
  nba := nba - 1
```

Figure 17 { Algorithme de Ricart-Agrawala { gestion des receptions

{ t_i^1 ou P_i envoie le message (request,i,pr) a P_j ;

{ t_j^0 ou P_j recoit (request,i,pr) et retourne (ok,j) a P_i (NB : en raison des hypotheses d'atomicite, on sait que cet instant existe : la reception du message de P_i est suivie immediatement { sans l'execution d'instructions de la procedure principale de P_j { par l'envoi de ok a P_i) ;

{ t_i^2 ou P_i recoit le message (ok,j) de P_j .

On de nit de meme t_j^0, t_j^1, t_i^0 et t_j^2 pour le processus P_j . On a clairement : $t_i^0 < t_i^1 < t_j^0 < t_i^2$, et $t_j^0 < t_j^1 < t_i^0 < t_j^2$.

Supposons $t_i^0 < t_j^0$. Alors on distingue deux cas :

- $t_j^0 < t_j^0$: alors le choix de pr_j sera fait apres la mise a jour de maxpr dans P_j et donc on aura $(pr_j ; j) >> (pr_i ; i)$. En consequence, il n'est pas possible que P_i envoie un message ok a P_j avant de quitter sa SC... Il ne peut y avoir de con it en section critique de cette maniere.
- $t_j^0 < t_j^0$: alors on a soit $(pr_j ; j) >> (pr_i ; i)$ ou $(pr_i ;) >> (pr_j ; j)$. Dans le premier cas, P_j ne pourra recevoir de message ok, et dans le second c'est P_i qui n'en recevra pas de la part de P_j. Dans les deux cas, un des deux processus devra attendre que l'autre sorte de la section critique.

Le cas $t_j^0 < t_i^0$ est similaire.

Il n'y a donc pas de con it en section critique. 2

L'absence de famine est aussi garantie :

Theoreme 12 [illegible]

- [illegible]
- [illegible]
- [illegible]

[illegible] $\hat{}_i \; 2 \; (p2^{(i)}) \; 3 \; p7^{(i)})$.

Preuve : Supposons que le processus P_i soit en situation de famine. Soit pr_i sa priorite. Si P_i reste bloque, c'est forcement en p6 : il attend d'avoir une reponse de chaque processus[1].

Il existe donc un processus P_j qui ne donne jamais son accord a P_i. Si tel est le cas, c'est que P_j veut aussi acceder a sa SC (ceci est necessaire pour di erer l'envoi de ok). Et si P_j bloque pour toujours P_i, c'est que lui aussi est bloque : car si P_j accedait a sa SC, il la terminerait et nirait par envoyer ok a P_i...

Notons au passage, que les autres processus niront par être aussi bloques lors de leur prochain pre-protocole : ils ne recevront plus de message ok de P_i et P_j car ils auront des numeros superieurs a pr_i et pr_j... Tout le monde nira donc bloque (ou en section non-critique).

Soit P_m le processus bloque dans son pre-protocole ayant la plus petite priorite pr_m (ie minimale pour $<<$) . Soit P_k un autre processus bloque qui n'a pas envoye son message ok a P_m. Pourquoi P_k a-t-il bloque P_m ? C'est qu'au moment de la reception de (request, pr_m,m), P_k voulait acceder a la SC (reqCS==True) et que son pr etait inferieur (pour $<<$) a pr_m (NB : c'est ici que l'hypothese d'atomicite du bloc p2,p3,p4 est utilisee). Mais depuis P_k a change de pr (puisque pr_m est le minimum { legalite est impossible grâce a la relation $<<$), c'est donc qu'il est passe en SC et donc qu'il a execute la boucle p9 et donc envoye un message ok a P_m. Il y a bien contradiction : P_m ne peut pas être bloque... 2

Dans cet algorithme, une demande d'acces en SC demande $2(n \; 1)$ messages.

Exercice 3 [illegible]

On peut trouver une preuve automatique de cet algorithme dans [3] (accessible sur le web).

---

1 [illegible] ok [illegible] ok [illegible] request [illegible] sinon : [illegible] ok.

A Abstraction et diagramme d'états

La taille du diagramme d'états est un problème majeur de l'analyse des algorithmes concurrents. Il peut être exponentiel dans le nombre de processus évoluant en parallèle et dans le nombre de variables... Il est donc important de réduire le plus possible ce graphe, en ne conservant que les parties pertinentes vis-à-vis de la propriété à vérifier. Ce travail n'est pas toujours facile à faire. Nous allons l'illustrer de deux manières sur l'exemple de l'algorithme A1 de la section 5.1.

Tout d'abord, nous pouvons simplifier l'algorithme en considérant celui de la figure 18.

```
int turn := 1        // variable partagée

-- Processus P1                  -- Processus P2
loop forever :                   loop forever :
p1': await turn==1               q1': await turn==2
p2': turn:=2                     q2': turn:=1
```

Figure 18 – Algorithme A1'

En effet, pour la propriété d'exclusion mutuelle, on ne s'intéresse pas à ce qui se passe dans les sections critiques et les sections non-critiques. Par exemple, si ces deux parties du code étaient vides, alors le protocole devrait toujours fonctionner (on sait que ces parties ne peuvent modifier la variable turn). On peut donc se limiter aux deux phases du protocole. Le diagramme d'états de A1' est alors décrit à la figure 19.

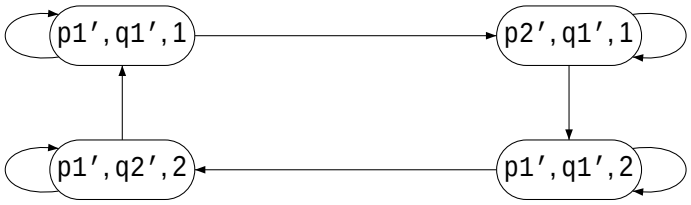

Figure 19 – Diagramme d'états de A1'.

Comment vérifier la propriété d'exclusion mutuelle ? En vérifiant que les états (p2';q2';1) et (p2';q2';2) ne sont pas accessibles depuis l'état de départ... Et c'est bien le cas !

Mais on ne peut pas utiliser ce graphe pour vérifier la propriété d'absence de famine pour A1. En effet, dans A1', on ne représente plus le fait que les SNC peuvent ne pas terminer... Or c'est précisément cela qui rend possible la famine des processus avec A1. Et ce n'est plus le cas dans A1' : cet algorithme vérifie bien l'absence de famine (on le constate dans le graphe de la figure 19 où le long de toutes les exécutions équitables, les deux processus accèdent infiniment souvent à la SC). Il faut donc être très prudent lorsqu'on simplifie (abstrait) un algorithme : il faut s'assurer que cette abstraction est compatible avec les propriétés étudiées.

Un autre graphe abstrait. Enfin, nous pouvons aussi obtenir un graphe simplifié représentant le comportement de l'algorithme A1 de la manière suivante. Il suffit de partir d'un autre point de vue : puisque l'objectif est d'analyser les accès à la section critique, nous

pouvons repr esenter un etat par un triplet ($_1$; $_2$; t) o u $_1$ est un bool een valant vrai ssi P_1 est dans sa SC, et $_2$ vaut vrai ssi P_2 est dans sa SC, et t repr esente la valeur de la variable turn . A partir de ces triplets, on peut d ecrire le comportement du programme A1 avec le graphe etiquet e de la gure 20.

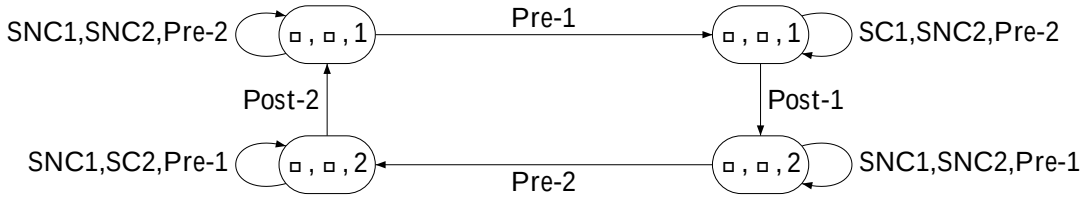

Figure 20 { Graphe simpli e de A1.

L'id ee des transitions etiquet ees est d'indiquer quelles instructions du programme sont e ectu ees lors de la transition. L'etiquette Pre-1 signi e que le processus P_1 e ectue son Pre-protocole, c'est-a-dire l'instruction await turn==1 (qui a pour e et de boucler si le test est n egatif, ou de passer en section critique sinon).

Ce graphe peut servir pour prouver l'exclusion mutuelle ainsi que la pr esence de famine (en consid erant le cycle SNC1= Pre-2 sur le premier etat : si le processus P_1 ne sort jamais de sa section non-critique, il ne permettra jamais a P_2 d'acc eder a sa SC. . .).

Notons que ce graphe simpli e n'est pas a proprement parler un diagramme d'etats comme nous les avons d e nis au premier cours. Il s'agit d'un autre type de construction qui d ecrit aussi le comportement du programme et qui peut servir pour analyser certaines propri et es. Cela montre qu'il existe plusieurs m ethodes alternatives possibles mais a chaque fois il faut etre s^ur que la construction est pertinente pour la propri et e etudi ee : un graphe trop simpli e ne marchera pas. . . Ce travail de construction d'un mod ele (diagramme d'etats abstrait) est un probl eme d elicat !

Exercice 4 [illegible]

```
boolean D1 := False      // variables partagees
boolean D2 := False

-- Processus P1                    -- Processus P2
loop forever :                     loop forever :
p1: section NC                     q1: section NC
p2: D1 := True                     q2: D2 := True
p3: await (D2==False)              q3: await (D1==False)
p4: section critique               q4: section critique
p5: D1:=False                      q5: D2:=False
```

Figure 21 { Algorithme A2

B Logique temporelle { mini kit de survie

Ici nous utilisons la logique temporelle LTL pour enoncer des proprietes sur le comportement des processus concurrents.

LTL est une logique temporelle de temps lineaire les formules de LTL s'interpretent sur des ehpx : Une execution est une sequence in nie d'etats : $q_0 ! q_1 ! q_2 ! : : :$ et chaque etat q_i est etiquete par un ensemble de fff ' (q_i) ($'$ est appelee la fonction d'etiquetage) qui correspondent a l'ensemble des proprietes elementaires que cet etat veri e.

On note (i) le $(i + 1)$-eme etat de (dans l'exemple ci-dessus on a $(i) = q_i$) et on note ^{i} le $(i + 1)$-eme su xe de , c'est-a-dire $q_i ! q_{i+1} ! : : :$

Dans notre cas, on va interpreter les formules de LTL sur des executions du diagramme d'etats (ou d'un graphe equivalent) associe a un algorithme. On a donc une sequence d'etats $q_0 ! q_1 ! q_2 ! : : :$ ou chaque transition correspond a un pas de l'algorithme et chaque etat est une structure de la forme $(p\text{-}; q\text{-}; v1; v2; : : :)$: on sait pour chaque processus quelle est la prochaine instruction qui sera executee, et on connaît la valeur de certaines variables. Ces caracteristiques de chaque etat sont vues comme des propositions atomiques de la logique temporelle : c'est a dire des propositions qui s'interpretent sur un etat (et non sur l'ensemble de l'execution) : ainsi la proposition p2 sera vraie dans un etat de la forme $(p2; : : :)$.

Un exemple d'execution du diagramme d'etats de l'algorithme A1 est :

$$(p1; q1; 1) ! (p1; q1; 1) ! (p1; q2; 1) ! (p2; q2; 1) ! (p3; q2; 1) ! (p3; q2; 1) ! (p4; q2; 1) : : :$$

Dans cet exemple, l'ensemble des propositions atomiques sera $fp1; p2; p3; p4; q1; q2; q3; q4;$ turn $== 1$; turn $== 2$ g. La proposition p3 sera vraie dans tout etat de la forme $(p3; q\text{-};)$, la proposition turn $== 2$ sera vraie dans tout etat de la forme $(p\text{-}; q\text{-}; 2)$, te .

La syntaxe de la logique LTL est de nie comme suit :

De nition 2 (syntaxe de LTL)

$$; ::= P \mid \wedge \mid : \mid U \mid ,$$

uo P te fff

Les operateurs booleens s'interpretent comme d'habitude. Une execution munie d'une fonction d'etiquetage ' veri e une proposition atomique P si l'etat (0) contient P dans son etiquetage (i.e. $P \in '((0))$). L'operateur U (pour until) est utilise de la maniere suivante : U est vraie pour une execution ssi il existe un su xe de veri ant et tous les su xes precedents veri ent (on a bien jusqu'a avoir !). L'operateur , permet de parler du su xe obtenu apres une transition.

On va maintenant de nir formellement cette semantique. Etant donnee une execution $: q0 ! q_1 ! q_2 ! : : :$, on note $\models$ pour signi er que la formule est vraie pour . On de nit cette relation comme suit :

- $\models P$ ssi $P \in '((0))$;
- $\models \wedge$ ssi $\models$ et $\models$;
- $\models :$ ssi il n'est pas vrai que l'on a : $\models$;
- $\models U$ ssi $9i \ 0$ tq $^i \models$ et $80 \ j < i$ on a : $^j \models$

{ $\models$, ssi $^1\models$.

On dénit quelques macros :

{ _ par : (: ^ :);

{) par (:) _ ;

{ > par P _: P pour n'importe quelle proposition atomique P : > est toujours vrai (pour n'importe quel état de n'importe quelle exécution) ;

{ 3 par > U : un jour (dans le futur), sera vraie (en suivant la dénition du Until, on exige qu'il existe un suxe vérifiant et que tous les suxes précédents vérifient > { ce dernier point étant toujours vrai).

{ 2 par : 3 : : est toujours (dans le futur) vraie .

Si l'on reprend l'exemple d'exécution donnée ci-dessus (pour l'algorithme A1), alors la propriété 3 (p3 ^ q2) est vériée. Il en est de même pour la formule (turn == 1) Up3.

Que signie qu'un programme vérie une formule de LTL ? Cela signie que toutes les exécutions de ce programme vérient la formule.

Dans le cours, nous utiliserons essentiellement les opérateurs 3 et 2 (et les opérateurs booléens). Cela permet d'énoncer des propriétés classiques comme :

{ des propriétés de sûreté qui expriment l'idée qu'une mauvaise chose n'arrive jamais.
Par exemple 2 : (p8 ^ q8) : il est toujours vrai que P_1 et P_2 ne peuvent pas être dans leur 8-ème instruction au même instant... C'est équivalent à écrire : : 3 (p8 ^ q8) ou encore 2 (: p8 _ : q8).

{ des propriétés de vivacité qui expriment l'idée qu'une bonne chose doit arriver (dans certaines circonstances).
Par exemple 2 (p2) 3 p8) : il est toujours vrai que si P_1 est en p2, alors plus tard il accèdera à sa 8-ème instruction (par ex. sa section critique)...

{ des propriétés d'équité qui expriment une forme d'équité sur l'exécution.
Par exemple, on pourrait dire que si un processus demande innimment souvent d'accéder à sa section critique, alors il y parviendra innimment souvent... Notez que c'est moins fort que la propriété précédente car avec cette nouvelle propriété, on peut imaginer un processus qui demande 100 fois d'accéder à sa SC et qui y renonce après quelque temps et nit par ne plus la demander. Ce type d'exécutions où la SC n'est jamais obtenue vérierait la propriété d'équité mais pas celle de vivacité ci-dessus.
Si la demande d'accès à sa SC se fait à l'instruction p15 et que sa SC se trouve en p23, on peut écrire la propriété d'équité décrite ci-dessus par : (2 3 p15)) (2 3 p23)

Ce ne sont que quelques exemples, la logique temporelle permet d'énoncer des propriétés très nes sur les exécutions d'un programme. Mais nous ne l'utiliserons que de manière très simple dans tout le cours...

References

[1] Moti Ben-Ari. . Addison-Wesley, second edition, 2006.

[2] Michel Raynal. . Dunod, 1984.

[3] Ekaterina Sedletsky, Amir Pnueli, and Mordechai Ben-Ari. Formal veri cation of the ricart-agrawala algorithm. In , volume 1974 of , pages 325{335. Springer, 2000.

[4] Gadi Taubenfeld. . Prentice Hall, 2006.