CHAP. 6 : TEST DE PROCESSUS

Exercice Préliminaire : comment tester si une implémentation satisfait la spécification ci-dessous?

```
process Pack2StupideAvecControle
   [input, output, control] (Max : Nat):
exit :=
( control?NewMax:Nat[NewMax = 0] ; exit
   [ ]
  control?NewMax:Nat[NewMax > 0] ;
      Pack2StupideAvecControle[input,output,
                               control] (NewMax )
   [ ]
  input?M1:Message ; input?M2:Message ;
   ([taille(M1) + taille(M2) ≤ Max] ->
     output!ajouter(ajouter(vide, M1), M2);
      Pack2StupideAvecControle[input, output,
                                control] (Max )
   [ ]
    [taille(M1)+taille(M2) > Max] ->
     output!ajouter(vide,M1)!ajouter(vide,M2);
      Pack2StupideAvecControle[input, output,
                                control] (Max)
   )
)
endproc (*Pack2StupideAvecControle*)
```

LE PROBLÈME

- On a une spécification Lotos SP
- On a une implémentation I (de SP ???)
- Comment tester I pour vérifier qu'elle est conforme à SP ?

LES QUESTIONS

- Quelles données choisir?
- Comment les soumettre?
- Comment décider si tout s'est bien passé?

(et il peut y avoir de l'indéterminisme!)

UNE PARTIE DES RÉPONSES

- On teste I par des processus "testeurs" que l'on construit à partir de la spécification
- On exécute I et chaque testeur T en parallèle

$$I \parallel T$$

et on observe ce qui se passe (blocages, actions observables)

PRINCIPES DU TEST À PARTIR D'UNE SPÉCIFICATION FORMELLE

- Pour construire les testeurs et décider du succès de I || T, on a besoin de définir ce qu'est une implémentation correcte : *I conf SP*

- À partir de la définition de conf et de SP, on cherche à définir un jeu de test "exhaustif", tel que :

I passe TestExhaust(SP) avec succès ⇔ I conf SP

- I doit être "testable" :

I testable ⇒(I passe TestExhaust(SP) avec succès ⇔I conf SP)

NB : dans le cas des processus, testeur = test

EXEMPLE D'HYPOTHÈSES DE TESTABILITÉ

- L'implémentation sous test est déterministe (ou raisonnablement non déterministe)

- Les actions de la spécification sont toutes implémentées, et de manière atomique

VALIDITÉ ET NON BIAIS

Validité :

I testable $\Rightarrow$*(I passe TestExhaust(SP) avec succès* $\Rightarrow$*I conf SP)*

"On détecte toutes les implémentations non conformes"

Non Biais

I testable $\Rightarrow$*(I passe TestExhaust(SP) avec succès* $\Leftarrow$*I conf SP)*

"On accepte toutes les implémentations conformes"

COMMENT SÉLECTIONNER UN SOUS-ENSEMBLE FINI DE TESTS?

• On fait des hypothèses plus fortes sur l'implémentation sous test : *hypothèses de sélection*

• On préserve la validité et le non biais

(NB : sélection d'un sous-ensemble de TestExhaust(SP) => le non biais est préservé)

• **Exemples :** uniformité, régularité, équité, etc

• Comment les trouver?

- règles ou normes de programmation

- expériences précédentes dans les mêmes conditions

- autres tests, statiques ou dynamiques

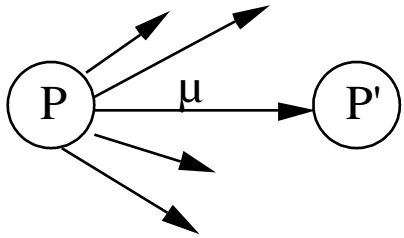
DÉFINITIONS PRÉLIMINAIRES (LOTOS DE BASE)

μ est une action quelconque

a est une action observable

s est une séquence d'actions observables

• P -μ-> P' : P *peut* faire μ et être ensuite dans l'état P'



• =ε => fermeture réflexive et transitive de **i**

- on a toujours P =ε => P

• P =as=> P' :

∃ Q et Q' t.q. P=ε =>Q -a->Q'=s=>P'

(on peut avoir des suites de **i** partout)

• P=s=> : *s est exécutable depuis P*

∃ P' t.q. P=s=>P'

NB: il peut y en avoir plusieurs

• P=/s=> : *s n'est pas exécutable depuis P*

TRACES ET REFUS

• Traces(P) = {s ∈ L* | P=s=> }

où L est l'ensemble des actions observables

• P **after** s = { P' | P=s=>P' }

ensemble des états possibles après s

• Ref(P, s) = {X | ∃ P' dans P **after** s t.q.

∀a∈ X, P'=/a=> }

C'est un ensemble d'ensembles d'actions

- Soit X = {a1, …, an}
- Si X ∈ Ref(P, s) le processus

$$P \parallel s ; [\,]_{i=1,\ldots,n}\ a_i$$

peut bloquer après s

- Sinon, *il ne bloque jamais* et exécute toujours un a_i

NB : quand P=/s=>,

Ref(P, s) = Ø car P **after** s = Ø

EXEMPLE

- L = {a, b, c}
- Q[a,b,c] := (a; b; **stop** [ ] a; c; **stop**)

↑ ↑ ↑

Q Q' Q''

On a Q-a->Q' et Q-a->Q'':

Q **after** a = {Q', Q''}

En Q', on a Q'-b->Q''', et c'est tout :

Ø, {a}, {c}, {a,c}

sont des ensembles X tels que

$\forall x \in X$, Q'=/x=>

En Q'', on a Q''-c->Q'''', et c'est tout :

Ø, {a}, {b}, {a,b}

sont des ensembles X tels que

$\forall x \in X$, Q''=/x=>

DONC

Ref(Q, a) = {Ø, {a}, {c}, {a,c}, {b}, {a,b}}

"Q peut bloquer si on lui propose un de ces ensembles d'actions après a"

Ref(P, s) est *l'ensemble des refus* de P après s

RELATION DE CONFORMITÉ

• I **conf** SP ssi

$\forall s \in Traces(SP), Ref(I, s) \subseteq Ref(SP, s)$

• autrement dit : *soit s une trace de SP ; si après avoir exécuté s, I peut bloquer sur un ensemble d'actions X, alors* $X \in Ref(SP,s)$

• encore autrement dit :

Si une trace observable de SP, s, est exécutable par I et *peut* mener à un état où toutes les actions d'un ensemble A sont refusées

Alors

s peut mener, à partir de SP, à un état où toutes les actions de A sont refusées

"On n'a pas ajouté de blocage par rapport à la spécification"

EXEMPLE 1

- SP := (a; b; **stop** [ ] a; c; **stop**)
- I := a; (b; **stop** [ ] c; **stop**)
- L = {a, b, c}
- Ref(I, ε) = {Ø, {b}, {c}, {b,c}} = Ref(SP, ε)
- Ref(I, a) et Ref(SP, a)

"SP **after** a" contient deux états

Pour Ref(SP, a), voir page 11 : {Ø, {a}, {c}, {a,c}, {b}, {a,b}}

"I **after** a" contient un état

Ref(I, a) = {Ø, {a}}

• *On a donc bien Ref(I, a)⊆Ref(SP, a)*

C'est à vérifier *pour toutes les traces de SP exécutables par I* (ici, toutes), *sans oublier ε*

• Ref(I, a; b) = P(L) = Ref(SP, a; b)

• Ref(I, a; c) = P(L) = Ref(SP, a; c)

DONC I **conf** SP,

mais pas l'inverse car Ref(I, a)≠Ref(SP, a)

Exemple 2

- SP := **i**; a; **stop**
- I := (b; **stop** [ ] **i**; a; **stop**)
- L = {a, b, c}

ATTENTION : I **after** ε = {I, I'} avec

I' = a; **stop**

Traces(SP) = {ε , a}, toutes exécutables par I

• Ref(I, ε) = ce qu'on refuse en I et ce qu'on refuse en I' = {Ø, {c}}∪{Ø, {b}, {c},{b,c}}
= {Ø, {b}, {c}, {b,c}}
= Ref(SP, ε)

• Ref(I, a) = P(L) = Ref(SP, a)

DONC I **conf** SP, ET l'inverse!

EXEMPLE 3

- SP := (b;c; **stop** [ ] **i**; a; **stop**)
- I := **i**; a; **stop**
- L = {a, b, c}

Traces de SP exécutables par I : {ε , a}

- SP **after** ε = {SP, SP'} avec SP' = a; **stop**
- I **after** ε = {I, I'} avec I' = a; **stop**

Ref(SP, ε) = {Ø, {b}, {c}, {b,c}}

Ref(I, ε) = {Ø, {b}, {c}, {b,c}}

• Ref(I, a) = P(L) = Ref(SP, a)

DONC I **conf** SP, et l'inverse!

MAIS ON N'A PAS :

(b; **stop** [ ] **i**; a; **stop**) **conf**

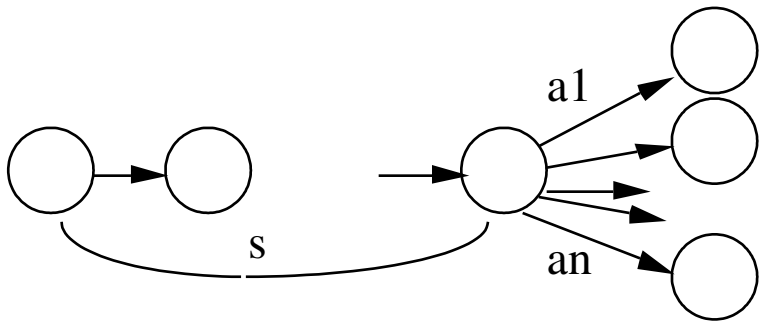
(b; c; **stop** [ ] **i**; a; **stop**)

car après b le premier bloque sur c

=> **conf** n'est pas transitive

TEST EXHAUSTIF POUR conf

exhaust(SP) = {s ; [ ]$_{ai \in A}$ ai; **stop** |
s∈ Traces(SP), A⊆L}



NB, en Lotos complet: L = {g!v | g port visible de SP, v valeur d'un type de SP}

Rappel : g!exp1 || g!exp2 s'exécute si exp1 = exp2

Verdict de I || T avec T∈ exhaust(SP) :

- on atteint un **stop** de T => *succès*
- on bloque avant
  - on bloque avant la fin de s => *succès*

  (s n'est pas exécutable par I) ou verdict *inconclusif*
  - on bloque à la fin de s
    - si SP peut bloquer à la fin de s

=> *succès*

- sinon => *échec*

AMÉLIORATION DE EXHAUST, TESTABILITÉ

• Les tests tels que A∈ Ref(SP, s) sont toujours des succès!

exhaust'(SP) = {s ; [ ]$_{ai \in A}$ ai; **stop** |
s∈ Traces(SP), A⊆L, A∉ Ref(SP, s)}

• il y a aussi des redondances sur les sous-séquences qui … ne bloquent pas

Que signifierait le succès de tous les tests de exhaust(SP) ?

Cela implique que I **conf** SP seulement si

- I a les mêmes actions observables que SP

(sinon on a des blocages non significatifs)

- Le non-déterminisme de I est “équilibré” et on passe chaque test “un nombre suffisant de fois”…

Exemple d’Hypothèse de testabilité : Après p exécutions d’un test tous les comportements possibles ont été couverts.

p dépend de |s| et du nb max de choix possibles à chaque action de I.

Autre exemple : I prend toujours le même choix…

OPTIMISATION DES TESTS

On factorise :

Exemple SP := a ; b ; c ; **stop**

exhaust'(SP) = {

Après ε on doit faire a : **a** ; **stop** ,

Après a on doit faire b : a; **b**; **stop** ,

Après b on doit faire c : a; b; **c**; **stop** }

Verdict : on peut bloquer partout, sauf sur les actions en gras

Jeu de test optimisé, en changeant le verdict :

{a ; b ; c ; **stop**}

Verdict : il faut atteindre le **stop**

Règle de factorisation :

Toute trace qui est préfixe d'une autre trace présente dans le jeu de test peut être retirée du jeu de test en aménageant le verdict pour que

les absences de blocage qu'elle testait soit toujours testées

Attention : ε est toujours une trace!

À PROPOS D'ε

SP := (i ; a ; **stop** [] i ; b ; **stop**)

a ; **stop** avec verdict "il faut atteindre le **stop**" est un test biaisé : on peut refuser a après ε

b ; **stop** est aussi biaisé avec ce verdict…

Test non biaisé : (a ; **stop** []b ; **stop**) avec verdict "il faut atteindre un **stop"** est un test non biaisé (et valide)

Donc, on déterminise

Mais pas seulement dans le cas d'ε !

Autre exemple :

SP := (i; a; c;**stop** [] i;a;b;**stop**)

(a;(c;**stop**[]b;**stop**)) avec verdict "il faut atteindre un **stop"** est un test non biaisé (et valide)

SP := (i; a; c;**stop** [] i;a;(b;**stop** [] i;d; **stop**))

test : a; (c;**stop**[]b;**stop**[]d;**stop**))

Exemple 1 de test : cas non disjoints, avec hypothèses d'uniformité

```
process P[g,h]: noexit:=
    (g?x:int[x<10]; g!x; stop
     [ ]
     g?y:int[y>5]; h!y; stop)
endproc
```

Traces(P) = {ε , <g,i>, <g,i><g,i> $_{i<10}$,
<g,i><h,i> $_{i>5}$}

Ref(P, ε) = P({h!i | i:int})

Ref(P, <g,i>) =

- si 5<i<10, P(toutsauf {h!i , g!i})
- si i≤5, P(tout sauf g!i)
- si i≥10, P(tout sauf h!i)

Ref(P, <g,i><g,i> $_{i<10}$) = P(tout)

Ref(P, <g,i><h,i> $_{i>5}$) = P(tout)

Jeu de test possible : 3 tests qui doivent terminer

- g!v1; (g!v1; **stop** [ ] h!v1; **stop**) *(* 5<v1<10 *)*
- g!v2; g!v2; **stop** *(* v2≤5 *)*
- g!v3; h!v3; **stop** *(* v3≥10 *)*

avec hypothèses d'uniformité sur v1, v2, v3

Exemple 2 : cas disjoints avec blocage, hypothèses d'uniformité et de régularité

---

process Q[g]: **noexit**:=

g?x:int; ([x<10]—> g!x; **stop** *(*ou **exit**)*

[ ]

[x>20]—> Q[g])

endproc

Traces(Q) =

{ε , <g,i>i quelconque, <g,i><g,i> i<10,

<g,i>s i>20 et s∈ Traces(Q)}

Jeu de test possible : n tests qui doivent terminer

- g!v1; g!v1; **stop** *(* ou **exit**, v1<10*)*
- g!v2; g!v3; g!v3; **stop** *(* ou **exit**, v2>20, v3<10*)*
- g!v4; g!v5; g!v6; g!v6; **stop** *(* etc*)*

Choix de n : hypothèse de régularité

Hypothèses d'uniformité sur v1, v2, …

Forme générale des tests :

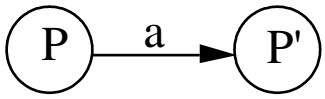
Tests(Q) =

{g!v1; g!v1; **stop** |*v1<10* } ∪ {g!v2;T |*v2>20, T∈Tests(Q)* }

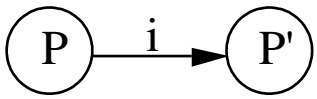
RÈGLES GÉNÉRALES POUR CONSTRUIRE Tests(P)

(en Lotos de base, se généralisent à Lotos complet)

Tests(**stop**) = {**Ø**} Tests(**exit**) = {**exit**}

P := a; P' 

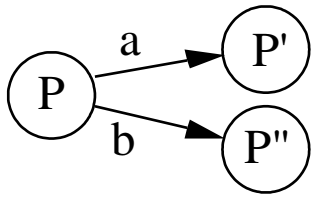
Tests(P) = {a ; T | T∈ Tests(P')}
On ne doit pas bloquer sur a

P := i; P' 

Tests(P) = {Tests(P')}

Cas plus compliqué : [ ]

1 - non-déterminisme contrôlable a ≠ b :



Tests (a;P'[ ] b;P") = {a;T', b; T" | T'∈ Tests(P'),
T"∈ Tests(P")}

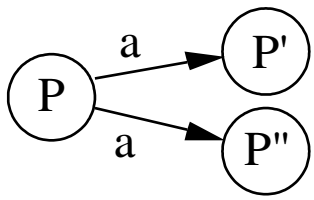
On ne doit bloquer ni sur a, ni sur b.

Attention : l'implémentation ne doit pas pouvoir contrôler le testeur!

- a; T'[ ] b; T" n'est pas un test complet.
- i; a; T'[ ] i; b; T" l'est, mais doit être passé "un nombre suffisant de fois" (donc à éviter).

2 - non-déterminisme non contrôlable :

2.1 : factoriser les cas communs



Tests (a;P'[ ] a;P") = {a; (T'[ ] T") |T'∈ Tests(P'),
T"∈ Tests(P")}

On ne doit pas bloquer sur a.

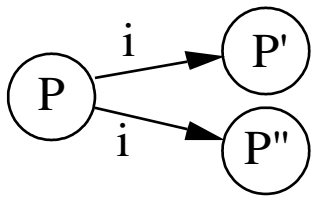
mais aussi :

Tests (a;P'[ ] i;a;P") = {a; (T'[ ] T") |T'∈ Tests(P'),
T"∈ Tests(P")}

Condition : les ensembles d'actions initiales de T' et T" sont disjoints et ≠ i.

Sinon, on factorise comme ci-dessus, et on retire les i comme ci-dessous.

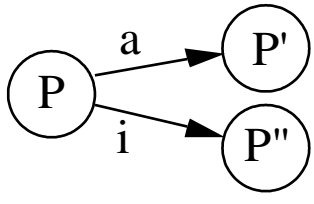
2.2 : supprimer les **i**



Tests (i;P'[ ] i;P") = {(T'[ ]T") |T'∈ Tests(P'), T"∈ Tests(P")} si les ensembles d'actions initiales de T' et T" sont disjoints et ≠ i. *Ces tests autorisent l'implémentation à choisir arbitrairement une branche.*

Sinon, on factorise et on retire les i.

Supprimer les i (suite)



Tests (a; P'[ ] i; P") = {(a; T'[ ]T"), T" | T'∈ Tests(P'), T"∈ Tests(P")}

avec les mêmes conditions sur T' et T".

Attention : si a n'est pas une action initiale de P", elle peut être refusée => a; T' peut bloquer sur a. *Ces tests autorisent l'implémentation à refuser a.*

NB : un seul T" dans les tests (a; T'[ ]T") suffit.

Exemple 3 : où on a un peu de tout

---

process P[g, h] : **noexit** :=

(g?x[x<20]; h!x+2; **stop** [ ] **i** ; g?y[y>10]; h!y ; **stop**)

end proc

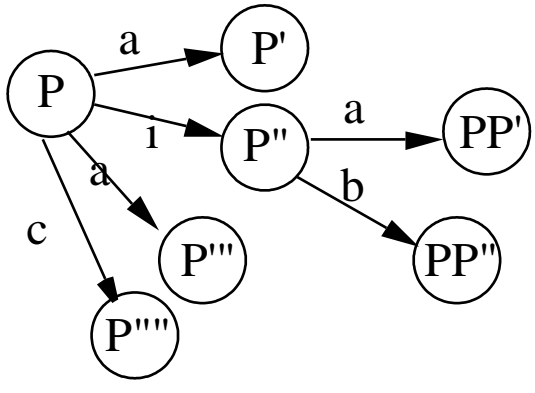
---

supression de **i,** factorisation des cas communs :

- (g!v1; h!v1+2; **stop** *(* v1≤10, g!v1 peut être refusé*)*
  [ ] g!v2; h!v2; **stop**) *(* v2≥20, toujours accepté*)*
- (g!v1; h!v1+2; **stop** *(* v1≤10, g!v1 peut être refusé*)*
  [ ] g!v3; (h!v3+2; **stop** [ ]h!v3; **stop**)) *(* 10<v3<20*)*
- g!v2; h!v2; **stop** *(* v2≥20*)*
- g!v3; (h!v3+2; **stop** [ ]h!v3; **stop**) *(* 10<v3<20*)*

Exemple récapitulatif 3bis

(heureusement, de tels cas sont rares en pratique!)



P = (a; P' [ ] i; (a;PP' [ ] b; PP") [ ] a; P"'[ ] c; P"")

Les actions observables possibles à partir de P sont a, b, c.

• a est toujours acceptée et peut mener à P', PP' ou P"'

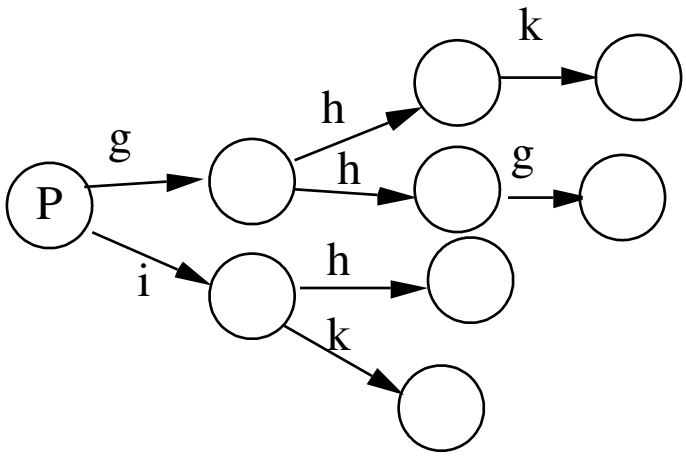
• b est toujours acceptée et mène à PP"

• c peut être refusée (en P"). Si elle est acceptée elle mène à P""

Tests(P) = {a; (Tests(P') [ ] Tests(PP') [ ] Tests(P"')),
b; Tests(PP"),
(c; Tests(P"") [ ]
a; (Tests(P') [ ] Tests(PP') [ ] Tests(P"'))),
(c; Tests(P"") [ ] b; Tests(PP")) }

NB : 3 tests suffisent. Un des 2 derniers est redondant.

Exemple récapitulatif 3ter



P[g, h, k] := (g; (h; k; **stop** [] h; g; **stop**)
[]
i ; (h; **stop** [] k; **stop**))

Actions initiales possibles : g, h, k

- g peut être refusée
- h doit être factorisée après g

Tests(P), version 1 :

---

h; **stop**

k; **stop**

(g; h; (k; **stop** [] g; **stop**) [] h; **stop**)

Tests(P), version 2 :

---

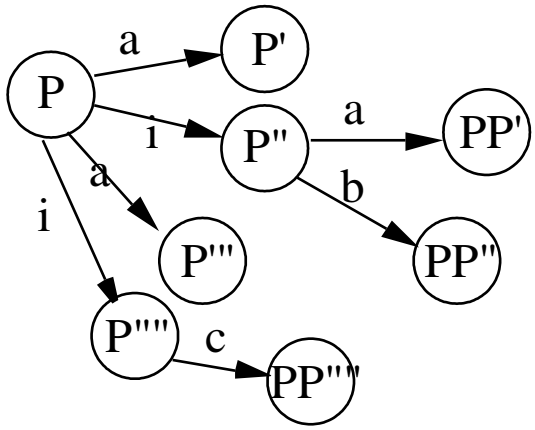
h; **stop**

---

k; **stop**

(g; h; (k; **stop** [ ] g; **stop**) [ ] k; **stop**)

exercice



P = (a; P' [ ] i; (a;PP' [ ] b; PP'') [ ] a; P'''[ ] i;c; P'''')

Les actions observables possibles à partir de P sont a, b, c.

• a peut être refusée (en P'''', mais alors c est acceptée) ; si elle est acceptée elle mène à P', PP' ou P'''

• b peut être refusée (en P'''', mais alors c est acceptée); si elle est acceptée elle mène à PP''

• c peut être refusée (en P'', mais alors a ou b sont acceptées); si elle est acceptée elle mène à PP''''

Tests possibles :

Tests(P) = {(a; (Tests(P')[ ] Tests(PP')[ ] Tests(P''))
[ ] c; Tests(PP'''')),
(b; Tests(PP'')[ ] c; Tests(PP'''')),

(c; Tests(PP””)[ ] a; Test(PP’)) }