# PC3R
# Cours 05 - Vérification

Romain Demangeon

PC3R MU1IN507 - STL S2

04/03/2021

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# Plan du Cours 5

- Vérification des systèmes concurrents.
- *Model-checking*: exemple de *SPIN*
- Typage: exemple des *sessions*.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# Difficulté de la programmation concurrente

- Programmation concurrente:
  - exponentialisation de l'espace d'états
  - par mémoire partagée ou passage de message
- Croissance de la difficulté:
  - bugs plus fréquents, plus difficile à détecter,
  - impossibilité de visualiser les comportements possibles.
- Apparition d'écueils propres à la concurrence:
  - compétition pour les ressources,
  - interbloquage entre les acteurs du système,
  - considération de performance supplémentaires:
    - utilisation optimale du processeur (attente active, préemption au bon moment, changement de contexte),
    - utilisation optimale des multi-coeurs (répartition des calculs, coûts de synchronisation),
- Des solutions (mutex et conditions, coopération, passage de message, annotations, typage, ...):
  - ajoute de la charge de travail pour le programmeur,
  - ajoute des nouveaux problèmes (canaux emmêlés, interbloquage dans les mutex, ...)
- La programmation concurrente est essentiellement génératrice à erreurs et de bugs.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Caractère critique de la programmation concurrente

- La programmation concurrente est nécessaire pour les systèmes composites:
  - systèmes d'exploitation, (assez critique)
  - systèmes embarqués, (souvent critique)
  - applications *web*, (souvent peu critique)
  - réseaux propriétaires (bancaires, scientifiques, ...). (parfois critique)
- Il n'y pas plus de repas gratuit.
  - *The Free Lunch is Over*, H. Sutter, 2005: fin du monde séquentiel
  - les limites physiques du calcul séquentiel sont (quasiment) atteintes.
  - le 21ème siècle est concurrent.
    - corroboré par les avancées technologiques, la recherche scientifique et le développement des langages.
- La programmation concurrente revêt un caractère ubiquitaire.
- en résumé: "c'est difficile mais c'est obligé".
  - désirabilité sur le marché de programmeurs concurrents expérimentés (et/ou intelligents),
  - nécessité de trouver des méthodes pour garantir la correction du code concurrent.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Propriétés

- on considère un système dont on veut assurer la correction
  - système réel dont on connait le fonctionnement,
  - système réel dont on ne connait pas complètement le fonctionnement,
  - système imaginé (futur).
- on énonce des propriétés désirables du système:
  - sûreté: "quelque chose de mal ne se produit jamais"
    - $x$ est toujours inférieur à $y$
    - les deux feux tricolores ne sont pas au vert en même temps.
    - les trois routines ne sont pas simultanément en train d'essayer d'envoyer sur le même canal.
  - vivacité: "quelque chose de bien finit par se produire"
    - si $z$ vaut 3 à un moment, alors, plus tard, $x$ finit par dépasser $y$,
    - un feu tricolore est vert infiniment souvent,
    - chaque routine qui cherche à entrer en section critique finit par pouvoir y accéder.
  - autres:
    - terminaison: le serveur traite chaque requête en temps fini,
    - comportement: les deux routines sont d'accord sur l'utilisation d'un canal commun,
    - ...

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Test

Comment s'assurer qu'un système valide ces propriétés ?

- Première méthode: le test
  - les propriétés sont codées avec le système
    - directement: avec des assertions dans le code,
    - subtilement: en branchant des contrats sur le système,
    - automatiquement: en utilisant un cadriciel de test.
  - de nombreuses exécutions partielles (tests unitaires) ou totale du système sont lancées successivement.
    - notion de couverture des tests,
    - intérêt de générer automatiquement des cas de tests pertinents et variés.
    - consomme énormément de ressources.
  - souvent, la correction ne peut être totalement atteinte (domaines de test infinis vs. moyens finis).
- dans le cas concurrent, l'exponentialisation de l'espace d'états se paye directement:
  - le nombre de tests nécessaires pour atteindre une couverture similaire devient exponentiel
- ça reste la principale méthode de vérification des programmes concurrents dans l'industrie.
  - techniquement facile, pas d'accès au code.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Moniteurs

- Variante du test pour les applications non-critique.
- les propriétés sont intégrées dans un composant spécial du système, le *moniteur* qui s'execute en simultanément.
  - il peut être écrit dans un langage différent,
- le moniteur à une visibilité sur le système sans pouvoir d'action directe:
  - accès à des variables en lecture seul,
  - interception (et relâche) des messages échangés entre les composants,
- le moniteur vérifie en temps réel des propriétés de sûreté.
- réaction prévue, si l'une est violée;
  - action indirecte: enregistrement des erreurs, contact d'un autre composant, ...
  - action directe: modification d'un message, arrêt du sytème, ...

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Analyse statique

- Principe: le programme est analysé avant d'être exécuté (à la compilation, ou avant).
  - demande un accès au code source,
  - les propriétés sont vérifiées à l'aide de programmes spécifique (vérifieurs de types, vérifieurs de modèles)
  - le programme est soit accepté, soit rejeté par l'analyse.
  - si un programme qui passe l'étape de vérification avec succès, on a la preuve qu'il satisfait les propriétés vérifiées.
- la plupart des propriétés intéressantes (correction, terminaison, accessibilité, progrès, ...) sont indécidables;
  - les analyses sont souvent correctes mais pas complètes
    - on rate des bons systèmes
    - on n'accepte pas de mauvais systèmes
  - le développement d'une analyse statique est un processus scientifique

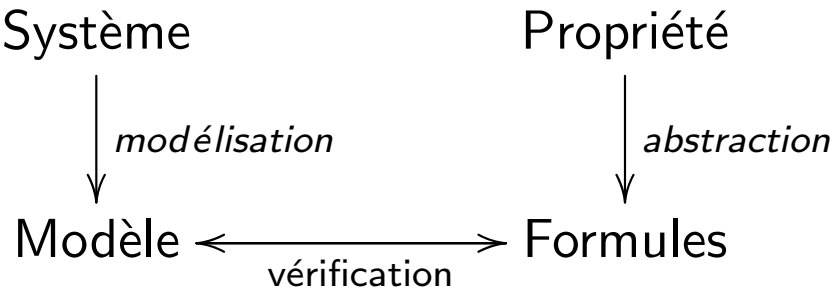
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# Typage et Interprétation Abstraite

- Typage:
  - principe: utiliser les types qui décorent les données pour transporter des informations qui permettent de garantir des propriétés
    - exemple de la notion de propriété (*ownership*) et des *reference counted pointers* en *Rust*.
    - le typage fort d'*OCaml* et de *Go* permet de rejeter à la compilation un programme composés de threads qui ne s'entendraient pas sur l'utilisation d'un canal.
  - les types peuvent être explicites (annotations en *Go*) ou implicites (inférence en *OCaml*)
  - les types sont vérifiés par un *type-checker*.
- Interprétation Abstraite:
  - principe: approximer l'espace des états pris par un programme pour vérifier des propriétés de sûreté.
  - utilisation d'outils mathématiques (points fixes) pour raisonner sur les abstractions de programmes.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Model-checking*: Principes




- le système est modélisé mathématiquement:
  - par un automate ou un système de transition
- les propriétés sont converties en formules décrites par un langage spécifique (LTL, $\mu$-calcul, ...)
- un vérifieur assure la validité des formules sur les états du modèle
- les *model-checkers* travaillent sur des ensemble infinis d'états (à croissance exponentielle dans le cas des systèmes concurrents).

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *SPIN*: Principes

- *model-checker* créé aux *Bell Labs* dans les années 80
  - initiative de Gerard Holzmann
  - toujours "à la mode" dans les années 2020.
- les modèles sont écrits en *Promela* (*Process Meta Language*)
  - langage de description de systèmes avec mémoire partagée et canaux synchrones.
  - ce n'est pas un langage de programmation.
- les formules sont écrites en LTL (*Linear Temporal Logic*)
- SPIN construit un model-checker spécifique en *C* à partir d'un modèle en *Promela*.
- nombreuses optimisations:
  - compression de l'espace d'états,
  - exploitation des ordres partiels (commutativité des actions atomiques)

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Promela*: syntaxe

- variables `int temp = 0; temp = compteur`
- canaux `ch?c; c!42`
  - ordre supérieur.
  - synchrones `chan ch = [0] of {chan}`
  - synchrones `chan ch2 = [3] of {int}`
- gardes `x == 0; ch!x`
  - bloquante tant qu'elles sont fausses.
- structures de contrôle: `if`, `do`

```
if
    :: cond1 -> ...
    :: cond2 -> ...
    :: true ->
fi
```

- flot de contrôle par états:

```
debut:
    ...
boucle:
    ...
    goto loop
fin:
    ...
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Modéliser en *Promela*

- on définit des processus `proctype` lancés depuis un processus principal `init`
  - `active proctype` définit un processus lancé directement
- on ne programme pas:
  - SPIN explore tous les chemins possibles:
    - pas d'ordre sur les branchements (tout est exploré)
    - l'aléatoire est explicite.
    - on peut bloquer des chemins avec des gardes.
  - on écrit des automates: chaque action atomique est un changement d'état, les branchements et les gardes permettent d'obtenir le contrôle et le non-déterminisme.
  - on ne modélise pas les données: souvent, juste le flot de contrôle du système nous intéresse.
- on utilise des processus observateurs pour réifier des propriétés

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Spin*: Machine à café

```
mtype = {b_cafe, b_the}

proctype machine(chan piece; chan sert; chan bc; chan bt){
    int argent
    int valeur
    int k
    dispo:
        if
        :: piece?valeur -> argent = argent + valeur; goto dispo
        :: argent >= 2 -> bc?k; goto the
        :: argent >= 1 -> bt?k; goto cafe
        fi
    cafe:
        sert!b_cafe; goto dispo
    the:
        sert!b_the; goto dispo
}
```

- Modèle de processus paramétré par quatre canaux.
- sur `sert` on fait passer une énumération, sur les autres des entiers (bidons, on ne s'intéresse pas à leur valeur).
- `->` et `;` ont la même sémantique.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Spin*: Machine à café (II)

```
proctype louis(chan piece; chan sert; chan bc; chan bt){
    mtype boisson
    do
    :: piece!1
    :: piece!2
    :: bc!0
    :: bt!0
    :: sert?boisson; if
        :: boisson == b_cafe -> printf("Louis boit du cafe.\n")
        :: boisson == b_the -> printf("Louis boit du the.\n")
        fi
    od
}
```

- do est un `if` répété
- ce modèle de client teste en permanence toutes les interactions.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Machine à café (III)

```
init{
    chan piece = [0] of {int}
    chan sert = [0] of {mtype}
    chan bc = [0] of {int}
    chan bt = [0] of {int}
    run machine(piece, sert, bc, bt)
    run louis(piece, sert, bc, bt)
}
```

- initialisation: création et passage des canaux synchrones
- on aurait pu faire des déclaration globales.
- on lance les processus.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Machine à café, exécution

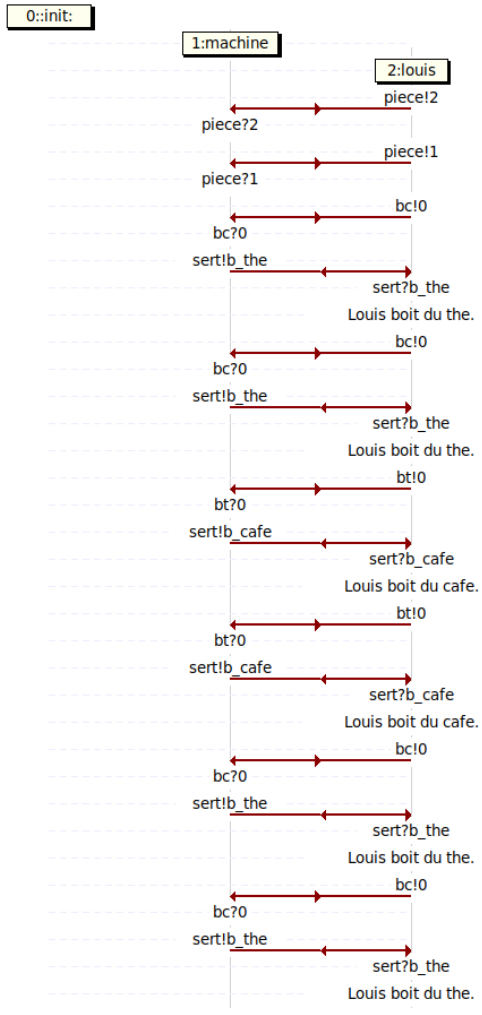
- à partir d'un fichier *Promela* on peut utiliser SPIN.
- `spin cafe.pml` simule une exécution aléatoire.
- `spin -c cafe.pml` propose une visualisation ASCII de l'exécution

```
proc 0 = :init:
proc 1 = machine
proc 2 = louis
q\p   0    1    2
  1   .    .    piece!1
  1   .    piece?1
  4   .    .    bt!0
  4   .    bt?0
  2   .    sert!b_cafe
  2   .    .    sert?b_cafe
                 Louis boit du cafe.
  4   .    .    bt!0
  4   .    bt?0
  2   .    sert!b_cafe
  2   .    .    sert?b_cafe
                 Louis boit du cafe.
  4   .    .    bt!0
  4   .    bt?0
  2   .    sert!b_cafe
  2   .    .    sert?b_cafe
                 Louis boit du cafe.
  4   .    .    bt!0
  4   .    bt?0
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Machine à café, exécution (II)

- `spin -M cafe.pml` propose une visualisation *postscript* de l'exécution




SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Machine à café, vérification

- on ne vérifie rien en exécutant (ce n'est pas le but).
- plusieurs manières de vérifier des propriétés.
- utilisation d'un observateur
  - on ajoute un processus observateur au système et des canaux pour lui envoyer des messages.
  - on ajoute de l'envoi d'information à l'observateur depuis des points-clefs du système.
  - l'observateur vérifie des propriétés à partir des information reçues

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Machine à café, vérification (II)

```
mtype = {p1, p2, caf, th}

proctype louis(chan piece; chan sert; chan bc; chan bt; chan obs){
    mtype boisson
    do
    :: piece!1; obs!p1
    :: piece!2; obs!p2
    :: bc!0
    :: bt!0
    :: sert?boisson; if
        :: boisson == b_cafe -> obs!caf; printf("Louis boit du cafe.\n")
        :: boisson == b_the -> obs!th; printf("Louis boit du the.\n")
        fi
    od
}
```

- le code du client est modifié pour envoyer à l'observateur des informations sur les actions effectuées.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Machine à café, vérification (III)

```
proctype observateur(chan obs){
    mtype m
    int cafes_bus = 0
    int thes_bus = 0
    int argent_depense = 0
    loop:
    do
    :: obs?m -> if
        :: m == p1 -> argent_depense = argent_depense + 1
        :: m == p2 -> argent_depense = argent_depense + 2
        :: m == caf -> cafes_bus = cafes_bus + 1
        :: m == th -> thes_bus = thes_bus + 1
        fi
        printf("Observateur: %d; %d; %d\n", argent_depense,
                                            cafes_bus,
                                            thes_bus)
        assert (argent_depense >= cafes_bus + thes_bus * 2)
    od
}
```

- l'observateur réagit aux messages du systèmes,
- on utilise une assertion dans le code de l'observateur.
- on vérifie que le client n'a pas bu pour plus cher que ce qu'il a dépensé.

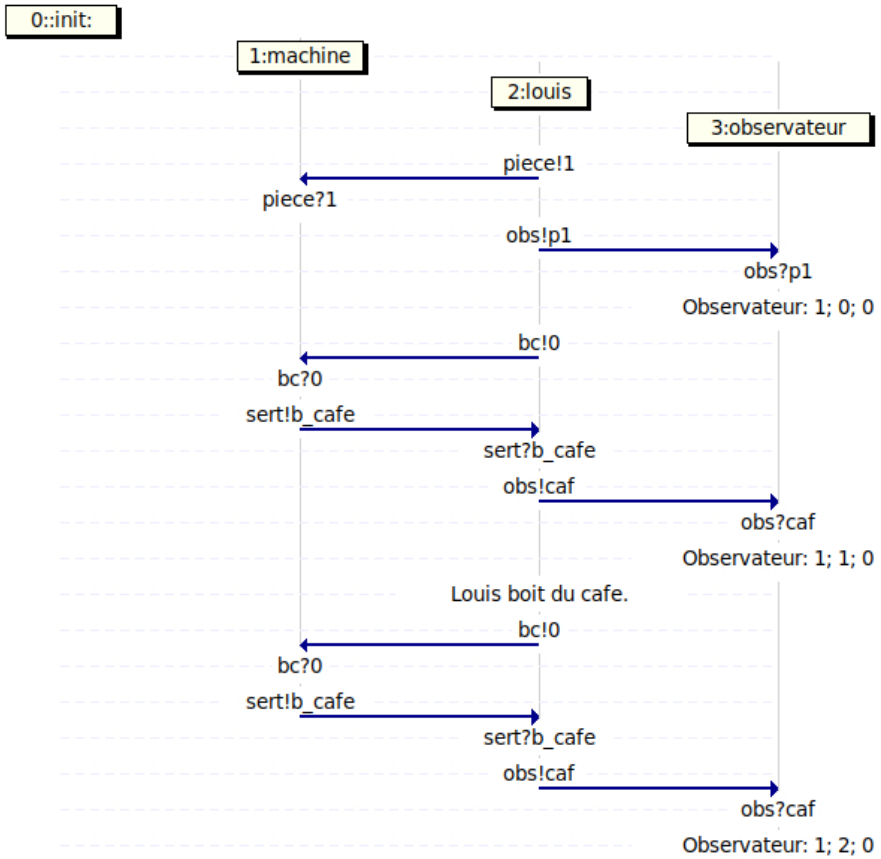
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Machine à café, vérification (IV)

- on génère un analyseur (*pan*) avec `spin -a cafe1.pml`
  - on doit ensuite compiler l'analyseur `gcc -o pan pan.c`
  - puis le lancer `./pan`
  - alternativement, directement `spin -search cafe1.pml`
- par défaut, l'analyseur cherche:
  - les violations d'assertion,
  - les interbloquages (*invalid end-states*)
  - par profondeur (option `-bfs` pour la largeur)
  - dans un espace d'état limité (10000 états) (option `-mN` pour changer la limite en `N`)
- si l'analyse trouve un problème, on récupère une trace qui y amène.
- `spin -c/M -t cafe1.pml` permet de visualiser la trace.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Spin*: Machine à café, vérification (V)




- on récupère une erreur d'assertion et une trace correspondante.
- on a oublié de mettre à jour l'argent quand on appuie sur le bouton.

```
:: argent >= 2 -> bc?k; goto the
:: argent >= 1 -> bt?k; goto cafe
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Spin*: Machine à café, vérification (VI)

```
...
    dispo:
        if
        :: piece?valeur -> argent = argent + valeur; goto dispo
        :: argent >= 0 -> bt?k; goto the
        :: argent >= 1 -> bc?k; goto cafe
        fi
    cafe:
        argent = argent - 1
        sert!b_cafe; goto dispo
    the:
        argent = argent - 2
        sert!b_the; goto dispo
...

proctype jeanne(chan piece; chan sert; chan bc; chan bt){
    mtype boisson
    int i
    piece!2;
    bt!0;sert?boisson
    bt!0;sert?boisson
}
```

- `argent >= 0` au moment de presser le bouton du thé.
- client déterministe

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Spin*: Machine à café, vérification (VII)

- on veut détecter si `argent` est toujours positif
- on définit une formule atomique `define q (argent < 0)`
- on passe à SPIN une formule de LTL
  `spin -a -f '<> q' cafe2.pml` (il existe un état qui vérifie `q`)
- SPIN trouve le contre-exemple:

```
proc 1 = machine
proc 2 = jeanne
q\p   0   1
  1   .   .   piece!2
  1   .   piece?2
  4   .   .   bt!0
  4   .   bt?0
  2   .   sert!b_the
  2   .   .   sert?b_the
  4   .   .   bt!0
  4   .   bt?0
spin: trail ends after 29 steps
-------------
final state:
-------------
#processes: 3
                argent = -2
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Philosophes

```
proctype philosophe(int id; chan pg; chan rg; chan pd; chan rd){
    int f1, f2
    dort:
        printf("Philosophe %d DORT.\n", id)
    mange:
        pg?f1;
        printf("Philosophe %d prend fourchette gauche: %d.\n", id, f1)
        pd?f2;
        printf("Philosophe %d prend fourchette droite: %d.\n", id, f2);
        amange[id-1] = 1;
        printf("Philosophe %d MANGE.\n", id)
        rd!f2;
        printf("Philosophe %d relache fourchette droite: %d.\n", id, f2)
        rg!f1;
        printf("Philosophe %d relache fourchette gauche: %d.\n", id, f1)
        goto dort
}
```

- la prise et la relache d'une fourchette se font à l'aide de canaux,
- définition standard: prendre fourchette gauche puis droite, relacher droite puis gauche.
- le philosophe met à jour un tableau quand il mange.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Philosophes (II)

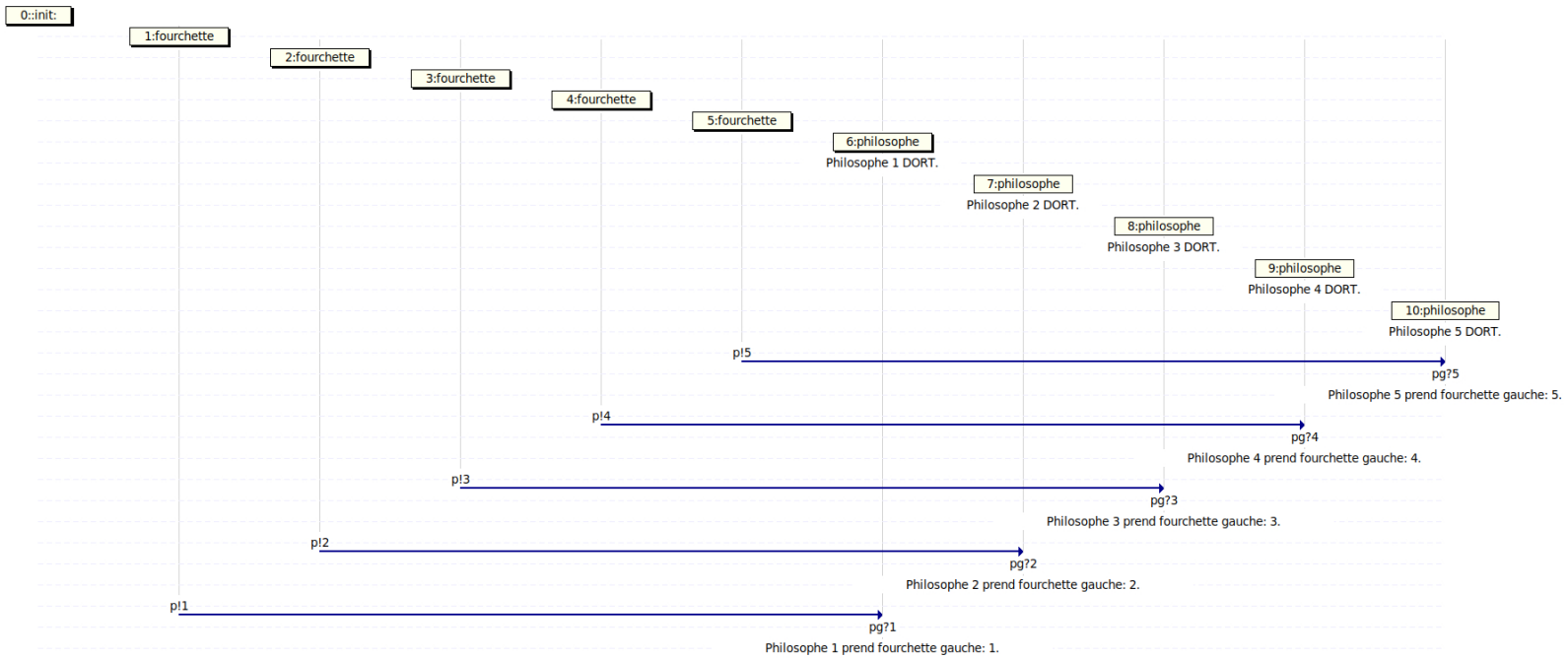
```
proctype fourchette(int id; chan p; chan r){
    int f = id
    loop:
        p!f;
        r?f;
        goto loop
}

init{
    int i
    int nphi = NB_PHI
    chan prend[NB_PHI] = [0] of {int}
    chan relache[NB_PHI] = [0] of {int}
    for(i:1..nphi){
        run fourchette(i, prend[i-1], relache[i-1])
    }
    for(i:1..nphi-1){
        run philosophe(i, prend[i-1], relache[i-1], prend[i], relache[i])
    }
    run philosophe(nphi, prend[nphi-1], relache[nphi-1],
                         prend[0], relache[0])
}
```

- fourchettes et initialisation.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Philosophes (II)

- *SPIN* cherche, de base, des *invalid end-states*




SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Philosophes (III)

```
active proctype verifieur(){
    int acc, i
    int nphi = NB_PHI
    do
    :: atomic{
        acc = 0
        for(i:1..nphi){
            acc = acc + amange[i-1]
        }
        printf("Ont mange: %d\n", acc)
        assert(acc < NB_PHI)
    }
    od
}
```

- un processus vérifieur avec une assertion fausse si tous les philosophes ont mangé.
- on lance `spin -search -bfs -E philo.pml` pour ignorer les interbloquages.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Spin*: Philosophes (IV)

```
 5    .    .    .    .    .    .    p!5
 5    .    .    .    .    .    .    .    .    .    .    .    pg?5
 3    .    .    .    .    p!3
 3    .    .    .    .    .    .    .    .    .    pg?3
 4    .    .    .    .    .    p!4
 4    .    .    .    .    .    .    .    .    .    pd?4
 9    .    .    .    .    .    .    .    .    .    rd!4
 9    .    .    .    .    .    r?4
 8    .    .    .    .    .    .    .    .    .    rg!3
 8    .    .    .    .    r?3
 4    .    .    .    .    .    p!4
 4    .    .    .    .    .    .    .    .    .    .    pg?4
 1    .    .    p!1
 1    .    .    .    .    .    .    .    .    .    .    .    pd?1
 6    .    .    .    .    .    .    .    .    .    .    .    rd!1
 6    .    .    r?1
10    .    .    .    .    .    .    .    .    .    .    .    rg!5
10    .    .    .    .    .    .    r?5
 5    .    .    .    .    .    .    p!5
 5    .    .    .    .    .    .    .    .    .    .    pd?5
 1    .    .    p!1
 1    .    .    .    .    .    .    .    pg?1
 2    .    .    .    p!2
 2    .    .    .    .    .    .    .    pd?2
 7    .    .    .    .    .    .    .    rd!2
 7    .    .    .    r?2
 2    .    .    .    p!2
 2    .    .    .    .    .    .    .    .    pg?2
 3    .    .    .    .    p!3
 3    .    .    .    .    .    .    .    .    pd?3
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Spin*: *Never Claims* et *LTL*

- les propriétés compliquées peuvent être encodés sous forme de *never claim*:
  - petits automates qui tournent de manière synchrone (ils ne s'entrelacent pas, mais avance avec le système)
  - ils vérifient la validité d'une formule,
  - ils sont utilisés négativement ("*never*")
- SPIN traduit automatiquement les formules de la LTL en *never claims*.
  - $\phi ::= \bot \mid \top \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$
    (faux, vrai, négation, conjonction, disjonction)
  - $\mid X\ \phi$
    $\phi$ est vraie dans l'*état suivant*.
  - $\mid \Box\ \phi$
    $\phi$ est vraie dans tous les états à partir de maintenant.
  - $\mid \Diamond\ \phi$
    $\phi$ est vraie dans un état dans le futur.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# *Spin*: *Never Claims* et *LTL*

- Génération de *never claim*: `spin -f '[](q || <>p)'`

```
never  {    /* [](q || <>p) */
T0_init:
        do
        :: ((((p)) || ((q)))) -> goto accept_S13
        :: (1) -> goto T0_S25
        od;
accept_S13:
        do
        :: ((((p)) || ((q)))) -> goto T0_init
        :: (1) -> goto T0_S25
        od;
accept_S25:
        do
        :: ((p)) -> goto T0_init
        :: (1) -> goto T0_S25
        od;
T0_S25:
        do
        :: ((p)) -> goto accept_S13
        :: (1) -> goto T0_S25
        :: ((p)) -> goto accept_S25
        od;
}
```

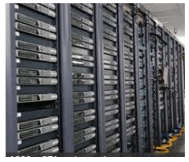
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# *Spin*: Bilan

- outil puissant
  - fondements mathématiques (automates de Büchi, ...),
  - beaucoup d'optimisations (ordres partiels, ...)
- pour bien utiliser *SPIN*:
  - savoir optimiser l'espace d'état:
    - ne pas sur-modéliser,
    - utiliser `atomic`,
    - optimiser les branchements et les transitions.
  - savoir quoi chercher:
    - traduction des propriétés.
    - options de recherche.
- SPIN est utilisé pour la vérification de systèmes concrets:
  - notamment des missions spatiales (*NASA*)

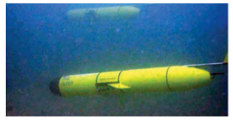
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Vérification de Protocoles acentralisés



San Diego

Brest

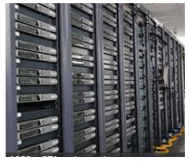


Ocean Pacifique

- Trois programmes indépendants (client, agent, instrument):
  - écrits dans des langages différents ,
  - avec librairies et compilateurs locaux,
  - interagissent par messages.
- Pas de contrôle global.
- Objectif: garantir le succès des interactions.
  - Méthode formelle: types de sessions.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

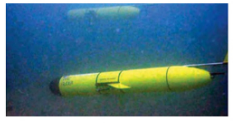
# Vérification de Protocoles acentralisés



San Diego

Negotiation

Brest

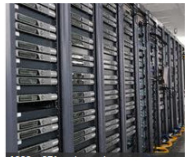


Ocean Pacifique

- Trois programmes indépendants (client, agent, instrument):
  - écrits dans des langages différents ,
  - avec librairies et compilateurs locaux,
  - interagissent par messages.
- Pas de contrôle global.
- Objectif: garantir le succès des interactions.
  - Méthode formelle: types de sessions.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

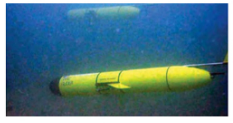
# Vérification de Protocoles acentralisés



San Diego

Initialisation

Brest

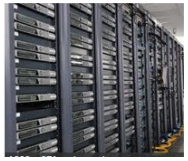


Ocean Pacifique

- Trois programmes indépendants (client, agent, instrument):
  - écrits dans des langages différents ,
  - avec librairies et compilateurs locaux,
  - interagissent par messages.
- Pas de contrôle global.
- Objectif: garantir le succès des interactions.
  - Méthode formelle: types de sessions.

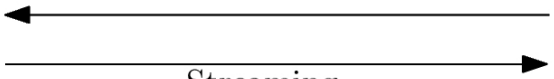
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

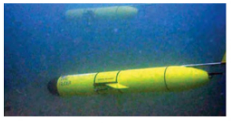
# Vérification de Protocoles acentralisés



San Diego

Brest



Ocean Pacifique

- Trois programmes indépendants (client, agent, instrument):
  - écrits dans des langages différents ,
  - avec librairies et compilateurs locaux,
  - interagissent par messages.
- Pas de contrôle global.
- Objectif: garantir le succès des interactions.
  - Méthode formelle: types de sessions.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# Types de Sessions Binaires

- *Languages Primitives and Type Discipline for Structured Communication-Based Programming*, Honda, Kubo, Vasconcelos, ESOP 1998
  - Domaine: algèbres de processus ($\pi$-calcul): les agents communiquent avec des messages sur des canaux.
  - Motivation: construire des types pour guider les interactions entre deux agents sur un même canal.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Types de Sessions Binaires

- *Languages Primitives and Type Discipline for Structured Communication-Based Programming*, Honda, Kubo, Vasconcelos, ESOP 1998
  - Domaine: algèbres de processus ($\pi$-calcul): les agents communiquent avec des messages sur des canaux.
  - Motivation: construire des types pour guider les interactions entre deux agents sur un même canal.
- Principes:
  - Décrire formellement les interactions entre deux participants (une *session*) sur un unique canal $s$.
    - Briques de bases: communications (direction, étiquette, type du message), choix, récursion, fin de session.
  - Séparation du type en deux extrémités symétriques (semblables à des processus CCS).
  - Validation, (système de types) de chaque participant par rapport à son type respectif.
- Originalité dans les types pour les canux: la séquence:
  - Types habituels: `'a * ('b channel) channel`
  - Types de session: `('a send);('b receive) channel`.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Types de sessions binaires - Exemple

- Type global:

$$\begin{array}{lll} \texttt{Seller} \to \texttt{Buyer} & (\texttt{price}); & \\ \texttt{Buyer} \to \texttt{Seller} & -(\texttt{ko}); & \texttt{end} \\ & -(\texttt{ok}); & \{\texttt{Seller} \to \texttt{Buyer}\ (\texttt{item}); \\ & & \texttt{end}\} \end{array}$$

- Types locaux (extrémités):
  - $\texttt{Buyer} : ?(\texttt{price}); !\{\texttt{ko}; \texttt{end}\ ,\ \texttt{ok}; ?\texttt{item}; \texttt{end}\}$
  - $\texttt{Seller} : !(\texttt{price}); ?\{\texttt{ko}; \texttt{end}\ ,\ \texttt{ok}; !\texttt{item}; \texttt{end}\}$
- Processus candidats:
  - $s_{\texttt{price}}(p).(\overline{s}_{\texttt{ok}}.s_{\texttt{item}}(i) + \overline{s}_{\texttt{ko}})$:

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# Types de sessions binaires - Exemple

- Type global:

$$\begin{array}{lll} \texttt{Seller} \to \texttt{Buyer} & (\texttt{price}); & \\ \texttt{Buyer} \to \texttt{Seller} & -(\texttt{ko}); & \texttt{end} \\ & -(\texttt{ok}); & \{\texttt{Seller} \to \texttt{Buyer}\ (\texttt{item}); \\ & & \texttt{end}\} \end{array}$$

- Types locaux (extrémités):
  - $\texttt{Buyer} : ?(\texttt{price}); !\{\texttt{ko}; \texttt{end}\ ,\ \texttt{ok}; ?\texttt{item}; \texttt{end}\}$
  - $\texttt{Seller} : !(\texttt{price}); ?\{\texttt{ko}; \texttt{end}\ ,\ \texttt{ok}; !\texttt{item}; \texttt{end}\}$
- Processus candidats:
  - $s_{\texttt{price}}(p).(\overline{s}_{\texttt{ok}}.s_{\texttt{item}}(i) + \overline{s}_{\texttt{ko}})$: bon $\texttt{Buyer}$.
  - $s_{\texttt{price}}(p).\overline{s}_{\texttt{ko}}$:

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# Types de sessions binaires - Exemple

- Type global:

$$\begin{array}{lll} \texttt{Seller} \to \texttt{Buyer} & (\texttt{price}); & \\ \texttt{Buyer} \to \texttt{Seller} & -(\texttt{ko}); & \texttt{end} \\ & -(\texttt{ok}); & \{\texttt{Seller} \to \texttt{Buyer}\ (\texttt{item}); \\ & & \texttt{end}\} \end{array}$$

- Types locaux (extrémités):
  - $\texttt{Buyer} : ?(\texttt{price}); !\{\texttt{ko}; \texttt{end}\ ,\ \texttt{ok}; ?\texttt{item}; \texttt{end}\}$
  - $\texttt{Seller} : !(\texttt{price}); ?\{\texttt{ko}; \texttt{end}\ ,\ \texttt{ok}; !\texttt{item}; \texttt{end}\}$
- Processus candidats:
  - $s_{\texttt{price}}(p).(\overline{s}_{\texttt{ok}}.s_{\texttt{item}}(i) + \overline{s}_{\texttt{ko}})$: bon $\texttt{Buyer}$.
  - $s_{\texttt{price}}(p).\overline{s}_{\texttt{ko}}$: bon $\texttt{Buyer}$.
  - $\overline{s}_{\texttt{price}}\langle 100\ \texttt{Fr}\rangle.s_{\texttt{ko}}$:

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Types de sessions binaires - Exemple

- Type global:

$$\begin{array}{lll} \texttt{Seller} \to \texttt{Buyer} & (\texttt{price}); & \\ \texttt{Buyer} \to \texttt{Seller} & -(\texttt{ko}); & \texttt{end} \\ & -(\texttt{ok}); & \{\texttt{Seller} \to \texttt{Buyer}\ (\texttt{item}); \\ & & \texttt{end}\} \end{array}$$

- Types locaux (extrémités):
  - $\texttt{Buyer} : ?(\texttt{price}); !\{\texttt{ko}; \texttt{end}\ ,\ \texttt{ok}; ?\texttt{item}; \texttt{end}\}$
  - $\texttt{Seller} : !(\texttt{price}); ?\{\texttt{ko}; \texttt{end}\ ,\ \texttt{ok}; !\texttt{item}; \texttt{end}\}$
- Processus candidats:
  - $s_{\texttt{price}}(p).(\overline{s}_{\texttt{ok}}.s_{\texttt{item}}(i) + \overline{s}_{\texttt{ko}})$: bon $\texttt{Buyer}$.
  - $s_{\texttt{price}}(p).\overline{s}_{\texttt{ko}}$: bon $\texttt{Buyer}$.
  - $\overline{s}_{\texttt{price}}\langle 100\ \texttt{Fr}\rangle.s_{\texttt{ko}}$: mauvais $\texttt{Seller}$.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Vers les MPST

- Les types extrémités sont parfaitement symétriques.
  - Ils s'assurent que les deux partis s'entendent sur les actions à effectuer.
- Les types de sessions binaires peuvent être mis en relation avec les logiques linéaires/intuitionnistes:
  - *Session Types as Intuitionistic Linear Propositions*, Caires, Pfenning, CONCUR 2010
  - Beaucoup de développement récents dans les dernières années: "Curry-Howard for sessions".
- Challenge: Les protocoles dans les réseaux impliquent souvent plus de deux participants:
  - la symétrie est cassée,
  - on introduit de l'asynchronie:
    - $\mathtt{A} \to \mathtt{B}(\mathrm{m}_1); \mathtt{A} \to \mathtt{C}(\mathrm{m}_2); \mathtt{C} \to \mathtt{B}(\mathrm{m}_3)$,
    - $\mathtt{B}$ peut recevoir $\mathrm{m}_3$ avant de recevoir $\mathrm{m}_1$.
  - *Multiparty Asynchronous Session Types*, Carbone, Honda, Yoshida, POPL 2008

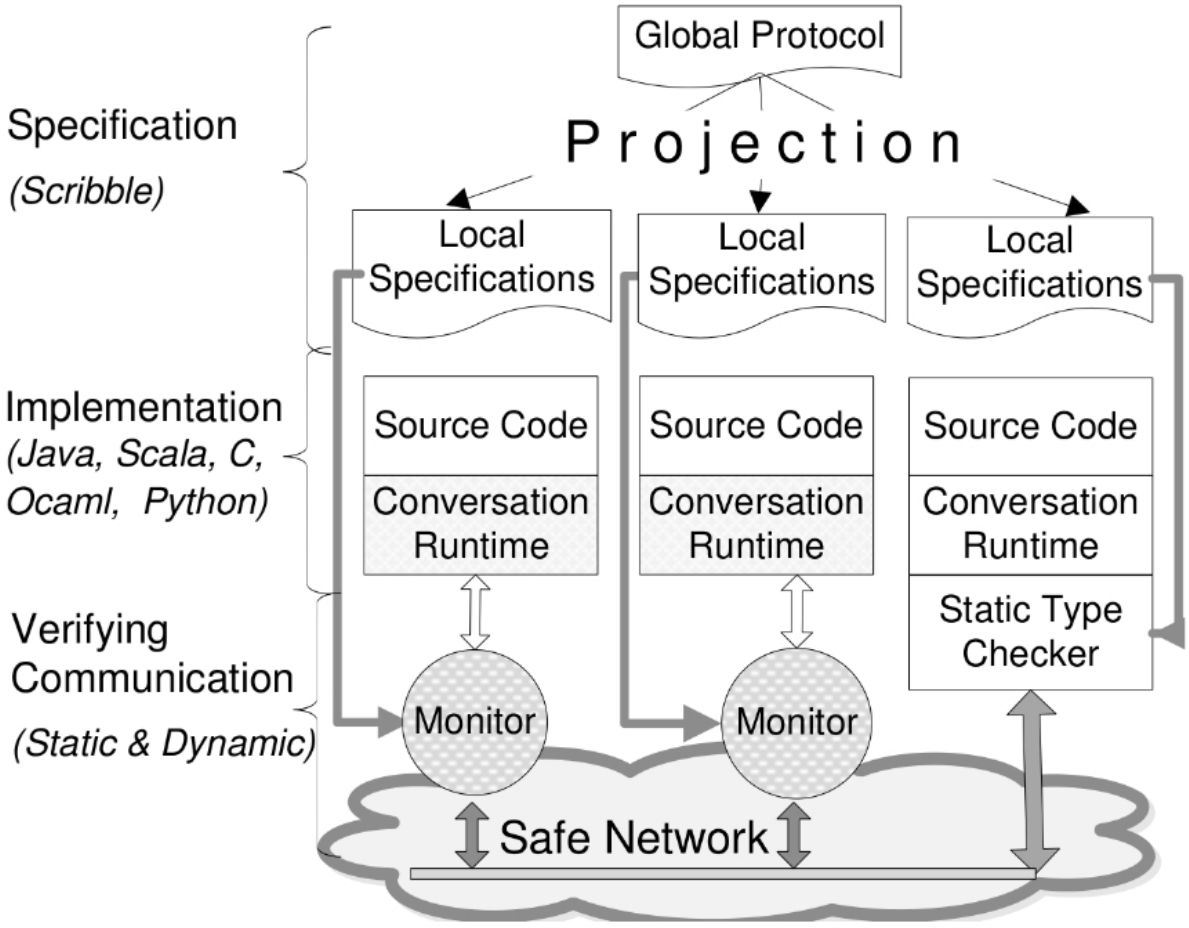
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# MPST comme Méthode de vérification

- Vérification pour des réseaux de services/applications:
  - réseaux acentralisés
    - communication par passage de message,
    - pas de contrôle global.
  - spécification: chorégraphies globales d'interactions entre plusieurs participants
    - des rôles interagissent dans une session.
    - les types globaux sont projetés en types locaux.
  - Thm: les agents suivent localement leurs types locaux
    $\Rightarrow$ le réseau suit globalement la spécification.
- Vérifier les types locaux aux extrêmités:
  - validation: analyser statiquement le programme (*typechecker*).
  - *monitoring*: analyser à la volée les messages entrants et sortants de l'application.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# MPST comme Méthode de vérification (II)



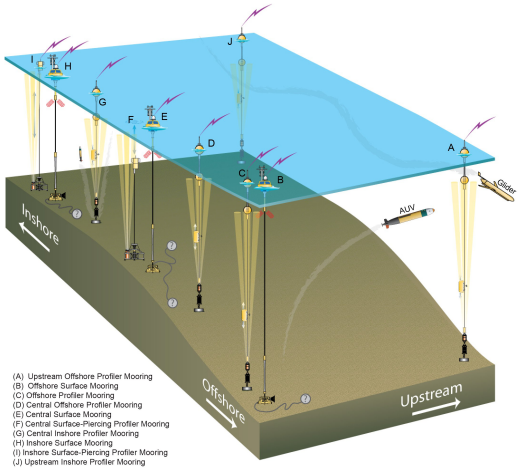

(extrait de *Monitoring Networks through Multiparty Session Types*)

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# Collaborations

- Réseaux de services, web.
  - difficulté d'obtenir des applications concrètes.
- Intéressants pour les banques (BoJ, UBS, JP Morgan)
  - propriétaires de larges infrastructures distribuées,
  - accent mis sur la correction, la sécurité.
- Principale collaboration: *Ocean Observatory Initiative*
  - Projet international d'océanographie.
  - Communication par passage de messages dans de grandes infrastructures.

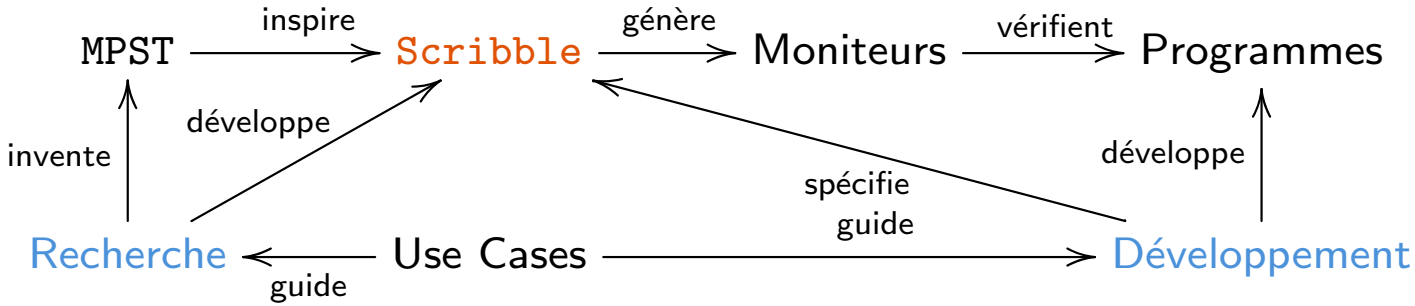
OCEAN OBSERVATORIES INITIATIVE




SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Collaboration avec OOI

- Développement d'un langage de protocoles (*Scribble*).
- Cycle de vie d'une *feature* (e.g. exceptions, modularité):
  - Echange avec les développeurs et les océanographes, validation par les chefs de projet.
  - Développement d'une théorie formelle ($\pi$-calculs) avec preuves, publication.
  - Intégration au langage *Scribble*: sémantique (mot-clefs), moniteurs, toolsuite.
  - Utilisation par les développeurs (spécification et création de moniteurs).




SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

# MPST: Théorie, types globaux

$$
\begin{array}{lcll}
G & ::= & \mathtt{r}_1 \to \mathtt{r}_2 : \Sigma_{i \in I} \{ l_i(x_i : S_i); G_i \} \mid \mathtt{end} & \text{(com,fin)} \\
& \mid & G_1 \oplus^{\mathtt{r}} G_2 \mid G_1 \parallel G_2 & \text{(choix,par)} \\
& \mid & \mu \mathtt{t}.G \mid \mathtt{t} & \text{(rec,rec-var)}
\end{array}
$$

- $\mathtt{r}$: rôles (participant d'une session).
- $li$: etiquettes des communications.
- $S_i$: type du message.
- $\oplus^{\mathtt{r}}$: choix d'un participant $\mathtt{r}$.
- Récursion par variables $\mathtt{t}$.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# MPST: Types Globaux - Exemples

$$
\begin{array}{l}
\mathtt{A} \to \mathtt{B} : \mathtt{ask}(x : \mathtt{title});\\
\mathtt{A} \to \mathtt{C} : \mathtt{ask}(x : \mathtt{title});\\
\mathtt{B} \to \mathtt{A}; \mathtt{price}(y_1 : \mathtt{nat});\\
\mathtt{C} \to \mathtt{A}; \mathtt{price}(y_2 : \mathtt{nat});\\
(\mathtt{A} \to \mathtt{B} : \mathtt{ok};\\
\ \mathtt{A} \to \mathtt{C} : \mathtt{ko};\\
\ \mathtt{B} \to \mathtt{A} : \mathtt{movie}(m_1 : .\mathtt{avi}); \mathtt{end}\\
) \oplus^{\mathtt{A}} (\\
\ \mathtt{A} \to \mathtt{C} : \mathtt{ok};\\
\ \mathtt{A} \to \mathtt{B} : \mathtt{ko};\\
\ \mathtt{C} \to \mathtt{A} : \mathtt{movie}(m_2 : .\mathtt{avi}); \mathtt{end})
\end{array}
$$

- Asynchronie: $\mathtt{A}$ peut recevoir $y_2$ de $\mathtt{C}$ avant $y_1$ de $\mathtt{B}$.
- Conditions de bonne formation:

$$
\begin{array}{l}
(\mathtt{A} \to \mathtt{B} : \mathtt{hello}; \mathtt{D} \to \mathtt{A} : \mathtt{black})\\
\oplus^{\mathtt{A}} (\mathtt{A} \to \mathtt{C} : \mathtt{hello}; \mathtt{D} \to \mathtt{A} : \mathtt{white})
\end{array}
$$

est mal-formé.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Types Locaux

Obtenus par projection automatique des types globaux:

$$\begin{array}{rcl} T & ::= & \mathtt{r}?_{i\in I}\{l_i(x_i : S_i); T_i\} \mid \mathtt{r}!_{i\in I}\{l_i(x_i : S_i); T_i\} \\ & \mid & T \parallel T \mid T \oplus T \mid \mu \mathtt{t}.T \mid \mathtt{t} \mid \mathtt{end} \end{array}$$

- $\mathtt{r}?_{i\in I}\{l_i(x_i : S_i); T_i\}$: reçoit de $\mathtt{r}$, projection d'une communication sur le receveur.
- $\mathtt{r}!_{i\in I}\{l_i(x_i : S_i); T_i\}$: envoie à $\mathtt{r}$, projection d'une communication sur l'émetteur.
- Exemple: projection sur $A$:

$T_{\mathtt{A}}$ : $\mathtt{B}!\mathtt{ask}(x : \mathtt{title}); \mathtt{C}!\mathtt{ask}(x : \mathtt{title}); \mathtt{B}?\mathtt{price}(y_1 : \mathtt{nat}); \mathtt{C}?\mathtt{price}(y_2 : \mathtt{nat});$
$(\mathtt{B}!\mathtt{ok}; \mathtt{C}!\mathtt{ko}; \mathtt{B}?\mathtt{movie}(m_1 : .\mathtt{avi}); \mathtt{end})$
$\oplus(\mathtt{C}!\mathtt{ok}; \mathtt{B}!\mathtt{ko}; \mathtt{C}?\mathtt{movie}(m_2 : .\mathtt{avi}); \mathtt{end})$

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Scribble

```
include bbs_aux;

global protocol BuyerBrokerSupplier2(role Buyer, role Broker, role Supplier) {
    rec START {
        do ForwardQuery(Buyer, Broker, Supplier); // Perform as inline protocol
        do ForwardPrice(Supplier, Broker, Buyer);
        price(int) from Broker to Buyer;
        choice at Buyer {
            do ForwardRedo(Buyer, Broker, Supplier);
            continue START;
        } or {
            accept() from Buyer to Broker;
            confirm() from Broker to Supplier;
            do ForwardDate(Supplier, Broker, Buyer);
        } or {
            reject() from Buyer to Broker;
            cancel() from Broker to Supplier;
        }
    }
}
```

- Traduction directe des types de sessions formels.
- Utilisé par les développeurs pour spécifier des protocoles.
- Le code d'une application est vérifié par rapport à la spécification en Scribbe:
  - Outils de projection pour construire les types locaux,
  - Outils de création de moniteurs, FSMs qui tournent en même temps que les applications,
  - *Type-checkers* qui valident des morceaux de code (Session C, OCaml, Go)

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Sessions en *Go*

- *Fencing off Go: Liveness and Safety for Channel-Based Programming*, *POPL'17*, par Lange, Ng, Toninho, Yoshida
- Principe:
  - appliquer la théorie des types de sessions pour typechecker des programmes Go.
  - création d'un langage formel modélisant *Go* (syntaxe et sémantique)
  - développement d'un système de types pour ce langage,
  - la décidabilité est obtenue par restriction (*fencing*) du langage (description finie des composants d'un système infiniment croissant)
- Développement d'outils
  - extracteur de comportement en *Go*,
  - analyseur en *Haskell*

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Conclusion

- Résumé:
  - difficulté et désirabilité de la vérification des systèmes concurrents
  - plusieurs méthodes: tests, typage, moniteurs, vérification de modèles, ...
  - *SPIN*: exemple de *model-checker*:
    - modélisation puis vérification
  - *session types*: exemple de discipline de types.
- TD / TME:
  - TD: modélisation en *Promela*
  - TME: prise en main de *SPIN*
    - \+ finir les interfaces distantes en *Go*.
- Séance prochaine:
  - Web: applications, messages, approches.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257