Le join-calcul,
un calcul pour la programmation répartie et mobile

Séminaire IRISA, Rennes, Avril 1999.

Cédric Fournet
Microsoft Research (Cambridge, U.K.)
et le projet PARA à l'INRIA (Rocquencourt)

La programmation des systèmes répartis

Un système réparti comporte plusieurs machines interconnectées par un réseau. Chaque machine exécute un fragment d'un même programme, de manière concertée.

Deux fragments de programme peuvent communiquer des valeurs, du code exécutable, voire une partie du programme en cours d'exécution.

La localisation est importante.

La programmation des systèmes répartis

- Parallélisme, entre machines et sur chaque machine.
- Caractère asynchrone :
  - Il n'y a pas d'opérations atomiques globales.
  - Chaque opération prend un temps très variable ;
    les débits augmentent mais la latence reste ;
    un aller-retour pour Tokyo vaut 10^8 instructions locales.
- Environnement hétérogène et incertain :
  - La structure du réseau est changeante, partiellement connue.
  - Certaines machines peuvent tomber en panne.
  - Certaines machines peuvent tricher.
  - Le déploiement et les tests sont difficiles.

La programmation des systèmes répartis

En pratique, c'est un assemblage de programmes locaux, de bibliothèques de protocoles, et de mécanismes fournis par le système.

- conventions d'adressage et de routage;
- vérification de l'intégrité des messages,
  ré-émission des messages égarés par le réseau;
- RPC, RMI;
- sessions, transactions;
- applets, objets répartis, agents mobiles.

Calculs de processus (CCS, pi-calcul)

- Simplicité formelle
- Grande expressivité
- Outils sophistiqués (équivalences, techniques de preuve, typage)
- Succès en modélisation (spécification et validation de systèmes)

Quelques langages s'inspirent de ces calculs (PICT, Oz).

Ces calculs ne sont pas directement destinés à la programmation.

Leurs abstractions ne sont pas adaptées à la programmation répartie ; leur implémentation en toute généralité est délicate et inefficace.

Le join-calcul

1 Un calcul asynchrone.

2 Un modèle opérationnel clair:

Chaque étape du calcul s'implémente par (au plus) un message asynchrone d'une machine à une autre.

3 Une expressivité suffisante pour programmer confortablement; (calcul-noyau d'un langage de programmation de haut niveau.)

4 Un cadre formel pour étudier les propriétés de ces programmes.

A Localisation explicite, migration, pannes.

B Implémentation formellement sécurisée.

Le join-calcul

Les seules valeurs sont des noms $x, y, \mathit{imprimer}, \ldots$.

Un processus peut créer des noms, envoyer des messages contenant des noms, recevoir des messages sur les noms qu'il a créés.

Un message $x\langle y_1, \ldots, y_n \rangle$ utilise les noms de deux manières :
- x est l'adresse du message,
- $y_1, \ldots, y_n$ est son contenu.

Par exemple, nous modélisons l'interface d'un serveur d'impression par deux noms, *imprimer* pour les requêtes d'impression, et *accepter* pour les imprimantes disponibles.

Plusieurs messages peuvent être assemblés par composition parallèle:

$$\mathit{imprimer}\langle 1 \rangle \mid \mathit{imprimer}\langle 2 \rangle \mid \mathit{accepter}\langle \mathit{laser} \rangle$$

Le join-calcul

Une règle de réaction $J \triangleright P$ consomme un ensemble de messages de la forme décrite dans le filtre J, et déclenche l'exécution d'une copie du processus P.

Par exemple, la règle

$$D \stackrel{\text{def}}{=} \mathit{accepter}\langle \mathit{imprimante} \rangle | \mathit{imprimer}\langle \mathit{fichier} \rangle \triangleright \mathit{imprimante}\langle \mathit{fichier} \rangle$$

décrit le comportement du serveur en définissant comment les messages envoyés sur les noms *imprimer* et *accepter* sont traités.

On peut regrouper la définition et l'état du serveur en un seul processus.

$$\begin{aligned} P &\stackrel{\text{def}}{=} \mathsf{def}\ D\ \mathsf{in}\ \mathit{accepter}\langle \mathit{laser} \rangle \mid \mathit{imprimer}\langle 1 \rangle \mid \mathit{imprimer}\langle 2 \rangle \\ &\rightarrow \mathsf{def}\ D\ \mathsf{in}\ \mathit{laser}\langle 1 \rangle \mid \mathit{imprimer}\langle 2 \rangle \end{aligned}$$

Syntaxe du join-calcul

P	$::=$		processus
		$x\langle v_1, \ldots, v_n \rangle$	message asynchrone
	$\mid$	$\mathsf{def}\ D\ \mathsf{in}\ P$	définition locale
	$\mid$	$P \mid P'$	exécution parallèle
D	$::=$		définitions
		$J \triangleright P$	règle de réaction
	$\mid$	$D \wedge D'$	composition de définitions
J	$::=$		filtre
		$x\langle y_1, \ldots, y_n \rangle$	message requis
	$\mid$	$J \mid J'$	synchronisation de messages

Quelques exemples

Le relais n'effectue aucun calcul; il fait simplement suivre les messages d'un canal à un autre:

$$\mathsf{def}\ x\langle u\rangle \triangleright y\langle u\rangle\ \mathsf{in}\ Q$$

Une continuation permet de nommer un processus et de déclencher ce processus par un message:

$$\mathsf{def}\ \kappa\langle u, v\rangle \triangleright P\ \mathsf{in}\ \kappa\langle 1, 2\rangle \mid plug\langle\kappa\rangle \mid \ldots$$

Implémentation répartie?

La machine abstraite chimique

Un programme s'exprime par des transformations de multi-ensembles.
Par analogie avec la chimie des solutions (Banâtre et Le Metayer),
- un processus est une molécule;
- une réaction consomme et produit des molécules.

Pour un ensemble de réactions donné,
- on répartit ces réactions entre les machines disponibles;
- on organise la circulation des molécules.

Il y a deux groupes de règles (Berry et Boudol) :
L'équivalence structurelle (notée $\rightleftharpoons$) décrit comment réarranger les processus. Ces étapes sont réversibles.
La réduction chimique (notée $\rightarrow$) décrit l'interaction locale des processus.

Une machine chimique répartie

Pour garantir l'implémentation répartie du join-calcul :

1. On ne fait aucune hypothèse sur la répartition des processus;
   Il y a de nombreuses règles de réaction simples et indépendantes.

2. Toutes les règles qui peuvent consommer un message donné
   se trouvent sur un même site.

3. Au cours du calcul, de nouvelles molécules apparaissent avec leurs
   règles de réactions.

La machine chimique réflexive

Notée $\mathcal{D} \vdash \mathcal{P}$, une solution chimique représente l'état du calcul par deux multi-ensembles - $\mathcal{P}$ contient les processus en cours d'exécution - $\mathcal{D}$ contient les règles de réaction actives.

La solution évolue selon les règles suivantes :

$$
\begin{array}{lrcl}
\textsc{Str-join} & \vdash P_1 \mid P_2 & \rightleftharpoons & \vdash P_1, P_2 \\
\textsc{Str-and} & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash \\
\textsc{Str-def} & \vdash \mathsf{def}\ D\ \mathsf{in}\ P & \rightleftharpoons & D\sigma_{\mathrm{dv}} \vdash P\sigma_{\mathrm{dv}}
\end{array}
$$

$$
\textsc{Red} \qquad J \triangleright P \vdash J\sigma_{\mathrm{rv}} \longrightarrow J \triangleright P \vdash P\sigma_{\mathrm{rv}}
$$

Le contexte est implicite (localité chimique).

Exemple : La réplication

On peut encoder la réplication en démarrant de nouvelles copies d'un même processus P, indéfiniment :

$$\mathsf{repl}\ P \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ \kappa\langle\rangle \triangleright P \mid \kappa\langle\rangle\ \mathsf{in}\ \kappa\langle\rangle$$

Chimiquement (en supposant que κ n'apparaît pas dans P)

$$\begin{array}{lll}
\vdash \mathsf{def}\ \kappa\langle\rangle \triangleright P \mid \kappa\langle\rangle\ \mathsf{in}\ \kappa\langle\rangle & \stackrel{\text{Str-def}}{\rightharpoonup} & \kappa\langle\rangle \triangleright P \mid \kappa\langle\rangle \vdash \kappa\langle\rangle \\
& \stackrel{\text{Red}}{\rightarrow} & \kappa\langle\rangle \triangleright P \mid \kappa\langle\rangle \vdash P \mid \kappa\langle\rangle \\
& \stackrel{\text{Str-join}}{\rightharpoonup} & \kappa\langle\rangle \triangleright P \mid \kappa\langle\rangle \vdash P,\ \ \kappa\langle\rangle \\
& \stackrel{\text{Str-def}}{\rightharpoondown} & \vdash P,\ \ \mathsf{def}\ \kappa\langle\rangle \triangleright P \mid \kappa\langle\rangle\ \mathsf{in}\ \kappa\langle\rangle \\
& \stackrel{\text{Str-join}}{\rightharpoondown} & \vdash P \mid \mathsf{repl}\ P
\end{array}$$

Vers un langage de programmation

Vers un langage de programmation

Le join-calcul est assez proche des langages fonctionnels de haut niveau.

1. Le join-calcul peut être équipé d'un système de types polymorphes et d'un système de modules à la ML.

2. Le contrôle séquentiel s'exprime par passage de continuation. En supprimant la composition parallèle, on obtient un petit calcul fonctionnel.

   join-calcul + appel par valeur = ML + parallélisme (fork/join)

3. On peut également encoder les valeurs mutables et les objets.

Notre implémentation utilise Objective Caml comme langage-support; cette proximité permet de compiler des programmes hybrides avec des modules de chaque langage, à partir de leurs interfaces typées.

Typage

Le typage permet de détecter les erreurs d'arité, et la mauvaise utilisation des primitives; la réduction préserve le typage.

Les types et schémas de types pour chaque variable sont

$$\begin{aligned} \tau &::= b \mid \alpha \mid \langle \tau_1, \ldots, \tau_p \rangle \\ \sigma &::= \tau \mid \forall\, \alpha.\, \sigma \end{aligned}$$

Par exemple, si l'on a *imprimer* : $\langle$`Entier`$\rangle$, alors

$$\mathit{imprimer\_paire}\langle x, y \rangle \triangleright \mathit{imprimer}\langle x \rangle \mid \mathit{imprimer}\langle y \rangle$$

impose le typage *imprimer_paire* : $\langle$`Entier`, `Entier`$\rangle$.

L'inférence des types est possible parce que toutes les occurences contravariantes sont présentes dans la définition.

Typage; polymorphisme paramétrique

La définition

$$\mathit{imprimer\_ici}\langle \mathit{imprimante}, \mathit{fichier}\rangle \triangleright \mathit{imprimante}\langle \mathit{fichier}\rangle$$

autorise le typage $\mathit{imprimer\_ici} : \forall\alpha.\langle\langle\alpha\rangle, \alpha\rangle$.

La définition du serveur d'impression

$$\mathit{accepter}\langle \mathit{imprimante}\rangle | \mathit{imprimer}\langle \mathit{fichier}\rangle \triangleright \mathit{imprimante}\langle \mathit{fichier}\rangle$$

impose le typage non généralisable $\mathit{accepter} : \langle\alpha\rangle$ et $\mathit{imprimer} : \alpha$.

On peut généraliser le type d'un nom défini lorsque la variable n'apparaît pas dans le type d'un nom co-défini.

Fonctions et contrôle séquentiel

Le contrôle séquentiel reste très utile pour programmer.

En utilisant des continuations, on peut encoder plusieurs stratégies d'évaluation dans le join-calcul.

Pour imprimer une paire d'entiers dans l'ordre, on peut définir :

$$\begin{aligned} \mathit{imprimer\_paire}\langle x, y, \kappa\rangle \quad \triangleright \quad & \mathsf{def}\ \kappa_y\langle\rangle \triangleright \kappa\langle\rangle\ \mathsf{in} \\ & \mathsf{def}\ \kappa_x\langle\rangle \triangleright \mathit{imprimer}\langle y, \kappa_y\rangle\ \mathsf{in} \\ & \mathit{imprimer}\langle x, \kappa_x\rangle \end{aligned}$$

Il est plus facile d'écrire :

$$\mathrm{imprimer\_paire}(x, y) \quad \triangleright \quad \mathrm{imprimer}(x); \mathrm{imprimer}(y); \mathsf{reply}$$

Nous intégrons l'appel par valeur dans notre langage.

Fonctions et contrôle séquentiel

Plus généralement, on peut traduire la définition d'un échange de valeurs et l'appel d'une fonction qui retourne une valeur :

$$\mathsf{let}\ v = \mathrm{a}(u)\ \mathsf{in}\ P \quad \stackrel{\text{def}}{=} \quad \mathsf{def}\ \kappa\langle v\rangle \triangleright P\ \mathsf{in}\ a\langle u, \kappa\rangle$$

$$\mathrm{a}(u) \mid \mathrm{b}(v)\ \triangleright\ \mathsf{reply}\ v\ \mathsf{to}\ \mathrm{a} \mid \mathsf{reply}\ u\ \mathsf{to}\ \mathrm{b} \quad \stackrel{\text{def}}{=} \quad a\langle u, \kappa_a\rangle \mid b\langle v, \kappa_b\rangle \triangleright \kappa_a\langle v\rangle \mid \kappa_b\langle u\rangle$$

L'encodage du contrôle séquentiel s'étend aux types avec

$$\langle \tau_1, \ldots, \tau_n\rangle \to \langle \sigma_1, \ldots, \sigma_m\rangle \quad \stackrel{\text{def}}{=} \quad \langle \tau_1, \ldots, \tau_n, \langle \sigma_1, \ldots, \sigma_m\rangle\rangle$$

Programmation impérative

La programmation impérative s'exprime en représentant l'état par un ou plusieurs messages.

La règle suivante alloue une cellule mutable.

$$\mathrm{cellule}(v_0) \triangleright \left(\begin{array}{lrl} \mathsf{def} & \mathrm{lire}() \mid s\langle v\rangle \triangleright & \mathsf{reply}\ v\ \mathsf{to\ lire} \mid s\langle v\rangle \\ & \wedge \quad \mathrm{écrire}(u) \mid s\langle v\rangle \triangleright & \mathsf{reply\ to\ écrire} \mid s\langle u\rangle \\ \mathsf{in} & \mathsf{reply\ lire, écrire\ to\ cellule} \mid s\langle v_0\rangle & \end{array} \right)$$

L'allocateur a un type polymorphe

$$\mathrm{cellule} \quad : \quad \forall\, \alpha.\, \langle \alpha \rangle \to \langle \langle\rangle \to \langle \alpha \rangle, \langle \alpha \rangle \to \langle\rangle \rangle$$

En revanche, chaque cellule allouée est monomorphe, du type de l'argument d'initialisation v_0.

Propriétés formelles des processus

Quelle équivalence pour le join-calcul?

Pour formaliser les propriétés de programmes écrits en join-calcul,
il faut des équivalences faciles à interpréter (égalités, inégalités).

Pour prouver ces propriétés, il faut des technique de preuve efficaces.

Le join-calcul bénéficie du cadre théorique des calcul de processus,
en particulier des équivalences étudiées dans le pi-calcul.
Néanmoins, la programmation répartie pose des problèmes nouveaux:

Processus asynchrones les réductions internes sont invisibles.

Messages asynchrones l'émetteur ne détecte pas la réception.

Récepteurs statique on ne peut pas redéfinir un nom.

Pas de comparaison de noms on peut juste envoyer des messages.

Observations élémentaires

Seuls les messages sur les noms libres sont observables.

- P émet un message sur x, noté $P \downarrow_x$
- P peut émettre un message sur x, noté $P \Downarrow_x$,
  défini par $\Downarrow_x = \rightarrow^* \downarrow_x$.
- P pourra toujours émettre un message sur x, noté $\Downarrow_{\Box x}$,
  défini par $P \rightarrow^* P'$ implique $P' \Downarrow_x$.

Les réductions internes et la divergence ne sont pas observables avec les prédicats $\Downarrow_x$ et $\Downarrow_{\Box x}$.

Equivalences de test

Les tests $\Downarrow_x$, $\Downarrow_{\Box x}$ révèlent le comportement superficiel des processus.

On obtient des équivalences faciles à interpréter en considérant
des congruences qui respectent tous les tests.

Si l'on s'intéresse aux messages possibles:

P et Q sont équivalents lorsque, pour tout contexte $C[\,\cdot\,]$,
$C[P] \Downarrow_x$ si et seulement si $C[Q] \Downarrow_x$
(noté $P \simeq_{may} Q$).

Bisimulations

En demandant une correspondance entre les états internes de processus équivalents, on obtient des équivalences plus fines à base de bisimulations.

Par exemple, on peut définir la "bisimulation observationnelle" (notée $\approx$) comme la plus grande relation symétrique telle que, si $P \approx Q$,

1. $C[P] \approx C[Q]$ pour tout contexte $C[\,\cdot\,]$
2. Si $P \Downarrow_x$ alors $Q \Downarrow_x$ pour tout nom x
3. Si $P \to P'$ alors il existe Q' tel que $Q \to^* Q'$ et $P' \approx Q'$.

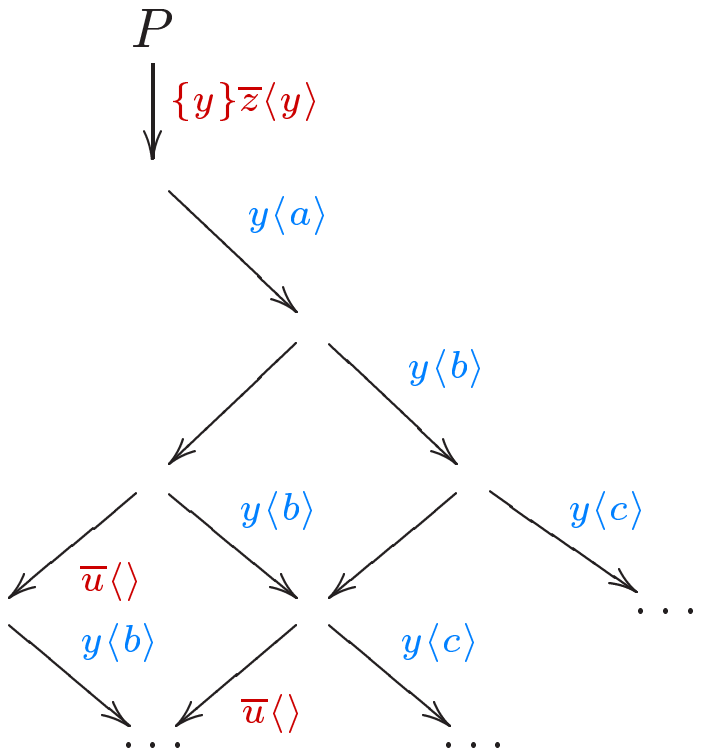
On retrouve $\simeq_{may}$ en demandant seulement 1. et 2.

Les preuves sont beaucoup plus faciles.

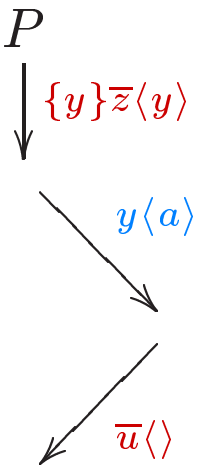
Un modèle plus explicite du join-calcul

Chaque interaction élémentaire entre le processus observé et le contexte peut être remplacée par une transition étiquetée.

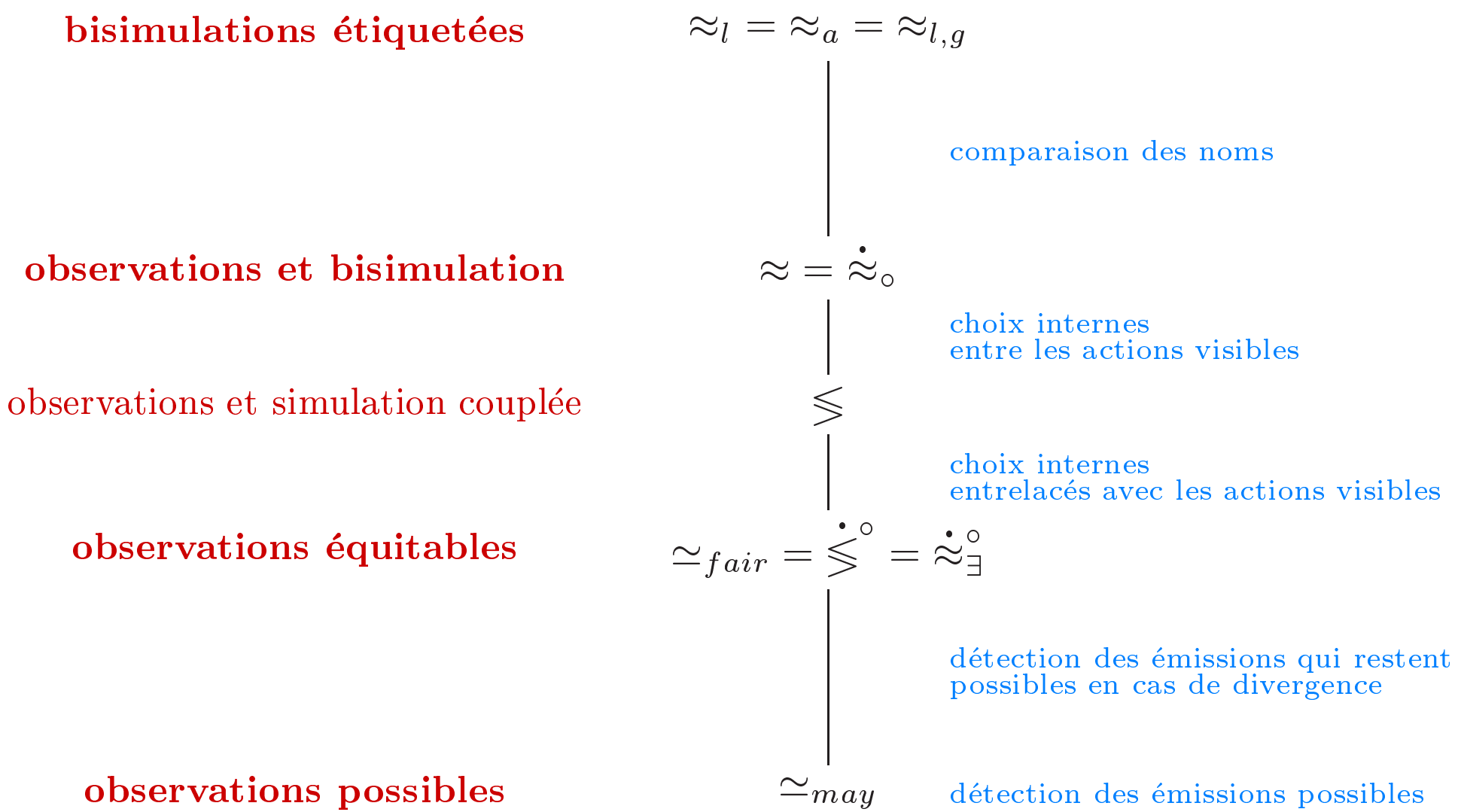
Par exemple, le processus $P \stackrel{\text{def}}{=} \mathsf{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright u\langle\rangle\ \mathsf{in}\ x\langle u\rangle \mid z\langle y\rangle$
peut être modélisé par (extrusions, intrusions, et transitions internes) :



ou simplement



Une hiérarchie d'équivalences de plus en plus fines.



Quelques équations sur les exemples

La présence de relais est indécelable:

$$\mathsf{def}\ x\langle u\rangle \triangleright y\langle u\rangle\ \mathsf{in}\ Q \quad \approx \quad Q\{^{y}/_{x}\}$$

La communication sur un nom simplement défini n'affecte pas l'équivalence (continuations, β-réduction):

$$\mathsf{def}\ \kappa\langle x, y\rangle \triangleright P\ \mathsf{in}\ Q \mid \kappa\langle u, v\rangle \quad \approx \quad \mathsf{def}\ \kappa\langle x, y\rangle \triangleright P\ \mathsf{in}\ Q \mid P\{^{u}/_{x}, ^{v}/_{y}\}$$

La réplication se comporte exactement comme ses dépliages:

$$\begin{aligned} \mathsf{repl}\ Q &\approx Q \mid \mathsf{repl}\ Q \\ &\approx Q \mid Q \mid \ldots \mid Q \mid \mathsf{repl}\ Q \end{aligned}$$

Complétude (1) : Un join-calcul minimaliste

P	::=		processus élémentaire
		$x\langle u\rangle$	message transmettant un seul nom
	$\mid$	$P_1 \mid P_2$	composition parallèle
	$\mid$	def $x\langle u\rangle \mid y\langle v\rangle \triangleright P_1$ in P_2	définition de deux noms par une règle

Il y a une traduction $[\![\cdot]\!]$ du join-calcul vers ce noyau;
cette traduction esst complète pour la bisimulation observationnelle :

$$P \approx Q \quad \text{si et seulement si} \quad [\![P]\!] \approx [\![Q]\!]$$

1. la récursion s'encode par l'ajout d'un message par filtre;
2. les définitions complexes et les filtres de synchronisation à $m \geq 3$ messages sont compilés (automate sur les filtres).
3. un message n-uple devient un dialogue à $n + 3$ messages simples.

Complétude (2) : Une comparaison avec le pi-calcul

Le join-calcul ressemble formellement au pi-calcul.

Les deux calculs utilisent les canaux de manières différentes, mais

1. Il y a des traductions complètes croisées.
2. Il y a des calculs typés intermédiaires (pi-1, pi-A, ...)

Quelques différences : Le join-calcul impose la définition statique de tous les récepteurs, combine tous les lieurs du join-calcul dans la définition, interdit la communication sur les noms libres, et autorise la réception de plusieurs messages à la fois.

Complétude (2) : Une comparaison avec le pi-calcul

Du join-calcul au pi-calcul, chaque définition binaire s'exprime en combinant les trois lieurs du pi-calcul.

$$[\![x\langle u\rangle]\!] \stackrel{\text{def}}{=} \overline{x}\langle u\rangle$$

$$[\![\mathsf{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright P\ \mathsf{in}\ Q]\!] \stackrel{\text{def}}{=} \nu x.\nu y.(!x\langle u\rangle.y\langle v\rangle.[\![P]\!] \mid [\![Q]\!])$$

Inversement, chaque canal du pi-calcul x permet à la fois l'envoi et la réception de messages. Son encodage en join-calcul utilise deux noms x_e et x_r définis par la règle

$$x_e\langle y_e, y_r\rangle \mid x_r\langle \kappa\rangle \quad \triangleright \quad \kappa\langle y_e, y_r\rangle$$

Localisation explicite

Localisation explicite (1) : Routage

L'état du calcul est réparti entre plusieurs solutions chimiques locales.

$$\mathcal{D} \vdash \mathcal{P} \qquad \text{devient} \qquad \mathcal{D}_1 \overset{1}{\vdash} \mathcal{P}_1 \;\|\; \mathcal{D}_2 \overset{2}{\vdash} \mathcal{P}_2 \;\|\; \cdots$$

Chaque nom est défini dans une seule solution chimique.
Chaque solution évolue comme avant (mêmes règles locales).

Il y a une règle de communication supplémentaire entre solutions:

$$\begin{array}{lll} \textsc{Comm} & \overset{1}{\vdash} x\langle a,b\rangle & \| \;\; x\langle u,v\rangle \triangleright P \overset{2}{\vdash} \\ \longrightarrow & \overset{1}{\vdash} & \| \;\; x\langle u,v\rangle \triangleright P \overset{2}{\vdash} x\langle a,b\rangle \end{array}$$

Le routage (Comm) est préalable au calcul local (Red);
il s'implémente en passant $2.x$ plutôt que x dans la règle (Comm).

Le serveur d'impression, plus en détail

Nous avons la série de réductions suivante :

$$
\begin{array}{lllll}
 & D \overset{s}{\vdash} & \| \overset{p}{\vdash} \mathsf{def}\ laser\langle f\rangle \triangleright P\ \mathsf{in}\ accepter\langle laser\rangle & \| \overset{u}{\vdash} imprimer\langle 1\rangle \\
\text{Str-Def} & D \overset{s}{\vdash} & \| laser\langle f\rangle \triangleright P \overset{p}{\vdash} accepter\langle laser\rangle & \| \overset{u}{\vdash} imprimer\langle 1\rangle \\
\underset{\rightarrow}{\text{Comm}} & D \overset{s}{\vdash} imprimer\langle 1\rangle & \| laser\langle f\rangle \triangleright P \overset{p}{\vdash} accepter\langle laser\rangle & \| \overset{u}{\vdash} \\
\underset{\rightarrow}{\text{Comm}} & D \overset{s}{\vdash} accepter\langle laser\rangle, imprimer\langle 1\rangle & \| laser\langle f\rangle \triangleright P \overset{p}{\vdash} & \| \overset{u}{\vdash} \\
\underset{\rightarrow}{\text{Red}} & D \overset{s}{\vdash} laser\langle 1\rangle & \| laser\langle f\rangle \triangleright P \overset{p}{\vdash} & \| \overset{u}{\vdash} \\
\underset{\rightarrow}{\text{Comm}} & D \overset{s}{\vdash} & \| laser\langle f\rangle \triangleright P \overset{p}{\vdash} laser\langle 1\rangle & \| \overset{u}{\vdash}
\end{array}
$$

(avec $D \overset{\text{def}}{=} accepter\langle imprimante\rangle \mid imprimer\langle fichier\rangle \triangleright imprimante\langle fichier\rangle$)

Localisation explicite (2) : Hiérarchie

Les emplacements sont nommés, et organisés hiérarchiquement;
ce sont des unités pour la migration et pour les pannes.

Les noms d'emplacement sont des valeurs comme les autres:

- On peut créer localement de nouveaux emplacements (Str-Def)
- On peut communiquer leur nom dans des messages
- On peut utiliser ces noms pour réorganiser la hiérarchie, pour détecter les pannes.

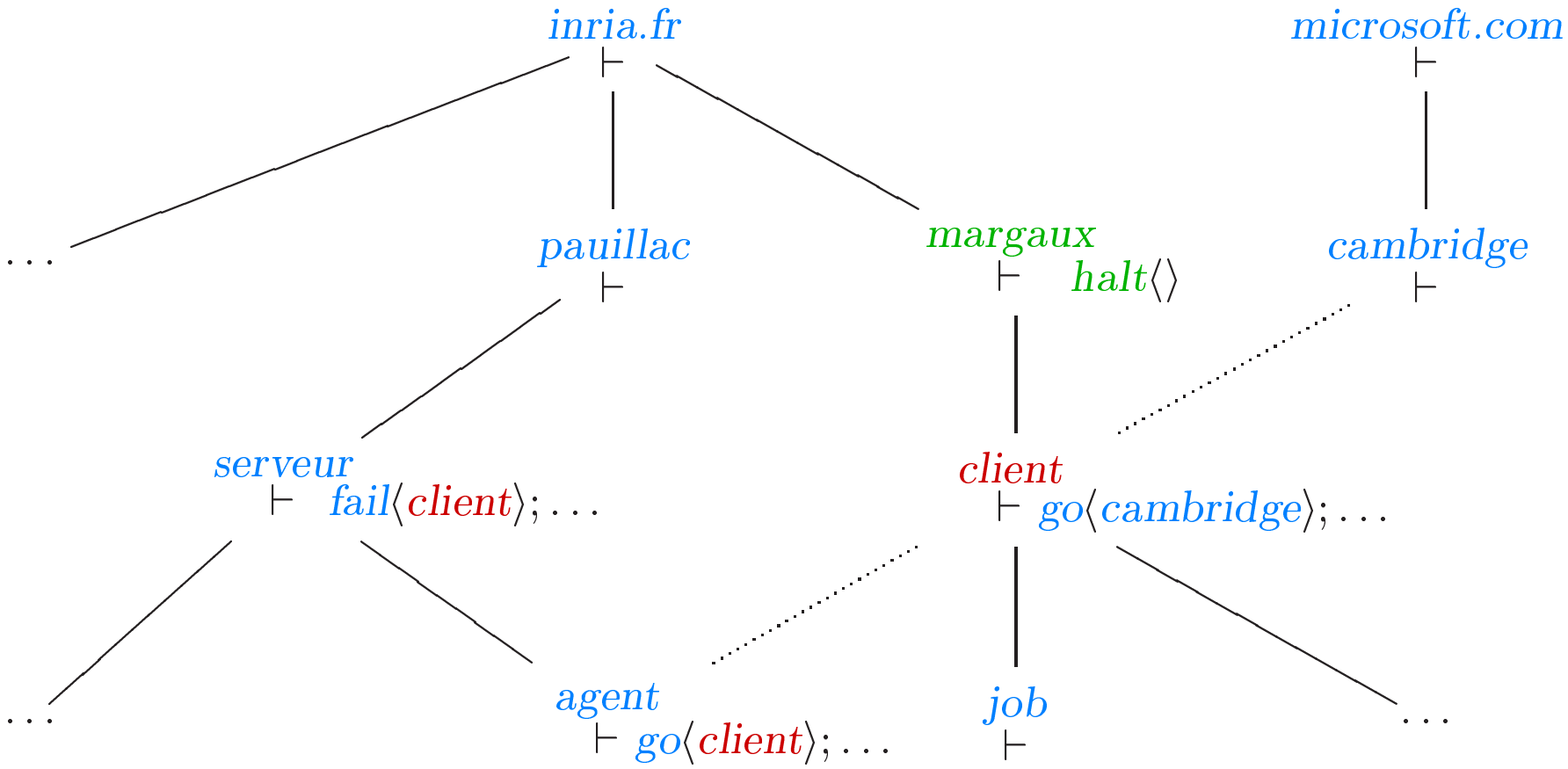
Localisation explicite : Syntaxe étendue

P	$::=$		processus
		$\ldots$	(comme avant)
	$\mid$	$go\langle a, \kappa\rangle$	migration de l'emplacement courant
	$\mid$	$halt\langle\rangle$	arrêt de l'emplacement
	$\mid$	$fail\langle a, \kappa\rangle$	détection de l'arrêt d'un emplacement
D	$::=$		définitions
		$\ldots$	(comme avant)
	$\mid$	$a\,[D : P]$	sous-emplacement
	$\mid$	$\Omega a\,[D : P]$	sous-emplacement arrêté

Une nouvelle règle structurelle permet de changer de représentation:

- solutions chimiques en parallèle pour la communication
- sous-emplacements imbriqués dans une définition pour la migration.

Localisation hiérarchique



Transparence

En l'absence de pannes ($halt\langle\rangle$) et de circularités dans la migration, la répartition du calcul est programmable mais transparente.

- L'envoi de messages et la migration ne dépendent pas de la localisation initiale.
- On peut fusionner les solutions locales et effacer les migrations, sans affecter l'équivalence observationnelle.

Sans l'hypothèse asynchrone, les performances sont très différentes!

Transparence ou opacité?

Récemment, plusieurs calculs de processus utilisent la localité pour contrôler le calcul plus directement, en bloquant la communication en cas de mauvaise localisation (Ambients, Seals, variantes du pi-calcul).

- description des mécanismes d' implémentation;
- utilisation d'un modèle local plus riche (communications synchrones);
- opacité des firewalls, des réseaux;
- environnement dynamique à l'exécution.

Il y a un prix à payer en complexité (typage, encodages explicites)
Comment maintenir alors un langage de haut niveau utilisable?

Comment modéliser les pannes?

1. Pas de mécanisme caché pour masquer les pannes (irréaliste).

2. Un emplacement s'arrête en exécutant le processus $halt\langle\rangle$.
   Par la suite, il ne participe plus à aucune réduction.

   L'arrêt potentiel de certains emplacements est plus simple à analyser
   qu'une multitude d'erreurs de plus bas niveau
   (pertes de messages, comportements incohérents.)

3. Le "time-out" est trivial dans un calcul asynchrone,
   mais souvent suffisant pour le traitement des pannes.

4. L'arrêt d'un emplacement est détectable par la garde $\mathrm{fail}(a);$.

   Cette primitive de détection est plus délicate à implémenter,
   mais garantit que certaines choses n'arriveront plus.

Implémentation Sécurisée
(avec Georges Gonthier et Martin Abadi)

Un exemple de programme réparti

$$
\begin{array}{ll}
\mathsf{def} & \mathit{bulletin}\langle a, n\rangle \,|\, \mathit{ouvert}\langle\rangle \quad \triangleright\ \mathit{gagnant}\langle n\rangle \,|\, a\langle \texttt{prix}\rangle \\
\wedge & \mathit{bulletin}\langle a, n\rangle \,|\, \mathit{gagnant}\langle m\rangle \ \triangleright\ \mathit{gagnant}\langle m\rangle \,|\, a\langle m + \texttt{a gagné}\rangle \ \mathsf{in} \\
\\
& \mathit{ouvert}\langle\rangle \,|\, \mathit{participant}_1\langle \mathit{bulletin}\rangle \,|\, \ldots \,|\, \mathit{participant}_k\langle \mathit{bulletin}\rangle
\end{array}
$$

L'organisateur du concours crée un canal *bulletin* et le diffuse.

Le premier participant qui envoie son nom gagne le concours.

Sécurité implicite

De nombreuses propriétés de sécurité sont exprimables par des équivalences, et faciles à vérifier.

Contrôle d'accès : Chaque participant doit avoir reçu le nom *bulletin*.

Intégrité : Tout les participants reçoivent le nom du gagnant.

Anonymat: Le nom des perdants disparaît.

Raffinements: Il est possible d'ajouter des relais, des proxies.

Dans l'énoncé de ces propriétés, l'attaquant, c'est le contexte.

Transparence?

Rennes	La Manche	*Cambridge*
$bulletin\langle x, \rangle$		$bulletin\langle a, n\rangle \triangleright \ldots$

Attaquant

Si le join-calcul garantit la transparence, toute propriété de sécurité reste valide. C'est utile, mais peu réaliste dans une implémentation répartie.

Il faut étudier la sécurité de l'implémentation, dans un cadre où les processus ne possèdent plus de canaux privés entre machines, et où les contextes peuvent exprimer toute attaque asynchrone.

Le s-join-calcul

Nous ajoutons la cryptographie à clé publique au join-calcul, sous la forme de nouvelles valeurs (les clés, les messages encryptés) et de nouveaux processus (décryption).

v	$::=$		valeurs
		$\ldots$	(comme avant)
	$\mid$	$\{v_1, \ldots, v_n\}_{v'}$	valeur encryptée
P	$::=$		processus
		$\ldots$	(comme avant)
	$\mid$	decrypt v using v' to $x_1, \ldots, x_n$ in P else P'	décryption
D	$::=$		définitions
		$\ldots$	(comme avant)
	$\mid$	keys x^+, x^-	paire de clés

Le s-join-calcul

La décryption réussit lorsque l'on utilise la bonne clé

$$\begin{array}{rl} & \mathsf{def\ keys}\ x^+, x^-\ \mathsf{in}\ \ldots \mid \mathsf{decrypt}\ \{v\}_{x^+}\ \mathsf{using}\ x^-\ \mathsf{to}\ u\ \mathsf{in}\ P\ \mathsf{else}\ P' \\ \longrightarrow & \mathsf{def\ keys}\ x^+, x^-\ \mathsf{in}\ \ldots \mid P\{^v/_u\} \end{array}$$

La décryption échoue autrement (checksum):

$$\begin{array}{rl} & \mathsf{def\ keys}\ x^+, x^-\ \mathsf{in}\ \ldots \mid \mathsf{decrypt}\ w\ \mathsf{using}\ x^-\ \mathsf{to}\ u\ \mathsf{in}\ P\ \mathsf{else}\ P' \\ \longrightarrow & \mathsf{def\ keys}\ x^+, x^-\ \mathsf{in}\ \ldots \mid P' \end{array}$$

Le contexte ne peut pas faire grand'chose sans la clé de décryption.

Implémentation sécurisée

Toute communication a lieu par l'intermédiaire d'un canal public.
L'attaquant peut intercepter chaque message, émettre tout message, et programmer dans le s-join-calcul.

Rennes	La Manche	*Cambridge*
$emit\langle\{a^+, n\}_{bulletin^+}\rangle$		$recv\langle m\rangle \triangleright$ decrypt m using $bulletin^-$ to a^+, n in $\ldots$

Attaquant($emit, recv$)

Cette implémentation est percée.
Par exemple, le nom n n'est pas secret, même si $bulletin^-$ reste secret.

Implémentation sécurisée

1. Un contexte $\mathcal{E}[\,\cdot\,]$ définit le réseau public $emit, recv$.
2. Nous programmons dans le s-join-calcul des protocoles de communication pour transmettre des messages sur ce réseau public sans divulguer plus d'information que dans le join-calcul.
3. Chaque processus P du join-calcul peut se traduire de deux manières
   - $[\![P]\!]$ remplace chaque communication par une exécution du protocole sur le réseau public. Ce codage est compositionnel.
   - $\mathcal{F}[P]$ filtre récursivement tous les messages qui arrivent ou qui partent de P, en exécutant le protocole pour ces messages seulement.

Implémentation sécurisée

Pour tout processus P du join-calcul, les deux traductions sont interchangeables, même en présence d'un attaquant A:

$$\mathcal{E}[A \mid [\![P]\!]] \quad \approx \quad \mathcal{E}[A \mid \mathcal{F}[P]]$$

Ces traductions sont complètes: pour tout P et Q dans le join-calcul,

$$P \approx Q \quad \text{si et seulement si, pour tout } A, \quad \mathcal{E}[A \mid [\![P]\!]] \approx \mathcal{E}[A \mid [\![Q]\!]]$$

En particulier, s'il y a une attaque dans l'implémentation, il y a une attaque dans le join-calcul aussi.

En combinant ces deux résultats, la répartition des processus est transparente, même en présence d'un attaquant, e.g.

$$\mathcal{E}[A \mid \mathcal{F}[P \mid Q]] \approx \mathcal{E}[A \mid [\![P \mid Q]\!]] \approx \mathcal{E}[A \mid [\![P]\!] \mid [\![Q]\!]] \approx \mathcal{E}[A \mid \mathcal{F}[P] \mid \mathcal{F}[Q]]$$

Un calcul pour la programmation répartie et mobile

Implémentations réparties, programmes, et papiers sont disponibles en

`http://join.inria.fr/`