Vérification de l'équivalence du π-calcul dans HOL

Otmane Ait-Mohamed

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Vérification de l'équivalence du π-calcul dans HOL

Otmane AIT-MOHAMED

N° 2412

Novembre 1994

PROGRAMME 2

ISSN 0249-6399

Vérification de l'équivalence du π-calcul dans HOL

Otmane AIT-MOHAMED*

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Eureca

Rapport de recherche n° 2412 — Novembre 1994 — 32 pages

Résumé : Dans ce rapport, on décrit une représentation purement *définitionnelle* du π-calcul dans le prouveur de théorèmes HOL. Les lois algébriques du π-calcul sont montrées comme étant des théorèmes dans la loqique d'ordre supérieur. Ainsi, on obtient un outil correte dans lequel on peut raisonner sur des applications utilisant le π-calcul comme outil de spécification. Comme application, on montre par induction la correction de la spécification de l'addition naturelle en π-calcul.

Mots-clé : π-calcul, HOL

(Abstract: pto)

*. amohamed@loria.fr

Unité de recherche INRIA Lorraine
Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : (33) 83 59 30 30 – Télécopie : (33) 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : (33) 87 20 35 00 – Télécopie : (33) 87 76 39 77

Verification of π-calculus equivalence in HOL

Abstract: This paper describes a *definitionnelle* presentation in higher order logic of the theory for Milner's π-calculus. The algebraic laws for the π-calculus are proved as theorems in the HOL logic and a set of proof tools is provided. So our system is in fact a secure environment for reasonning about equivalence in the π-calulus. As a case study, we show by induction the correction of the specification of the addition in π-calculus.

Key-words: π-calculus, HOL

1 Introduction

Les algèbres de processus sont un cadre naturel pour décrire et analyser les systèmes concurrents. Elles fournissent plusieurs descriptions d'un même système à des niveaux d'abstraction différents, et des techniques permettant de montrer leurs équivalence. Un problème typique est de montrer l'équivalence entre la spécification d'un système et son implantation.
Plusieurs environnements de preuves ont été développés ces dernières années. Ces outils permettent de vérifier si deux processus sont équivalent ou non. La vérification est accomplie soit *automatiquement*, soit *interactivement*.
Les outils *automatiques*, voir par exemple, [1, 2, 3] sont basés sur des algorithmes raisonnablement *efficaces* et manipulent des spécifications représentées par des automates. Cependant, cette approche présente quelques problèmes relatifs à l'*explosion combinatoire des états* et une limitations de leurs champs d'applications aux seules spécifications non paramétriques et dont l'ensembles des états est finis.
Les outils *interactifs* [4, 5, 6] sont étudiées pour surmonter ces différents problèmes en se basant sur la représentation abstraite des processus et utilisent pleinement la nature algébrique des formalismes sous jacents. Ils se basent sur l'approche de *preuve de théorèmes* par opposition au *model checking* [7].
Le travail que nous décrivons dans ce rapport, s'inscrit dans cette dernière approche concernant la vérification de systèmes concurrents. Le formalisme de spécification que nous avons choisi est le π-calcul [8].
Le π-calcul est une extension de CCS [9], dans la mesure où les processus (les termes du π-calcul) peuvent échanger des noms de canaux. Cette possibilité augmente le pouvoir expressif de ce langage, et permet de décrire les systèmes dont la topologie du réseau de communications change *dynamiquement*. Le prix à payer pour ce gain d'expressivité est la complexité de la vérification et de l'analyse de ces systèmes.
Plutôt que de construire un outil de preuve spécial pour le π-calcul, on a décidé d'utiliser un outil de preuve déjà existant. L'idée est d'enrichir le prouveur de théorème choisi par la syntaxe des objets du π-calcul, et d'utiliser tous les outils développées pour ce système hôte afin d'accomplir nos preuves d'équivalence.
Le *prouveur de théorème* que nous avons choisi est le système HOL [10]. Le codage que nous présentons est basé sur celui *M. Nesi* [5] et de *T. Melham* [11]. En effet, nous avons suivi une approche purement définitionnelle dans notre formalisation de la théorie du π-calcul. Les différentes notations concernant le π-calcul sont ajoutés à la logique de HOL en les définissant par des entités déjà existantes et dont la sémantique est connue, plutôt que d'en postuler leurs propriétés par des axiomes. Cette façon d'étendre la théorie de HOL préserve sa cohérence.
La finalité de notre travail est le développement d'un outil *flexible* et *correcte*, dans lequel on peut raisonner sur plusieurs notions d'équivalence du π-calcul avec la possibilité de définir des stratégies de preuves à différent degrés d'interactions pour des classes particulières de processus.
Le rapport est organisé comme suit: On commence par une brève description du système HOL et de sa logique et les différents moyens de son extension. On présente en section 3 le π-calcul: sa syntaxe, sa sémantique et la définition de l'égalité entre processus. La section 4 décrit le codage du π-calcul dans HOL . Dans la section 5 on présente un exemple complet concernant la preuve de la spécification de l'addition en π-calcul. On termine par une comparaison avec des travaux antérieurs et par des perspectives de notre travail.

terme	*notation* HOL	*notation standard*	*description*
Vrai	`T`	$\top$	vrai
Faux	`F`	$\bot$	faux
Négation	`~P`	$\neg P$	non P
Disjonction	`P \/ Q`	$P \vee Q$	P ou Q
Conjonction	`P /\ Q`	$P \wedge Q$	P et Q
Implication	`P ==> Q`	$P \Rightarrow Q$	P implique Q
Égalité	`P = Q`	$P = Q$	P égale Q
Quantification -$\forall$	`!x.P`	$\forall x.P$	pour tous x:P
Quantification -$\exists$	`?x.P`	$\exists x.P$	il existe x:P
terme-ϵ	`@x.P`	$\epsilon x.P$	un x tel que P
Conditionnelle	`P => Q| R`	$(P => Q, R)$	si P alors Q sinon R

Table 1 : Correspondance entre les notations de HOL et les notations standards

2 Le Système HOL

On a choisi comme prouveur de théorème le système HOL de Cambridge développé par Gordon [12]. Le système HOL est basé sur une logique expressive et générale nous permettant d'avoir une formulation correcte et pratique. En effet, nous avons adopté une approche purement définitionnelle, pour garantir la cohérence de notre extension et assurer la correction des preuves d'équivalence de π-termes.
Le système HOL est basé sur le prouveur de théorème LCF [13] et hérite de plusieurs de ses concepts. Les sections qui suivent décrivent brièvement la logique de HOL et le système HOL . Pour plus de détail, consulter [10]. A ce jour il existe deux versions du système HOL . La première version (HOL88) développé sous ML [14], et une version (HOL90) développé par *Konrad Slind* sous SML[15, 16]. Nous utilisons dans ce travail la version 2.02 du système HOL88.

2.1 La Logique HOL

La logique de HOL est une variété de la logique d'ordre supérieur basée sur une formulation de la théorie simple des types de Church [17]. Elle se présente comme une extension de la logique des prédicats parce que:

- Les variables peuvent êtres instanciées par des fonctions, et les arguments des fonctions peuvent êtres des fonctions.
- Les fonctions peuvent êtres représentées par des λ-abstractions.
- Chaque terme possède un type qui peut être polymorphe (au sens du langage SML).

Les notations utilisées dans la logique de HOL sont présentées dans la table 1.

2.2 Le prouveur HOL

Le système HOL est le résultat du codage de la logique décrite ci-dessus dans le langage fonctionnel ML. Les termes sont représentés comme des objets ML de type *term*. Le langage ML permet de manipuler ces termes et de contrôler leur bonne formation au moyen d'un algorithme d'inférence de type. Un terme de la logique HOL doit être présenté au système

HOL entre guillemets. Si ML lui attribue le type *term*, alors cela veut dire qu'il est bien formé. Les types de la logique HOL sont codés comme des objets ML de type *type*. ML fait la différence entre un objet de type *term* et un objet de type *type*, en faisant précéder de deux points verticales touts objet de type *type*.

La fonction principale du système HOL est de montrer que certain termes de la logique HOL sont des théorèmes. Les théorèmes prouvés dans le système sont des objets d'un autre type du langage ML appelé *thm*. Ce qui permet de distinguer ces deux catégories de termes de HOL. Un théorème est représenté par un ensemble de termes qu'on appelle *hypothèses* et un terme qu'on appelle *conclusion*. Étant donné un ensemble d'hypothèses Γ et une conclusion t, on note le théorème correspondant par $\Gamma \vdash t$. Si Γ est vide, on notera simplement $\vdash t$. Les théorèmes sont introduits dans le système, soit en les postulant comme des *axiomes*, soit en les déduisant des *théorèmes* existants par des *règles d'inférences* décrites dans le metalangae ML et qu'on appelle *tactiques*. La preuve d'un théorème est une succession de règles d'inférences appliquées aux axiomes ou aux théorèmes déjà prouvés. Les règles d'inférences vérifient que la déduction de la conclusion d'un théorème à partir de ses hypothèses est conforme aux règles logiques de HOL. Le noyau du système HOL est constitué de cinq axiomes et de huit règles d'inférences primitives à partir desquelles toutes les autres règles de la logiques sont dérivées.

Le résultat d'une session avec le système HOL est un objet appelé *théorie*. Cet objet constitue un ensemble de types, de constantes, d'axiomes, de définitions et de théorèmes. Le système fournit des possibilités d'étendre des théories existantes et de former des hiérarchies de théories. Si des résultats d'autres théories doivent être utilisé dans la théorie en question, on doit alors déclarer ces théories comme *parents* de la théorie qu'on développe. Les théories permettent une structuration des faits.

Un autre concept très utile, est celui de *librairie*. Une *librairie* est une collection de théories, de théorèmes, de tactiques et de fonctions ML. Elle n'est pas nécessairement chargée lors du lancement du système HOL, néanmoins elle peut être chargée dynamiquement lors d'une session avec HOL. Dans notre cas, on a utilisé les librairies suivantes:

- `sets`: la théorie des ensembles finis et infinis [18].
- `string`: Type logique des codes ASCII des caractères et des "strings" [19].
- `ind_defs`: Outil pour construire des définitions inductives [20].
- `taut`: Outil de preuve de certaines tautologies [21].

2.3 Les définitions dans HOL

Une définition est un axiome de la forme $\vdash c = t$ où c est une constante nouvelle et t un terme clos. Cette forme particulière de l'axiome représente la spécification de la constante c dans la théorie en question. Ce mode d'introduction d'un nouveau fait préserve la cohérence de la théorie et évite l'introduction de contradictions. Il est possible d'avoir des axiomes sous la forme: $f x_1 \cdots x_n = t$, où l'ensemble des variables libres de t est inclus dans l'ensemble $\{x_1, \cdots, x_n\}$, puisque cette forme est équivalente à $f = \lambda x_1 \cdots x_n . t$. On peut aussi introduire des définitions primitives récursives, puisqu'elle admettent une définition non récursive équivalente [12]. Une théorie dont toutes les constantes sont définies par des axiomes qui sont des définitions est dites une *théorie définitionnelle*. Aux yeux des logiciens cette théorie est une *extension conservative*.

Puisque le mode d'introduction de nouvelles définitions est très restrictifs, il existe des outils automatiques facilitant l'introduction d'un certain type de définitions récursives, par

des preuves formelles. Notamment, des procédures pour définir de nouveaux types récursives, des fonctions primitives récursives sur ces types et certain types de définitions inductives [22, 20]. Ce genre d'outils a beaucoup facilité notre mécanisation. On reviendra sur chacune de ces procédures à l'occasion de leur utilisation.

2.4 Autres modes de raisonnement offert par HOL

Le système HOL offre un autre moyen pour construire des preuves. Il s'agit de la construction de preuves en utilisant une approche dirigées par le but. L'idée est de développer la preuve en commençant par le résultat désiré (but) et de le réduire à un certain nombre de sous buts plus simples à résoudre au moyen de programmes ML qu'on appelle communément des tactiques et dont le concept est dû à *R Milner* [23]. Le système gère cette situation au moyen d'une pile (subgoal package) qui peut être manipuler interactivement. Ce genre d'outil est dû à *L Paulson* [23]. Étant donné un but $\Gamma ?t$ où Γ est une liste d'hypothèse et t un terme à prouver. On commence par initialiser la pile à ce but au moyen d'une fonction spécifique puis on applique des tactiques adéquates pour se ramener à résoudre des sous buts dont la preuve est soit évidente soit plus simple à trouver. Le système permet de nous fournir la preuve désiré en sauvegardant à chaque étape la justification par laquelle on a accomplit la transformation d'un but en un ensemble de sous buts. Cette façon de trouver la preuve réincarne le vieux concept: "diviser pour régner".

Le système HOL fournit un autre type de fonction appelé *conversion* [24]. Ce type de fonction met en correspondance un terme t avec un autre terme u au moyen du théorème $\vdash t = u$. Cet outil s'avère très utile dans la définition de stratégies de simplification de termes compliqués. Souvent, la définition de ces stratégies est basée sur des conversions plus simple qu'on combine par des opérateurs définis dans le système HOL .

3 π-calcul

Dans cette section, on donne une brève description de la syntaxe, la sémantique opérationnelle et les différentes équivalences du π-calcul. Pour plus de détail, voir [8]. La seule différence qui existe entre la syntaxe des π-termes que nous présentons et celle présentée dans [8] est la manière avec laquelle est codé un comportement récursif. Dans notre présentation on a utilisé l'opérateur `Rec`, dans [11], l'auteur utilise l'opérateur `!(bang)` et dans [8], la récursion est codé par des identificateurs de processus. Toutes ces notations sont équivalentes, mais la difficulté du codage varie d'une représentation à une autre. La syntaxe du π-calcul est défini par la grammaire suivantes:

$$
\begin{array}{rlll}
P ::= & Nil & \text{(inactif)} \\
 & X & \text{(variable)} \\
| & c(x).P & \text{(réception d'un canal)} \\
| & \overline{c}d.P & \text{(envoi d'un canal)} \\
| & \tau.P & \text{(action inobservable)} \\
| & [x = y]P & \text{(conditionnelle positive)} \\
| & \nu c.P & \text{(restriction)} \\
| & P + P & \text{(somme non déterministe)} \\
| & P \mid P & \text{(composition parallèle)} \\
| & Rec\ X\ P & \text{(récursion)}
\end{array}
$$

Où Nil est le processus inactif qui ne fait rien. $c(x).P$ est le processus qui peut recevoir n'importe quel canal, soit d, puis se comporte comme le processus $P\{d/x\}$. Le processus

$\overline{c}d.P$ envoie le canal d sur c et évolue vers le processus P . Le processus $\tau.P$ exécute une action interne, inobservable par son environnement puis se comporte comme le processus P. Le processus $[x = y]P$ se comporte comme P si $x = y$ sinon comme Nil. $P + Q$ décrit le processus qui peut exécuter d'une façon non déterministe une action de P ou une action de Q. $P \mid Q$ représente deux processus qui évoluent simultanément avec la possibilité de communiquer entre eux. Le terme $\nu c.P$ permet de déclarer que le canal c est local à P. Le processus $Rec\ X\ P$ permet de définir des processus récursifs. On abrégera le processus $\nu d.\overline{c}d.P$ à $\overline{c}(d).P$, i.e, le processus qui envoie son canal local à son environnement.

Les définitions de l'ensemble des noms *libres* et *liés* sont standards. (Dans $c(x).P$ et dans $\nu x.P$ la variable x est liée). On notera par $Fn(P)$ et $Fn(\alpha)$ les noms libres de P et α; $Bn(P)$ et $Bn(\alpha)$ les noms liés de P et α; et $N(P)$ et $N(\alpha)$ l'ensemble des noms de P et de α. On identifie deux processus qui ne diffèrent que par leurs noms liés. On note par $P[y/x]$ le processus obtenu aprés la substitution de toutes les occurrences libres de x dans P par y.

La sémantique des termes du π-calcul est définie par un système de transition étiqueté. Ce système comprend un ensemble d'axiomes et de règles d'inférences permettant de calculer l'ensemble des transitions d'un processus.

On présente le système de transition étiqueté sous forme d'un système formel dans lequel on dérive les transitions $P \xrightarrow{\alpha} P'$ qui sont valides. Ce système est présenté à la figure 1.

in	$w \notin Fn(\nu x.P)$	$\Rightarrow$	$c(x).P \xrightarrow{c(w)} P[w/x]$
out		$\Rightarrow$	$\overline{c}d.P \xrightarrow{\overline{c}d} P$
tau		$\Rightarrow$	$\tau.P \xrightarrow{\tau} P$
$match$	$P \xrightarrow{\alpha} P'$	$\Rightarrow$	$[c = c]P \xrightarrow{\alpha} P'$
$par1$	$P \xrightarrow{\alpha} P' \wedge Bn(\alpha) \cap Fn(Q) = \emptyset$	$\Rightarrow$	$P \mid Q \xrightarrow{\alpha} P' \mid Q$
$sum1$	$P \xrightarrow{\alpha} P'$	$\Rightarrow$	$P + Q \xrightarrow{\alpha} P'$
rec	$P[Rec\ X\ P/X] \xrightarrow{\alpha} P'$	$\Rightarrow$	$Rec\ X\ P \xrightarrow{\alpha} P'$
$com1$	$P \xrightarrow{c(w)} P' \wedge Q \xrightarrow{\overline{c}d} Q'$	$\Rightarrow$	$P \mid Q \xrightarrow{\tau} P'[d/w] \mid Q'$
$close1$	$P \xrightarrow{c(w)} P' \wedge Q \xrightarrow{\overline{c}(w)} Q'$	$\Rightarrow$	$P \mid Q \xrightarrow{\tau} \nu w.(P' \mid Q')$
nu	$P \xrightarrow{\alpha} P' \wedge c \notin N(\alpha)$	$\Rightarrow$	$\nu c.P \xrightarrow{\alpha} \nu c.P'$
$open$	$P \xrightarrow{\overline{d}c} P' \wedge d \neq c \wedge w \notin Fn(\nu c.P)$	$\Rightarrow$	$\nu c.P \xrightarrow{\overline{d}(w)} P'[w/c]$

Figure 1 : Système de transition étiqueté. Les règles $sum2$,$par2$,$com2$,$close2$ sont omises.

3.1 Équivalence forte et faible

Plusieurs définitions d'équivalence entre π-termes existent en littérature [8, 25, 26, 27, 28]. Chacune d'elle est caractérisée par son traitement de l'instanciation des noms de canaux, et par son traitement de l'action inobservable τ. Pour notre cas, on a retenu la définition de l'équivalence basée sur l'instanciation précoce dans sa version forte et faible. La raison de ce choix est que cette équivalence peut se caractériser d'une façon naturelle en termes d'autres équivalences [27, 29] et elle se présente comme la plus large équivalence définie dans la théorie du π-calcul. Ces notions d'équivalence sont toutes basées sur la notion de bissimulation [30]. La bissimulation forte traite toutes les actions de la mêmes manière et est basée sur la relation de transition définie dans la figure 1. La bissimulation faible

fait abstraction aux actions inobservables τ et est définie en terme de transition faible. Rappelons dans ce qui suit quelques définitions.

Transition faible

Étant donné deux processus P et Q et une séquence d'actions $t = a_1, \cdots, a_n (n \geq 0)$, la transition faible est définie par: $P \stackrel{t}{\Longrightarrow} Q$ si $P(\stackrel{\tau}{\longrightarrow})^* \stackrel{a_1}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^* \cdots \stackrel{a_n}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^* Q$, où $(\stackrel{\tau}{\longrightarrow})^*$ représente la clôture réflexive et transitive de la transition $\stackrel{\tau}{\longrightarrow}$. Si $t = \epsilon$ (séquence vide), alors $P \stackrel{\epsilon}{\Longrightarrow} Q$ si et seulement si $P(\stackrel{\tau}{\longrightarrow})^* Q$.

Équivalence précoce forte

Une relation $\Re$ est dite une simulation précoce si pour tous termes P, Q tel que $P \Re Q$ alors:

- si $P \stackrel{\alpha}{\longrightarrow} P'$ et $\alpha \equiv \tau$ ou $\alpha \equiv \overline{c}d$ alors il existe Q' tel que $Q \stackrel{\alpha}{\longrightarrow} Q'$ et $P' \Re Q'$.
- si $P \stackrel{c(x)}{\longrightarrow} P'$ et $x \notin N(P) \cup N(Q)$ alors pour tous y, il existe Q' tel que $Q \stackrel{c(x)}{\longrightarrow} Q'$ et $P'\{y/x\} \Re Q'\{y/x\}$.
- si $P \stackrel{\overline{c}(d)}{\longrightarrow} P'$ et $d \notin N(P) \cup N(Q)$ alors il existe Q' tel que $Q \stackrel{\overline{c}(d)}{\longrightarrow} Q'$ et $P' \Re Q'$.

Une relation $\Re$ est une bissimulation précoce si $\Re$ et $\Re^{-1}$ sont des simulations précoces.
On dit que P et Q sont équivalent ($P \sim Q$) si et seulement si il existe une bissimulation précoce $\Re$ tel que $P \Re Q$. Cette relation d'équivalence est préservé par tous les opérateurs du π-calcul sauf l'opérateur de réception de canal. À titre d'exemple, on a $[x = a]\tau.Nil \sim Nil$, mais $c(x).[x = a]\tau.Nil \not\sim c(x).Nil$. En effet, il suffit d'instancier le paramètre x par le canal a et constater que $\tau.Nil \not\sim Nil$. Pour remédier à ce problème, il suffit de quantifier sur les substitutions et on aura la définition de la congruence comme suit:

$$P \sim_c Q \ \ Si \ \forall \sigma.P\sigma \sim Q\sigma$$

où σ est une substitution.

Équivalence précoce faible

Une relation $\Re$ est dite une simulation précoce faible si pour tous termes P, Q tel que $P \Re Q$ alors:

- si $P \stackrel{\tau}{\longrightarrow} P'$, alors il existe un Q' tel que $Q \stackrel{\epsilon}{\Longrightarrow} Q'$ et $P' \Re Q'$.
- si $P \stackrel{\overline{c}d}{\longrightarrow} P'$, alors il existe Q' tel que $Q \stackrel{\overline{c}d}{\Longrightarrow} Q'$ et $P' \Re Q'$.
- si $P \stackrel{c(x)}{\longrightarrow} P'$ et $x \notin N(P) \cup N(Q)$, alors pour tous y, il existe Q' tel que $Q \stackrel{c(x)}{\Longrightarrow} Q'$ et $P'\{y/x\} \Re Q'\{y/x\}$.
- si $P \stackrel{\overline{c}(d)}{\longrightarrow} P'$ et $d \notin N(P) \cup N(Q)$, alors il existe Q' tel que $Q \stackrel{\overline{c}(d)}{\Longrightarrow} Q'$ et $P' \Re Q'$.

Une relation $\Re$ est une bissimulation précoce faible si $\Re$ et $\Re^{-1}$ sont des simulations précocess faibles.
On dit que P et Q sont faiblement équivalent ou observationnellement équivalent ($P \approx Q$)

si et seulement si il existe une bissimulation précoce faible $\Re$ tel que $P \Re Q$.
Donc, le problème qui consiste à montrer que deux processus sont équivalent est un problème d'existence, i.e trouver une bonne instance de la relation $\Re$ qui vérifie les conditions d'une bissimulation. Une autre approche consiste à utiliser un raisonnement équationnelle basé sur un certain nombre d'équivalence simple préétablit, en les appliquant comme des règles de réécritures selon le principe de "la substitution des égaux par des égaux". Malheureusement, la bissimulation précoce faible n'est pas une congruence. Le problème est dû à l'action interne τ et à l'opérateur du choix non déterministe. Pour plus de détail voir [9].

Congruence observationnelle

La congruence observationnelle est définie par:

$$P = Q \Leftrightarrow \forall \sigma . P\sigma =_g Q\sigma,$$

où σ est une substitution, et $=_g$ est définie pour tous P et Q par:
$P =_g Q$ si,

- $P \xrightarrow{\alpha} P'$ et $\alpha \equiv \tau$ ou $\alpha \equiv \overline{c}d$ alors il existe Q' tel que $Q \stackrel{\alpha}{\Longrightarrow} Q'$ et $P' \approx Q'$.
- $P \xrightarrow{c(x)} P'$ et $x \notin N(P) \cup N(Q)$ alors pour tous y, il existe Q' tel que $Q \stackrel{c(x)}{\Longrightarrow} Q'$ et $P'\{y/x\} \approx Q'\{y/x\}$.
- $P \xrightarrow{\overline{c}(d)} P'$ et $d \notin N(P) \cup N(Q)$, alors il existe Q' tel que $Q \stackrel{\overline{c}(d)}{\Longrightarrow} Q'$ et $P' \approx Q'$.
- $Q \xrightarrow{\alpha} Q'$ et $\alpha \equiv \tau$ ou $\alpha \equiv \overline{c}d$ alors il existe P' tel que $P \stackrel{\alpha}{\Longrightarrow} P'$ et $P' \approx Q'$.
- $Q \xrightarrow{c(x)} Q'$ et $x \notin N(P) \cup N(Q)$ alors pour tous y, il existe P' tel que $P \stackrel{c(x)}{\Longrightarrow} P'$ et $P'\{y/x\} \approx Q'\{y/x\}$.
- $Q \xrightarrow{\overline{c}(d)} Q'$ et $d \notin N(P) \cup N(Q)$, alors il existe P' tel que $P \stackrel{\overline{c}(d)}{\Longrightarrow} P'$ et $P' \approx Q'$.

4 Codage du π-calcul dans HOL

L'approche que nous avons adopté dans notre mécanisation de la théorie du π-calcul est celle basée sur la séparation de la définition de la syntaxe des π-termes de leur sémantique, ce qui permet de définir plusieurs sémantiques pour le même langage. Cette approche est basée sur le package de *T. Melham* [22] qui permet d'introduire des nouveaux types récursives dans la logique.

Les nouveaux types introduits dans le système sont considérés comme des sous ensembles de valeurs de types déjà existants dans la logique et possédants une certaine propriété. Concrètement, soit ty un type définit dans la logique, et P un prédicat sur les valeurs de type ty qui définit un sous ensemble de valeurs non vide de ty, alors le nouveau type ty peut être définit comme un type isomorphe possédant les mêmes propriétés que le sous ensemble définit par P. Le problème est alors de trouver une représentation adéquate du type qu'on veut définir et de montrer un certain nombres de théorèmes qui caractérisent ce nouveau type. Le système offre une règle ML qui permet d'automatiser toutes ces opérations complexes à réaliser manuellement. Il s'agit de la fonction `define_type:string -> string ->thm`. Le premier argument de cette fonction est le nom du type qu'on veut définir, le second argument représente sa spécification en terme d'un ensemble de *constructeurs*. Le système

fournit comme résultat un théorème caractérisant d'une manière abstraite ce nouveau type et offrant la base de tous raisonnement le concernant.

Avant de présenter, les différentes étapes de notre mécanisation, nous donnons un aperçu général sur l'équivalent de chaque objet π-calcul en logique dans le tableau de la figure 2.

π-calcul	HOL
Noms de canaux	Type logique `Ch`
Processus	Type logique `Pr`
Action	Type logique `Act`
L'α équivalence	`CONV:Pr-> Pr ->bool`
L'ensemble des noms libres	`FN_Pr:Pr-> (Ch)set`
L'ensemble des noms liées	`BN_Pr:Pr-> (Ch)set`
L'ensemble des noms	`N_Pr:Pr-> (Ch)set`
La substitution	`Subt1_Pr:Pr -> Ch # Ch -> Pr`
Relation de transition	`Trans:Pr-> Act-> Pr ->bool`
Strong_Sim	`Strong_Sim:(Pr -> (Pr -> bool)) -> bool`
Strong_Bisim	`Strong_Bisim:(Pr -> (Pr -> bool)) -> bool`
Strong_Equiv	`Strong_Equiv:Pr -> (Pr -> bool)`
Strong_Cong	`Strong_Cong:Pr -> (Pr -> bool)`
Weak_Equiv	`Weak_Equiv:Pr -> (Pr -> bool)`
OBS_CONG	`OBS_CONG:Pr -> (Pr -> bool)`

Figure 2 : Représentation des objets du π-calcul dans HOL

4.1 Représentation des canaux en logique

Les canaux sont représentés par le type logique `string` préétabli dans le système dans la librairie `strings`. On définit alors le type `Ch` par renommage de ce type en utilisant la fonction `new_type_abbrev: (string # type) -> void`. Pour pouvoir renommer des canaux, nous avons introduit la constante `VARIANT:(Ch)set->Ch->Ch`, tel que, étant donné un ensemble de canaux `vs` et un canal `c` alors le canal `VARIANT vs c` est un nouveau canal qui n'appartient pas à l'ensemble `vs`. Le second argument de cette fonction sert seulement à guider la construction de ce nouveau canal. L'existence d'une telle fonction découle du fait que l'ensemble logique `string` est infini. Une définition constructive de cette fonction est: `Variant s x = x IN s => VARIANT (PRIME x) s | x`. où `PRIME x` donne une version primé du canal `x`. Pour tous ensemble fini `s` cette fonction doit terminer, mais malheureusement elle n'est pas primitive récursive et donc ne peut être définie directement dans HOL. Donc, il a fallu développer une preuve pour la définir dans notre théorie.

4.2 Représentation des processus en logique

On introduit les processus en définissant un objet `Pr` en utilisant le package de *T. Melham* permettant de définir de nouveaux types récursives. Le type est présenté au package sous forme d'une équation dont la partie gauche spécifie le nom du type à définir et la partie droite spécifie le type en terme d'un ensemble de constructeurs distincts.

```
Pr = Nil
       |Var string
```

```
        |In Ch Ch  Pr
        |Out Ch Ch Pr
        |Oute Ch Ch Pr
        |Tau Pr
        |Sum Pr Pr
        |Par Pr Pr
        |Nu Ch Pr
        |Rec string Pr
```

La syntaxe que nous avons implanté diffère de celle que nous avons introduit dans la section 3 par l'ajout du constructeur `Oute`. Ceci pour faciliter la définition d'un certain nombre de fonctions nécessaires dans la formalisation du théorème d'expansion. Le constructeur `Oute` n'est autre qu'une abbrèviation du terme `Nu c (Out d c P)`. Pour expliciter ce lien entre ces deux opérateurs, on a ajouté l'axiome suivant dans notre théorie, en invoquant la fonction `new_axiom`:

```
new_axiom(‘EXT_ABBREV‘,"!c d P. ~(c=d) ==> Nu c(Out d c P) = Oute d c P");;
```

Le package rend, en réponse un théorème qui caractérise ce type et qui forme la base du raisonnement sur ce type. Le théorème qui caractérise le type `Pr` est:

```
⊢ ∀e f0 f1 f2 f3 f4 f5 f6 f7 f8 f9.
    ∃unique fn.
     (fn Nil = e)           ∧
     (∀s. fn(Var s) = f0 s) ∧
     (∀s0 s1 P. fn(In s0 s1 P) = f1(fn P)s0 s1 P)   ∧
     (∀s0 s1 P. fn(Out s0 s1 P) = f2(fn P)s0 s1 P)  ∧
     (∀s0 s1 P. fn(Oute s0 s1 P) = f3(fn P)s0 s1 P) ∧
     (∀P. fn(Tau P) = f4(fn P)P)                     ∧
     (∀s0 s1 P. fn(Match s0 s1 P) = f5(fn P)s0 s1 P) ∧
     (∀P1 P2. fn(Sum P1 P2) = f6(fn P1)(fn P2)P1 P2) ∧
     (∀P1 P2. fn(Par P1 P2) = f7(fn P1)(fn P2)P1 P2) ∧
     (∀s P. fn(Nu s P) = f8(fn P)s P)                ∧
     (∀s P. fn(Rec s P) = f9(fn P)s P)
```

4.3 Représentation des actions en logique

Le type logique `Act` est introduit de la même manière que le type `Pr`. Sa spécification est:

```
Act = tau |out Ch Ch |in  Ch Ch |oute Ch Ch
```

Le théorème qui le caractérise est alors:

```
⊢ ∀e f0 f1 f2.
    ∃unique fn.
     (fn tau = e) ∧
     (∀s0 s1. fn(out s0 s1) = f0 s0 s1) ∧
     (∀s0 s1. fn(in s0 s1) = f1 s0 s1) ∧
     (∀s0 s1. fn(oute s0 s1) = f2 s0 s1)
```

Ces deux théorèmes permettent de montrer l'existence de fonctions primitives récursives sur ces types. Le système offre une fonction `new_recursive_definition` pour automatiser l'introduction de nouvelles constantes dont la spécification est définie récursivement sur le

type `Pr`. En particulier, nous avons introduit les constantes `FN_Pr,BN_Pr,N_Pr` qui calculent, respectivement, l'ensemble des canaux libres, des canaux liées et de tous les canaux d'un processus.

4.4 Substitution

La complexité de la substitution définit dans le π-calcul est dû à la présence des deux opérateurs `Nu` et `In` qui lient les canaux. Lors d'une application d'une substitution, il faudra s'assurer qu'aucun canal libre ne devient lié. La définition standard de la substitution est récursive; on ne peut pas la définir directement dans HOL. Une autre approche de *substitution simultané* introduite par *A Stoughton* dans [31] nous donne une définition primitive récursive, qu'on peut introduire dans HOL, en utilisant simplement la règle dérivé d'introduction de nouvelles fonctions primitives récursives: `new_recursive_definition`. Ce qui simplifie les preuves concernant la substitution en utilisant le principe d'induction structurelle sur les termes définissant les processus. La définition de la substitution simultané est:

```
⊢def (∀s. Subt_Pr Nil s = Nil) ∧
   (∀x s. Subt_Pr(Var x)s = Var x) ∧
   (∀c x P s. Subt_Pr(In c x P)s =
   (let vs = IMAGE s((FN_Pr P) DELETE x)
    in
     let y = (x IN vs →  VARIANT vs x | x)
     in
      In(s c)y(Subt_Pr P(λn. ((n = x) = y | s n))))) ∧
 (∀c d P s. Subt_Pr(Out c d P)s = Out(s c)(s d)(Subt_Pr P s)) ∧
 (∀c d P s. Subt_Pr(Match c d P)s = Match(s c)(s d)(Subt_Pr P s)) ∧
 (∀P s. Subt_Pr(Tau P)s = Tau(Subt_Pr P s)) ∧
 (∀P Q s. Subt_Pr(Sum P Q)s = Sum(Subt_Pr P s)(Subt_Pr Q s)) ∧
 (∀P Q s. Subt_Pr(Par P Q)s = Par(Subt_Pr P s)(Subt_Pr Q s)) ∧
 (∀x P s. Subt_Pr(Rec x P)s = Rec x(Subt_Pr P s)) ∧
 (∀x P s.
   Subt_Pr(Nu x P)s =
   (let vs = IMAGE s((FN_Pr P) DELETE)
    in
     let y = (x IN vs →  VARIANT vs x| x)
     in
      Nu y(Subt_Pr P(λn. ((n = x) →  y | s n))))
```

Cette définition utilise les constantes définit dans la librairie `set`, en particulier la constante `IN:* -> ((*)set -> bool)"` qui représente le test d'appartenance d'un élément dans un ensemble et la constante `IMAGE:(*->**)->(*)set->(**)set` qui calcule l'image des éléments d'un ensemble. Elle utilise aussi, les fonctions `FN_Pr` et `VARIANT` que nous avons introduit dans la section précédente.
Maintenant, on peut introduire facilement la substitution d'un canal unique. C'est justement suffisant pour notre mécanisation.

```
⊢def ∀P x y. Subt1_Pr P(x,y) = Subt_Pr P(λn. ((n = x) →  y | n))
```

Nous avons implanté une conversion basé sur cette définition pour simplifier les termes contenant une substitution. Cette conversion `SUBT_CONV` automatise la suite de règles d'inférences nécessaire pour dériver le théorème qui représente le résultat d'une application d'une substitution.

4.5 L'alpha congruence

La relation de l'α congruence, $=_\alpha$ est la plus petite relation close par rapport à l'ensemble de règles de la figure 3. La fonction `FV_Pr` calcule les variables de processus libre, et $P[Y/X]$

$$
\begin{array}{lcl}
 & \Rightarrow & Nil =_\alpha Nil \\
P =_\alpha Q & \Rightarrow & \tau.P =_\alpha \tau.Q \\
P =_\alpha Q & \Rightarrow & \overline{c}d.P =_\alpha \overline{c}d.Q \\
((x = y) \wedge P =_\alpha Q) \vee & & \\
(y \notin FN\_Pr(P) \wedge P[y/x] =_\alpha Q) & \Rightarrow & c(x).P =_\alpha c(y).Q \\
((x = y) \wedge P =_\alpha Q) \vee & & \\
(y \notin FN(P) \wedge P\iota[y/x] =_\alpha Q) & \Rightarrow & \nu x.P =_\alpha \nu y.Q \\
P_1 =_\alpha Q_1 \wedge P_2 =_\alpha Q_2 & \Rightarrow & P_1 + Q_1 =_\alpha P_2 + Q_2 \\
P_1 =_\alpha Q_1 \wedge P_2 =_\alpha Q_2 & \Rightarrow & P_1 \mid Q_1 =_\alpha P_2 \mid Q_2 \\
P =_\alpha Q & \Rightarrow & [c = d]P =_\alpha [c = d]Q \\
((X = Y) \wedge P =_\alpha Q) \vee & & \\
((Y \notin FV\_Pr(P) \wedge P[Y/X] =_\alpha Q & \Rightarrow & Rec\ X.P =_\alpha Rec\ Y.Q
\end{array}
$$

Figure 3 : Définition de la relation de l'α-congruence

est la substitution de la variable X par Y dans P. Leurs définitions dans notre théorie est similaire aux définitions des fonctions `FN_Pr` et `Subt_Pr`.
On dit que P et Q sont α-convertible *si et seulement si* $P =_\alpha Q$.
Nous avons introduit la relation de l'α-congruence dans notre théorie en utilisant la règle d'introduction de *predicat inductif*. Ce principe est décrit brievement dans la section suivante. La relation de α-conversion est une proprieté décidable, nous avons developpé une conversion qui vérifie si deux processus sont α-congruent ou non.

4.6 Système de transition étiqueté

Le système de transition étiquetté présenté dans la figure 1 est codé dans HOL en utilisant le package développé par *T. Melham* [20] qui permet l'introduction de nouvelles *relations inductives*. Ce package permet de prouver *automatiquement* l'existence de la relation définit inductivement, par un ensemble de règles d'inferences. Ce dernier est présenté à ce package sous forme d'une liste de couples de la forme:
(<listes d'hypothèses>, <conclusion>), où la première composante du couple est une liste d'hypothèses d'une règle avec ces "conditions d'application" et la deuxième composante représente la conclusion de la règle. A chaque règle correspond un statut logique sous forme d'implication de la conclusion de la règle par la conjonction de ces hypothèses avec éventuellement ces "conditions d'application". Le système définit une constante pour nommer cette relation et prouve automatiquement une série de théorèmes montrant que la nouvelle relation est close par rapport aux règles d'inférences et un théorème montrant que c'est la plus petite relation satisfaisant ces règles. Le système offre, aussi, un certain nombres de tactiques permettant de montrer des propriétés relatives à ce genre de définition. On retiendra, la tactique permettant de raisonner par induction structurelle et des tactiques permettant de montrer la validité d'une transition. Un autre théorème, relatif à l'analyse des cas et qui très intéressant dans la conduite des preuves relatives au système

de transition étiquetté est montré automatiquement.

5 Équivalence du π-calcul dans HOL

Les trois équivalences, décrites dans la section 3 se traduisent facilement dans HOL , en utilisant le mécanisme de base d'introduction de nouvelles constantes:
`new_definition`. La notion d'équivalence est basée sur la notion de la relation de bissimulation [30] qui est elle même basée sur la relation de transition définit dans la section précédente pour sa version forte, et sur la notion de transition faible pour sa version faible. Cette dernière étant la clôture réflexive et transitive de la transition $P \xrightarrow{\tau} P'$. Alors, introduisant la constante $TransRc : Pr \longrightarrow Pr \longrightarrow bool$ dans la théorie en invoquant la règle dérivée pour l'introduction de nouvelles *définitions inductives*. Les trois règles que doit satisfaire cette constante sont:

```
∀P   . TransRc P P  ∧
        ∀P P'. Trans P tau P' ⇒  TransRc P P' ∧
        ∀P P'. (∃P1. TransRc P P1 ∧ TransRc P1 P') ⇒  TransRc P P'
```

La définition de la transition faible est alors:

```
⊢def ∀P a P'.Weak_Trans P a P' =
     (∃P1 P2. TransRc P P1 ∧ Trans P1 P2 ∧ TransRc P2 P')
```

A ce point, nous sommes en mesure de définir toutes les autres constantes qui correspondent aux différentes notions que nous avons présenté dans la section 3:

Simulation

```
⊢def ∀R.  Strong_Sim R =
    ∀P Q. R P Q ⇒
       (∀P'. Trans P tau P' ⇒
                 ∃Q'. Trans Q tau Q' ∧ R P' Q') ∧
       (∀c x P'. Trans P(in c x)P' ∧ ¬x IN ((N_Pr P) UNION (N_Pr Q)) ⇒
              ∀y. ∃Q'. Trans Q(in c x)Q' ∧
                       R(Subt1_Pr P'(x,y))(Subt1_Pr Q'(x,y))) ∧
      (∀x y P'. Trans P(out x y)P' ⇒
                 ∃Q'. Trans Q(out x y)Q' ∧ R P' Q') ∧
      (∀x y P'. Trans P(oute x y)P' ∧ ¬y IN ((N_Pr P) UNION (N_Pr Q)) ⇒
                 ∃Q'. Trans Q(oute x y)Q' ∧ R P' Q')
```

Bissimulation

```
⊢def ∀R. Strong_Bisim R = Strong_Sim R ∧ Strong_Sim(λx y. R y x)
```

Équivalence forte

```
⊢def ∀P Q. Strong_Equiv P Q = (∃R. R P Q ∧ Strong_Bisim R)
```

Congruence forte

```
⊢def ∀P Q. Strong_Cong P Q = ∀s.Strong_Equiv (Subt_Pr P s)(Subt_Pr Q s)
```

Simulation faible

```
⊢def ∀R. Weak_Sim R = ∀P Q. R P Q ⇒
        (∀P'. Trans P tau P' ⇒
               ∃Q'. TransRc Q Q' ∧ R P' Q') ∧
        (∀c x P'. Trans P(in c x)P' ∧ ¬x IN ((N_Pr P) UNION (N_Pr Q))⇒
             ∀y.∃Q'. Weak_Trans Q(in c x)Q' ∧
                     R(Subt1_Pr P'(x,y))(Subt1_Pr Q'(x,y))) ∧
        (∀c d P'. Trans P(out c d)P' ⇒
               ∃Q'. Weak_Trans Q(out c d)Q' ∧ R P' Q') ∧
        (∀c d P'. ¬d IN ((N_Pr P) UNION (N_Pr Q)) ∧
          Trans P(oute c d)P' ⇒
               ∃Q'. Weak_Trans Q(oute c d)Q' ∧ R P' Q')
```

Bissimulation faible

```
⊢def ∀R. Weak_Bisim R = Weak_Sim R ∧ Weak_Sim(λx y. R y x)
```

Équivalence faible

```
⊢def ∀P Q. Weak_Equiv P Q = (∃R. R P Q ∧ Weak_Bisim R)
```

Congruence observationnelle

```
⊢def ∀P Q. OBS_CONG P Q = ∀s. Obs_Cong (Subt_Pr P s) (Subt_Pr Q s)
```

```
⊢def ∀P Q. Obs_Cong P Q =
         (∀P'. Trans P tau P' ⇒
                ∃Q'. Weak_Trans Q tau Q' ∧ Weak_Equiv P' Q') ∧
         (∀c x P'. Trans P (in c x) P' ∧
                           ¬x IN ((N_Pr P) UNION (N_Pr Q)) ⇒
             ∀y.∃Q'. Weak_Trans Q (in c x) Q' ∧
                           Weak_Equiv(Subt1_Pr P'(x,y))(Subt1_Pr Q'(x,y))) ∧
         (∀c d P'. Trans P (out c d)P' ⇒
                ∃Q'. Weak_Trans Q (out c d)Q' ∧ Weak_Equiv P' Q') ∧
         (∀c d P'. Trans P (oute c d)P ∧
                           ¬d IN ((N_Pr P) UNION (N_Pr Q)) ⇒
                ∃Q'. Weak_Trans Q (oute c d)Q' ∧ Weak_Equiv P' Q'))) ∧
        (∀P'.Trans Q tau P' ⇒
                ∃Q'. Weak_Trans P tau Q' ∧ Weak_Equiv P' Q') ∧
        (∀c x P'. Trans Q (in c x) P' ∧
                           ¬x IN ((N_Pr Q) UNION (N_Pr P)) ⇒
             ∀y.∃Q'. Weak_Trans P(in c x)Q' ∧
                           Weak_Equiv(Subt1_Pr P'(x,y))(Subt1_Pr Q'(x,y))) ∧
        (∀c d P'. Trans Q (out c d)P' ⇒
                ∃Q'. Weak_Trans P(out c d)Q' ∧ Weak_Equiv P' Q') ∧
        (∀c d P'.Trans Q(oute c d)P ∧
                          ¬d IN ((N_Pr Q) UNION (N_Pr P))⇒
                ∃Q'. Weak_Trans P(oute c d)Q' ∧ Weak_Equiv P' Q')
```

5.1 Règles algébriques du π-calculus

Toutes ces règles algébriques sont formellement dérivées dans le système HOL . La preuve est basée sur les définitions de l'équivalence forte et la bissimulation forte. L'idée de la preuve est d'exhiber une relation qui vérifie les conditions d'une bissimulation. Les preuves

sont souvent assez longues et necessitent beaucoup de temps et de patiences parce qu'elles requiert l'analyse de beaucoup de cas. Cette situation se retrouve souvent dans la preuve des lois relatives à la composition parallèle.
Toutes ces lois ainsi montrées forment la base du raisonnement sur les applications spécifiées dans le π-calcul. Les lois relatives à l'équivalence faible et à la congruence observationnelle se dérivent facilement à partir des lois de l'équivalence forte en utilisant des tactiques de bases du système de HOL et les deux théorèmes suivants:

```
⊢ ∀P Q. Strong_Cong P Q ⇒  OBS_CONG P Q

⊢ ∀P Q. OBS_CONG P Q  ⇒  Weak_Equiv P Q
```

Dans ce qui suit on présente la liste des théorèmes déjà prouvés dans notre système.

CONG

```
⊢ ∀P Q. CONV P Q ⇒  Strong_Equiv P Q
⊢ ∀P Q. Strong_Equiv P Q ⇒
              ∀c d.Strong_Equiv (Out c d P) (Out c d Q)
⊢ ∀P Q. Strong_Equiv P Q ⇒
              ∀c d.Strong_Equiv (Oute c d P) (Oute c d Q)
⊢ ∀P Q. Strong_Equiv P Q ⇒   Strong_Equiv (Tau P) (Tau Q)
⊢ ∀x P Q y. Strong_Equiv (Subt1_Pr P (x,y)) (Subt1_Pr Q (x,y)) ⇒
              ∀c.Strong_Equiv (In c x P) (In c x Q)
⊢ ∀P Q. Strong_Equiv P Q ⇒  ∀R.Strong_Equiv (Sum P R) (Sum Q R)
⊢ ∀P Q. Strong_Equiv P Q ⇒  ∀R.Strong_Equiv (Par P R) (Par Q R)
⊢ ∀P Q. Strong_Equiv P Q ⇒  ∀c.Strong_Equiv (Nu c P) (Nu c Q)
```

SUM

```
⊢∀P. Strong_Equiv (Sum P Nil) P
⊢∀P Q. Strong_Equiv (Sum P Q) (Sum Q P)
⊢∀P. Strong_Equiv (Sum P P) P
⊢∀P Q R.Strong_Equiv (Sum (Sum P Q) R) (Sum P (Sum Q R))
```

PAR

```
⊢∀P. Strong_Equiv (Par P Nil) P
⊢∀P Q. Strong_Equiv (Par P Q) (Par Q P)
⊢∀P Q R. Strong_Equiv (Par (Par P Q) R) (Par P (Par Q R))
```

NU

```
⊢∀c d P. Strong_Equiv (Nu c (Out c d  P)) Nil
⊢∀c d P. Strong_Equiv (Nu c (Oute c d P)) Nil
⊢∀c x P. Strong_Equiv (Nu c (In c x   P)) Nil
⊢∀c P  . Strong_Equiv (Nu c (Tau P)) (Tau (Nu c P))
⊢∀c d e P. ¬(c=d) ∧ ¬(c=e) ⇒
          Strong_Equiv (Nu c (Out d e P )) (Out d e (Nu c P))
⊢∀c d x P. ¬(c=d) ∧ ¬(c=x) ⇒
⊢         Strong_Equiv (Nu c (In d x P )) (In d x (Nu c P))
⊢∀c d P. Strong_Equiv (Nu c (Nu d P)) (Nu d (Nu c P))
⊢∀c P Q. Strong_Equiv (Nu c (Sum P Q)) (Sum (Nu c P) (Nu c Q))
⊢∀c .  Strong_Equiv (Nu c Nil) Nil
⊢∀c P Q. ¬(c IN FN_Pr P) ⇒
           Strong_Equiv (Nu c (Par P Q)) (Par P (Nu c Q))
```

MATCH

```
⊢∀c P. Strong_Equiv (Match c c P) P
⊢∀c d P. ¬(c=d) ⇒  Strong_Equiv (Match c d P) Nil
```

REC

```
⊢ ∀x E. Strong_Equiv(Rec x E)(PI_SUBT E(Rec x E)x)
```

Où la constante `PI_SUBT:Pr->Pr->string->Pr` calcule le processus `(PI_SUBT P1 P2 X)` obtenu après le remplacement de toutes les occurrences de `X` par le processus `P2` dans `P1`. La fonction `PI_SUBT` est introduite dans la théorie par la règle derivée d'introduction de nouvelles *fonctions récursives*. Dans la définition de cette fonction, nous avons supposé que toutes les variables de processus liées sont différentes.

5.2 Le théorème d'expansion

Le théorème d'expansion nous permet d'éliminer l'opérateur de composition parallèle en le remplaçant par l'opérateur du non déterminisme. Cette simplification permet de mettre en évidence les actions succeptibles d'être exécuter par un processus.

Dans ce qui suit on dira qu'un processus `P` est en forme préfixé s'il existe `c,x,Q` tel que `P = Tau Q` ∨ `P = In c x Q` ∨ `P = Out c x Q` ∨ `P = Oute c x Q`.

La formalisation du théorème d'expansion dans le système HOL nécessite l'introduction de nouvelles définitions pour supporter les notations syntaxiques introduites dans sa formulation. La première définition correspond à l'introduction de la somme de `n` processus en forme préfixé qu'on appelle une somme indéxé . Étant donné un entier `n` et une fonction `P:num -> Pr`, on définit la fonction `SIGMA` par récursion primitive tel que: `SIGMA n P` correspond au processus `(P 0) + (P 1) + ...  + (P n)`. Cette fonction est définit par:

```
⊢def SIGMA 0 E       = PREF (ACT_PREF (E 0)) (CONT_PREF (E 0)) ∧
   SIGMA (SUC m) E = Sum (SIGMA m E)
                        (PREF (ACT_PREF (E (SUC m))) (CONT_PREF (E (SUC m))))
```

où `ACT_PREF`, `CONT_PREF` représentent les fonctions de projections qui calculent l'action ou la continuation d'un processus en forme préfixé et la fonction `PREF:Act->Pr-> Pr` est définie de tel façon que le processus `PREF a P` exécute l'action `a` puis se comporte comme `P`.

```
⊢def ∀c x P.ACT_PREF (In c x P)  = in c x  ∧
  ∀c d P.ACT_PREF (Out c d P) = out c d ∧
  ∀P.ACT_PREF (Tau P)     = tau    ∧
  ∀c d P.ACT_PREF (Oute c d P)  = oute c d

⊢def  ∀c x P.CONT_PREF (In c x P)  = P ∧
  ∀c d P.CONT_PREF (Out c d P) = P ∧
  ∀P.CONT_PREF (Tau P)     = P  ∧
  ∀c d P.CONT_PREF (Oute c d P)= P

⊢def  ∀c x P.PREF (in c x)   P    = In c x P ∧
  ∀c d P.PREF (out c d)  P    = Out c d P) ∧
  ∀P.    PREF  tau       P    = Tau P)    ∧
  ∀c d P.PREF (oute c d) P    = Oute c d P)
```

Dans la formalisation du théorème d'expansion, on doit prendre en compte des problèmes de capture des noms de canaux qui peuvent surgir dans les situations suivantes:

1. Soit $P = c(x).P1$ et $Q = \overline{x}d.Q1$, alors une naive application du théorème d'expansion au terme $P \mid Q$ le reduit au terme $c(x).(P1 \mid Q) + \overline{x}d.(P \mid Q1)$. Le canal x qui était libre dans Q se retrouve liée dans l'expression obtenu.

2. Soit $P = \nu c.\overline{d}c.P1$ et $Q = d(x).\overline{c}x.Q1$. On appliquant le théorème d'expansion au terme $P \mid Q$, on obtient le terme $\nu c.\overline{d}c.(P1 \mid Q) + d(x).(P \mid \overline{c}x.Q1) + \tau.\nu c.(P1 \mid \overline{c}c.Q1\{c/x\})$. Encore une fois l'occurence libre de c dans Q est capturée par l'operateur ν.

Pour prevenir ces problèmes on doit renommer les canaux liées à chaque fois que c'est nécessaire. Le théorème d'expansion est alors formalisé par le théorème suivant:

```
⊢ ∀m P n Q.
    Strong_Equiv
    (Par(SIGMA m P)(SIGMA n Q))
    (Sum
     (SIGMA m
      (λi.
        ((ACT_TYPE(P i) = `in`) →
         let x = RCV_CH(P i) in
         let c = CH_COM(P i) in
         let vs = FN_Pr(SIGMA n Q) in
         let y = VARIANT vs x in
             In c y(Par(Subt1_Pr(CONT_PREF(P i))(x,y))(SIGMA n Q)) |
        ((ACT_TYPE(P i) = `oute`) →
         let x = SEND_CH(P i) in
         let c = CH_COM(P i) in
         let vs = FN_Pr(SIGMA n Q) in
         let y = VARIANT vs x in
             Oute c y(Par(Subt1_Pr(CONT_PREF(P i))(x,y))(SIGMA n Q)) |
         PREF(ACT_PREF(P i))(Par(CONT_PREF(P i))(SIGMA n Q))))))
     (Sum
      (SIGMA n
       (λi.
         ((ACT_TYPE(Q i) = `in`) →
          let x = RCV_CH(Q i) in
          let c = CH_COM(Q i) in
          let vs = FN_Pr(SIGMA m P) in
          let y = VARIANT vs x in
              In c y(Par(SIGMA m P)(Subt1_Pr(CONT_PREF(Q i))(x,y))) |
         ((ACT_TYPE(Q i) = `outè`) →
          let x = SEND_CH(Q i) in
          let c = CH_COM(Q i) in
          let vs = FN_Pr(SIGMA m P) in
          let y = VARIANT vs x in
              Oute c y(Par(SIGMA m P)(Subt1_Pr(CONT_PREF(Q i))(x,y))) |
          PREF(ACT_PREF(Q i))(Par(SIGMA m P)(CONT_PREF(Q i)))))))
      (ALL_SYNC m P n Q)))
```

Où les fonctions SEND_CH,RCV_CH,CH_COM,ACT_TYPE sont définies par :

```
⊢def ∀c d P.SEND_CH (Out c d P) = d ∧
           ∀c d P.SEND_CH (Oute c d P) = d
⊢def ∀c x P.RCV_CH (In c x P) = x)
⊢def ∀c x P.CH_COM (In c x P)  = c ∧
           ∀c d P.CH_COM (Out c d P) = c ∧
           ∀c d P.CH_COM (Oute c d P) = c
⊢def ∀c x P.ACT_TYPE (In c x P)  = `in` ∧
           ∀c d P.ACT_TYPE (Out c d P) = `out` ∧
           ∀P.ACT_TYPE (Tau P)     = `tau` ∧
           ∀c d P.ACT_TYPE (Oute c d P) = `outè
```

La fonction SYNC calcule toutes les synchronisations possibles entre un processus en forme préfixé et un processus en forme de somme indexé. Elle est définit dans HOL , par récursion primitive. La complexité de cette fonction est dûe aux problèmes de renommage auxquelles il faut faire attention dans le cas où un processus envoit un canal lié à son partenaire. La définition de cette fonction est la définition la plus compliquée dans notre théorie.

```
⊢def (∀P E.
     SYNC P 0 E =
     (((ACT_TYPE P = `in`) ∧ (ACT_TYPE(E 0) = `out`) ∧
       (CH_COM P = CH_COM(E 0))) →
    Tau (Par (Subt1_Pr(CONT_PREF P)(RCV_CH P,SEND_CH(E 0)))
             (CONT_PREF(E 0)))
  | (((ACT_TYPE P = `in`) ∧ (ACT_TYPE(E 0) = `oute`) ∧
      (CH_COM P = CH_COM(E 0))) →
      let c = SEND_CH(E 0)in
      let x = RCV_CH P in
      let vs = FN_Pr(Nu x P) in
      let y = VARIANT vs c in
          Tau (Nu y(Par(Subt1_Pr(CONT_PREF P)(x,y))
                       (Subt1_Pr(CONT_PREF(E 0))(c,y))))
  | (((ACT_TYPE P = `out`) ∧ (ACT_TYPE(E 0) = `in`) ∧
       (CH_COM P = CH_COM(E 0))) →
      Tau (Par (CONT_PREF P)
               (Subt1_Pr(CONT_PREF(E 0))(RCV_CH(E 0),SEND_CH P)))
  | (((ACT_TYPE P = `oute`) ∧ (ACT_TYPE(E 0) = `in`) ∧
        (CH_COM P = CH_COM(E 0))) →
       let c = SEND_CH P in
       let x = RCV_CH(E 0) in
       let vs = FN_Pr(Nu x(E 0)) in
       let y = VARIANT vs c in
           Tau (Nu y(Par(Subt1_Pr(CONT_PREF P)(c,y))
                        (Subt1_Pr(CONT_PREF(E 0))(x,y))))
  | Nil))))) ∧
   (∀P m E.
     SYNC P(SUC m)E =
     (((ACT_TYPE P = `in`) ∧ (ACT_TYPE(E(SUC m)) = `out`) ∧
       (CH_COM P = CH_COM(E(SUC m)))) →
      Sum
      (Tau(Par(Subt1_Pr(CONT_PREF P)(RCV_CH P,SEND_CH(E(SUC m))))
              (CONT_PREF(E(SUC m)))))
      (SYNC P m E)
   | (((ACT_TYPE P = `in`) ∧ (ACT_TYPE(E(SUC m)) = `oute`) ∧
       (CH_COM P = CH_COM(E(SUC m)))) →
      let c = SEND_CH(E(SUC m)) in
      let x = RCV_CH P        in
      let vs = FN_Pr(Nu x P)  in
      let y = VARIANT vs c    in
          Sum
          (Tau(Nu y(Par (Subt1_Pr(CONT_PREF P)(x,y))
                        (Subt1_Pr(CONT_PREF(E(SUC m)))(c,y)))))
         (SYNC P m E)
   | (((ACT_TYPE P = `out`) ∧ (ACT_TYPE(E(SUC m)) = `in`) ∧
       (CH_COM P = CH_COM(E(SUC m)))) →
      Sum
      (Tau (Par(CONT_PREF P)
               (Subt1_Pr(CONT_PREF(E(SUC m)))(RCV_CH(E(SUC m)),SEND_CH P))))
      (SYNC P m E)
   | (((ACT_TYPE P = `oute`) ∧ (ACT_TYPE(E(SUC m)) = `in`) ∧
        (CH_COM P = CH_COM(E(SUC m)))) →
       let c = SEND_CH P in
       let x = RCV_CH(E(SUC m)) in
       let vs = FN_Pr(Nu x(E(SUC m))) in
       let y = VARIANT vs c in
           Sum
           (Tau(Nu y(Par (Subt1_Pr(CONT_PREF P)(c,y))
                         (Subt1_Pr(CONT_PREF(E(SUC m)))(x,y)))))
           (SYNC P m E)
    | SYNC P m E))))
```

La fonction fonction `ALL_SYNC` calcule toutes les communications possibles entre deux processus en forme de somme indexé. Elle est définie aussi par récursion primitive.

```
⊢def ∀P m Q.ALL_SYNC  0  P m Q =  SYNC (P 0) m Q                ∧
   ∀n P m Q.ALL_SYNC (SUC n) P m Q =  Sum (SYNC (P (SUC n)) m Q)
                                  (ALL_SYNC n P m Q)
```

5.3 Les outils de preuves associés aux lois algébriques du π-calcul

Nous avons implanté plusieurs conversions basées sur les lois algébriques du π-calcul, voir [32]. Chaque conversion peut être vue comme une procédure de preuve paramétrique qui simplifie automatiquement un terme quelconque par rapport aux lois qui lui sont associées. La plus importante de ces conversions est celle qui reduit le parallelisme au choix non deterministe. Un des réelles problèmes qu'on doit résoudre est celui de *l'explosion des états*. Le nombre de termes obtenus aprés la reduction du parallélisme croit exponentiellement par rapport aux nombres de ces sous termes. Pour remédier à ce problème, on utilise une approche plutôt naturelle: d'abord appliquer une étape d'expansion puis on réduit le terme obtenue en utilisant d'autres théorèmes appropriées. Le but est d'éliminer, dès leurs formations, tous les sous termes qui ne sont pas importants dans la suite du processus de simplification. Cette conversion constitue l'étape la plus importante dans la preuve que nous allons présenter dans la section suivante.

6 Preuve par induction de la spécification de l'addition en π-calcul

Notre premier exemple concerne la spécification des entiers naturels et de l'addition naturelle en π-calcul. Notre objectif est d'appliquer notre formalisation à la preuve de l'équivalence faible de processus dont la structure est définie par induction. Cette preuve est construite interactivement dans le système HOL, en utilisant particulièment la tactique prédefinie `INDUCT_TAC`.

Le codage que nous allons présenter est celui décrit par Milner dans [33]. Un entier est représenté par un processus paramètré par deux canaux. Chaque synchronisation sur le premier canal comptabilise la valeur de cet entier et le deuxième canal permet de tester sa valeur. L'étape suivante concerne la définition de l'addition de deux entiers (processus), en π-calcul et la preuve de sa correction.

Dans ce qui suit, on montre comment utiliser notre système pour accomplir cette preuve. Notre présentation suit un ordre chronologique, ce qui permet d'exprimer l'ordre dans lequel les commandes sont introduites dans le système. On suppose que les systèmes HOL et `PIC` ont été chargés. Le prompt d'invitation est '`PIC>`'.

Les lignes commençant par ce prompt et se terminant par deux points virgules représentent les commandes de l'utilisateur, et les autres lignes montrent la réponse du système.
On commence par déclarer la théorie dans laquelle on va raisonner sur les entiers, puis on décrit le comportement d'un entier en introduisant la constante `PI` par récursion primitive.

1

```
PIC>let PI = new_prim_rec_definition (‘PI‘,
    "(PI    0    x z = Out z ‘a‘ Nil) /\
     (PI (SUC n) x z = Out x ‘a‘ (PI n x z))");;

PI =
|- (!x z. PI 0 x z = Out z ‘a‘ Nil) /\
   (!n x z. PI(SUC n)x z = Out x ‘a‘(PI n x z))
```

La spécification qui décrit l'addition (`Add`) en π-calcul n'est pas primitive récursive et dépend de la définition des deux constantes `Copy` et `Succ` qui sont elles mêmes mutuellement récursive. On ne peut pas les introduire directement dans HOL mais on peut les déclarer en invoquant la fonction ML `new_constant` puis on postule leurs propriétés en utilisant la fonction `new_axiom`.

2

```
PIC>new_constant(‘Add‘,":Ch # Ch # Ch # Ch # Ch # Ch ->Pr")
and new_constant(‘Succ‘,":Ch # Ch # Ch # Ch->Pr")
and new_constant(‘Copy‘,":Ch # Ch # Ch # Ch->Pr");;

() : void

PIC>let
Add_Def = new_axiom(‘Add‘,"Add(x1,z1,x2,z2,y,w) =
             Sum (In x1 ‘a‘(Out y ‘a‘(Add(x1,z1,x2,z2,y,w))))
                 (In z1 ‘a‘(Copy(x2,z2,y,w)))")
and
Copy_Def = new_axiom(‘Copy‘,"Copy(x,z,y,w) =
              Sum(In x ‘a‘(Succ(x,z,y,w)))(In z ‘a‘(Out w ‘a‘ Nil))")
and
Suc_Def = new_axiom(‘Succ‘,"Succ(x,z,y,w) = Out y ‘a‘(Copy(x,z,y,w))");;

Add_Def =
|- !x1 z1 x2 z2 y w.
    Add(x1,z1,x2,z2,y,w) =
    Sum
    (In x1 ‘a‘ (Out y ‘a‘(Add(x1,z1,x2,z2,y,w))))
    (In z1 ‘a‘(Copy(x2,z2,y,w)))
Copy_Def =
|- !x z y w.
    Copy(x,z,y,w) =
    Sum(In x ‘a‘ (Succ(x,z,y,w)))(In z ‘a‘ (Out w ‘a‘ Nil))
Suc_Def = |- !x z y w. Succ(x,z,y,w) = Out y ‘a‘ (Copy(x,z,y,w))
```

Pour montrer la correction de la spécification de l'addition en π-calcul, on doit montrer que pour tous entiers n1 et n2, la représentation de la somme (n1+n2) de ces deux entiers est faiblement équivalente au processus obtenu par la mises en parallèle de la représentation de n1, la représentation de n2 et le processus représentant la spécification de l'addition. Ceci se traduit par la formule suivante

```
!n1 n2.
  Weak_Equiv
```

```
(Nu ‘x1‘(Nu ‘z1‘(Nu ‘x2‘(Nu ‘z2‘
 (Par(PI n1 ‘x1‘ ‘z1‘)(Par(PI n2 ‘x2‘ ‘z2‘)
      (Add(‘x1‘,‘z1‘,‘x2‘,‘z2‘,‘y‘,‘w‘))))))))
(PI(n1 + n2)‘y‘ ‘w‘)
```

Pour montrer que cette formule est un thèorème on a utilisé plusieurs tactiques prédéfinit dans le système et des tactiques spécifiques que nous avons implanté pour pouvoir raisonner sur les spécifications en π-calcul. En outre on a été amené à montrer les lemmes ci-dessous qui spécifient des propriétés des constantes `Copy`, `Succ` et `Add` et dont la preuve est accomplie, également par induction. Ces lemmes sont:

```
Lemma1
|- !n. Weak_Equiv
        (Nu ‘x‘(Nu ‘z‘(Par(PI n ‘x‘ ‘z‘)(Succ(‘x‘,‘z‘,‘y‘,‘w‘)))))
       (PI(SUC n)‘y‘ ‘w‘)
Lemma2
|- !n.Weak_Equiv
        (Nu ‘x‘(Nu ‘z‘(Par(PI n ‘x‘ ‘z‘)(Copy(‘x‘,‘z‘,‘y‘,‘w‘)))))
        (PI n ‘y‘ ‘w‘)
Lemma3
|- !n.Weak_Equiv
       (Nu ‘x1‘(Nu ‘z1‘(Nu ‘x2‘(Nu ‘z2‘(Par(PI 0 ‘x1‘ ‘z1‘)
              (Par(PI n ‘x2‘ ‘z2‘)(Add(‘x1‘,‘z1‘,‘x2‘,‘z2‘,‘y‘,‘w‘)))))))
       (PI n ‘y‘ ‘w‘)
Lemma4
|-!n1. Weak_Equiv
 (Nu ‘x1‘ (Nu  ‘z1‘
        (Par (PI n1 ‘x1‘ ‘z1‘)
             (Out ‘y‘  ‘a‘ (Add(‘x1‘,‘z1‘,‘x2‘,‘z2‘,‘y‘,‘w‘))))))
  (Out ‘y‘ ‘a‘(Nu  ‘x1‘ (Nu  ‘z1‘
        (Par (PI n1 ‘x1‘ ‘z1‘) (Add(‘x1‘,‘z1‘,‘x2‘,‘z2‘,‘y‘,‘w‘))))))
Lemma5
|- !n P. Weak_Equiv
            (Nu ‘x2‘(Nu ‘z2‘(Par (Tau (Out ‘y‘ ‘a‘ P))
                                 (PI n ‘x2‘ ‘z2‘))))
            (Tau (Out ‘y‘ ‘a‘ (Nu ‘x2‘(Nu ‘z2‘(Par P (PI n ‘x2‘ ‘z2‘))))))
```

La preuve que nous allons présenter est un “script” montrant le dialogue qui s’établit entre l’utilisateur et le système, au moyen du “subgoal package”.

Le but à prouver est introduit dans le système par la fonction `g`. Puis on applique l’induction sur le premier entier au moyen de la tactique `INDUCT_TAC` qui implante cette stratégie.

```
PIC>g"!n1 n2.
      Weak_Equiv
       (Nu `x1`(Nu `z1`(Nu `x2`(Nu `z2`
            (Par(PI n1 `x1` `z1`)(Par(PI n2 `x2` `z2`)
            (Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))
     (PI(n1 + n2)`y` `w`)";;
"!n1 n2.
  Weak_Equiv
  (Nu`x1`(Nu `z1`(Nu `x2`(Nu `z2`
      (Par (PI n1 `x1` `z1`)
       (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))
  (PI(n1 + n2)`y` `w`)"
() : void
Run time: 0.1s
PIC>e(INDUCT_TAC);;
OK..
2 subgoals
"!n2.
  Weak_Equiv
  (Nu`x1`(Nu`z1`(Nu `x2`(Nu `z2`
      (Par
       (PI(SUC n1)`x1` `z1`)
       (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))
  (PI((SUC n1) + n2)`y` `w`)"
  1  ["!n2.
        Weak_Equiv
        (Nu`x1`(Nu`z1`(Nu`x2`(Nu `z2`
            (Par
             (PI n1 `x1` `z1`)
             (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))
        (PI(n1 + n2)`y` `w`)" ]

"!n2.
  Weak_Equiv
  (Nu`x1`(Nu`z1`(Nu`x2`(Nu`z2`
      (Par
       (PI 0 `x1` `z1`)
       (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))
  (PI(0 + n2)`y` `w`)"

() : void
Run time: 0.2s
Garbage collection time: 1.4s
Intermediate theorems generated: 24
```

Le sous but de base est une instance du lemme 3. Donc, il est résolu immediatement après la simplification du terme `0 + n2` par la définition de l'addition naturelle définit dans la théorie `arithmetic` qui est un ancêtre de notre théorie.

```
PIC>e(REWRITE_TAC[ADD] THEN MATCH_ACCEPT_TAC LEMMA3);;
OK..
goal proved
|- !n2.
    Weak_Equiv
    (Nu`x1`(Nu`z1`(Nu`x2`(Nu`z2`
        (Par
         (PI 0 `x1` `z1`)
         (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))))
    (PI(0 + n2)`y` `w`)

Previous subproof:
"!n2.
  Weak_Equiv
  (Nu`x1`(Nu`z1`(Nu`x2`(Nu`z2`
      (Par
       (PI(SUC n1)`x1` `z1`)
       (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))))
  (PI((SUC n1) + n2)`y` `w`)"
  1  ["!n2.
        Weak_Equiv
        (Nu`x1`(Nu`z1`(Nu`x2`(Nu`z2`
            (Par
             (PI n1 `x1` `z1`)
             (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))))
        (PI(n1 + n2)`y` `w`)" ]

() : void
Run time: 0.3s
Garbage collection time: 1.3s
Intermediate theorems generated: 26
```

Une fois que ce sous but est prouvé, le système nous présente le cas inductive. L'hypothèse inductive est présenté entre crochets. Dans la suite de la preuve on la représente uniquement par trois points. La preuve de ce cas est composé particulièrement de manipulations syntaxiques basées sur les lois algébriques du π-calcul, et sur la conversion `RED_PAR_CONV` qui implante la règle qui transforme un processus composé de deux processus mis en parallèles en leurs équivalents sous forme de choix non déterministe. Les grandes étapes de la preuve sont:

Applications de la définition de PI et ADD

5

```
PIC>e(REWRITE_TAC[definition `pi_num` `PI`]
  THEN ONCE_REWRITE_TAC[axiom `pi_num` `Add`]);;
OK..
"!n2.
  Weak_Equiv
  (Nu `x1`(Nu `z1` (Nu `x2` (Nu `z2`
      (Par (Out `x1` `à(PI n1 `x1` `z1`))
       (Par (PI n2 `x2` `z2`)
        (Sum
         (In `x1` `a `(Out `y` `à(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`))))
         (In `z1` `a`(Copy(`x2`,`z2`,`y`,`w`)))))))))
  (PI((SUC n1) + n2)`y` `w`)"
  1  [ ... ]

() : void
Run time: 0.3s
Garbage collection time: 1.3s
Intermediate theorems generated: 62
```

Manipulation syntaxique avant l'application de RED_PAR_CONV

La conversion RED_PAR_CONV simplifie un terme qui est en forme d'une composition parallèle de deux processus en forme préfixé en son équivalent calculé par l'application du théorème d'expansion. Mais avant de pouvoir l'appliquer, on effectue quelques opérations syntaxiques, à savoir une application du théorème de commutativité et d'associativité de l'opérateur de la composition parallèle.

6

```
PIC>e(MY_REWRITE_TAC[SPEC "PI n2 `x2` `z2`" PAR_COM]
  THEN MY_REWRITE_TAC[PAR_ASSOC]);;
OK..
"!n2.
  Weak_Equiv
  (Nu `x1` (Nu `z1` (Nu `x2` (Nu `z2`
       (Par
       (Par (Out `x1` `à(PI n1 `x1` `z1`))
        (Sum
         (In `x1` `a`(Out `y` `à(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`))))
         (In `z1` `a`(Copy(`x2`,`z2`,`y`,`w`)))))
       (PI n2 `x2` `z2`))))))
  (PI((SUC n1) + n2)`y` `w`)"
  1  [ ... ]

() : void
Run time: 0.4s
Garbage collection time: 1.4s
Intermediate theorems generated: 44
```

Application de RED_PAR_CONV


```
PIC>e(CONV_TAC(DEPTH_CONV RED_PAR_CONV));;
OK..
"!n2.
  Weak_Equiv
  (Nu `x1` (Nu `z1` (Nu `x2` (Nu `z2`
       (Par
       (Sum
        (Out `x1` `a`
         (Par
          (PI n1 `x1` `z1`)
          (Sum
           (In `x1` `a`(Out `y` `à(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`))))
           (In `z1` `a`(Copy(`x2`,`z2`,`y`,`w`))))))
        (Sum
         (Sum
          (In `x1`
           (VARIANT(FN_Pr(Out `x1` `a`(PI n1 `x1` `z1`)))`a`)
           (Par
            (Out `x1` `a`(PI n1 `x1` `z1`))
            (Out `y`
             (VARIANT(FN_Pr(Out `x1` `a`(PI n1 `x1` `z1`)))`a`)
             (Subt_Pr
              (Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`))
              (\n.
                ((n = `a`) =>
                 VARIANT(FN_Pr(Out `x1` `a`(PI n1 `x1` `z1`)))`a` |
                 n))))))
          (In `z1`
           (VARIANT(FN_Pr(Out `x1` `a`(PI n1 `x1` `z1`)))`a`)
           (Par
            (Out `x1` `a`(PI n1 `x1` `z1`))
            (Subt_Pr
             (Copy(`x2`,`z2`,`y`,`w`))
             (\n.
               ((n = `a`) =>
                VARIANT(FN_Pr(Out `x1` `a`(PI n1 `x1` `z1`)))`a` |
                n))))))
         (Tau
          (Par
           (PI n1 `x1` `z1`)
           (Out `y` `a`
            (Subt_Pr
             (Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`))
             (\n. ((n = `a`) => `a` | n))))))))
       (PI n2 `x2` `z2`))))))
  (PI((SUC n1) + n2)`y` `w`)"
  1  [ ... ]

() : void
Run time: 566.1s
Garbage collection time: 668.3s
Intermediate theorems generated: 272082
```

Simplifcation du terme obtenu


```
PIC>e(REWRITE_TAC[FN_Pr;FN_PI]
  THEN CONV_TAC(DEPTH_CONV (UNION_CONV string_EQ_CONV))
  THEN CONV_TAC(DEPTH_CONV VARIANT_CONV)
  THEN MY_REWRITE_TAC[NU_SUM]
  THEN ONCE_REWRITE_TAC[RES_SWAP]
  THEN REWRITE_TAC[ REWRITE_RULE[string_EQ_CONV "`x1`=`x2`";
                                 string_EQ_CONV "`x1`=`z2`";
                                 string_EQ_CONV "`z1`=`x2`";
                                 string_EQ_CONV "`z1`=`z2`"]
             (SPECL["n2:num";"`x1`";"`z1`";"`x2`";"`z2`"]RES_LIFT)]
  THEN MY_REWRITE_TAC[NU_SUM]
  THEN REPEAT NU_IN_TAC
  THEN REPEAT NU_TAU_TAC
  THEN REPEAT NU_OUT_TAC
  THEN CONV_TAC(DEPTH_CONV NU_FREE_PR_CONV)
  THEN MY_REWRITE_TAC[NIL_SUM;SUM_NIL]
  THEN REWRITE_TAC[REWRITE_RULE[Subt1_Pr](SPECL ["P:Pr";"`à"] SUBT1_PR_ID)]);;
OK..
"!n2.
  Weak_Equiv
  (Nu `x2`(Nu `z2`
    (Par
     (Tau
      (Nu `x1` (Nu `z1`
        (Par
         (PI n1 `x1` `z1`)
         (Out `y` `a`(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))
     (PI n2 `x2` `z2`))))
  (PI((SUC n1) + n2)`y` `w`)"
  1  [ ... ]

() : void
Run time: 10.8s
Garbage collection time: 5.1s
Intermediate theorems generated: 1748
```

Application des lemmes 4 et 5

9

```
PIC>e(MY_REWRITE_TAC[LEMMA2]
  THEN MY_REWRITE_TAC[LEMMA3]);;
OK..
"!n2.
  Weak_Equiv
  (Tau
   (Out `y` `a`
    (Nu `x2`(Nu `z2`
      (Par
       (Nu `x1`(Nu `z1`
         (Par(PI n1 `x1` `z1`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))
       (PI n2 `x2` `z2`))))))
  (PI((SUC n1) + n2)`y` `w`)"
  1  [ ... ]
() : void
Run time: 0.3s
Intermediate theorems generated: 37
```

Manipulation syntaxique avant l'application de l'hypothèse inductive

10

```
PIC>e(REWRITE_TAC[ADD_CLAUSES;definition `pi_num` `PI`]
  THEN REWRITE_TAC[(SYM o SPEC_ALL)(REWRITE_RULE[string_EQ_CONV "`x1`=`x2`";
                              string_EQ_CONV "`x1`=`z2`";
                              string_EQ_CONV "`z1`=`x2`";
                              string_EQ_CONV "`z1`=`z2`"]
             (SPECL["n2:num";"`x1`";"`z1`";"`x2`";"`z2`"]RES_LIFT))]
  THEN ONCE_REWRITE_TAC[RES_SWAP]
  THEN REWRITE_TAC[SYM(SPEC_ALL PAR_ASSOC)]
  THEN MY_REWRITE_TAC[SPEC
  "Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)"PAR_COM]);;
"!n2.
  Weak_Equiv
  (Tau
   (Out `y` `a`
    (Nu `x1`(Nu `z1`(Nu `x2`(Nu`z2`
        (Par
         (PI n1 `x1` `z1`)
         (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))))
  (Out `y` `à(PI(n1 + n2)`y` `w`))"
  1  [ "!n2.
        Weak_Equiv
         (Nu `x1`(Nu `z1`(Nu `x2`(Nu`z2`
           (Par
             (PI n1 `x1` `z1`)
             (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))
        (PI(n1 + n2)`y` `w`)" ]

() : void
Run time: 1.0s
Intermediate theorems generated: 352
```

Application de l'hypothèse inductive

11

```
PIC>e(ASSUM_LIST(\thl. MY_REWRITE_TAC [el 1 thl]));;
OK..
"!n2.
  Weak_Equiv
  (Tau(Out `y` `a`(PI(n1 + n2)`y` `w`)))
  (Out `y` `a`(PI(n1 + n2)`y` `w`))"
  1  [ ... ]

() : void
Run time: 0.2s
Intermediate theorems generated: 17
```

Le but auquel on est arrivé est une instance d'un théorème déja montré dans notre système et est résolu immediatement.

```
PIC>e(MATCH_ACCEPT_TAC TAU_P);;

|- !P. Weak_Equiv(Tau P)P

goal proved
|- !n2.
    Weak_Equiv
    (Tau(Out `y` `a` (PI(n1 + n2)`y` `w`)))
    (Out `y` `a`(PI(n1 + n2)`y` `w`))
....
|- !n1 n2.
    Weak_Equiv
    (Nu`x1`(Nu`z1`(Nu `x2`(Nu`z2`
        (Par
         (PI n1 `x1` `z1`)
         (Par(PI n2 `x2` `z2`)(Add(`x1`,`z1`,`x2`,`z2`,`y`,`w`)))))))
    (PI(n1 + n2)`y` `w`)
Previous subproof:
goal proved
() : void
Run time: 1.3s
Garbage collection time: 2.8s
Intermediate theorems generated: 71
```

7 Conclusion

Nous avons décrit une représentation de la théorie du π-calcul dans l'environnement de preuve HOL en suivant une approche purement *définitionnelle* afin d'assurer la cohérence de notre extension de la théorie de HOL et de permettre la construction de preuves correctes. En théorie cette approche nous permettra de prouver l'équivalence de deux processus dont l'ensemble des états est infinis. Ceci est dû essentiellement au fait qu'on ne manipule que la representation abstraite des processus. Ces manipulations sont basées sur les règles algèbriques du π-calcul.

Dans [5], un environnement de preuve pour CCS en HOL est présenté. Cette approche est étendue au π-calcul par *T. Melham* dans [11]. Notre travail diffère de la représentation de [11] dans le choix de la syntaxe du π-calcul que nous avons codé dans HOL. Par exemple, pour spécifier des comportements récursifs, nous avons choisi l'opérateur `Rec` plutôt que d'utiliser l'opérateur !(*bang*) parceque `Rec` semble plus naturel à utiliser dans les spécifications des systèmes concurrents. Un autre point de différence entre notre codage et celui de *T. Melham* concerne l'opérateur `Oute` qui nous a facilité la formalisation du théorème d'expansion. Dans [11], l'auteur n'a pas présenté sa formalisation du théorème d'expansion dans HOL . Ce qui rend toute comparaison impossible sur l'utilité de l'opérateur `Oute`. Un autre point de différence entre ces deux travaux réside dans la nature de l'équivalence codé dans HOL . Dans [11], l'auteur présente la définition de l'équivalence basée sur l'instanciation retardé, par contre dans notre travail on a codé l'équivalence précoce dans sa version forte et faible, et on a présenté une définition d'une congruence observationnelle. Nous avons implanté un certain nombre d'outils permettant d'automatiser certaines opérations sur les π-termes.

En perspective, nous souhaitons développer des stratégies de preuves générales relatives

à une classe particulière de processus, un travail dans ce sens est en cours. En se basant sur la stratégie décrite dans [29], on développe une procédure automatique pour décider l'équivalence précoce de deux processus finis. Ceci serait un premier pas dans ce que nous voulons accomplir, à savoir si les algorithmes implanté dans les systèmes de preuve automatiques, tel que le MBW [34] peuvent être formalisé en tant que règles d'inférences pour les adapter à notre cadre. Une autre possibilité est de voir si on peut avoir un système *hybride* où on peut décomposer la preuve d'un problème en des parties dont la preuve peut être accomplie *interactivement* et des parties dont elle peut être accomplie *automatiquement*. Ceci suppose que les deux systèmes implantent la même notion d'équivalence ou des équivalences différentes mais dont on connaît le lien existant entre elles. Enfin, nous souhaitons tester notre approche sur des situations réelles tel que les protocoles de communications.

Remerciements

Je tiens à remercier le docteur *T. Melham* pour avoir partagé avec moi le code de la fonction qui permet de génerer de nouveaux canaux et pour sa gentillesse à répondre à mes messages électroniques, et mon directeur de recherche R. Amadio pour sa disponibilité et ses précieuses remarques.

References

[1] F Moller. The Edinburgh Concurrency Workbench (version 6.0). *University of Edinburgh*, August 1991.

[2] J C Godskesen, K G Larsen, and M Zeeberg. TAV users manual. Technical report, Aalborg University Center, Danmark, 1989.

[3] R De simone and D Vergamini. Aboard AUTO. Technical Report 111, Institut National de Recherche en Informatique et Automatique, 1989.

[4] H Lin. PAM: A Process Algebra Manipulator. Technical Report 2/91, University of Sussex, 1991.

[5] M Nesi. A formalisation of the CCS process algebra in Higher Order Logic. Technical Report 278, Computer Laboratory, University of Cambridge, December 1992.

[6] A J Camilleri. Mechanizing CSP trace theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.

[7] P Inverardi and C Priami. Evaluation of tools for the analysis of communicating systems. Bulletin of the EATCS no 45, 1992. pp. 158-185.

[8] R Milner, J Parrow, and D Walker. A calculus of mobile process, part 1-2. *Information and Computation*, 100(1):1–77, 1992.

[9] R Milner. Communication and Concurrency. In *Lecture Notes in Computer Science*. Springer Verlag, 1980. 92.

[10] M J C Gordon and T F Melham. *Introduction to HOL: A theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[11] T M Melham. A mechanized theory of π-calculus in HOL. Technical Report 244, Computer Laboratory, University of Cambridge, January 1992.

[12] M J C Gordon. HOL A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P.Subrahmanyam, editors, *VLSI Specification, verification and synthetis*, pages 73–128, Boston, 1988. Kluwer Academic Publishers.

[13] M J C Gordon, A J Milner, and C P Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York (NY, USA), 1979.

[14] G Cousineau, L C Paulson, G Huet, R Milner, M Gordon, and C Wadsworth. *The ML Handbook*. INRIA, Rocquencourt, May 1985.

[15] R Milner and M Tofte. *The definition of Standard ML*. The MIT press, Cambridge, Massachusetts and London, England, 1991.

[16] L C Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[17] A Church. A formulation of a simple theory of type. *Journal of Symbolic Logic*, 5:55–68, 1940.

[18] T M Melham. The HOL sets library. Technical report, Cambridge Computer Laboratory, 1992.

[19] T M Melham. The HOL string library. Technical report, Cambridge Computer Laboratory, 1992.

[20] T M Melham. A package for inductive relation definitions in HOL. In P.J. Windly, M. Archer, K.N. Levitt, and J.J Joyce, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 350–357. IEEE Computer Society Press, 1992.

[21] R J Boulton. The HOL taut library. Technical report, Cambridge Computer Laboratory, 1991.

[22] T M Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[23] L C Paulson. Logic and Computation- interactive proof with Cambridge LCF. *Cambridge Tracts in Theoretical Computer Science (2)*, 1987. Cambridge University Press.

[24] L C Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[25] R Milner and D Sangiorgi. Barbed bisimulation. In Kuich, editor, *Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 685–707, Wein,Austria, July 1992. Springer-Verlag.

[26] R Amadio. A uniform presentation of CHOCS and π-calculus. Research Report 1726, Inria-Lorraine, 1992.

[27] D Sangiorgi. *Expressing mobility in process algebras: first-order and higher order paradigms*. PhD thesis, University of Edinburgh, September 1992.

[28] D Sangiorgi. A theory of bisimulation for the π-calculus. In E. Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142, Hildeshein, Germany, August 1993. Springer-Verlag.

[29] R Amadio and O Aït-Mohamed. An analysis of π-calculus bisimulation. Technical Report 94-2, ECRC, 1994.

[30] D M R Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Karlsruhe, March 1980. Springer-Verlag.

[31] A Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.

[32] O Aït-Mohamed. The HOL π-calculus library. Unpublished note, 1994.

[33] R Milner. The polyadic π-calculus: a tutorial. Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, The King's Buildings, Edinburgh EH9 3 JZ, UK, October 1991.

[34] B Victor and F Moller. The mobility workbench –a tool for the π-calculus–. In David L. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440, Stanford, California, USA, June 1994. Springer-Verlag.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399