Formaliser l'exécution de *LeanMachines* à l'aide du π-calcul

Danaël Carbonneau[1]

[1]*Laboratoire d'Informatique de Paris 6 – Sorbonne Université, CNRS, UMR 7606 LIP6, 4 place Jussieu, Paris Cedex 05, 75252, France*

Résumé

LeanMachines [CP26] est un *framework* s'inspirant de la méthode Event B afin de spécifier des programmes réactifs dans l'assistant de preuves *Lean4*. En *LeanMachines*, il est possible de spécifier des événements, de prouver des propriétés sur ces derniers, mais également de composer ces spécifications. Il n'est cependant pas encore possible de faire de même pour les machines qui exécutent ces événements : le cadre actuel ne formalisant que leurs transitions, et pas leur comportement à l'exécution.

Afin de résoudre ce problème, nous proposons d'intégrer des machines, dont les événements ont été spécifiés en *LeanMachines*, dans des processus de π-calcul. Cette approche permettrait à la fois de spécifier et raisonner sur des réseaux composés de plusieurs machines interagissant entre elles, mais aussi de pouvoir exprimer et vérifier des propriétés sur leur exécution à l'aide de techniques issues du domaine des algèbres de processus.

1 Introduction

Ce travail se place dans le cadre de *LeanMachines*, un *framework* pour la modélisation de systèmes réactifs basés sur une notion d'états, écrit dans l'assistant de preuves *Lean4*[MU21]. Le *framework* s'inspire de la méthode formelle *Event B*. Dans cette méthode formelle, un système réactif est représenté par une machine avec un état et des transitions gardées, appelées *événements*. On y suit un principe de correction par construction : un invariant est spécifié sur l'état de la machine et les événements définis sur une machine doivent vérifier un certain nombre d'obligations de preuves (ne pas briser l'invariant, respecter les principes d'une notion de raffinement, *etc.*) afin de garantir la correction du système vis-à-vis de sa spécification.

Dans [CP26], nous avons présenté une approche visant à composer les événements à l'aide de combinateurs algébriques, issus de la programmation fonctionnelle. Cette approche répondait à plusieurs problèmes. D'une part, nous souhaitions pouvoir réutiliser du code, ainsi que des preuves, la composition d'événements préservant leur propriété de *safety*. D'une autre, les combinateurs algébriques permettent d'avoir un langage dédié pour décrire le flot de contrôle des événements. Cependant, cette approche ne permet que de composer les événements d'une seule machine. Cet article vise à donner une sémantique à l'exécution des machines de *LeanMachines*, afin de pouvoir les manipuler, les composer et raisonner sur leur comportement.

Pour ce faire, nous intégrons les machines, spécifiées au préalable en *LeanMachines* à des processus de π-calcul [SW01], une algèbre de processus qui permet de décrire des processus concurrents et réactifs, et de raisonner formellement sur ces derniers. Le π-calcul permet notamment d'exprimer les communications faites entre processus, qui sont les transitions réalisables par un système. En intégrant nos machines au π-calcul, nous pouvons ainsi nous concentrer sur la manière dont ces dernières s'exécutent, c'est-à-dire leurs comportements observables (transitions). Nous pouvons ainsi d'une part profiter de l'expressivité du π-calcul du point de vue du contrôle et des communications. D'une autre, ce cadre nous permettrait d'utiliser les outils théoriques du domaine des algèbres de processus, afin de vérifier des propriétés sur l'exécution de nos systèmes.

Dans cet article, nous présentons les travaux en cours sur cette formalisation. Dans la section 2, nous présentons le formalisme (syntaxe du langage et sémantique de transition). Puis, la section 3, nous présentons des exemples de processus exprimables dans ce formalisme. Enfin, la section 4 présente les pistes envisagées pour la suite de ce travail.

2 Étendre le π-calcul avec les *LeanMachines*

2.1 *LeanMachines*

En *LeanMachines*, la spécification d'un modèle se fait en deux temps. On définit d'abord une machine, représentée par le type de son état interne M, le type de son contexte CTX et un invariant portant sur cet état interne. Par la suite, on définit les différentes transitions réalisables par la machine, c'est-à-dire ses événements. Un événement est défini par une garde, qui définit les conditions pour que la transition soit valide, et une action, qui est une fonction qui prend en argument l'entrée de l'événement (de type α), l'état courant (de type M), ainsi qu'une preuve que la garde de l'événement est respectée pour ces entrées (de type $guard\ m\ x$), et renvoie la sortie de l'événement (de type β) et l'état suivant de la machine, sous forme de produit. Le code *Lean4* présenté en *(fig. 1)* implémente cette définition.

```
class Machine (CTX : outParam (Type u)) (M) where
  context : CTX
  invariant : M → Prop
structure Event (M) [Machine CTX M] (α : Type) (β : Type)
  guard (m : M) (x : α) : Prop
  action (m : M) (x : α) (grd : guard m x): (β × M)
```

FIGURE 1 – Encodage des machines et des événements dans la bibliothèque *LeanMachines*

2.2 Syntaxe

Dans notre encodage, chaque événement $ev : Event\ M\ \alpha\ \beta$ est associé à un nom (ev) dans le π-calcul. Ce nom représente un canal qui reçoit deux entrées : l'argument de la transition et le canal de retour qui permet de produire une sortie. $\lfloor m \rfloor_{ctx,E}$ représente l'instance d'une machine d'état interne $m : M$, de contexte $ctx : CTX$ pour laquelle a été défini un ensemble d'événements E. Nous ajoutons cette représentation des machines à la syntaxe du π-calcul polyadique avec réplication réactive [1] inspiré de [SW01]*(fig. 2)*.

$$P ::= (\nu z)P \mid P \| P \mid !x(\tilde{y}).P \mid M \mid \lfloor m \rfloor_{ctx,E} \qquad \pi ::= \tau \mid \bar{x}\langle \tilde{y} \rangle \mid x(\tilde{y}) \mid [x = y].\pi$$
$$M ::= M + M \mid \pi.P \mid \mathbf{0}$$

FIGURE 2 – Syntaxe de notre extension du π-calcul

Nous utilisons $\tilde{y}$ pour dénoter $y_1, \ldots, y_n$ pour un certain n. L'opérateur $\|$ représente la mise en parallèle de processus, $+$ le choix non déterministe, et $(\nu z)P$ permet de rendre privé le nom z dans P. Un processus peut être préfixé par une action invisible τ, l'envoi d'un nombre arbitraire de noms sur un canal $\bar{x}\langle \tilde{y} \rangle$, la réception de noms sur un canal $x(\tilde{y})$, ou bien par une garde $[x = y].\pi$ sur un autre préfixe.

2.3 Sémantique

Nous étendons les règles de transitions du π-calcul avec une règle permettant de donner une sémantique aux transitions gardées spécifiées en *LeanMachines (fig. 3)*. $m, x \xrightarrow{ev} y, m'$ signifie que depuis un état m et avec une entrée x, la garde de ev est vraie, et que l'action renvoie une sortie y et un nouvel état m'. La règle permet ainsi d'intégrer les transitions gardées spécifiées en *LeanMachines* à la sémantique de notre algèbre de processus.

Puisque la seule règle de transition permettant de modifier l'état des machines est (EVENT), et que cette transition ne peut se faire que si la garde est vraie, les garanties de correction vis-à-vis de l'invariant prouvées en *LeanMachines* sont préservées dans notre sémantique. Si l'invariant est vrai pour toutes les machines d'un processus P, alors pour toute action μ et tout processus P' tels que $P \xrightarrow{\mu} P'$, l'invariant est vrai pour toutes les machines de P'.

1. les canaux peuvent avoir plusieurs arguments et la réplication ne peut se faire qu'après réception sur un canal.

$$\frac{}{!a(X).P \xrightarrow{a\ V} !a(X).P \| P\{V/X\}} \quad (\text{REPL}) \qquad \frac{P \xrightarrow{\bar{a}\ X} P' \qquad Q \xrightarrow{a\ X} Q'}{P\|Q \xrightarrow{\tau} P'\|Q'} \quad (\text{COMM-L})$$

$$\frac{}{a(x_1,\ldots,x_n).P \xrightarrow{a\ v_1,\ldots,v_n} P\{v_1/x_1,\ldots,v_n/x_n\}} \quad (\text{IN}) \qquad \frac{}{\bar{a}\langle X\rangle.P \xrightarrow{\bar{a}\ X} P} \quad (\text{OUT})$$

$$\frac{P \xrightarrow{\bar{a}\ n} P'}{\nu(n)P \xrightarrow{\bar{a}(n)} P'} \quad (\text{OPEN}) \qquad \frac{ev \in E \qquad m, x \xrightarrow{ev} y, m'}{\lfloor m \rfloor_{ctx,E} \xrightarrow{ev\ x,s} \lfloor m' \rfloor_{ctx,E} \mid \bar{s}\langle y\rangle} \quad (\text{EVENT})$$

FIGURE 3 – Extrait des règles de transition du π-calcul [SW01], et nouvelle règle pour les événements (EVENT)

3 Exemples de processus

3.1 Communication entre machines

Ce formalisme nous permet généraliser l'approche présentée dans [CP26], qui visait à combiner les événements entre eux pour en définir de plus complexes. Là où dans *LeanMachines*, les événements combinés sont définis sur une seule machine, il est désormais possible de combiner des événements de plusieurs machines mises en parallèle dans un processus. Nous pouvons, par exemple, décrire un processus qui permet à deux machines de communiquer entre elles *(fig. 4)* : La sortie du premier événement est utilisée pour déclencher le second.

$$!sync(x, s, ev_1, ev_2).(\nu c)\overline{ev_1}\langle x, c\rangle.c(y).\overline{ev_2}\langle y, s\rangle$$

FIGURE 4 – Processus permettant de faire un enchaînement entre deux événements

3.2 Partage d'événements et renommage

Une des motivations principales à notre formalisme était de permettre de décrire des situations où plusieurs instances de machines sont présentes en parallèle dans un processus [2]. Il est alors possible d'avoir un processus où deux instances d'une même machine sont présentes. Ces deux instances partageant la même collection d'étiquettes (événements), elles offrent les mêmes transitions à l'extérieur *(fig. 5)*.

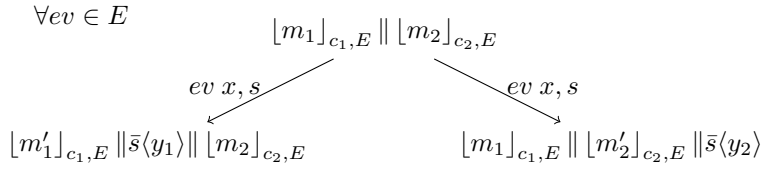

FIGURE 5 – Comportement de deux instances de machines partageant leurs événements

Cependant, si on souhaite qu'un événement soit spécifique à une instance de machine, il est possible d'écrire un processus ayant ce comportement. En rendant privés les événements à l'aide de l'opérateur de restriction du π-calcul, on peut empêcher leur utilisation par l'extérieur. Afin de pouvoir contacter la machine sur ces canaux, il faut alors faire une indirection : un nouveau nom, visible de l'extérieur, qui envoie la demande d'événement en interne *(fig. 6)*. Pour éviter les collisions entre noms liés et noms libres, il faut cependant α-convertir le processus. Nous notons $E\{^{ev'}/_{ev}\}$ la collection d'événements où le nom ev a été remplacé par ev' :

$$((\nu\, ev)(\lfloor m_1 \rfloor_{c_1,E} \| !ev'(x,s)).\overline{ev}\langle x, s\rangle) \| \lfloor m_2 \rfloor_{c_2,E} \overset{\alpha}{=} ((\nu\, ev'')(\lfloor m_1 \rfloor_{c_1,E\{^{ev''}/_{ev}\}} \| !ev'(x,s)).\overline{ev''}\langle x, s\rangle) \| \lfloor m_2 \rfloor_{c_2,E}$$

2. Il s'agit d'une composition parallèle, qui correspond à la décomposition de machines proposée dans [But09]

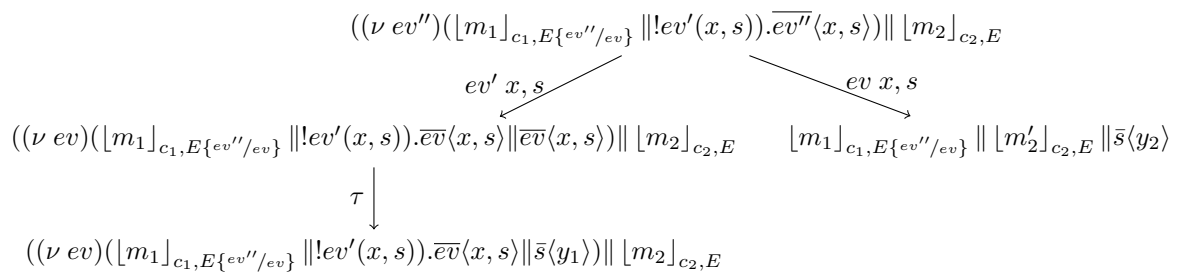

FIGURE 6 – Comportement de deux instances de machines partageant tous leurs événements sauf ev

3.3 Machines inactives et initialisation dynamique

En *LeanMachines*, les machines sont définies avec une valeur par défaut, pour laquelle l'invariant n'est pas forcément respecté. Un événement d'initialisation sert alors à fixer un état qui respecte l'invariant.

Dans notre formalisme, nous souhaitons représenter une instanciation de machines dynamique, avec des valeurs initiales fournies par l'extérieur, en utilisant les événements d'initialisation de *LeanMachines*. Nous considérons les trois processus suivants :

$$P_1 \stackrel{\text{def}}{=} \lfloor x_0 \rfloor_{ctx, E\cup\{init\}} \qquad P_2 \stackrel{\text{def}}{=} (\nu E)\, \lfloor m \rfloor_{ctx,E} \qquad P_3 \stackrel{\text{def}}{=} (\nu E)\, \lfloor m \rfloor_{ctx, E\cup\{init\}}$$

P_1 permet à l'extérieur d'initialiser la machine. Cependant, il permet également de faire toutes les transitions de E, alors même que l'état x_0 peut être invalide. Afin d'empêcher à une machine non initialisée de faire des transitions, il nous faut un mécanisme permettant de rendre une machine inactive, c'est-à-dire invisible depuis l'extérieur et un moyen de la rendre visible après initialisation. Le processus P_2 n'offre aucune transition à l'extérieur : on rend une machine inactive en rendant privé tous ses noms d'événements. En considérant qu'un événement d'initialisation renvoie les noms des événements de la machine initialisée, on peut encoder l'initialisation de machines dynamique dans notre formalisme : grâce à la règle (OPEN) *(fig. 3)*, une fois la machine initialisée, l'envoi sur un canal public s de la collection d'événements E rend la machine publique. Ainsi, P_3 est un processus qui représente une machine qui peut être initialisée, mais dont aucun autre comportement n'est observable avant initialisation *(fig. 7)*.

$$\text{Si } \_, x \xrightarrow{init} E, m \text{ alors} \quad (\nu E)\, \lfloor x_0 \rfloor_{ctx, E\cup\{init\}} \xrightarrow{init\ x,s} (\nu E)\, \lfloor m \rfloor_{ctx, E\cup\{init\}} \| \bar{s}\langle E\rangle \xrightarrow{\bar{s}(E)} \lfloor m \rfloor_{ctx, E\cup\{init\}}$$

FIGURE 7 – Initialisation dynamique d'une machine

4 Conclusion et travaux futurs

Dans cet article, nous avons présenté un cadre pour raisonner sur la sémantique de *LeanMachines* en intégrant les spécifications au π-calcul. En nous basant sur le π-calcul, nous nous concentrons sur la manière dont les machines peuvent communiquer, et sur leur comportement observable, par leur sémantique de transition. Cette approche nous permet de profiter de la grande expressivité permise par les algèbres de processus, notamment pour composer les spécifications.

L'idée de décrire le contrôle d'une machine spécifiée en *Event B* à l'aide d'une algèbre de processus, afin de séparer le modèle de calcul du modèle d'exécution est présente dans [STW10], où les auteurs se concentrent sur l'ordonnancement d'événements d'une même machine. Par ailleurs, notre démarche visant à utiliser une algèbre de processus et les outils associés afin de vérifier des propriétés sur des échanges, est proche de celle présentée dans [OR16] : les auteurs utilisent des types de session multipartites afin de modéliser les communications, puis les traduisent en une machine abstraite *Event B*. Enfin, notre approche visant à donner une sémantique de transition à *LeanMachines* se rapproche de la manière dont la sémantique de *Event B* est définie à l'aide de *labelled transition systems* dans [Cas21] afin de prouver des métapropriétés sur le raffinement.

Nous considérons plusieurs axes pour la suite des travaux. Tout d'abord, nous aimerions identifier des propriétés et garanties de sécurité sur les processus décrits dans notre formalisme, afin de proposer des analyses statiques, par exemple par des systèmes de type. Nous souhaitons également définir une notion d'équivalence de processus. Enfin, nous travaillons à un encodage du π-calcul en *Lean4*, qui s'appuie sur la bibliothèque *CSlib* [BCM$^+$26], ce qui nous permettrait, à terme, de pouvoir ajouter ce formalisme à *LeanMachines*, et ainsi de pouvoir raisonner sur l'exécution des machines dans notre *framework*.

Références

[BCM+26] Clark Barrett, Swarat Chaudhuri, Fabrizio Montesi, Jim Grundy, Pushmeet Kohli, Leonardo de Moura, Alexandre Rademaker, and Sorrachai Yingchareonthawornchai. CSLib : The Lean Computer Science Library, February 2026. arXiv :2602.04846 [cs].

[But09] Michael Butler. Decomposition Structures for Event-B. In Michael Leuschel and Heike Wehrheim, editors, *Integrated Formal Methods*, volume 5423, pages 20–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title : Lecture Notes in Computer Science.

[Cas21] Pierre Castéran. An Explicit Semantics for Event-B Refinements. In Yamine Ait-Ameur, Shin Nakajima, and Dominique Méry, editors, *Implicit and Explicit Semantics Integration in Proof-Based Developments of Discrete Systems : Communications of NII Shonan Meetings*, pages 155–173. Springer, Singapore, 2021.

[CP26] Danaël Carbonneau and Frederic Peschanski. LeanMachines : State-based Modeling with Refinement (a Lean4 Framework). *Form. Asp. Comput.*, February 2026. Just Accepted.

[MU21] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28 : 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, pages 625–635, Berlin, Heidelberg, July 2021. Springer-Verlag.

[OR16] Carlos Olarte and Camilo Rueda. Session types for communicating systems in event-B. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1686–1693, Pisa Italy, April 2016. ACM.

[STW10] Steve Schneider, Helen Treharne, and Heike Wehrheim. A CSP Approach to Control in Event-B. October 2010.

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus : A Theory of Mobile Processes*. Cambridge University Press, Cambridge, 2001.