Support linguistique à la constitution de bibliothèques de méta-composants

Claude MICHEL — Jean-Pierre REGOURD

I3S - Université de Nice-Sophia Antipolis
Bât. ESSI
650, route des Colles
BP. 145
06903 SOPHIA ANTIPOLIS Cedex
{cpjm, rjp}@lolita.unice.fr

RÉSUMÉ. *Alors que la réflexivité semble être un candidat prometteur pour résoudre de nombreux problèmes, son utilisation n'est pourtant pas aussi répandue qu'on pourrait l'espérer. Nous croyons que cela est dû à l'accroissement de la complexité qu'elle introduit. Afin d'en faciliter l'utilisation, nous proposons de construire des bibliothèques de programmes réflexifs. Cependant, les langages d'acteurs ne disposent pas des outils nécessaires à leur construction. Aussi avons nous introduit le support linguistique requis pour la construction de ces bibliothèques. Notre proposition est basée sur une spécification du lien causal. Chaque comportement de méta-composant, un acteur appartenant à l'ensemble des acteurs qui composent le méta, inclut une spécification des éléments réifiés. En modifiant la vision du lien causal, cette spécification offre une séparation claire entre les différents niveaux d'utilisation et libère l'utilisateur final de l'obligation de connaître parfaitement le méta-protocole.*

ABSTRACT. *Although reflexion is promising as a candidate to solve a large set of problems, its use is not as widely spread as it could be expected. We believe this is due to the increase of complexity it introduces. To ease the use of reflexion, we propose to build some set of libraries of reflective programs. Unfortunately, actor languages do not hold the required tools to build them. That is why we introduced the linguistic support required to build such libraries. Our proposal is based on a specification of the causal link. Each behavior of a meta-component, an actor which is a part of the meta of an actor, includes such a specification. With this modification of the causal link's sight, this specification introduces a clear separation between differents levels of uses and releases the end-user from the duty to perfectly know the meta-protocol.*

MOTS-CLÉS : *acteurs, réflexivité, bibliothèques de méta-composants.*

KEY WORDS: *actors, reflexion, meta-components libraries.*

1. Introduction

La réflexivité [SMI 83] apporte aux langages d'acteurs [AGH 86] la souplesse nécessaire à l'étude et à l'introduction de nouveaux mécanismes linguistiques. De nombreux travaux se sont d'ailleurs intéressés à l'introduction de la réflexivité dans cette classe de langages [YON 89, TAN 90, MAS 92, HAS 93, GIR 93, MAS 96, MIC 97, MIG 98].

Cependant, la réflexivité reste peu répandue au regard de ses multiples bénéfices. Nombreuses sont les applications qui pourraient tirer avantage de son utilisation. En particulier, l'intelligence artificielle, discipline en constante évolution, devrait trouver dans les interprètes réflexifs l'adaptabilité qu'elle réclame tant au niveau de l'introduction de mécanismes linguistiques spécifiques que dans l'utilisation des capacités d'introspection naturellement offertes par ces systèmes. Or, si nous croyons au fort potentiel de la réflexivité, un rapide état des lieux amène à une toute autre conclusion.

Étant donné son potentiel, si la réflexivité n'est pas très largement utilisée, c'est qu'elle introduit un degré supplémentaire de complexité dans la réalisation d'une application. C'est pourtant dans les réalisations les plus complexes qu'elle a le plus de chances de montrer son utilité. L'accumulation de différentes sources de complexités constitue un frein à une large utilisation de la réflexivité. La complexité des applications candidates à l'utilisation d'interprètes réflexifs ne pouvant pas être réduite, c'est donc à celle de la réflexivité même qu'il nous faut nous attaquer.

L'écriture de programmes réflexifs est une tâche délicate. Un interprète réflexif constitue un système ouvert dont l'ensemble des mécanismes linguistiques peuvent être modifiés. Toute modification introduite au sein du système est susceptible d'en perturber le fonctionnement au point de le rendre inapte à l'exécution du plus simple des programmes. Pour éviter cet écueil, le programmeur doit posséder une maîtrise parfaite de l'interprète réflexif qu'il modifie [PAZ 97]. Si la représentation qu'offre l'interprète de lui-même constitue déjà un pas vers la simplification de la programmation réflexive, elle n'est pourtant pas suffisante pour permettre une utilisation aisée de la réflexivité.

Par ailleurs, le point de vue de l'utilisateur de programmes réflexifs est trop rarement pris en compte. Ce dernier a presque toujours l'obligation de connaître parfaitement le modèle de réflexivité pour faire bon usage des programmes réflexifs proposés [PAZ 97]. Un modèle de réflexivité complet devrait permettre une distinction claire des différents niveaux d'utilisation.

Pour faciliter l'utilisation de la réflexivité, nous proposons de construire des bibliothèques de méta-composants prêts à être assemblés. La mise en œuvre des capacités réflexives d'un interprète passe alors par l'assemblage de composants pré-définis qui réduisent d'autant le travail d'écriture de programmes réflexifs et, conséquemment, les sources possibles d'erreurs. Ces bibliothèques de méta-composants nous semblent être un outil capable de favoriser l'utilisation extensive de la réflexivité en offrant un environnement plus sécurisé.

Dans cet article, nous décrivons le support linguistique que nous avons défini afin de permettre la construction de bibliothèques de *méta-composants*, les entités constitutives du méta d'un acteur. Après une analyse du problème posé par l'écriture de bibliothèques de méta-composants, nous présentons notre architecture à base de méta-composants que nous illustrerons d'exemples. Enfin nous comparerons notre approche avec l'existant.

2. Réflexivité et bibliothèques

Le concept de bibliothèque est largement utilisé dans les langages de classes pour lesquels elles foisonnent. L'héritage y constitue l'outil idéal qui permet d'adapter l'offre aux besoins au moyen de la spécialisation ou de l'héritage multiple. Le développement rapide d'applications, l'une des clefs du succès des langages de classes, est ainsi rendu possible.

La réflexivité dans les langages de classes est naturellement supportée par le prolongement de la relation d'instanciation de l'objet à la classe. En tant que classe, la méta-classe offre les mêmes avantages que la classe. Il est donc courant de constituer des bibliothèques de méta-classes [GIR 93, McA 95] que l'utilisateur de base peut mettre en œuvre pour adapter le langage à ses propres besoins sans qu'il lui soit nécessaire de développer ses propres méta-classes.

Cependant, si les langages de classes fournissent, non sans soulever des problèmes spécifiques, le support linguistique nécessaire à la constitution de bibliothèques, les langages d'acteurs souffrent de l'absence d'outils capables de supporter un tel concept.

Les langages d'acteurs [AGH 86] ne disposent pas de classes. Le comportement d'un acteur, bien que partiellement comparable à une classe, reste dépourvu des outils qui font la spécificité de la classe. En particulier, et pour satisfaire au besoin d'autonomie de l'acteur, l'héritage ne s'applique pas au comportement. Aussi, les seuls travaux qui suggèrent la construction de bibliothèques dans le monde des acteurs sont en fait des implémentations basées sur un langage de classes [GIR 93, McA 95, McA 96].

La réflexivité dans les langages d'acteurs est le plus souvent associée au concept de méta-acteur [MAT 91, MAS 92, MIC 97]. Le méta-acteur est alors un acteur qui implémente sa dénotation i.e. l'acteur dont il explicite les mécanismes. Une modification du méta-acteur entraîne une modification conséquente des mécanismes de sa dénotation. Si cette architecture a fait ses preuves, elle représente pourtant un obstacle à la construction de bibliothèques. En effet, elle concentre au sein d'un même objet l'ensemble des mécanismes d'une dénotation. Pour construire une bibliothèque, il faudrait donc non seulement décrire l'ensemble des méta-acteurs contenant les mécanismes élémentaires proposés, mais aussi l'ensemble des combinaisons possibles de ces mécanismes. Une telle bibliothèque, si elle est envisageable, atteindrait une taille disproportionnée par rapport à son utilité. Et, par exemple, son utilisation serait rendue difficile, chaque combinaison possible de fonctionnalités portant un nom spécifique. Elle serait donc contraire à l'objectif de simplification que nous nous sommes fixé.

Afin de réduire la taille de notre bibliothèque, il nous faut la constituer non pas de méta-acteurs, mais des éléments constitutifs de ces méta-acteurs. Une bibliothèque devant servir de base à la programmation réflexive doit proposer un ensemble de modifications des mécanismes de l'acteur qui satisfont les utilisations les plus courantes. Il apparaît naturel d'y inclure un certain nombre de mécanismes de réception des messages, un certain nombre de mécanismes d'activation des messages reçus, etc. ainsi qu'une forme de *composition* de ces mécanismes. En combinant ces éléments, l'utilisateur de notre interprète peut alors assembler un méta d'acteur correspondant à ses propres besoins et cela, sans qu'il lui soit vraiment nécessaire d'écrire une ligne de code.

Si, comme cela l'a été proposé dans plusieurs travaux [GIR 93, MIG 98], il est possible de décomposer le méta d'un acteur à l'aide de plusieurs acteurs dont chacun représente une des activités de la machine acteur [1], le couplage faible des acteurs entre eux n'offre aucune garantie quant au respect du protocole proposé. Le seul garant du bon fonctionnement du système est le concepteur de la bibliothèque. Et si l'utilisateur doit combiner les éléments de la bibliothèque avec des éléments qu'il a créés, rien dans le langage ne lui assure la compatibilité de ses propres éléments avec ceux déjà existants. Enfin, une décomposition du méta d'un acteur uniquement basée sur un protocole de communication acteur transfère sur l'utilisateur l'entière responsabilité de la mise en place des éléments constitutifs du méta. Il ne peut alors se passer de la connaissance complète de ce protocole.

Pour satisfaire à notre objectif de simplification, nous pensons qu'il est nécessaire d'introduire un support linguistique adéquat. L'objet de ce dernier est de permettre une utilisation très simplifiée de la réflexivité sans qu'il soit nécessaire de maîtriser l'auto-représentation de l'interprète. Une bibliothèque correctement construite à partir d'un tel support doit offrir un ensemble complet de fonctionnalités que l'utilisateur final met en place aussi simplement que possible.

L'examen de programmes réflexifs montre que la mise en œuvre de la réflexivité se fait non autour des structures qu'elle offre mais autour des *fonctionnalités* qui doivent être modifiées. Il s'agit d'abord de modifier le protocole de réception des messages. Ce n'est que dans une seconde étape que ce besoin est traduit en termes de méta-acteur.

Afin de distinguer clairement les différents niveaux d'utilisation de la réflexivité, notre approche s'appuie sur une modification de la vision habituelle du lien causal, le lien qui lie une méta-entité à sa dénotation en s'assurant de la cohérence entre représenté et représentant. Plutôt que de baser le lien causal sur une simple relation d'implémentation, la méta-entité implémentant la dénotation, nous introduisons un support linguistique qui laisse la méta-entité expliciter sa relation avec sa dénotation. Cette explicitation s'appuie sur une spécification des mécanismes élémentaires de la dénotation réifiés par la méta-entité ainsi que de la façon dont ils sont réifiés.

Notre architecture permet d'organiser les éléments du méta, les méta-composants, non autour des aspects structurels ou opérationnels mais autour des fonctionnalités qu'ils introduisent. La place d'un méta-composant dans le méta d'un acteur n'est plus de la responsabilité de l'utilisateur final mais de son concepteur.

1. Par machine acteur, il faut comprendre l'ensemble des mécanismes du système qui donnent à un acteur ses capacités d'acteur : autonomie, capacité de calcul, aptitude à recevoir des messages etc. Cette notion de machine acteur est définie plus précisément dans [AGH 86].

3. Support linguistique

La constitution de bibliothèques comme facteur simplificateur de l'utilisation de la réflexivité nécessite la mise en place d'outils linguistiques dédiés qui rendent implicites les particularités de l'implémentation. En effet, l'utilisation des éléments de la bibliothèque doit être aussi simple que possible. Dans ce but, nous proposons ici une décomposition de la machine acteur qui explicite les éléments réifiés par chacune des composantes du méta.

Cependant, plutôt que de construire cette décomposition comme un système d'acteurs [AGH 86] liés par un protocole particulier et dont la configuration est fixée à l'avance, notre décomposition est basée sur la description du lien causal. Cette dernière, détenue par les entités qui composent le méta d'un acteur, offre un degré de souplesse supplémentaire dans l'expression et l'agencement des méta-composants sans imposer à l'utilisateur final la connaissance du protocole qui les lie.

3.1. *Décomposition de la machine acteur*

Dans les modèles de réflexivité courants, la machine acteur d'un acteur est représentée par un seul et unique acteur, le méta-acteur [MAS 92, HAS 93, MIC 97]. À travers ses accointances et ses méthodes, le méta-acteur traduit les différents aspects de la machine. Les accointances représentent l'état de la machine acteur i.e. des variables comme la liste des messages en attente de traitement ou l'adresse du comportement actuel de la dénotation. Les méthodes correspondent aux activités de la machine acteur comme la réception d'un message ou son activation.

L'activité de la machine acteur se traduit par le déclenchement et l'enchaînement des méthodes qui composent le méta-acteur. Le déclenchement initial de la machine acteur est dû à la demande de réception d'un message provenant de l'extérieur. Les méthodes s'enchaînent ensuite les unes après les autres réalisant ainsi le travail de la machine acteur.

Cette vision monolithique, en imposant une réification globale de la machine acteur, présente plusieurs aspects négatifs. D'une part, elle oblige le programmeur à manipuler toute la machine acteur là où il lui serait suffisant de ne manipuler que certains de ses éléments. Pourquoi réifier toute la machine acteur lorsqu'il n'est question que de tracer les messages reçus ? Outre la réécriture d'un nouveau comportement qui décrit l'ensemble des éléments de la machine, modifiés ou pas, cette approche se traduit par une inefficacité inhérente au modèle : une machine optimisée et efficace est remplacée par une autre moins efficace [MIC 97]. Enfin, pour obtenir un méta-acteur qui comptabilise les messages reçus et qui limite les ressources accaparées par la dénotation, un comportement spécifique doit être décrit qui combine ces deux fonctionnalités [PAZ 97].

Cet ensemble de considérations nous a amenés à décomposer la machine acteur en éléments manipulables un à un à l'aide de la réflexivité. De façon simplifiée, et bien que cela ne soit pas la seule décomposition possible (cf. [McA 95, GIR 93, MIG 98]), les accointances et les méthodes d'un méta-acteur classique sont assimilables aux éléments de base de la machine acteur. Chaque accointance, chaque méthode est alors un élément insécable de la machine acteur.

Notre décomposition du méta-acteur permet de manipuler ces éléments un à un i.e. de ne réifier que la liste des messages reçus ou l'adresse du comportement de la dénotation. Elle n'est donc pas organisée autour d'une architecture particulière d'un système d'acteurs [GIR 93, MIG 98] mais sur un agrégat d'acteurs du méta, nommés *méta-composants*, dont chacun réifie certains des éléments de la machine acteur. Dans ce modèle, tout méta-composant a un comportement qui ne se différencie du comportement d'un acteur classique que par la présence d'une description du lien causal qu'il entretient avec sa dénotation.

3.2. *Protocole de description du lien causal*

Le modèle de décomposition du méta-acteur proposé s'organise autour d'une description du lien causal qu'entretient un méta-composant avec sa dénotation. Cette description s'effectue au moyen d'une déclaration insérée dans le corps d'un comportement. Elle prend la forme suivante :

> (**declare** (*reify* (*acquaintance* ...) (*activity* ...))

Le champ **declare** introduit toute déclaration liée à un comportement. La sous-liste introduite par le symbole *reify* constitue la description des éléments réifiés par un méta-composant. L'utilisation de cette dernière est rendue nécessaire par l'existence d'autres déclarations qui ne seront pas décrites ici car elles restent hors du sujet de cet article. Le champ *acquaintance* décrit la liste des accointances de la machine acteur que le méta-composant réifie. Le champ *activity* contient la liste des activités de la machine acteur réifiées par le méta-composant.

La réification d'un élément de la machine acteur est décrite par un mot-clé qui désigne de façon unique cet élément suivi d'un autre mot-clé qui précise le type de la réification. Une description est donc constituée d'une suite de mots-clés, les mots-clés de rang impair désignant des éléments de la machine acteur et ceux de rang pair le type de réification associé à chaque élément.

Tous les comportements de méta-composants contiennent une telle déclaration. D'un point de vue syntaxique, c'est là l'unique moyen de distinguer le comportement d'un acteur classique de celui d'un méta-composant.

3.2.1. *Réification des accointances*

La réification des accointances de la machine acteur par un méta-composant intervient à travers ses propres accointances. Les accointances réifiées constituent le début de la liste des valeurs des accointances du méta-composant. L'ordre dans lequel elles sont déclarées correspond à leur position dans cette liste. Les autres accointances déclarées dans le comportement du méta-composant correspondent aux accointances propres au méta-composant. Il détient directement les valeurs de ces dernières qu'il manipule de manière identique aux accointances classiques d'un acteur.

Chaque réification d'une accointance de la machine acteur par un méta-composant est accompagnée de son type. Ce dernier spécifie le rapport qu'entretient le méta-composant avec l'accointance réifiée.

Les types de réification des accointances sont :

— :read, l'accointance est lue par le méta-composant. La lecture s'effectue de manière totalement asynchrone.

— :get, l'accointance est détenue par le méta-composant le temps d'un traitement. Durant ce laps de temps, il est garanti au méta-composant qu'il est l'unique détenteur de l'accointance. Lorsqu'il change de comportement, le méta-composant libère l'accointance en modifiant ou non sa valeur.

Le plus souvent, cette typologie n'est pas explicitement utilisée. Le type de réification d'une accointance est alors déduit du type du méta-composant. Si ce dernier est non sérialisé, la réification sera de type :read, tandis qu'elle sera de type :get pour un méta-composant sérialisé.

Dans le cas d'un méta-composant sérialisé, le type de réification de l'accointance peut être spécifié de manière explicite. Le type de réification d'une accointance prime alors sur le type du méta-composant. Si un méta-composant sérialisé peut utiliser n'importe lequel des deux types possibles de réification, un méta-composant non sérialisé est restreint au premier type.

3.2.2. *Réification des activités*

La réification des activités permet à un méta-composant de se substituer pour une part à la machine acteur. Une des méthodes du méta-composant représente alors une réification de l'activité concernée. Un même méta-composant peut réifier une ou plusieurs des activités de la machine acteur.

Bien que l'approche habituelle de la réflexivité consiste à substituer à l'implémentation un programme de l'utilisateur, un nombre non négligeable de programmes réflexifs peuvent se contenter d'une option moins radicale. Par exemple, un méta-composant chargé de tracer les messages reçus remplira parfaitement son rôle si, au lieu de remplacer l'activité de réception des messages, il est notifié à chaque déclenchement de cette activité. Cette constatation se traduit par une typologie étendue de la réification des activités.

Deux types de réification des activités sont proposés :

— :extension, le méta-composant demande à recevoir les messages correspondant à l'activation de l'activité réifiée. Il est simplement notifié[2] à chaque fois que l'activité qui l'intéresse est déclenchée.

— :main, le méta-composant implémente l'activité réifiée. À un instant `t`, un seul méta-composant peut implémenter une activité donnée.

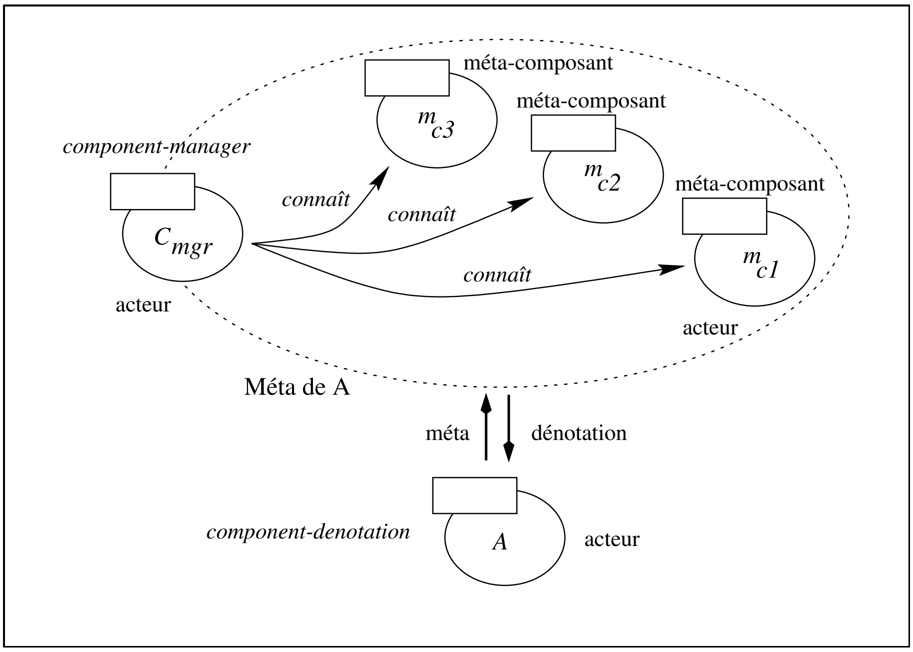

Figure 1. *Architecture du méta*

Contrairement au type :main qui n'autorise qu'un seul méta-composant par activité, le type :extension accepte autant de méta-composants que désiré. C'est ainsi qu'un méta-composant qui trace les messages reçus peut être combiné avec un autre méta-composant chargé de comptabiliser ces mêmes messages.

3.3. *Architecture et mécanismes*

Le méta-composant, doté de la description du lien causal qu'il entretient avec sa dénotation, vient s'insérer dans un ensemble de méta-composants qui participent tous au méta d'un acteur. Pour assurer un fonctionnement cohérent à cet ensemble, les éléments qui le constituent doivent être capables de communiquer entre eux. Il existe donc une entité spécifique, qui connaît et qui est connue de l'ensemble des méta-composants, qui assure le rôle de point de rendez-vous, le *component-manager* (Fig. 1).

Le *component-manager* accomplit deux tâches essentielles à la cohésion du couple machine acteur/méta-composants. D'une part, il reçoit et distribue les messages qui

2. Un message asynchrone qui dénote le déclenchement de l'activité est envoyé au méta-composant. Aucune réponse n'est attendue de ce message. Le méta-composant traite ce message en parallèle aux autres traitements.

déclenchent les activités de la machine acteur. Lorsque la dénotation doit recevoir un message, c'est à lui qu'est adressée une demande de réception de message. Et c'est toujours lui qui envoie cette demande de réception aux méta-composants qui la réifient ou qui déclenche les fonctions correspondantes de la machine acteur. D'autre part, le *component-manager* gère les demandes d'accès aux accointances de la machine acteur. Lorsqu'un méta-composant qui réifie certaines des accointances de la machine acteur doit exécuter l'une de ses méthodes, il demande les valeurs de ces accointances au *component-manager*. Ce dernier s'assure de la cohésion des accès concurrents aux accointances de la machine acteur.

Le *component-manager* représente le point d'entrée dans le système d'acteurs qu'il constitue avec les méta-composants (Fig. 1). En tant que tel, il apparaît comme étant un acteur. Cependant, et tant qu'il n'est pas réifié, il bénéficie d'une implémentation particulière qui lui permet de se fondre dans la machine acteur dont il présente la vision acteur. Grâce à cette implémentation particulière, la réification par un méta-composant de certains des éléments de la machine acteur reste confinée à ces éléments.

Une autre information partagée par l'ensemble des méta-composants constitutifs du méta d'un acteur est l'adresse de leur dénotation. Chaque méta-composant y accède à travers le symbole *component-denotation*.

Ces deux variables, *component-manager* et *component-denotation*, ne font pas partie des accointances d'un méta-composant. Elles ne sont connues de lui qu'à l'instant où il exécute l'une de ses méthodes et sont transportées par les messages qui déclenchent les activités de la machine acteur.

La réification d'une dénotation n'est effective que lorsqu'une des méthodes d'un méta-composant est activée. La mise en place d'un méta-composant ne représente pas, comme dans le modèle basé sur les méta-acteurs, une réification de la machine acteur. Ce retard dans la réification autorise le partage des méta-composants entre différentes dénotations. Ainsi partagés, les méta-composants occupent moins de place en mémoire mais peuvent aussi effectuer pour plusieurs dénotations ce qu'ils savent effectuer pour une seule. C'est ainsi qu'un compteur des messages reçus peut comptabiliser ceux reçus par l'unique dénotation qu'il réifie ou par un ensemble de dénotations réifiées.

4. Exemples d'utilisations des méta-composants

Sous forme de méta-composants, un mécanisme de comptabilisation des messages reçus est plus concis et plus direct que sous forme de méta-acteurs (cf. [MIC 97]). L'ensemble du comportement ne décrit que les aspects relatifs au mécanisme de comptabilisation sans qu'il soit fait appel, directement ou indirectement, aux autres mécanismes de la machine acteur. En LActE/R [MIC 97], la couche acteurs réflexifs que nous avons utilisée pour implémenter notre prototype, le code d'un méta-composant qui comptabilise les messages reçus est :

```
; Comportement d'un méta-composant
(def-behavior a-received-message-counter(number-of-messages) t
    ; Spécification du lien causal
  (declare (reify '((activity :receive-message :extension))))
    ; Méthode de comptabilisation des messages reçus
  ((:receive-message a-message)
   (become a-received-message-counter () (+ number-of-messages 1)))
    ; Méthode de lecture du nombre de messages reçus
  ((:get-number-of-messages) (become) (result number-of-messages)))
```

Le mot-clé **def-behavior** introduit la description du comportement *a-received-message-counter*. Ce comportement d'un acteur sérialisé utilise une seule accointance, *number-of-messages*, afin de mémoriser le nombre des messages reçus.

La description du lien causal fait apparaître l'unique activité réifiée, la réception des messages. Le méta-composant ne réalise pas l'implémentation de cette dernière et ne nécessite que d'être informé du déclenchement de cette activité. Aussi, le type de réification choisi est :extension.

La première des méthodes décrites incrémente le nombre de messages reçus à chaque déclenchement de l'activité de réception. Elle correspond au traitement de l'activité réifiée par le méta-composant. La seconde méthode autorise la lecture du nombre de messages reçus.

L'ajout d'une capacité de comptage des messages reçus s'opère de la manière suivante :

```
; Création du compteur de messages
(new rmc a-received-message-counter () 0)
; Établissement de la connexion entre le méta-composant et sa dénotation
(send [:reify fact] :to rmc)
```

Après établissement de la connexion, l'ensemble des messages reçus par la factorielle seront comptabilisés. Ce qu'il convient de remarquer ici est l'absence, lors de l'établissement de la connexion causale, de spécification de la place du méta-composant dans le méta de l'acteur réifié. L'utilisateur final ne se préoccupe donc pas de l'architecture du méta.

Si ce même méta-composant réifie d'autres acteurs que *fact*, il ne comptabilisera plus uniquement les messages reçus par *fact*, mais les messages reçus par l'ensemble des acteurs qu'il réifie. Cette utilisation étendue d'un même méta-composant qui se base sur la possibilité de les partager ne nécessite pas l'écriture d'un comportement supplémentaire. Par exemple, en supposant l'existence d'un acteur *fib*, l'expression suivante

```
(send [:reify fib] :to rmc)
```

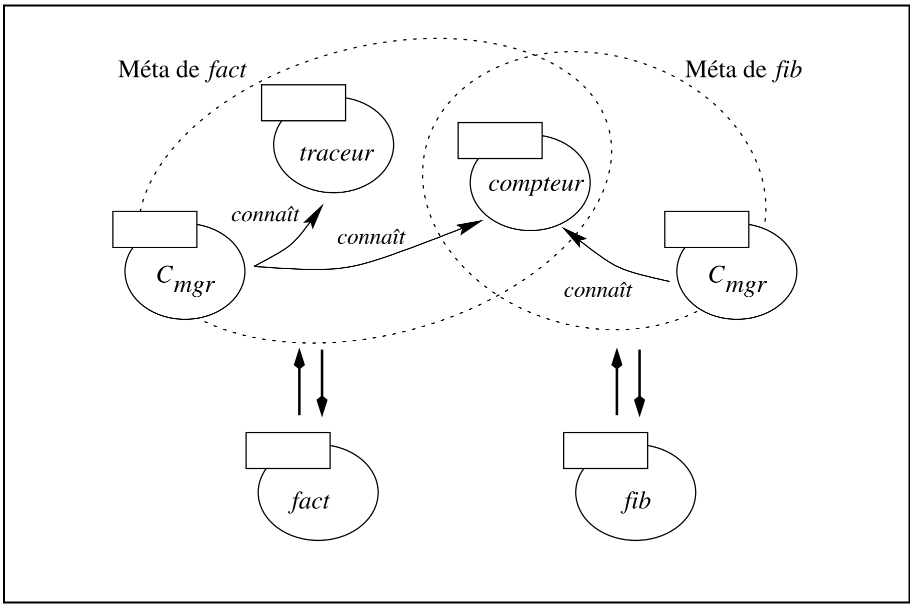

Figure 2. *Exemple de partage de méta-composants*

fera de *rmc* le compteur des messages reçus par *fact* et par *fib*. Le résultat de ce partage est illustré par la figure 2.

Plusieurs méta-composants peuvent bien sûr être combinés pour former le méta d'un seul et même acteur. Par exemple, l'acteur factorielle peut être doté d'un méta-composant qui trace les messages qu'il reçoit et d'un méta-composant qui séquentialise l'exécution de ses méthodes sans que l'utilisateur n'ait rien d'autre à faire que de créer ces méta-composants et d'établir leur lien avec la dénotation. Cependant, pour une activité donnée, un seul de ces méta-composants peut être de type :main.

5. Comparaison avec d'autres travaux

Plusieurs travaux organisent leur modèle de réflexivité autour d'une décomposition opérationnelle du méta-acteur [MIG 98, GIR 93, McA 95]. Bien que les motivations soient sensiblement différentes, ces décompositions visent à obtenir une manipulation plus fine des mécanismes de la dénotation.

L'introduction de la réflexivité dans Plasma II est basée sur un méta constitué d'un système d'acteurs [MIG 98]. Chaque composante de ce système remplit un rôle particulier, l'un étant le récepteur des messages, l'autre responsable de la sélection d'une méthode, etc. L'ensemble de ces acteurs collaborent entre eux pour implémenter la machine acteur de leur dénotation. La décomposition proposée ici peut servir de base à la constitution de bibliothèques. La composition des entités contenues dans la bibliothèque résulte du respect du protocole qui régit le méta d'un acteur. Cependant cette composition est laissée à la charge de l'utilisateur qui doit en connaître le protocole. Cette approche s'oppose à la simplification de l'utilisation de la réflexivité attendue par la constitution de bibliothèques.

ReActalk [GIR 93] décompose le méta d'un acteur sous la forme d'usines. Bien que le protocole qui régit le méta d'un acteur soit différent de celui de Plasma II, ces deux architectures sont assez comparables. Néanmoins, ReActalk s'appuie sur Smalltalk pour son implémentation. Il utilise habilement les classes pour implémenter les comportements des acteurs. Cette démarche lui permet de construire une hiérarchie de classes qui implémentent différents mécanismes acteurs. Un mécanisme d'hybridation des classes vient compléter son système en offrant un mécanisme de composition adéquat qui cache les détails de l'implémentation. Cependant, sa démarche repose sur l'utilisation des classes, outil qui reste délicat à mettre en œuvre dans un cadre distribué.

CodA [McA 95] se rapproche de notre modèle par l'accent qu'il met sur les activités de la machine acteur. Chaque activité y est implémentée par un méta-composant et le méta d'un objet résulte de leur agrégation. L'architecture de CodA est issue de la prise en compte des techniques objets qu'il applique à la réflexivité. En cherchant à obtenir un système support à l'implémentation d'un ensemble aussi large que possible d'interprètes réflexifs, CodA choisit de rompre l'uniformité linguistique. L'implémentation s'appuie sur les classes de Smalltalk.

L'ensemble des modèles précédents se caractérisent par une mise en place des méta-composants qui fait apparaître explicitement les activités qu'ils réifient. L'utilisateur ne peut donc utiliser ces systèmes en faisant abstraction d'une connaissance du méta-protocole. Chaque entité y est décrite par le programmeur en tenant compte des activités qu'elle va devoir réifier alors que sa place au sein du méta d'un acteur ne lui sera attribuée que lors de son utilisation. Ce manque de distinction entre les différents niveaux d'utilisation de la réflexivité ne facilite pas la mise en place de bibliothèques [PAZ 97].

Notre modèle se distingue des architectures précédentes par la mise en valeur qu'il fait du lien causal. En associant à nos méta-composants une spécification du lien causal, nous offrons à l'implémenteur la liberté de choisir son architecture tout en laissant à l'utilisateur la possibilité d'ignorer les détails de l'implémentation.

6. Perspectives et travaux futurs

À partir de notre modèle de réflexivité, nos recherches se dirigent dans plusieurs directions.

Actuellement, nous nous attachons à expérimenter notre modèle dans le cadre de la définition du contenu d'une bibliothèque de programmes réflexifs. Par cette démarche, nous cherchons, d'une part, à soulever les problèmes liés à sa constitution et, d'autre part, à affiner notre spécification du lien causal. La définition du contenu d'une bibliothèque est en effet une question délicate. C'est essentiellement son contenu qui fait son intérêt. À travers les travaux effectués dans notre équipe [PAZ 97, MIC 97], nous disposons de données non négligeables que nous allons exploiter à cette fin.

Le protocole de spécification du lien causal, tel qu'il est décrit dans cet article, ne supporte qu'un système limité de composition. L'étude du problème de la composition appliquée à la réflexivité effectuée dans notre équipe [PAZ 97] a permis de distinguer deux types de programmes réflexifs : les adjonctions et les noyaux. Les types :extension et :main de réification des activités sont le reflet de ces distinctions. Cependant, les catégories d'adjonctions supportées par le type :extension sont très limitées. Nous voudrions étendre la typologie de réification des activités afin d'augmenter le nombre de programmes réflexifs qui peuvent se décrire en termes d'adjonctions.

Une autre direction cherche à exploiter les informations offertes par la spécification afin d'optimiser le fonctionnement du système. En effet la spécification du lien causal peut être vue comme une description des différences entre la machine acteur initiale et la machine acteur modifiée. Nous pensons que l'évaluation partielle peut avantageusement exploiter ces informations pour supprimer une part du surcoût dû à la réflexivité.

7. Conclusion

En organisant la décomposition du méta-acteur non autour d'une architecture particulière, mais autour d'un système de spécification du lien causal, notre approche permet la mise en place de bibliothèques pour lesquelles il n'est pas nécessaire que l'utilisateur final connaisse les détails liés à l'implémentation. Par ailleurs, l'implémenteur conserve la maîtrise sur l'organisation qu'il veut donner au méta ainsi que sur le rôle de chaque méta-composant et le lien qu'il entretient avec sa dénotation. En modifiant la vision du lien causal, notre architecture représente selon nous un équilibre souhaitable entre ce qu'il est possible de faire et ce qu'il est nécessaire de connaître. Nous pensons que cette approche de la réflexivité offre un support apte à permettre son utilisation extensive.

Bibliographie

[AGH 86] AGHA G., « Actors: A Model of Concurrent Computation in Distributed Systems », PhD thesis, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, 1986, Published by MIT Press.

[GIR 93] GIROUX S., « Agents et sytèmes, une nécessaire unité », PhD thesis, Département d'informatique et de recherche opérationnelle, Faculté des Arts et Sciences, Université de Montréal, Mars 1993.

[HAS 93] HASSAS S., « GMAL, Un Modèle Réflexif pour la Conception de Systèmes d'Intelligence Artificielle Distribuée », Thèse de Doctorat d'Université, LISI, Université Claude Bernard, Lyon I, Septembre 1993.

[MAS 92] MASUHARA H., MATSUOKA S., WATANABE T., YONEZAWA A., « Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently », In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*, pp. 127–144, Vancouver, Canada, Octobre 1992, ACM, ACM Press.

[MAS 96] MASUHARA H., MATSUOKA S., YONEZAWA A., « Implementing Parallel Language Constructs Using a Reflective Object-Oriented Language », In *Reflection'96*, San Francisco, Avril 1996.

[MAT 91] MATSUOKA S., WATANABE T., ICHISUGI Y., YONEZAWA A., « Object-Oriented Concurrent Reflective Architectures », In TOKORO M., Ed., *Workshop on Object-Based Concurrent Programming*, Geneve, Switzerland, Juillet 1991, LNCS, Springer-Verlag.

[McA 95] MCAFFER J., « Meta-level Programming with CodA », In *in Proceedings of OOPSLA'95*, Octobre 1995.

[McA 96] MCAFFER J., « Engineering the Meta-Level », In *Proceedings of the Reflection'96 conference*, San Francisco, Avril 1996, ParcPlace-Difitalk, Object Technology International, Xerox Palo Alto Research Center.

[MIC 97] MICHEL C., « Modèles et Implémentations d'interprètes réflexifs », Thèse de Doctorat d'Université, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S), Université de Nice-Sophia Antipolis, Décembre 1997.

[MIG 98] MIGEON F., SALLÉ P., « Acteurs et Réflexivité », In *Journées Francophones des Languages Applicatifs*, Février 1998.

[PAZ 97] PAZZAGLIA J.-C., « Méthodologie et outils pour la programmation réflexive dans les langages à objets », Thèse de Doctorat d'Université, Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S), Université de Nice-Sophia Antipolis, Décembre 1997.

[SMI 83] SMITH B. C., « Reflection and Semantics in Lisp », In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pp. 23–35, ACM Press, 1983.

[TAN 90] TANAKA T., « Actor Reflection without Meta-object », In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, pp. 253–272, Ottawa, Canada, Octobre 1990.

[YON 89] YONEZAWA A., WATANABE T., « An Introduction to Object-Based Reflective Concurrent Computation », In *Proceedings od the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, vol. 24, pp. 50–54, SIGPLAN Notices, ACM Press, Avril 1989.