Typage de JavAct

Approche dérivée du typage de Cap

Vincent Hennebert, Marc Pantel

1 Problématique

L'évolution de l'architecture des calculateurs et des réseaux a rapidement permis de passer d'une informatique séquentielle et centralisée à une informatique concurrente, répartie et mobile. Nos travaux se situent dans le cadre du génie logiciel qui accompagne ces évolutions technologiques.

Le développement d'applications mobiles demande des méthodes et des outils adéquats. D'une part, leur mise au point est nettement plus complexe que celle des programmes séquentiels centralisés. Leurs fonctions sont réparties au sein de différents processus qui doivent collaborer. Les erreurs peuvent alors apparaître au sein des composants concurrents répartis ou bien être issues de problèmes de communication ou de synchronisation. D'autre part les applications mobiles doivent faire face à l'hétérogénéité matérielle et logicielle, ainsi qu'à de fortes variations (qualitative et quantitative) de leur environnement d'exécution.

L'utilisation d'approches plus abstraites de la programmation combinée avec des systèmes de type des programmes permet de faciliter l'introduction de la concurrence, de la répartition et de la mobilité. Nous nous appuyons sur un modèle de programmation à base d'objets actifs, les *acteurs*, qui constitue un support simple, réaliste, bien adapté à la mise en œuvre d'applications réparties à grande échelle ou mobiles. Un calcul de processus, Cap ([CPS96]), a tout d'abord été conçu afin d'étudier les possibilités de typage sur un tel modèle. Une bibliothèque Java inspirée de Cap, JavAct, a ensuite été développée pour pouvoir appliquer nos résultats à un langage réel de programmation. Nous exposons dans cette communication la structure de JavAct issue des systèmes de type développés pour Cap.

2 Rappels sur le modèle d'acteurs

Le modèle des acteurs a été proposé par Hewitt et Agha ([Agh86, HBS73]). Un acteur est une entité autonome pourvue de son propre espace mémoire contenant son code d'exécution et ses données. Il communique avec d'autres acteurs au moyen de messages asynchrones. Il est caractérisé par un certain *comportement* correspondant à l'ensemble des messages qu'il reconnaît à un instant donné. Ce comportement peut évoluer au cours de son existence, et c'est lorsqu'il en change qu'il consulte sa boîte aux lettres (voir fig. 1). Lorsqu'il accepte un message, il peut :

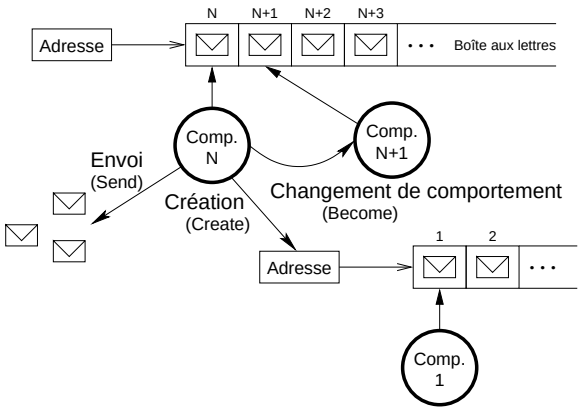

FIG. 1 – Les actions possibles d'un acteur lors du traitement d'un message

- envoyer de nouveaux messages à d'autres acteurs ou à lui-même ;
- changer de comportement ;
- créer de nouveaux acteurs.

Les *acquaintances* ou *connaissances* de l'acteur sont l'ensemble des autres acteurs dont il connaît l'adresse soit parce qu'il les a créés, soit parce qu'il les a reçus comme paramètres d'un message. Le support de communication est supposé *fiable* (les messages arrivent à destination dans leur intégralité, en un temps fini et en un seul exemplaire) et équitable (quant à la manière de traiter les messages, en général dans leur ordre d'arrivée). Par contre, l'ordre de réception peut être différent de l'ordre d'émission — cela permet de représenter de manière réaliste les réseaux comme Internet, qui font emprunter aux messages des chemins différents selon leur charge instantanée.

Ce modèle simple et réaliste est particulièrement adapté à la description de réseaux dont la topologie change dynamiquement (GSM, WiFi, BlueTooth, ...). Cependant une partie de la synchronisation entre les acteurs doit être programmée et il faut assurer que celle-ci est correctement effectuée. Le problème principal est de vérifier qu'un acteur acceptera les messages qui lui sont envoyés par les autres acteurs. Ce mécanisme est particulièrement complexe car un acteur peut à tout instant changer de comportement, *i.e.* changer l'ensemble des messages qu'il accepte (nous parlons alors de comportement non uniforme). Cela peut conduire à l'apparition de messages dits *orphelins*, qui ne seront jamais acceptés par leur destinataire.

La validation par typage dans le modèle d'acteurs consiste d'une part à contrôler que les messages sont bien formés (typage classique dans tous les langages), d'autre part à détecter les orphelins.

$$
\begin{array}{lll}
\nu a, b\,(& a \rhd [put(v) = \zeta(e, s_{empty})(& \\
 & \qquad e \rhd [get(c) = \zeta(e', s_{full})(c \lhd value(v) \parallel e' \rhd s_{empty})]) & \\
\parallel & b \rhd [value(v) = \zeta(e, s_{print})(print\ \text{"received } v\text{"} \parallel e \rhd s_{print})] & \\
\parallel & a \lhd put(5) & \\
\parallel & a \lhd put(7) & \\
\parallel & a \lhd get(b)) &
\end{array}
$$

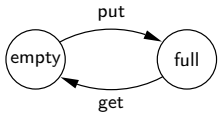

FIG. 2 – Exemple de programme CAP : une cellule

3 CAP

3.1 Le langage

CAP (Calcul d'Acteurs Primitifs) a été développé par J.-L. Colaço au sein de notre équipe ([CPS96]). Il se situe dans la lignée des calculs de processus : π-calcul de Milner ([MPW92]), calcul d'objets concurrents de Vasconcelos et Tokoro ([VT93]). Il combine plus précisément les objets primitifs de Cardelli ([AC94]) et les calculs asynchrones ([Bou92], [HT91]) ; le fait de travailler sur un calcul dédié permet de disposer des constructions propres aux acteurs (les noms, messages, etc.), et d'obtenir un typage efficace des expressions.

CAP possède la notion de comportement et de messages étiquetés. Plutôt que de donner sa grammaire nous présenterons sa syntaxe au moyen d'un exemple simple, celui d'une cellule tampon alternativement vide et pleine (fig. 2) :
Deux acteurs sont créés ($\nu a, b\ \ldots$), a étant la cellule et b un client. a adopte un premier comportement (cellule vide, $a \rhd [\ldots]$) dans lequel il accepte le message *put* ($a \rhd [put(v) = \ldots]$). La construction $\zeta(e, s)$ permet de capturer l'adresse (« ego », e) et le comportement (« self », s) courants.
Après avoir traité le message *put*, l'acteur adopte un nouveau comportement (celui de la cellule pleine), dans lequel il accepte le message *get* ($e \rhd [get(c) = \ldots]$). Ce message possède un paramètre correspondant au client auquel sera envoyé le contenu de la cellule ($c \lhd value(v)$). Parallèlement à cela l'acteur reprend le comportement vide ($e' \rhd s_{empty}$) du début.

Le client boucle sur le même comportement dans lequel il affiche la valeur qu'il a reçue de la cellule.

Trois messages sont envoyés à a ($a \lhd put(5) \parallel a \lhd put(7) \ldots$), qui les acceptera nécessairement dans l'ordre suivant : un premier *put*, le *get*, le second *put*. Selon l'ordre d'arrivée des messages, la valeur renvoyée par le *get* sera 5 ou 7.

3.2 Typage de Cap

À chaque acteur est associé un type calculé par inférence ; il correspond à l'ensemble des messages que l'acteur peut traiter, avec pour chaque message le type de ses paramètres : $\langle m_i(\widetilde{\alpha_i})^{i\in I}\rangle$ (ici le $\widetilde{}$ exprime la séquence). Le sous-typage existe et est lié au fait qu'un acteur puisse toujours être remplacé par un autre acteur offrant plus de méthodes et imposant moins de contraintes sur les arguments (notion de *subsumption*). Cela nous amène à la définition suivante :

Définition (Inclusion de type de noms) *Un nom de type α peut remplacer un nom de type α' ($\alpha \supseteq \alpha'$) si et seulement si :*

$$\langle m_i(\widetilde{\alpha'_i})^{i\in I}\rangle \subseteq \langle m_j(\widetilde{\alpha_j})^{j\in J}\rangle \iff I \subseteq J \wedge ((\forall k \in I)\ \widetilde{\alpha'_k} \widetilde{\supseteq} \widetilde{\alpha_k})$$

où $\widetilde{\supseteq}$ représente l'inclusion terme à terme entre séquences de même longueur.

Le problème est : « *Comment représenter fidèlement dans le type d'un acteur l'ensemble des comportements qu'il est susceptible d'adopter au cours de son exécution* ». Il s'agit de « *fusionner* » les ensembles de messages associés à chaque comportement. Une première idée peut être de prendre l'intersection de ces ensembles, pour aboutir à un sous-ensemble commun dont on est sûr qu'il sera accepté dans toutes les branches d'exécution possibles. Ce sous-ensemble est trop restrictif et un typage par cette méthode rejettera trop de programmes corrects ([Col97] p. 50). La solution opposée, consistant à prendre l'union des types de comportement, est trop permissive et ne permet pas non plus de faire un typage pertinent.

Une solution intermédiaire a donc été développée : l'*aplatissement* :

Définition (Aplatissement) *L'aplatissement, noté $\uplus$, est défini par :*

$$\langle m_i(\widetilde{\alpha_i})^{i\in I}\rangle \uplus \langle m_j(\widetilde{\alpha'_j})^{j\in J}\rangle = \langle m_k(\widetilde{\alpha_k}\widetilde{\cup}\widetilde{\alpha'_k})^{k\in I\cap J}\, m_i(\widetilde{\alpha_i})^{i\in I\setminus J}\, m_j(\widetilde{\alpha'_j})^{j\in J\setminus I}\rangle$$

De même ici $\widetilde{\cup}$ correspond à l'union terme à terme entre séquences de même longueur.

Contrairement à l'union, l'aplatissement est *sûr* : il n'accepte pas de programmes incorrects. Il est également moins restrictif que l'intersection et rejettera donc moins de programmes corrects (voir [CPS97]).

La figure 3 permet d'illustrer l'aplatissement ; un acteur est représenté sous la forme d'un arbre éventuellement infini mais régulier (un automate, voir fig. 2) ; les nœuds représentent les divers comportements qu'il peut adopter, et les arcs correspondent aux changements de comportement liés à la réception d'un message. Le comportement de départ est α_e (la racine de l'arbre ou l'état initial de l'automate), et selon les messages qu'il reçoit il peut adopter ensuite les comportements $\alpha_{e_1}, \alpha_{e_2}, \ldots, \alpha_{e_n}$. Son type est donc :

$$\alpha_e = \langle m_1(\alpha_1), \ldots, m_n(\alpha_n)\rangle \uplus \alpha_{e_1} \uplus \cdots \uplus \alpha_{e_n}$$

et les types de $\alpha_{e_1}, \ldots, \alpha_{e_n}$ sont définis récursivement de la même manière.

Pour la cellule de la fig. 2, on obtient les types suivants :

$$\begin{aligned} \alpha_a &= \langle put(), get(\langle value()\rangle)\rangle \\ \alpha_b &= \langle value()\rangle \end{aligned}$$

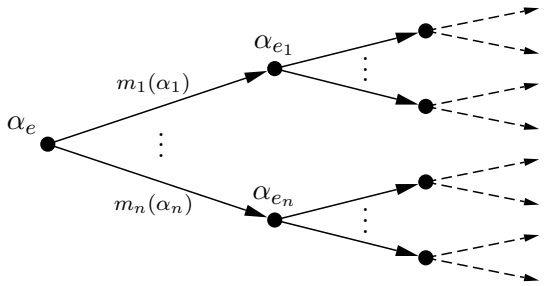

FIG. 3 – Les comportements d'un acteur vus comme les nœuds d'un arbre

Ceux-ci indiquent que la cellule accepte les messages *get* et *put* et que l'acteur passé en paramètre d'un message *get* doit être capable d'accepter le message *value*.

Les messages orphelins

Le typage permet d'assurer que toute communication potentielle se fera sans erreur sémantique (nombre et type corrects des arguments de messages). Selon la branche d'exécution que l'acteur adoptera certains messages ne pourront plus être pris en compte, et de ce fait deviendront *orphelins*. On peut donner la définition informelle suivante : « *Un message est orphelin s'il est destiné à un acteur dont ni le comportement actuel, ni les comportements futurs (quelle que soit l'exécution) ne l'accepteront* » — un orphelin trivial étant un orphelin avant même toute exécution du programme. Une définition formelle équivalente est présentée dans [CTP03].

Le système de type présenté ici repère les orphelins triviaux ; la détection (d'une partie) des autres orphelins fait l'objet d'un système plus évolué évoqué en conclusion.

4 JAVACT

JAVACT est une bibliothèque permettant d'implanter en JAVA des programmes respectant le modèle d'acteurs. Elle fournit des mécanismes pour la création d'acteurs, le changement de comportement, la répartition, la mobilité et la communication. La mobilité ne change pas la sémantique des acteurs. Elle est « forte » [1] mais limitée en pratique au moment du changement de comportement.

La bibliothèque JAVACT a été conçue afin d'être minimale et maintenable à moindre frais. Nous essayons de ne pas développer notre propre compilateur qui poserait inévitablement des problèmes de compatibilité avec le compilateur standard et qui nécessiterait une trop lourde maintenance pour l'adapter aux évolutions de JAVA. Nous tentons donc de concevoir la structure de la bibliothèque de façon à pouvoir s'appuyer sur le système de type standard du langage.

[1] au sens où l'acteur peut à tout moment décider de stopper son exécution pour la reprendre après s'être déplacé sur un autre site

Notre objectif est d'appliquer à JavAct les analyses effectuées sur Cap. Pour cela nous avons retranscrit en Java les structures essentielles de Cap, c.-à-d. les messages et les comportements. Pour résumer, un message correspond à un objet Java *serializable* ; les comportements sont définis par des classes dont chaque méthode correspond à un message reconnu par ledit comportement. L'intergiciel se charge, lors de la réception d'un objet message, de faire appel à la méthode adéquate de son destinataire.

Alors qu'en Cap nous faisions de l'*inférence* de type, ici nous utilisons le système de type de Java, où il n'est pas possible de calculer le multi-aplatissement défini précédemment. Nous disposons par contre de la surcharge qui nous permet de conserver séparément les différentes signatures fusionnées par l'aplatissement. Nous effectuons donc une sorte d'union, en rassemblant l'ensemble des signatures définies dans les diverses interfaces de comportement.

4.1 Développement d'une application JavAct

Les principales étapes qu'il convient de respecter sont les suivantes :

4.1.1 Profils de comportements et d'acteurs

Chaque comportement est déclaré au moyen d'une interface héritant de `BehaviorProfile`, contenant d'une part les méthodes correspondant aux messages acceptés par ce comportement, d'autre part les changements de comportement possibles (méthodes `become`).

Il faut ensuite associer à chaque acteur de l'application une interface héritant d'`ActorProfile`. Cette interface de marquage doit hériter des profils de tous les comportements que l'acteur est susceptible d'adopter (voir fig. 4).

Exemple : *Reprenons notre cellule, dont le code* Cap *a été donné en fig. 2. L'acteur a adopte alternativement deux comportements : une cellule pleine (s_{full}) et une cellule vide (s_{empty}). Le code pour ces deux interfaces sera le suivant :*

```
interface Full extends BehaviorProfile {
   public void get(Actor c) ;
   public void become(Empty bp) ;
}
interface Empty extends BehaviorProfile {
   public void put(int v) ;
   public void become(Full bp) ;
}
```

Et le profil de l'acteur sera déclaré ainsi :

```
interface Cell extends Empty, Full, ActorProfile { }
```

L'acteur b est destiné à tester la cellule, il n'adopte qu'un seul comportement. Il suffit de déclarer une seule interface, comme suit :

```
interface Client extends BehaviorProfile, ActorProfile
{ public void reply(int v) ; }
```

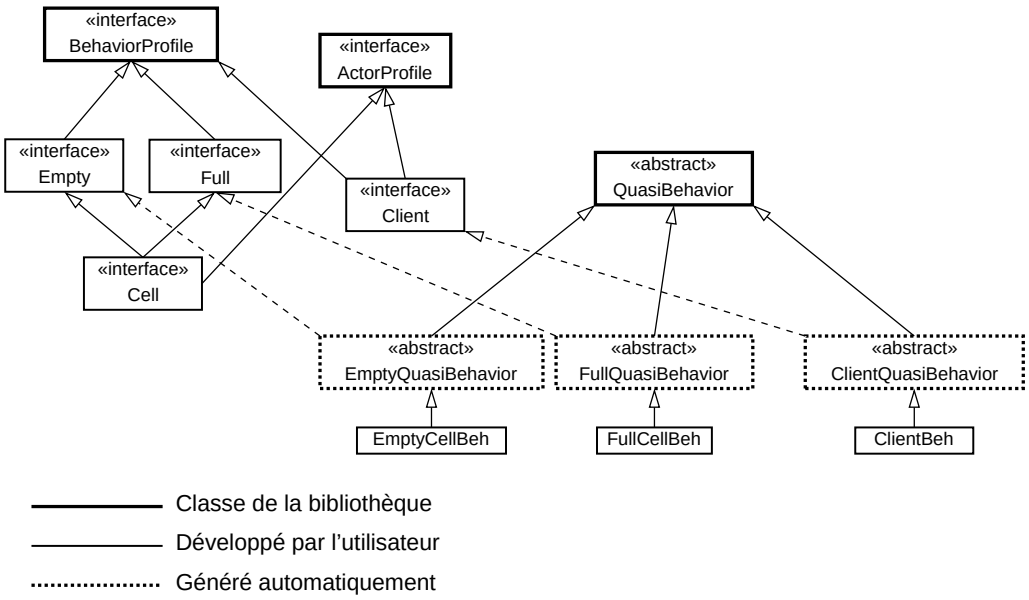

FIG. 4 – Structure de l'application JAVACT correspondant à la cellule

4.1.2 Génération des quasi-comportements et des messages

Les profils permettent de générer d'une part les réalisations partielles des comportements contenant le code des méthodes `become` (classes nommées `XxxQuasiBehavior`) ; d'autre part les classes représentant les messages (ces classes sont nommées `JAMXxx` dans le cas asynchrone et `JSMXxx` dans le cas synchrone).

Les messages sont des objets JAVA *serializable*. Ils possèdent une méthode `handle` appelée au moment où leur destinataire les traitera. Ils réagiront en fonction du comportement dudit destinataire. Ils peuvent être *asynchrones* (messages classiques), ou *synchrones* [2] à base de *futurs* : les premiers correspondent aux méthodes renvoyant `void`, et les seconds à celles renvoyant une valeur. L'acteur ayant envoyé un message synchrone se bloquera en attente de la réponse lorsqu'il aura besoin du résultat ; auparavant il peut effectuer d'autres actions en concurrence.

4.1.3 Implantation des comportements

La fig. 4 présente le diagramme de classes de la cellule. Le corps des méthodes déclarées dans les profils de comportement doit être écrit dans des classes héritant des classes abstraites `XxxQuasiBehavior` générées automatiquement. Celles-ci héritent des profils de comportement, et de la classe `QuasiBehavior` factorisant les traitements propres à tous les comportements.

[2] les messages synchrones ont été ajoutés à JAVACT uniquement pour simplifier la programmation ; on peut s'en passer en utilisant une continuation (l'acteur indique son adresse dans le message envoyé, et adopte ensuite un comportement dans lequel il attend la réponse)

Exemple : *Pour le comportement* `Full` *de la cellule :*

```
class FullCellBeh extends FullQuasiBehavior {
   int v ;    // Contenu de la cellule

   FullCellBeh(int v) { this.v = v ; }

   public void get(Actor c) {
      send(new JAMreply(v), c) ;
      become(new EmptyCellBeh()) ;
}}
```

4.1.4 Et pour finir...

Il reste à amorcer l'application en créant les acteurs et en leur envoyant les premiers messages. Ce démarrage, indépendant des acteurs, est effectué à l'aide de composants standards de création et d'émission (les `STD`). Ensuite, tout appel à `create`, `send`, `become`... doit être effectué dans un comportement d'acteur.

Exemple :

```
/* Création de la cellule */
Actor cell = CreateCt.STD.create(new EmptyCellBeh()) ;
SendCt.STD.send(new JAMput(2003), cell) ;
/* Création d'un client */
Actor customer = CreateCt.STD.create(new ClientBeh()) ;
SendCt.STD.send(new JAMget(customer), cell) ;
```

4.2 Typage de JavAct

Si l'utilisateur de la bibliothèque respecte ces conventions de programmation il est possible de procéder aux contrôles suivants :

4.2.1 Cohérences des déclarations

Lors de la génération automatique il est possible de contrôler facilement l'adéquation entre, d'une part, les `become` déclarés dans les profils de comportement et, d'autre part, les interfaces héritées par le profil d'acteur correspondant : les paramètres des `become` doivent se trouver dans l'ensemble des interfaces héritées. De plus, l'implantation des comportements respectera cet automate puisqu'elle hérite des `become` typés générés dans les `XxxQuasiBehavior`. Un acteur ne pourra donc prendre que les comportements déclarés dans son profil.

4.2.2 Paramètres des messages

Les messages sont codés par des objets Java, et pour chaque signature déclarée dans un profil de comportement un constructeur équivalent est généré. Un envoi de message correspond à une ligne du type :

```
send(new JAMget(client), cellule);
```

Le compilateur JAVA peut donc contrôler que les messages sont bien formés, et que l'arité et les types des paramètres sont corrects.

Ce contrôle est cependant moins fort que le multi-aplatissement : en effet ce dernier permet de vérifier en plus que le client est capable de traiter un message `reply`. Nous évoquons en section 4.3 une possibilité de renforcer la vérification.

4.2.3 Détection des orphelins triviaux

La détection des orphelins triviaux est pour l'instant un peu particulière : si un message a été déclaré dans *un* profil de comportement, alors il est possible d'envoyer ce message à n'importe quel autre acteur, puisque la classe `JxMmessage` correspondante aura été générée. Le compilateur JAVA ne détectera pas cette erreur qui n'apparaîtra que dynamiquement.

La section suivante présente quelques ébauches de solution dont la réalisation se heurte essentiellement à des problèmes techniques.

4.3 Améliorations possibles du typage

4.3.1 Analyse complète du code source

Une première solution serait de réaliser notre propre analyse sémantique du code source. On peut pour cela s'appuyer sur des outils d'analyse lexicale et syntaxique tel JAVACC [3].

Cette méthode présente plusieurs inconvénients : d'une part elle nécessite d'être régulièrement adaptée aux évolutions de JAVA, ce qui demanderait une maintenance trop importante. D'autre part JAVA est un langage plutôt complexe et l'implantation d'un contrôle complet est assez difficile.

4.3.2 Améliorer la structure de la bibliothèque

Il est possible, en améliorant encore les règles de programmation, de confier au compilateur JAVA standard un contrôle plus évolué que ce qui est fait actuellement.

Un premier point concerne les messages. Actuellement toutes les signatures d'un message `msge` sont regroupées dans une seule classe `JxMmsge` ; la seule distinction faite correspond au caractère synchrone/asynchrone du message. Cela autorise donc par exemple à envoyer à a un message avec une signature déclarée par b. Ce problème peut être facilement résolu en séparant les signatures provenant de profils différents. Il existerait alors plusieurs `JxMmsge` qu'il faudrait pour les distinguer placer dans des paquetages différents (un par profil d'acteur), ou bien déclarer en tant que classes internes à une classe de type « boîte à outils ». Pour la cellule nous aurions par exemple la classe suivante :

```
class CellToolBox {
    class JAMput extends Message { ... }
    class JAMget extends Message { ... }
    ...
}
```

[3] `http://javacc.dev.java.net/`

Une seconde amélioration plus importante consisterait à simuler complètement le multi-aplatissement de CAP pour, d'une part, contrôler le type des acteurs paramètres de messages et, d'autre part, détecter les orphelins triviaux. Nous illustrerons nos propos en reprenant l'exemple de la cellule.
Dans le premier cas il faudrait remplacer, dans l'interface `Full`, la déclaration

```
void get(Actor c);
```

par une déclaration du genre

```
void get(Client c);
```

On informe ainsi le système de type qu'on attend du client qu'il sache traiter un message `reply`. Il est alors possible de générer une encapsulation de la méthode `send` standard — dont la signature est `send(Message, Actor)` — par une version typée, par exemple `send(JAMget, CellActor)`.

Le même mécanisme peut être utilisé pour les orphelins triviaux. Actuellement on ne peut pas contrôler quel message est envoyé à un acteur. Si on utilisait l'encapsulation typée précédente il deviendrait impossible d'écrire par exemple `send(new JAMreply, cell)`.

La mise en œuvre pratique de ces solutions est en cours d'étude. L'utilisation des classes internes semble prometteuse puisqu'elle offre les possibilités d'encapsulation recherchées. En dépit de quelques problèmes techniques nous sommes optimistes quant à la possibilité de faire effectuer par le compilateur JAVA la totalité du multi-aplatissement de CAP, et même de lever certaines restrictions imposées par celui-ci.

Un contrôle plus poussé (détection d'orphelins non triviaux) devra lui faire appel à des méthodes spécifiques d'analyse évoquées en conclusion.

5 Conclusion et perspectives

De nombreux travaux ont eu lieu dans le cadre du transfert d'un système de type sophistiqué vers un langage dédié. Cependant les outils de programmation dérivés des calculs formels sont, en général, de nouveaux langages qui imposent un effort d'apprentissage coûteux incompatible avec une pratique industrielle (citons Obliq [Car94], JoCaML [CLF99], DiTyCo [LSÀFV99], Nomadic Pict [WS99]...). En outre les bibliothèques JAVA dérivées du modèle d'acteurs (citons ProActive [Pro99], JavaParty [PZ97], Aglets [LO98], Voyager [Obj99], Salsa [VA01]...) offrent, à notre connaissance, une abstraction limitée semblable aux versions précédentes de JAVACT, dans lesquelles il n'y avait aucun contrôle sur les acteurs et les messages. L'originalité de nos travaux réside dans l'utilisation d'un langage industriel existant à travers une bibliothèque, sans nouvelle construction syntaxique et sans modification de la sémantique du langage ou du système de type. Les limites de cette approche se trouvent au niveau de la précision de ce dernier. Mais elle reste suffisamment grande pour détecter la plupart des erreurs et les types calculés sont utiles comme point de départ de systèmes plus précis.

Il est ensuite intéressant d'élargir le contrôle en tentant de déterminer les orphelins non triviaux. Ceux-ci sont de deux types : les orphelins de *vivacité* (L'acceptation d'un message est subordonné à la réception d'un autre message qui ne sera jamais envoyé), et

les orphelins de *sûreté* (par ex. plusieurs messages seront envoyés alors qu'un seul sera accepté).

Ces deux problèmes sont nettement plus complexes. Ils ne peuvent pas être résolus par le système de type de JAVA même avec un codage plus sophistiqué que celui évoqué dans cette contribution. Il sera donc nécessaire de développer des outils plus évolués indépendants du compilateur JAVA. Nous envisageons de poursuivre la coopération entre règles de codage, génération automatique de code et outils de typage pour éviter de construire un analyseur complet du code JAVA. En effet, les règles de codage devraient nous permettre de nous contenter d'une analyse du code binaire JVM avec des outils tels BCEL [4] ou ASM [5] et éviter le coût des changements syntaxiques importants de JAVA (voir l'introduction future de la généricité dans la version 1.5).

D'autres systèmes de type plus précis ont été développés par notre équipe (voir [CPDS99, CTP03]) pour le calcul CAP. Les différentes formes de messages orphelins sont représentées par une logique temporelle adaptée à la description des boîtes aux lettres qui permet ensuite de déduire des techniques d'abstraction des acteurs. Le type associé à un acteur prend alors en compte la multiplicité des messages, c'est-à-dire le nombre de fois que ce message peut être émis vers l'acteur et accepté par celui-ci (un nombre inconnu éventuellement infini étant représenté par ω). La version la plus récente qui est en cours d'adaptation à JAVACT repose sur l'utilisation d'automates (plus précisément de transducteurs) pour représenter les comportements (messages acceptés/émis) puis à calculer une abstraction du langage associé et à vérifier l'adéquation entre les messages acceptés par un acteur et les messages émis vers cet acteur.

Références

[AC94] Martín Abadi and Luca Cardelli. A theory of primitive objects — untyped and first-order systems. In M. Hagiya and John C. Mitchell, editors, *proceedings of the International Symposium on Theoretical Aspects of Computer Software TACS '94, Sendai, Japan, April 19–22, 1994*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320, New York, NY, USA, 1994. Springer-Verlag.

[Agh86] Gul Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. The MIT Press, Cambridge, MA, USA, 1986.

[Bou92] Gérard Boudol. Asynchrony and the π-Calculus(note). Rapport de Recherche 1702, INRIA-Sophia Antipolis, May 1992.

[Car94] Luca Cardelli. Obliq A language with distributed scope. Technical Report 122, 1994.

[CLF99] Sylvain Conchon and Fabrice Le Fessant. Jocaml : mobile agents for objective-caml. In *Proceedings of the Joint Symposium ASA/MA '99. First International Symposium on Agent Systems and Applications (ASA '99). Third International Symposium on Mobile Agents (MA '99).* IEEE-Computer Society, October 1999.

[4] `http://jakarta.apache.org/bcel/`

[5] `http://asm.objectweb.org/`

[Col97] Jean-Louis Colaço. *Analyses statiques par typage de langages d'Acteurs*. Thèse de doctorat, Institut National Polytechnique de Toulouse, October 1997.

[CPDS99] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. Static safety analysis for non-uniform service availability in actors. In *Proceedings of FMOODS '99*, February 1999.

[CPS96] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. CAP : An actor dedicated process calculus. In *Proceedings of Proof Theory of Concurrent Object-Oriented Programming*, May 1996.

[CPS97] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. A set-constraint based analysis of actors. In *Proceedings of FMOODS '97*. International Federation for Information Processing, Chapman and Hall, July 1997.

[CTP03] Matthias Colin, X. Thirioux, and Marc Pantel. Temporal logic based static analysis for non uniform behaviors. In *Proceedings of FMOODS '03*, November 2003.

[HBS73] Carl E. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of Int. Joint Conference on Artificial Intelligence*, 1973.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, July 1991.

[LO98] D. B. Lange and M. Oshima. Programming and deploying java mobile agents with aglets. Reading ISBN 0-201- 32582-9, AddisonWesley Longman, 1998.

[LSÀFV99] Luís Lopes, Fernando Silva, Àlvaro Figueira, and Vasco Thudichum Vasconcelos. DiTyCO : Implementing mobile objects in the realm of process calculi. In *Proc. of ECOOP workshop MOS*, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100 :1–77, September 1992.

[Obj99] Objectspace. Voyager ORB 3.3 – developer guide, 1999.

[Pro99] ProActive. INRIA, 1999. http ://www-sop.inria.fr/oasis/ProActive.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency : Practice and Experience*, 9(11) :1225–1242, 1997.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12) :20–34, 2001.

[VT93] Vasco Thudichum Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In Shojiro Nishio and Akinori Yonezawa, editors, *Object technologies for advanced software : first JSSST international symposium, Kanazawa, Japan, November 4–6, 1993 : proceedings*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474, New York, NY, USA, 1993. Springer-Verlag.

[WS99] Pawel Wojciechowski and Peter Sewell. Nomadic pict : Language and infrastructure design for mobile agents. In *Proc. of ASA and MA*, 1999.