# ARM : un modèle réflexif asynchrone pour les objets répartis et réactifs

**Jacques Malenfant — Simon Denier**

*Laboratoire VALORIA, Université de Bretagne sud*
*BP 92116, F-56321 Lorient cedex*
*Jacques.Malenfant@univ-ubs.fr*

RÉSUMÉ. *L'approche procédurale traditionnelle pour la réflexion dans les langages de programmation, où la connexion entre les niveaux de base et méta est réalisée par une relation de nature implantatoire, est remise en cause par les systèmes à base de composants, les systèmes répartis et les systèmes réactifs. Dans tous ces cas, une partie du système ne peut être capturée par une relation de nature implantatoire, où le méta-niveau est un processeur de langage (par exemple, un interprète), soit parce que la notion d'état global y échappe, soit parce que des éléments essentiels se situent en dehors de l'implantation du système. Nous proposons un nouveau* modèle de réflexion asynchrone, *ARM, où la connexion entre les niveaux est fondée sur une communication d'événements par publication/souscription. Nous démontrons non seulement que ce modèle est mieux adapté aux systèmes répartis ou réactifs, mais qu'il permet aussi de généraliser les formes de réflexions possibles en adoptant et s'adaptant à la «juste combinaison de connexion et de détachement» entre les niveaux. Nous présentons une première implantation d'ARM en Java, sous J2EE avec JMS.*

ABSTRACT. *The traditional procedural approach to programming language reflection, where the connection between the base and the meta levels is of an implementation nature, is challenged by component-based, distributed and reactive systems. In all of these fields, part of the system to reflect upon cannot be captured in an "implements" relation, where the metalevel operates as a language processor (e.g. interpreter) either because we lack a centralized state or an essential ingredient lies outside the system. We introduce a novel* asynchronous reflective model, *ARM, where the connection between levels use an asynchronous publish/subscribe communication model. We show not only that this model is better suited to distributed and reactive systems, but that it also generalizes the possible forms of reflection by adopting and adapting to the "right combination of connection and detachment" between the base and the meta level. We present a first implementation of ARM in Java, under J2EE with JMS.*

MOTS-CLÉS : *réflexion, publication/souscription, Java, connexion causale, adaptation dynamique.*

KEYWORDS: *reflection, publish/subscribe, Java, causal connection, dynamic adaptation.*

RSTI - L'objet – 9/2003. LMO 2003, pages 91 à 103

## 1. Introduction

Réseaux ultra-rapides, mobilité et omniprésence des systèmes embarqués, trois forces majeures qui révolutionnent aujourd'hui l'informatique. Bien qu'une partie importante de la programmation soit encore séquentielle, nous basculons irréversiblement dans le monde des systèmes répartis, embarqués et mobiles, s'exécutant en continu sur des périodes allant de plusieurs heures à des années. Parmi les formidables défis à relever, jamais les aspects de qualité de service et d'adaptabilité dynamique n'auront été plus prégnants. L'adaptabilité dynamique étant l'un des crédos centraux de la réflexion, cette discipline est-elle bien armée pour affronter ces défis ?

En fait, la plupart des travaux sur la réflexion en programmation sont fondés sur l'approche procédurale traditionelle de la réflexion, où la connexion entre les niveaux de base et «méta» est réalisée par une relation de nature «*implantatoire*». Selon cette idée, le méta-niveau est un processeur de langage (par exemple, un interprète) pour le niveau de base. Malheureusement, cette vision des choses s'avère inappropriée à la programmation répartie ou réactive. Dans ces cas, une partie du système sur lequel on doit «réfléchir» ne peut être capturée par une relation de nature implantatoire.

Pour répondre à ces nouveaux défis, nous revenons aux principes de base de la réflexion énoncés par Brian Smith au début des années 80 [SMI 82]. Ces principes, qui ont récemment inspirés des travaux dans les domaines des acteurs et des agents, ont finalement été un peu oubliés au profit de la vision procédurale dans les autres domaines. Postulant que la répartition et l'embarqué nécessitent de revoir le cœur même de hypothèses réflexives, nous introduisons un nouveau modèle de réflexion asynchrone, ARM (*Asynchronous Reflection Model*). Ce modèle est fondé sur l'hypothèse selon laquelle la généralisation des formes de représentations réifiées et l'adoption de la communication d'événements par publication/souscription, sont mieux adaptées à la réalisation de la relation «méta» dans ces domaines.

Dans la suite de cet article, nous étudions les motivations et les conséquences de ces choix sur les fondements de la réflexion et nous démontrons qu'ils permettent d'adopter et de s'adapter à la «*juste combinaison de connexion et de détachement*» entre le niveau de base et le niveau méta, notion infatiguablement défendue par Brian Smith [SMI 84] de manière à généraliser les formes de réflexion rendues possibles. Nous présentons enfin une première implantation de ce modèle en Java, sous J2EE avec JMS.

## 2. Motivations

### 2.1. *La réflexion procédurale*

La réflexion a été définie par Brian Smith comme «*an entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter*». Le comportement réflexif est réalisé par un méta-niveau placé en relation «méta» avec le niveau de base. Bien que

Smith n'ait pas imposé de manière générale pour réaliser cette relation méta, son langage 3-Lisp [SMI 84] et la plupart de ses descendants ont adopté une relation «*implantatoire*» pour ce faire. Cette restriction de la réflexion à des systèmes où le méta-niveau est en relation d'implantation avec le niveau de base a été appelée *reflexion procédurale* par des Rivières (et Smith) [RIV 84] et elle a été définie de manière plus affinée par Bobrow et al. sous la forme suivante [BOB 93] :

> «*La réflexion est la capacité pour un programme à manipuler comme données quelque chose représentant l'état de ce programme durant son exécution. Il y a deux aspects dans cette manipulation : l'*introspection *et l'*intercession*. L'introspection est la capacité pour un programme d'observer et donc de raisonner sur son propre état. L'intercession est la capacité pour ce programme de modifier son propre état d'exécution ou d'altérer son interprétation ou sa signification. Ces deux aspects requièrent un mécanisme pour encoder l'état d'exécution comme données ; fournir un tel encodage est appelé* réification.»

Précisons que la réification comprend à la fois la définition préalable d'une représentation (par exemple, une hiérarchie de classes) et l'obtention à l'exécution des objets qui représentent effectivement l'état courant. Pour être effectives, l'introspection et l'intercession doivent opérer sur des données constamment remises à jour. La *connexion causale* est cette propriété qui veut que toute modification au niveau de base se reflète au niveau de sa représentation réifiée, et vice versa.

### 2.2. *Limites de l'approche procédurale*

Depuis plus de vingt ans que la réflexion en programmation est étudiée, c'est l'approche *procédurale* de la réflexion qui est la plus présente dans les langages à objets, séquentiels ou répartis. Son principal avantage est de réaliser directement une connexion causale complète. Lorsque le méta-niveau est un interprète méta-circulaire, il est facile de rendre visibles au programme et de maintenir à jour la représentation réifiée puisqu'elle est formée des structures de données mêmes qui sont utilisées pour exécuter le programme et qui sont implantées dans le langage de ce dernier.

Cette approche a cependant le désavantage d'imposer un couplage fort et très strict entre niveaux. Par définition, toute relation implantatoire suppose que le méta-niveau exécute et donc *est* le niveau de base. Cette observation a été formalisée par Danvy et Malmkjaer par la propriété d'*unifilarité* («*single-threadedness*») des langages à la 3-Lisp [DAN 88] : en tout temps, seul l'un ou l'autre du niveau de base ou du niveau méta s'exécute. Un corollaire usuel évitant le recouvrement introspectif est que le niveau de base est interrompu pendant l'exécution des calculs réflexifs ; toutes les modifications à la représentation réifiée prennent donc effet avant la prochaine étape de calcul au niveau de base. En d'autres termes, les étapes de calcul au méta-niveau et au niveau de base sont exécutées de manière parfaitement *synchrones*, au sens où elles s'exécutent en exclusion mutuelle. Un autre corrolaire, expliquant le couplage strict entre niveaux, est que le méta-niveau exécutant le niveau de base, aucun «détache-

ment» ne lui est possible puisqu'exécution signifie connaissance complète et parfaite de l'état courant du niveau de base.

Malheureusement, cette approche ne passe pas à l'échelle de la répartition et des systèmes réactifs. Dans les systèmes répartis (ou à base de composants), la relation implantatoire ne peut rendre compte de l'ensemble du système puisqu'il n'existe pas d'état global du fait des caractéristiques inhérentes à ces systèmes. En conséquence, la plupart des langages répartis réflexifs se sont restreints à une réflexion locale à chaque objet actif (voir §5). Dans les systèmes réactifs, l'environnement non-informatique, qui est partie prenante à la réflexion, ne peut être mis en relation implantatoire avec un méta-niveau. Dans les deux cas, les restrictions imposées par la nature implantatoire de la relation méta empêchent de traiter le problème de l'adaptation dynamique dans sa généralité, puisque celle-ci suppose une réification de ce qui se passe autour de l'objet, dans les autres objets et dans l'environnement au sens large.

### 2.3. *Avantages d'une approche asynchrone non-implantatoire*

L'idée de fonder la communication entre les niveaux de base et méta sur le modèle de publication/souscription peut sembler cosmétique. Techniquement, l'intégration au noyau réflexif d'un intersticiel de publication et souscription (par exemple, JMS du J2EE) ne pose pas de problèmes. Le point crucial concerne plutôt la révolution dans la façon de mettre en œuvre et d'utiliser la réflexion, bref dans ce que *cela permet de faire et qui était difficile voire impossible de faire dans l'approche procédurale*.

Sur la forme, l'utilisation de messages asynchrones et l'abandon du rôle de processeur de langage par le méta-niveau rendent caduques les vieilles discussions sur la concurrence ou la non-concurrence entre le niveau de base et le niveau méta. Le méta-objet de comportement, déchargé de son rôle d'exécution de son objet de base, peut s'exécuter de manière concurrente et désynchronisée par rapport à celui-ci.

Plus fondamentalement, le rejet du synchronisme strict entre les niveaux au profit d'un choix plus nuancé de sémantiques plus ou moins synchronisée, plus ou moins tolérante aux pannes, respectant différentes notions de causalité entre événements va introduire autant de nuances dans la notion de connexion causale. Mais voilà justement l'objectif recherché par Smith d'un juste équilibre entre «connexion» et «détachement». L'importance de cet aspect apparaît de manière évidente en programmation réactive. Un processus contrôlant des procédés en temps réel doit respecter ses délais de réaction même s'il fait appel à des calculs réflexifs. Ces calculs doivent être suffisamment «détachés» pour maintenir le caractère temps réel du système.

S'ouvre ainsi tout un champ d'exploration dans la nature même de la représentation réifiée et de son utilisation pour réaliser un très large spectre de formes d'introspection et d'intercession. N'étant plus imposée par les nécessités de l'exécution, la représentation réifiée peut se définir en fonction d'un *modèle* au plus pur sens du terme, c'est-à-dire une abstraction choisie pour sa finalité propre. Dans le monde complexe des application réparties embarquées, il est illusoire d'espérer construire des modèles

complets, prenant en compte tous les aspects du niveau de base. En fonction des besoins, un modèle est défini par nécessité, autant pour la représentation réifiée que des moyens de construction du modèle, d'introspection et d'intercession. Ce modèle n'a pas à prétendre à l'unicité ; nous croyons plus à la composition des modèles, facilitée d'ailleurs par la communication par publication/souscription.

En tant que modèle au sens large, la représentation réifiée va plus naturellement se construire par la *perception* d'informations pertinentes sur l'entité de base, une notion limitée ici à la réception et à l'interprétation d'événements par le méta-niveau. Ces événements notifient des changements d'état ou des actions prises au niveau de base, à partir de quoi, le méta-niveau va construire la représentation réifiée.

Ayant toute autonomie d'exécution, le méta-niveau peut percevoir des événements provenant également d'autres entités de base ou de l'environnement non-informatique, concrétisant ainsi la réflexion dans les systèmes répartis et réactifs. L'autonomie d'exécution du méta-objet lui permet aussi de sonder de lui-même l'environnement extérieur de manière à affiner sa représentation de l'état du calcul et de son environnement. Ces derniers peuvent les utiliser pour construire leur modèle de ce qui se passe et pour déclencher de l'adaption dynamique, déclenchement qui n'est alors plus du seul ressort du niveau de base (comme dans les langages à la 3-Lisp).

L'utilisation de l'approche publication/souscription apporte une grande souplesse. En utilisant les capacités de filtrage intégrées dans les intersticiels de messagerie, en hiérarchisant les événements et en définissant des propriétés de sûreté de transmission et d'ordre de causalité des évènements, un méta-objet peut choisir le niveau de complétude et de déterminisme avec lequel il veut percevoir des événements de telle nature ou de tel objet. Bref, le modèle de publication/souscription et les propriétés de communication permettent au méta-objet de déterminer le niveau de connexion et de détachement avec lequel il veut suivre ces événements. Tout cela peut évoluer dans le temps, en modifiant dynamiquement les propriétés, selon les capacités de l'intersticiel de messagerie sous-jacent (que l'on ne suppose pas limitées *a priori*).

Enfin, la nature même des modèles de représentation réifiée est fondée à changer drastiquement. L'expérience accumulée dans les applications de la réflexion à l'adaptation dynamique des systèmes, mais aussi dans les architectures délibératives/réactives des systèmes multi-agents, montrent la nécessité de la prévision dans l'intercession [MAL 01]. Modifier le comportement d'une application en fonction du niveau de ressources physiques disponibles est un problème de contrôle classique au sens de la théorie du contrôle. Or, de mauvaises politiques de contrôle peuvent dégrader singulièrement la performance. S'adapter répétitivement à des changements dont la durée est trop courte peut provoquer une sorte de «*thrashing*» où le système ne fait que s'adapter et rien d'autre. Par un choix approprié de modèles, le méta-niveau sera en mesure de calculer des politiques viables, voire optimales, qui seront appliquées lors de l'intercession. C'est donc vers un mariage entre réflexion et contrôle qu'il faut aller pour réussir l'adaptabilité dynamique des applications.

Le passage à un modèle de réflexion asynchrone n'abolit pas complètement l'approche procédurale. De fait, postulons la conjecture selon laquelle la réflexion procédurale est parfaitement réalisable dans un contexte de réflexion asynchrone (ce que nous faisons). Mais plus fondamentalement, l'intercession sur un programme en cours d'exécution nécessitera bien de la réflexion procédurale si on veut modifier son comportement et son état d'exécution. Plutôt qu'un méta-niveau procédural à part entière, nos expériences montrent qu'un ensemble limité d'opérations réflexives peut suffir, même s'il est plus satisfaisant esthétiquement de rechercher des systèmes purement et complètement réflexifs. En Java par exemple, l'addition des API définies par la «*Java Platform Debugging Architecture, JPDA*» (dont le changement à chaud des classes, dit «*hotswap*») et des capacités de modification du code données par des compilateurs JIT réflexifs s'avère bien souvent suffisante.

## 3. Le modèle ARM

### 3.1. *Les entités du noyau*

Le noyau d'ARM est largement inspiré du modèle ObjVlisp de Cointe et Briot [BRI 87, COI 87], auquel sont adjoints des méta-objets selon une approche de Ferber [FER 89]. Le noyau est donc conçu autour de trois classes, `Entity`, `StructuralMeta` et `BehavioralMeta`, et d'un premier méta-objet de comportement, ci-après nommé `basicBehavioralMeta`, dont le rôle consiste à être le méta-objet de comportement de tous les objets du noyau, c'est-à-dire les trois classes et lui-même. Nous avons évité de reprendre les noms classiques d'`Object`, `Class` et `MetaObject` pour bien marquer qu'ARM étend la «réflexion procédurale» d'ObjVlisp par une généralisation du modèle de représentation d'une entité computationnelle à partir de laquelle plusieurs choix de représentation spécifiques peuvent être développés. Le modèle ARM n'a pas pour objectif d'être trop directif. Il a plutôt vocation à être étendu pour réaliser des applications particulières, soit en réflexion procédurale ou autre. Il s'attache donc à exprimer les relations entre ses entités, plutôt que leur contenu définitif et complet.

La figure 1 reprend les principales relations d'héritage, d'instantiation et de «*méta-de*» comportemental du noyau d'ARM. Il faut comprendre ces relations en ARM avec une sémantique adaptée à la forme de réflexion employée, qui n'est pas procédurale en général. La relation «instance» s'entend donc comme ayant pour méta-objet structurel, la relation «hérite» s'entend comme l'extension d'un méta-objet structurel, et la relation «méta-de» s'entend comme est le méta-objet comportemental de. À charge des noyaux spécifiques de définir plus précisément la sémantique de ces relations. Ce graphe subsume celui d'ObjVlisp. S'ajoute aux relations isomorphes à celles d'ObjVlisp tout ce qui se rapporte à `BehavioralMeta`. `BehavioralMeta` est instance de `StructuralMeta` et hérite de `Entity`. Le premier méta-objet comportemental, `basicBehavioralMeta` est instance de `BehavioralMeta` et il est le méta-objet comportemental de toutes les entités du noyau, y compris lui-même.

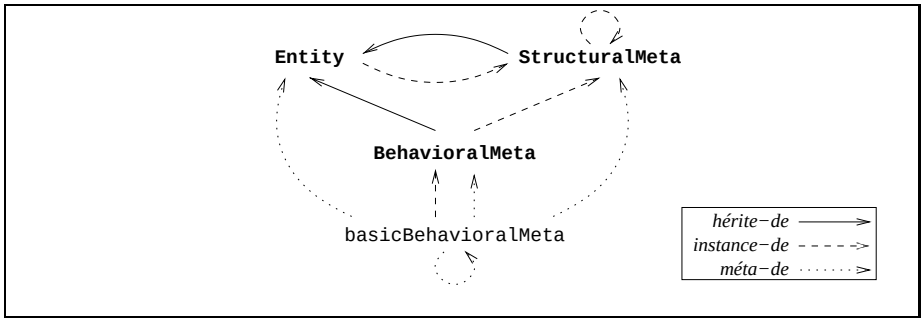


**Figure 1.** *Noyau du modèle ARM*

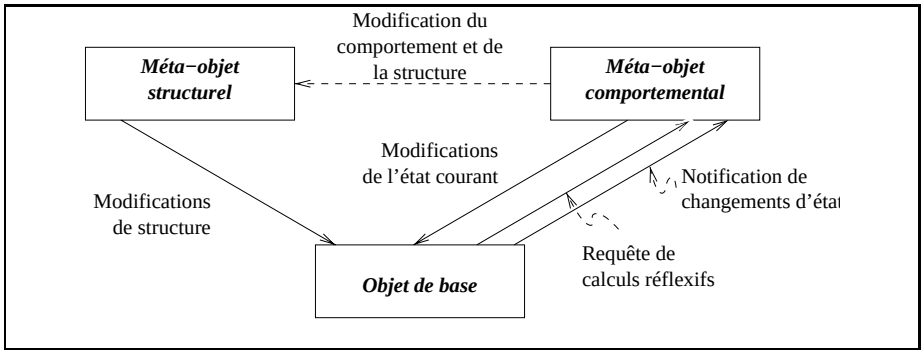


**Figure 2.** *Communication asynchrone entre entités*

La figure 2 complète ces relations en présentant les principaux liens de communication entre entités du noyau : (1) la notification de changements d'états ou d'actions depuis l'objet de base vers son méta-objet comportemental, (2) les requêtes des objets de base vers le méta-objet comportemental pour l'exécution de calculs réflexifs, (3) les modifications de l'état courant du calcul requises par le méta-objet comportemental sur l'objet de base, (4) les modifications du comportement (par exemple méthodes en réflexion procédurale) et de la structure (par exemple, variables d'instances) requises par le méta-objet comportemental au méta-objet structurel, et enfin (5) les modifications de la structure de l'objet de base requises par son méta-objet structurel.

Outre les requêtes de modifications de comportement et de structure qui peuvent être adressées au méta-objet structurel par le méta-objet comportement, au nom de l'intercession, les autres relations de communication traversent la frontière des niveaux. Il sont donc assimilables aux opération classiques «*reify*» et «*reflect*» permettant de passer du niveau de base au méta-niveau et inversement. Se pose ainsi le problème de la désignation des entités réifiées par rapport à la désignation de l'entité ainsi représentée, un aspect sur lequel nous devrons revenir dans une étude plus approfondie des fondements du modèle ARM.

### 3.2. *Généralisation de la notion de représentation réifiée*

La représentation réfiée est l'élément central du méta-niveau. On la décompose généralement entre une partie structurelle et une partie comportementale. L'intérêt de cette décomposition est que la représentation structurelle peut généralement être partagée par plusieurs entités du niveau de base (mais pas toujours, comme dans les prototypes), alors que la représentation comportementale, tenant compte de l'exécution de chacune des entités, ne se partage généralement pas par nature.

Pour se donner des points de repères, en réflexion procédurale séquentielle la représentation réifiée est assurée par les classes (bien connues) et par les méta-objets de comportement. Elle comporte une description de la structure des entités de base, par la liste des variables d'instance (y compris leurs types), et une description du comportement, par l'ensemble des méthodes applicables à ces entités de base. La représentation comportementale, elle, comporte l'état d'exécution d'une entité de base représentée par la continuation et l'environnement courants (sachant que la continuation contient les environnements de tous les calculs suspendus en attente d'un résultat). En réflexion procédurale répartie, la représentation comportementale comporte aussi des éléments propres à la concurrence et à la répartition, tels la file des messages à traiter, etc.

ARM est un modèle générique ouvert à plusieurs choix de représentation réifiée pour les entités de base, pouvant éventuellement cohabiter dans une même application. ARM est ainsi construit comme un «*framework*» qui est conceptuellement un générateur $\mathcal{ARM}(\cdot)$ de noyaux réflexifs spécifiques selon le choix de représentation réifiée. Le noyau définit un ensemble de classes abstraites pour la représentation réifiée, imposant ainsi ses contraintes. ARM est donc un modèle paramétré par l'ensemble de classes concrètes dérivées des classes abstraites de représentation.

Cet ensemble des classes abstraites résulte d'une induction pour faire ressortir ce qui transcende différentes formes de représentation possibles. Nous nous sommes basés sur trois noyaux spécifiques possibles : un basé sur une représentation procédurale classique $\mathcal{ARM}(\mathcal{P})$, un sur une représentation par un automate d'états fini $\mathcal{ARM}(\mathcal{DFA})$, et un sur une représentation par un automate de type «*statechart*» $\mathcal{ARM}(\mathcal{SC})$, où il est possible de voir la représentation à plusieurs niveaux de granularité selon une sémantique relativement bien établie. L'analyse de ces trois formes de représentation fait ressortir les concepts de représentation suivants :

– l'*ensemble des états* possibles pour un objet de base,
– un *comportement* applicable à un objet le faisant passer d'un état à un autre,
– l'*ensemble des comportements* applicables à une entité de base,
– l'*activation* d'un objet de base, et
– un *état* possible pour un objet de base.

Les trois premiers concepts font partie de la représentation structurelle, alors que l'activation fait partie de la représentation comportementale. Le concept d'état, peut rendre compte de l'état courant d'un objet, faisant donc partie de la représentation comportementale, mais l'ensemble des états possibles peut aussi être décrit en extension, par énumération des états individuels, ce qui rejoint la représentation structurelle.

Le modèle ARM représente ces concepts par les classes `State`, `StateSpace`, `Behavior`, `Behaviors` et `Activation`, qui se spécialisent naturellement aux différentes formes de représentation réifiée. L'état d'un objet peut être un vecteur de valeurs de variables d'instance (procédurale), un état d'un automate ou encore dans un «*statechart*» (pouvant alors admettre une relation d'inclusion entre états). Un ensemble d'états peut être décrit par ses dimensions (noms de variables d'instances) et l'ensemble des valeurs admises dans chacune de ces dimensions (types des variables). Un comportement peut être vu comme une méthode (procédurale), ou encore comme une transition dans un automate. L'ensemble des comportements pouvant être un dictionnaire de méthodes (procédurale) ou l'ensemble des transitions dans l'automate.

### 3.3. *Les événements du protocole de communication*

Le protocole de communication entre niveaux et entre entités suppose l'utilisation d'événements. Outre la communication entre les entités qui s'apparente à de l'invocation de méthodes par messages asynchrones, le modèle ARM, dans sa volonté de généralisation propre à un «*framework*», propose deux grands types d'événements aux propriétés différentes pour gérer la notification de l'objet de base vers son méta-objet comportemental : les événements notifiant une action faite au niveau de base et les événements notifiant plutôt son état courant. Ces deux modes de notification sont également envisageables : l'entité notifie son méta-objet qu'il exécute une certaine action (par exemple, l'invocation d'un certain comportement) ou encore il notifie son état courant. Le modèle propose deux classes abstraites représentant ces deux formes de notification : `StateEvent` et `ActionEvent`.

La notification d'une action nécessite peu d'information : un nom de comportement, et des arguments. Dans la mesure où le méta-objet de comportement peut suivre de manière fiable ces actions, il pourra reconstituer la séquence des états et des actions exécutées par le niveau de base. Cette forme de notification donne au méta-niveau un maximum d'information. La notification d'un état suppose souvent le transfert de plus de données du niveau de base vers le méta-niveau, et ne dit rien sur les actions qui ont été exécutées par le niveau de base. Par contre, la notification d'état est plus robuste au sens où elle n'est pas sensible à la perte de messages (bien que sensible à l'ordre d'arrivée, ce qui est le cas également de la notification d'action).

## 4. Implantation en Java, sous J2EE et JMS

La déclinaison concurrente et répartie du modèle ARM est implantée en Java, version J2EE, sur la base d'un modèle d'objets actifs avec communication asynchrone et synchronisation sur des futurs. En fait, le modèle sur lequel nous nous sommes fondés est emprunté au langage Hybrid de Nierstrasz [NIE 87], avec des objets actifs autour desquels sont formés des ilôts d'objets passifs qui ne peuvent communiquer qu'en passant par l'objet actif qui les détient. La mise en œuvre s'inspire fortement du patron de conception *Active Object* formalisé par Schmidt [SCH 00].

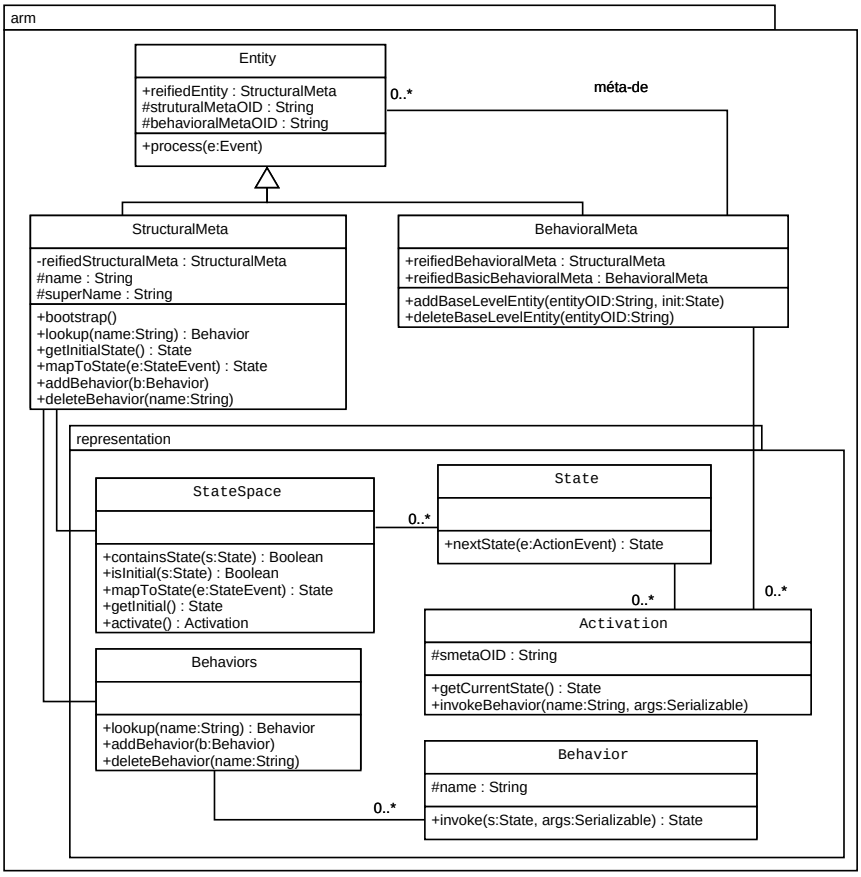


**Figure 3.** *Mise en œuvre du noyau d'ARM en Java*

La classe `ActiveObject` joue le rôle central : elle est un fil d'exécution Java (sous-classe de `Thread`) et elle met en œuvre la connexion avec JMS. Les classes de messages ont pour racine `Event`. Un événement a un expéditeur et un destinataire qui font partie des propriétés du message filtrables en JMS. Il peut être synchronisé (`SynchronizedEvent`) par des objets futurs (`Futur`, `FuturValue`). Il peut correspondre à un appel de méthode (interface `Callable` et classe `MethodRequest`) éventuellement synchronisé (`SynchronizedMethodRequest`).

La publication/souscription en JMS est organisée en sujets ou «*topics*». Pour le substrat objets actifs, nous avons retenu un seul sujet appelé `"active/Futur"` servant à publier et à souscrire aux événements délivrant les valeurs futures pour la synchronisation des objets actifs. Pour les communications du modèle ARM, nous avons choisi de les diviser en trois sujets : `"arm/Entity"` pour la commmunication entre entités

de même niveau, "arm/SMeta" pour celle entre entités et leur méta-objet structurel et "arm/BMeta" pour celle entre entités et leur méta-objet comportemental.

La figure 3 présente le noyau d'ARM en Java. Outre le fait non-illustré qu'Entity est sous-classe d'ActiveObject, on retrouve les liens et les méthodes attendues après la définition précédente du modèle. L'auto-amorçage en deux phases est ici réalisé par la méthode statique bootstrap de StructuralMeta. Les représentations réifiées sont bien entendues très peu élaborées, cette définition étant à la charge des noyaux étendus des modèles spécifiques. Les classes de dérivation des trois modèles avec représentation procédurale, par automate fini et par automate «*statechart*» sont regroupés dans trois paquetages que nous ne détaillons pas ici faute d'espace.

## 5. Travaux connexes

En parallèle à l'adoption industrielle de la communication par publication/souscription en programmation répartie, comme l'illustre l'API JMS de Java qui inclut ce mode de communication, quelques travaux ont introduit la communication par événements dans les systèmes et intersticiels réflexifs [BLA 98, BLA 00, KON 00, WAN 00] ainsi que plus généralement les systèmes cherchant à faire de l'adaptation dynamique [MAL 01]. Mentionnons aussi les travaux sur les interfaces homme-machine, l'architecture MVC et les patrons de conception Observateur et État qui ont également popularisé les concepts et utilisations de la notification en général. Ces travaux, dont leur utilisation en adaptation dynamique [BRA 99, FER 98], ont inspiré et préfigurent une généralisation comme celle proposée par ARM.

Dans les langages réflexifs, LEAD++ [AMA 99] propose une approche tendant vers les idées d'ARM, sans cependant se détacher de l'approche procédurale. L'utilisation d'événements a aussi inspiré Dynascope [SOS 92], un système de supervision dont Sosič a fait une lecture réflexive. On peut voir ce système comme un ancêtre de la «*Java Platform Debugger Architecture (JPDA)* [JPD02]. MetaXa [GOL 98] utilise les événements pour faire de la réflexion à partir d'un machine virtuelle notifiant ses activités, mais lui aussi reste lié à la vision procédurale.

En réflexion pour objets répartis, la plupart des langages se sont restreints à une réflexion locale à chaque objet actif où le méta-niveau peut être un processeur du niveau de base (voir [BRI 98] pour une synthèse sur ce vaste domaine). Ou encore ils ont limité la réification à certains aspects précis de l'implantation (envoi de messages, «*proxy*», «*stubs*», ...) sur lesquelles des calculs réflexifs peuvent être localement réalisés. Ces travaux demeurent intimement liés à la vision procédurale de la réflexion.

La communauté des acteurs et des agents a identifié et commence à explorer de nouvelles formes de réflexion subsumant les besoins identifiés ici. Sans partager l'inspiration IA des agents, de nombreuses convergences seront trouvées entre ARM et les techniques d'implantation des agents. Enfin, peu de travaux font le pont entre le domaine du contrôle et la réflexion, le travail de Pii Lunau fournissant une exception notable [LUN 97], tout en restant attaché à une vision procédurale de la réflexion.

## 6. Conclusions et perspectives

Nous avons présenté un nouveau modèle de réflexion asynchrone se fondant sur la communication d'événements par publication/souscription. Le modèle ARM, dérivé du modèle ObjVlisp et y intégrant des méta-objets de comportement, délaisse l'unique approche procédurale de la réflexion pour généraliser les formes de réflexion possibles par une exploration de diverses formes de représentation réifiée, d'introspection et d'intercession. Nous avons montré que ce modèle est mieux à même que l'approche procédurale de s'attaquer aux besoins d'adaptabilité dynamique des systèmes répartis et réactifs. Le modèle ARM est en fait un «*framework*» ou un générateur de modèles de réflexion, selon la représentation réifiée. Trois modèles ont été définis à ce jour : $\mathcal{ARM}(\mathcal{P})$ avec représentation procédurale classique, $\mathcal{ARM}(\mathcal{DFA})$ avec une représentation par automate d'états fini, et $\mathcal{ARM}(\mathcal{SC})$ avec une représentation par un automate de type «*statechart*». Nous avons proposé une première implantation du modèle ARM en Java avec JMS. Nous avons décrit les problèmes et les choix techniques de cette mise en oeuvre, qui est disponible au téléchargement pour évaluation à `http://www.univ-ubs.fr/valoria/Jacques.Malenfant/ARM`.

Ces travaux ouvrent de nombreuses perspectives, dont l'exploration des possibilités offertes par les nouvelles formes de représentation réifiées, d'introspection et d'intercession et une meilleure conceptualisation du modèle. Cela ne pourra se concrétiser que par l'utilisation d'ARM pour des applications réelles, comme celles que nous développons dans le cadre du projet robotique MAAM. Une piste à étudier pour la conceptualisation est de considérer la représentation réifiée comme partie d'une interprétation abstraite du niveau de base, ou encore d'une vérification de modèle («*model-checking*»). Cette vision nous laisse entrevoir les moyens de poser ses nouvelles formes de réflexion non-prodécurales sur des fondations solides et générales.

Remerciements

Simon Denier, élève-ingénieur à l'INSA de Rennes, a contribué à ces travaux lors d'un stage au sein du pré-projet MAAM dirigé par Dominique Duhaut et financé par le programme Robotique et Entitiés Artificielles (ROBEA) du CNRS.

## 7. Bibliographie

[AMA 99] AMANO N., WATANABE T., « An Approach for Constructing Dynamically Adaptable Component-Based Software Systems using LEAD++ », CAZZOLA W., STROUD R., TISATO F., Eds., *Actes électroniques du Workshop OORaSE'99*, 1999, p. 1–16.

[BLA 98] BLAIR G., COULSON G., ROBIN P., PAPATHOMAS M., « An Architecture for Next Generation Middleware », *Middleware'98*, IFIP, Springer-Verlag, 1998.

[BLA 00] BLAIR G., ANDERSEN A., BLAIR L., COULSON G., SÁNCHEZ D., « Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform », *IEE Proceedings — Software*, vol. 147, n° 1, 2000, p. 13–21.

[BOB 93] BOBROW D., GABRIEL R., WHITE J., « CLOS in Context — The Shape of the Design Space », PAEPCKE A., Ed., *Object-Oriented Programming : the CLOS Perspective*, chapitre 2, p. 29–61, MIT Press, 1993.

[BRA 99] BRAUX M., NOYÉ J., « Changement dynamique de comportement par composition de schémas de conception », *"Langages et Modèles à Objets, LMO'99"*, Hermès, 1999.

[BRI 87] BRIOT J.-P., COINTE P., « A Uniform Model for Object-Oriented Languages Using the Class Abstraction », *IJCAI'87*, 1987, p. 40–43.

[BRI 98] BRIOT J.-P., GUERRAOUI R., LOHR K., « Concurrency and Distribution in Object-Oriented Programming », *ACM Computing Surveys*, vol. 30, n° 3, 1998, p. 291–329.

[COI 87] COINTE P., « Metaclasses are First Class : the ObjVLisp Model », *OOPSLA'87, ACM Sigplan Notices*, vol. 22, n° 12, 1987, p. 156–167.

[DAN 88] DANVY O., MALMKJAER K., « Intensions and Extensions in a Reflective Tower », *ACM LFP'88*, 1988, p. 327–341.

[FER 89] FERBER J., « Computational Reflection in Class Based Object-Oriented Languages », *OOPSLA'89, ACM Sigplan Notices*, vol. 24, octobre 1989, p. 317–326.

[FER 98] FERREIRA L., RUBIRA C., « Reflective Design Patterns to Implement Fault Tolerance », *«OOPSLA'98 Workshop on Reflective Programming in C++ and Java»*, octobre 1998, p. 81–85.

[GOL 98] GOLM M., KLEINÖDER J., « MetaXa and the Future of Reflection », *«OOPSLA'98 Workshop on Reflective Programming in C++ and Java»*, octobre 1998, p. 1–5.

[JPD02] « Sun Microsystems, Java Platform Debugger Architecture Home Page », `http ://java.sun.com/products/jpda`, 2002, SDK release 1.4.

[KON 00] KON F., ROMÀN M., LIU P., MAO J., YAMANE T., AES L. M., CAMPBELL R., « Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB », *Middleware 2000*, n° 1795 LNCS, Springer-Verlag, 2000, p. 121–143.

[LUN 97] LUNAU C., « A Reflective Architecture for Process Control Applications », *ECOOP'97*, n° 1241 LNCS, Springer-Verlag, 1997, p. 170–189.

[MAL 01] MALENFANT J., SEGARRA M.-T., ANDRÉ F., « Dynamic Adaptability : the MolèNE Experiment », *Reflection 2001*, LNCS, Springer-Verlag, 2001.

[NIE 87] NIERSTRASZ O., « Active Objects in Hybrid », *OOPSLA'87, ACM Sigplan Notices*, vol. 22, n° 12, 1987, p. 243–253.

[RIV 84] DES RIVIÈRES J., SMITH B. C., « The implementation of procedurally reflective languages », *ACM LFP'84*, août 1984, p. 331–347.

[SCH 00] SCHMIDT D., STAL M., ROHNERT H., BUSCHMANN F., *Pattern-Oriented Software Architecture : Patterns for Concurrent and Networked Objects*, Wiley & Sons, 2000.

[SMI 82] SMITH B., « Reflection and Semantics in a Procedural Language », PhD thesis, MIT Laboratory for Computer Science, 1982, Research report 272.

[SMI 84] SMITH B., « Reflection and Semantics in Lisp », *ACM POPL'84*, p. 23–35.

[SOS 92] SOSIČ R., « Introspective computer systems », *Electrotechnical Review*, vol. 59, n° 5, 1992, p. 292–298.

[WAN 00] WANG N., KIRCHER M., SCHMIDT D., « Towards a Reflective Middleware Framework for QoS-Enabled CORBA Component Model Applications », *Actes électroniques du «Reflective Middleware Workshop»*, 2000.