
OpenCorba : un bus logiciel réflexif adaptable

Thomas Ledoux

*École des Mines de Nantes
4 rue Alfred Kastler, BP 20722
F-44307 Nantes Cedex 3 France
Thomas.Ledoux@emn.fr*

RÉSUMÉ. Aujourd'hui, l'architecture CORBA constitue la solution industrielle la plus prometteuse pour réaliser l'interopérabilité entre des composants logiciels répartis hétérogènes. Paradoxalement, alors que le projet CORBA cherche à fédérer des mécanismes de distribution au sein d'une même architecture, son modèle est peu flexible et paraît inadapté aux futures évolutions. Dans cet article, nous présentons OpenCorba, un ORB réflexif, capable d'adapter dynamiquement les stratégies de représentation et d'exécution du bus logiciel. Nous exposons tout d'abord les fondements réflexifs sous-jacents à la réalisation de OpenCorba. Nous montrons comment le concept de métaclasse permet l'isolement de propriétés spécifiques liées aux classes améliorant ainsi leur réutilisabilité. Puis, pour répondre aux problèmes des systèmes devant évoluer à l'exécution sans interrompre leur fonctionnement, nous introduisons un protocole de changement dynamique de métaclasse. S'appuyant sur ce cadre réflexif, OpenCorba permet de rendre plus « malléable » les caractéristiques internes du bus logiciel dans le but d'intervenir sur son modèle d'exécution répartie (e.g. invocation, contrôle de type IDL, gestion d'erreurs). OpenCorba favorise ainsi la construction d'architectures réparties réutilisables et adaptables, i.e. ouvertes.

ABSTRACT. Today, CORBA architecture brings the major industrial solution for realizing the interoperability between distributed software components in heterogeneous environments. While the CORBA project attempts to federate distributed mechanisms within a unique architecture, its model is not very flexible and seems to be not suitable to future evolutions. In this paper, we present OpenCorba, a reflective ORB, enabling users to adapt dynamically the representation and the execution policies of the software bus. We first expose the reflective foundations underlying the realization of OpenCorba : i) metaclasses which bring a better separation between specific properties of classes improving their reuse ; ii) a protocol for dynamically changing the metaclasses which allows run-time evolution of systems without stopping their execution. Based on this reflective environment, OpenCorba enables the adaptability of the internal characteristics of the bus in order to change its run-time model of distribution (e.g. invocation, IDL type checking, error handling). OpenCorba encourages the building of reusable and adaptable (i.e. open) distributed architectures.

MOTS-CLÉS : adaptabilité dynamique, CORBA, métaclasses

KEY WORDS: dynamic adaptability, CORBA, metaclasses

1. Introduction

Depuis quelques temps, l'engouement pour le réseau Internet souligne la nécessité de trouver rapidement des solutions pour supporter l'*interopérabilité* des environnements *répartis hétérogènes*. Aussi, l'une des préoccupations actuelles des acteurs de l'informatique est de réutiliser et d'assembler des composants logiciels, indépendamment de leurs langages d'implémentation, de leurs systèmes d'exploitation et de leurs sites d'exécution.

Les concepts de la technologie objet offrent de bonnes bases pour répondre à de tels défis. En normalisant des spécifications pour la portabilité et l'interopérabilité des composants objets, le consortium OMG (*Object Management Group*) apporte une réponse intéressante pour aborder le problème de la construction d'applications réparties à objets [OMG 95]. En effet, les efforts de standardisation de l'OMG ont permis de définir les spécifications d'une architecture globale modulaire. Celles-ci décrivent à divers degrés de détail les composantes indépendantes d'un système d'information, depuis le cœur technique jusqu'aux objets métiers, en passant par des services spécifiques (*CORBA services* [OMG 97]) comme la sécurité, les transactions, la notification d'événements, etc. Le bus logiciel CORBA (*Common Object Request Broker Architecture*), artère centrale de cette architecture, est responsable de la communication transparente entre objets distants, à travers des environnements répartis hétérogènes [OMG 98]. La *modularité* des composantes de l'architecture globale, proposée ainsi par l'OMG, constitue pour nous l'un des avantages majeurs de cette solution.

Cependant, cette volonté de *flexibilité* est en pleine contradiction avec la rigidité des spécifications du bus logiciel lui-même. Ce dernier est une « boîte noire » dont les spécifications sont volumineuses et le modèle objet figé. L'exemple du mécanisme d'invocation à distance – dont les spécifications sont fixées dans la norme – est significatif puisqu'un appel à propositions a été récemment lancé par l'OMG pour améliorer son modèle¹. C'est pourquoi nous nous interrogeons sur la pérennité de cette éventuelle amélioration. Aussi, pour faciliter les évolutions du modèle, il nous paraît nécessaire de trouver une réponse globale dans la capacité du bus à s'adapter.

La classification proposée par Jean-Pierre Briot et Rachid Guerraoui dans [BRI 96] constitue un cadre intéressant pour répondre à notre problème. Les auteurs distinguent trois approches pour l'intégration des concepts objets dans la programmation parallèle et répartie :

- l'approche applicative, basée sur le développement des concepts de distribution et de concurrence à travers des bibliothèques de classes ;
- l'approche intégrée, qui correspond à une extension des modèles objets pour la prise en compte de ces concepts ;
- l'approche réflexive, qui consiste en la définition de bibliothèques de méta-protocoles modélisant les concepts de distribution et de concurrence.

¹ C'est l'objectif de la proposition RFP Messaging Service.

En adoptant la classification proposée par Briot et Guerraoui, il apparaît que le modèle de l'OMG correspond à la fois à une *approche intégrée* (optique minimaliste du modèle de l'OMG) et à une *approche applicative* (optique bibliothèques des CORBA services). L'*approche réflexive*, qui permet de combiner les avantages des deux approches, est pour nous le meilleur choix conceptuel pour réaliser les futures évolutions du bus. La réflexion [SMI 82] [MAE 87] [KIC 91] permet en effet d'étendre le modèle initial de l'OMG avec des bibliothèques de méta-protocoles spécialisant les mécanismes de la programmation répartie. Il est alors possible d'introduire de nouvelles sémantiques sur ce modèle initial comme la concurrence, la réplication, la sécurité...

Dans cet article, nous présentons le bus logiciel OpenCorba [LED 98], une réalisation d'un bus CORBA basée sur une approche réflexive. Son architecture permet l'introspection et la modification des stratégies de représentation et d'exécution du bus logiciel CORBA. L'objectif de notre réalisation est d'abord de réifier les caractéristiques internes du bus logiciel dans le but de les modifier (intercession), mais surtout de les faire évoluer à l'exécution. En effet, pour s'adapter à des contextes d'exécution changeants (e.g. équilibrage de charge, tolérance aux fautes), les systèmes répartis doivent supporter la modification dynamique de leurs mécanismes. L'adaptabilité dynamique du bus rend alors possible l'instauration de nouvelles stratégies d'exécution (e.g. migration des objets). Celles-ci, mieux adaptées au comportement de l'application, contribuent ainsi à la construction d'un système ouvert. OpenCorba est ce bus logiciel réflexif *adaptable*.

OpenCorba est basée sur le langage réflexif NeoClasstalk [RIV 97]. Ce dernier est le résultat de l'implémentation d'un MOP (*Meta Object Protocol*) [KIC 91] dans le langage Smalltalk [GOL 89]. Sa principale contribution est d'étendre les aspects dynamiques de Smalltalk en proposant d'une part, une solution efficace pour contrôler l'envoi de messages et, d'autre part, un protocole de changement dynamique de classe.

Cet article est présenté comme suit. La section 2 expose les fondements réflexifs sous-jacents à la réalisation de OpenCorba et montre comment la réflexion contribue à la construction d'architectures ouvertes. Puis, après une brève introduction à OpenCorba, la section 3 décrit plus précisément trois aspects réflexifs du bus. La section 4 présente les travaux connexes alors que la section 5 introduit nos perspectives de travail. Enfin, nous concluons sur les apports de l'adaptabilité dynamique des systèmes dans le contexte des architectures réparties.

2. Un cadre réflexif pour la construction d'architectures ouvertes

Dans cette section, nous démontrons l'intérêt des métaclasse pour la construction d'architectures réutilisables et adaptables, i.e. *ouvertes*. Nous sommes profondément convaincus que – contrairement à ce qui a été dit dans le passé [STR 87] – la réflexion est un outil pertinent pour le génie logiciel. Dans ce qui suit,

nous allons montrer pourquoi les métaclasses peuvent être vues comme des composants logiciels réutilisables et adaptables.

2.1 Métaclasses et propriétés de classe

La réflexion permet de distinguer *ce que* fait un objet (son niveau de base) de *comment* il le fait (son méta-niveau) [McA 95]. Dans un langage réflexif, il existe donc une séparation clairement définie entre les fonctionnalités de l'application, programmées au niveau de base, et leurs représentations et contrôles, programmés au méta-niveau.

Dans le contexte des langages à classes réflexifs [COI 87] [DAN 94], la classe d'une classe – une *métaclasse* – va définir des propriétés liées à la création des objets, à leur encapsulation, aux règles d'héritage, à la résolution de l'envoi de messages, ... Nous nommons alors par *propriétés de classe*, ces propriétés propres à la classe, qui sont indépendantes du code que la classe décrit pour ses instances. Dans [LED 96], nous présentons une taxonomie de métaclasses modélisant des propriétés de classes : le fait de ne posséder qu'une seule instance, d'interdire la création de sous-classe, de réaliser des pré/post conditions sur les méthodes, ...

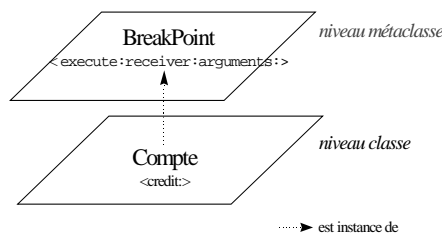


Figure 1. *Propriété de classe et (méta)classes*

La Figure 1 montre l'exemple d'une classe `Compte`, instance de la métaclasse `BreakPoint`, chargée de fixer des points d'arrêts sur les méthodes de la classe `Compte`². Le contrôle de l'envoi de messages réifié par le MOP de NeoClasstalk – via la méthode `execute:receiver:arguments:` – va permettre à la métaclasse `BreakPoint` d'intercepter les messages reçus par une instance de la classe `Compte`. La méthode `execute:receiver:arguments:` décrite ci-dessous, ouvre un débogueur sur le contexte d'exécution des méthodes de `Compte` (e.g. `credit:`), puis appelle la méthode standard d'invocation de messages grâce à l'héritage.

² Le temps de la mise au point d'un programme, il est intéressant de contrôler les interactions d'une classe avec le reste du système.

```
BreakPoint>>execute: cm receiver: rec arguments: args
    "Fixer un point d'arrêt sur les méthodes"
    self halt: 'BreakPoint for ', cm selector.
    ^super execute: cm receiver: rec arguments: args
```

Code métaclasse

```
Compte>>credit: aFloat
    "Réaliser un crédit sur le solde courant"
    self solde: self solde + aFloat
```

Code classe

En considérant les classes comme objet de plein droit, nous avons pu éviter l'entrelacement entre le code métier (compte bancaire) et le code décrivant une propriété spécifique sur la classe métier (point d'arrêt). Implémentées par les métaclasses, les *propriétés de classe* encouragent la lisibilité et la réutilisabilité du code dans le développement des classes.

2.2 Changement dynamique de métaclasse

Le *changement dynamique de classe* de NeoClasstalk est un protocole permettant aux objets de changer de classe à l'exécution [RIV 97]. Ce protocole a pour but de tenir compte de l'évolution du comportement des objets dans le temps, et permet d'améliorer ainsi l'implémentation de leurs classes respectives. L'association des classes comme objet de plein droit avec le protocole de changement dynamique de classe permet le *changement dynamique de métaclasse* à l'exécution. Puisque les propriétés de classe sont implémentées par des métaclasses, notre mise en œuvre autorise ainsi l'*adaptabilité dynamique des propriétés de classe*. Le protocole de changement dynamique de classe rend en effet possible l'ajout et le retrait des propriétés de classe à l'exécution sans re-génération de code au niveau des classes.

En reprenant l'exemple précédent, on peut associer temporairement la propriété BreakPoint à une classe le temps de sa mise au point. La classe Compte « bascule » de sa métaclasse d'origine³ vers la métaclasse BreakPoint pour déboguer les messages, puis revient vers son état antérieur par un nouveau basculement une fois le programme réalisé (cf. Figure 2).

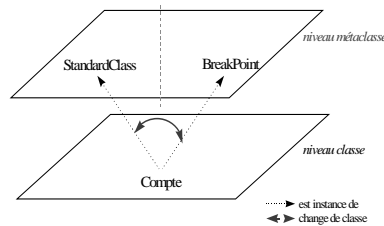


Figure 2. Adaptabilité dynamique des propriétés de classe

³ StandardClass est la racine de toutes les métaclasses. Elle décrit le comportement standard des classes.

Le protocole de changement dynamique de métaclasse permet à un système de remplacer une propriété de classe par une autre, pendant son exécution, sans affecter le reste du système. Pour nos travaux sur les bus logiciels *adaptables*, ce protocole est extrêmement important car il autorise la modification des mécanismes des architectures réparties à l'exécution. Ainsi, une invocation synchrone peut devenir asynchrone, un objet volatil peut devenir persistant, un objet passé en valeur dans une requête peut être passé par référence, un objet *proxy* peut gérer un cache, etc.

3. OpenCorba : un ORB ouvert

Notre première réalisation d'une architecture ouverte s'est déroulée dans le contexte de la plate-forme répartie CORBA [OMG 98] et a donné lieu à l'implémentation d'un bus logiciel nommé OpenCorba [LED 98]. OpenCorba est une application implémentant les API CORBA dans NeoClasstalk, i.e. un *ORB* (*Object Request Broker*) *réflexif*. Il réifie différentes propriétés du bus CORBA par le biais des métaclasses afin de favoriser la séparation des caractéristiques internes de l'ORB. L'utilisation du protocole de changement dynamique de métaclasse permet alors d'adapter le comportement du bus à l'exécution en modifiant les mécanismes de l'ORB représentés par des métaclasses.

Dans les paragraphes suivants, nous allons présenter trois aspects du bus parmi ceux que nous avons réifiés : le mécanisme d'invocation à distance via un *proxy*, la gestion d'erreurs lors de la création du dépôt d'interfaces⁴, et le contrôle de type IDL sur la classe serveur. Auparavant, nous présentons la projection IDL OpenCorba et les classes Smalltalk DirectToOpenCorba qui permettent la mise en œuvre de ces trois aspects réflexifs.

3.1 Création des classes proxies et serveurs dans OpenCorba

3.1.1 Projection IDL OpenCorba

Le pré-compilateur IDL OpenCorba génère une classe *proxy* sur le client et une classe patron sur le serveur en respectant le *mapping* Smalltalk de la norme CORBA [OMG 98]. La classe *proxy* est alors associée à la métaclasse `ProxyRemote` implémentant l'invocation à distance ; la classe patron, à la métaclasse `TypeChecking` implémentant le contrôle de type sur le serveur. La partie gauche de la Figure 3 présente les résultats de la projection IDL de l'interface `Compte` dans OpenCorba : la classe *proxy* `CompteProxy` est instance de `ProxyRemote` et la classe patron `Compte` est instance de `TypeChecking`.

⁴ Rappelons que le dépôt d'interfaces est semblable à une base de données contenant l'ensemble des interfaces IDL sous la forme d'objets accessibles à l'exécution.

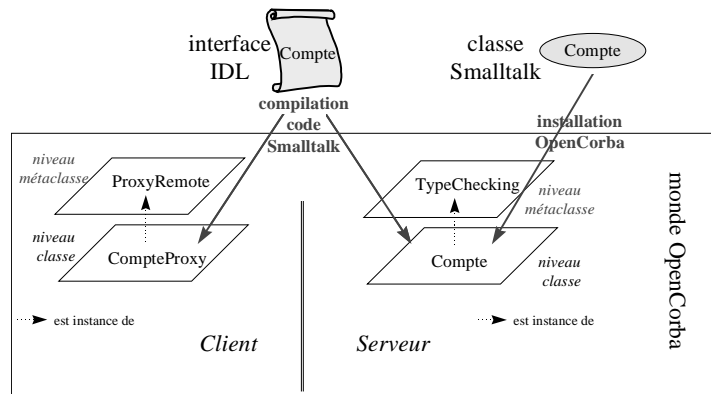


Figure 3. Création des classes proxies et serveurs dans OpenCorba

3.1.2 Classe Smalltalk DirectToOpenCorba

OpenCorba permet aux développeurs Smalltalk de transformer toute classe standard Smalltalk en une classe serveur Smalltalk dans l'ORB, nommée classe Smalltalk DirectToOpenCorba⁵. Cette fonctionnalité est introduite d'une part pour réutiliser du code Smalltalk existant, d'autre part pour décharger le programmeur de l'écriture du fichier IDL. Pour posséder le statut de classe serveur dans OpenCorba, une classe DirectToOpenCorba est instance de `TypeChecking` (cf. Figure 3, partie droite).

3.2 Proxy OpenCorba

Dans les architectures réparties, l'objet *proxy* est une représentation locale côté client de l'objet serveur. Sa mission est d'assurer la création des requêtes et leur acheminement vers le serveur, puis de retourner le résultat au client. Pour cela, une classe *proxy* adapte le style d'appel local au mécanisme d'invocation à distance [SHA 86].

3.2.1 Propriété de classe

L'invocation à distance est totalement indépendante de la sémantique de l'interface IDL d'origine. Aussi, l'invocation à distance se rapporte au contrôle de l'application et non au code de l'application : elle relève de la méta-programmation et constitue une propriété de classe. Le pré-compilateur IDL OpenCorba génère la

⁵ Cette terminologie fait référence au compilateur DirectToSOM C++ dont nous reprenons la philosophie

classe *proxy* au niveau de base de l'application et associe la classe *proxy* à la métaclasse `ProxyRemote`, chargée d'invoquer l'objet réel. L'invocation à distance reste ainsi transparente au client. Ce dernier peut alors manipuler la classe *proxy* avec les outils classiques Smalltalk (butineur, inspecteur).

Détails au niveau de base

Les méthodes de la classe *proxy* générée sont purement descriptives et ne présentent qu'une apparence d'interface comme les opérations IDL [LED 97]. Dans le butineur Smalltalk, les méthodes ne contiennent aucun code, seulement un commentaire indiquant qu'elles ont été générées.

<i>Attribut/opération IDL</i>	<i>méthodes d'une classe proxy OpenCorba</i>
<code>readonly attribute float solde;</code>	<code>solde "Generated by OpenCORBA *** DO NOT EDIT ***" "aFloat "</code>
<code>void credit(in float montant);</code>	<code>credit: aFloat "Generated by OpenCORBA *** DO NOT EDIT ***" "^nil"</code>

Tableau 1. Exemples de correspondance IDL vers proxy

Le Tableau 1 présente des exemples de projection pour un attribut et une opération de l'interface IDL `Compte` vers les méthodes Smalltalk correspondantes de la classe *proxy* `CompteProxy`. Le type des valeurs retournées est mis en commentaires pour signaler les informations IDL au programmeur.

Détails au méta-niveau

Par définition, l'exécution d'un message sur un objet *proxy* entraîne une invocation sur l'objet serveur qu'il représente. L'idée est donc d'intercepter le message au moment de sa réception pour lancer une invocation à distance. Aussi, la technique du contrôle de l'envoi de messages répond bien à cette attente. La métaclasse `ProxyRemote` redéfinit la méthode `execute:receiver:arguments:` du MOP `NeoClasstalk` pour intercepter les messages reçus par un *proxy*. Cette dernière réalise alors l'invocation à distance en utilisant les API du DII CORBA conformément à [OMG 98].

3.2.2 Adaptabilité dynamique des propriétés de classe

Pour permettre l'adaptabilité dynamique de l'invocation dans OpenCorba, il est possible de développer d'autres métaclasses. Nous les avons distinguées en deux catégories :

- La première traite des *variations* possibles sur la gestion des *proxies*. Nous pensons à des mécanismes modélisant Java RMI, une future version du DII CORBA ou tout simplement une invocation en local.

Ce dernier mécanisme a fait l'objet d'une implémentation dans OpenCorba. Nous adaptons le comportement du *proxy* suite au chargement de la classe serveur dans l'espace mémoire du client. Pour cela, nous avons développé une nouvelle métaclasse, `ProxyLocal`, chargée de réaliser des invocations sur un objet serveur local. La Figure 4 montre qu'une instance d'une classe *proxy* peut effectuer des invocations distantes à l'instant t , puis des invocations locales à l'instant $t+1$, uniquement grâce au changement dynamique de métaclasse (i.e. *sans* modifier le code du niveau de base).

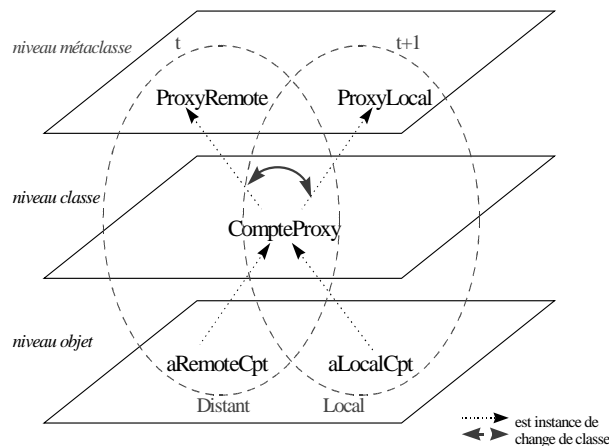


Figure 4. *Adaptabilité dynamique des proxies*

– La deuxième catégorie traite des *extensions* de la notion de *proxy* introduisant de nouveaux mécanismes comme la *migration* [JUL 88] [OKA 94] ou la *réplication active* [BIR 91] [FEL 97].

La migration d'objet consiste à transférer un objet serveur⁶ vers la plate-forme cliente afin d'optimiser les performances du système réparti. Ce mécanisme allège les goulots d'étranglement du réseau et minimise les communications distantes. Le client manipule toujours un *proxy*, mais celui-ci se comporte comme un cache local de l'objet serveur.

La réplication est un autre mécanisme de gestion de la répartition des objets. Elle consiste en une duplication du serveur en plusieurs réplicas qui sont la représentation exacte du serveur original. Le mécanisme de réplication active suppose que le message est envoyé par le client aux réplicas – via le *proxy* – grâce à un protocole de diffusion atomique (*broadcast*).

En conclusion, l'adaptabilité dynamique des propriétés de classe permet d'implémenter des variations sur le mécanisme d'invocation du système réparti, sans bouleverser l'architecture existante.

⁶ Ou une partie de l'objet serveur, c'est à dire soit ses méthodes, soit ses valeurs.

3.3 Dépôt d'interfaces et gestion d'erreurs

La compilation IDL et l'installation d'une classe Smalltalk DirectToOpenCorba sont les deux moyens utilisés pour remplir le dépôt d'interfaces. Dans le premier cas, la génération du *proxy* est accompagnée par la création des objets du dépôt. Dans le second cas, une table d'équivalence Smalltalk vers IDL (*rétro-mapping*) permet cette création. Techniquement, ces deux mécanismes ne possèdent pas le même degré d'intercession :

- Dans le cas des classes DirectToOpenCorba, l'analyse syntaxique et la vérification sémantique de la classe ont déjà été effectuées par le compilateur Smalltalk. Aussi, l'installation d'une classe Smalltalk dans l'ORB ne provoque pas d'erreurs lors de la création des objets dans le dépôt ;

- Par contre, pour le cas IDL, il s'agit d'une vraie compilation où les actions sémantiques se chargent entre autres de vérifier l'intégrité du fichier IDL avant la création des objets dans le dépôt (e.g. pas deux fois le même attribut).

Ainsi, nous devons encapsuler les API de création CORBA du dépôt d'interfaces afin de différencier le cas Smalltalk du cas IDL où la création peut conduire à une gestion d'erreurs. Rappelons que ces API de création sont implémentées par les classes *contenantes* du dépôt d'interfaces spécifiées par la norme CORBA [OMG 98].

3.3.1 Propriété de classe

En analysant les tests d'intégrité nécessaires à la compilation IDL, il apparaît qu'ils sont génériques et indépendants du code des API de création. Ainsi, ils peuvent être externalisés des classes contenantes pour constituer une propriété de classe qui sera implémentée par une métaclasse. Le code des classes contenantes n'effectue donc aucun test dans les méthodes de création. Il est alors plus simple de différencier le cas IDL du cas Smalltalk en associant ou non la métaclasse aux classes contenantes du dépôt.

Soit `IRChecking`, cette métaclasse. Elle spécialise la méthode `execute:receiver:arguments:` du MOP `NeoClasstalk` pour intercepter les messages reçus par les instances de la classe contenante. Pour les messages de type création, les tests d'intégrité sont réalisés et une exception est levée si ils ne sont pas vérifiés.

3.3.2 Adaptabilité des propriétés de classe

Notre conception conduit à une plus grande flexibilité pour effectuer ou faire évoluer les tests d'intégrité sans avoir à modifier les API de création. Ainsi, elle permet la distinction entre le cas Smalltalk et le cas IDL à l'exécution. Dans le cas Smalltalk, la création s'effectue normalement (métaclasse par défaut `StandardClass`); dans le cas IDL, il y a adaptation du comportement pour effectuer la création seulement si les tests d'intégrité ne lèvent pas d'erreur

(métaclasse `IRChecking`). OpenCorba échange les métaclasses à l'exécution selon les besoins du système.

3.4 *Serveur et contrôle de type IDL*

Dans la norme CORBA, il est spécifié que le dépôt d'interfaces est utilisé pour vérifier la conformité de la signature de la requête, avant et après application de la méthode par le serveur [OMG 98]. Par contre, elle ne précise pas comment doit être effectuée cette vérification de type.

3.4.1 *Propriété de classe*

Nous sommes convaincus qu'il est du ressort de la classe serveur de vérifier si une de ses méthodes peut être appliquée ou non. En effet, si ce genre de vérification est effectué plus en amont par l'ORB (au moment du dépaquetage de la requête par le serveur par exemple), cela introduit une gestion plus rigide puisque le moindre changement dans le mécanisme de contrôle de type entraîne une réécriture des couches basses de l'ORB. Au contraire, nous proposons d'externaliser le contrôle de type de ces couches, pour le déléguer à la classe serveur qui va tester si l'application de ses méthodes est possible ou non.

De plus, le contrôle de type est indépendant des fonctionnalités définies par la classe serveur : il peut être séparé du code de l'application et constitué une propriété de classe qui sera implémentée par une métaclasse. Ainsi, le code de la classe serveur n'effectue aucun test sur le type des données passées en arguments. Le développeur peut alors écrire, modifier, récupérer son code sans se soucier du contrôle de type effectué par la métaclasse `TypeChecking`.

Techniquement, cette métaclasse contrôle l'envoi de messages sur la classe serveur – via la méthode `execute:receiver:arguments:` – pour interroger le dépôt d'interfaces avant et après application de ses méthodes. Le dépôt nous permet alors de vérifier le type des arguments à l'appel et le type du résultat au retour de la méthode.

En conclusion, OpenCorba externalise le contrôle de type, à la fois des couches basses de l'ORB, et de la classe serveur.

3.4.2 *Adaptabilité des propriétés de classe*

Grâce à notre approche, nous pouvons apporter de nouveaux mécanismes de contrôle de type, sans remettre en cause l'implémentation existante. Par exemple, nous pouvons développer une nouvelle métaclasse gérant un *système de cache* pour les types des paramètres. À la première interrogation du dépôt d'interfaces, OpenCorba sauvegarde le type de chaque argument de la méthode localement⁷. Puis,

⁷ Par exemple, dans un dictionnaire partagée Smalltalk ou encore dans le code octal de la méthode compilée.

aux prochaines invocations de la méthode, la métaclasse interroge les informations mémorisées afin de réaliser le contrôle de type.

Autre exemple : nous pouvons aussi nous débarrasser du contrôle de type pour des raisons de performance ou lorsque le type des paramètres est connu. Un changement dynamique de métaclasse permet alors d'associer la classe serveur à la métaclasse par défaut du système OpenCorba.

4. Travaux connexes

Programmation par aspects

L'un des obstacles majeurs à la réutilisabilité des composants logiciels est lié à l'entrelacement dans le programme fonctionnel de plusieurs *aspects* techniques (e.g. répartition, synchronisation, gestion mémoire) dépendant du domaine de l'application. Une solution possible est alors d'envisager l'isolement de ces aspects spécifiques pour leur réutilisation. Le programme fonctionnel et les aspects se retrouvent ainsi séparés et peuvent évoluer indépendamment, formulant ainsi le paradigme de séparation des aspects (*separation of concerns* [HUR 95]).

Il existe un certain nombre de modèles et techniques permettant la séparation des aspects : les filtres de composition [BER 94], la programmation adaptative [LIE 96], la programmation orientée aspects (AOP) [KIC 97]. Cependant, ces derniers mettent en œuvre des constructions particulières pour achever la programmation par aspects. Aussi, nous avons préféré une approche réflexive qui n'impose pas un nouveau modèle, mais étend les langages existants pour les rendre ouverts. De plus, contrairement à ces modèles, notre solution rend possible l'adaptabilité des aspects à l'exécution.

Systèmes distribués réflexifs

L'utilisation de la réflexion dans les systèmes concurrents et répartis n'est certes pas récente [WAT 88], mais elle semble prendre un nouvel essor depuis quelques années. En effet, de nombreux projets de recherche dans le domaine de la réflexion ou des systèmes répartis abordent la réification des mécanismes de distribution pour les modifier et les spécialiser. Citons entre autres les mécanismes de migration [OKA 94], de *marshalling* [McA 95], de réplication [GOL 97], de sécurité [PER 97], ... L'architecture CORBA constitue une plate-forme fédératrice des différents mécanismes de distribution. Aussi, il nous semble intéressant de nous inspirer de ces projets pour implémenter ces mécanismes dans OpenCorba.

Middleware réflexifs adaptatifs

Parallèlement au bus OpenCorba, d'autres projets de recherche sont en cours de réalisation dans le domaine des *middleware* adaptatifs. Le projet ADAPT de l'Université de Lancaster s'intéresse à l'implémentation de plates-formes adaptatives pour les applications multimédia mobiles [BLA 97] et propose une approche pour la

conception de *middleware* ouverts [BLA 98]. Fort de son expérience dans l'élaboration de MOP [GOW 96], l'équipe systèmes distribués du Trinity College de Dublin a récemment démarré le projet Coyote qui a pour but de satisfaire l'adaptation des systèmes d'information (e.g. administration de réseaux de télécommunications, bus CORBA). Enfin, une équipe de l'Université de l'Illinois expérimente la réflexion pour étendre le bus logiciel CORBA avec des mécanismes de temps réel [SIN 97]. Chacun de ces projets utilise les techniques de réflexion pour mettre en œuvre l'adaptabilité des *middleware* à l'exécution.

5. Perspectives

Composition des métaclasses

Les architectures réparties sont complexes et nécessitent de prendre en compte de nombreux mécanismes pour leur modélisation. La question de la composition de ces mécanismes s'impose alors inéluctablement. Par exemple, la fonctionnalité « tracer les méthodes d'une classe réalisant des invocations distantes » utilise le mécanisme de « trace » conjuguée avec le mécanisme d'« invocation à distance ». Comme nous l'avons vu précédemment, chacun de ces mécanismes correspond à une propriété de classe et est implémenté par une métaclasse. Aussi, la combinaison de plusieurs mécanismes soulève le problème de la composition des métaclasses : cette composition est à priori source de conflits quand il existe un chevauchement des comportements [BS 98a].

Pour aborder ce problème, nos récents travaux ont consisté en la définition d'un « modèle de compatibilité » de métaclasses afin d'offrir un cadre fiable pour la composition des métaclasses [BS 98b]. Nos travaux actuels s'attachent à définir une classification des différents mécanismes de répartition pour prévenir des éventuels chevauchements de comportement.

Perspectives autour de OpenCorba

Pour élargir l'expérimentation de l'adaptabilité dynamique à notre système, il est intéressant d'implanter d'autres stratégies de communication entre objets répartis. Nous pensons plus particulièrement à des modèles de communication orientée messages du type *publish/subscribe* ou aux objets mobiles.

Par ailleurs, à l'instar des *patterns* de conception pour la programmation [GAM 95], il est important de s'abstraire d'un langage particulier et de raisonner sur des schémas de méta-programmation. Bien sûr, chaque langage possède sa propre sémantique, mais nous considérons qu'il est possible de spécifier des schémas réutilisables de métaclasses pour la construction d'applications réparties.

6. Conclusion

Dans cet article, nous avons développé l'idée selon laquelle la réflexion constitue un vecteur formidable pour la construction des architectures ouvertes. Les langages considérant les classes comme entités de plein droit mettent en relief le concept de *propriété de classe*. Implémentées par des métaclasse, les propriétés de classe encouragent la lisibilité, la réutilisabilité et la qualité du code en améliorant la séparation des différents aspects d'une classe. Les métaclasse, associées au *changement dynamique de métaclasse*, constituent alors un cadre réflexif privilégié pour la construction d'architectures réutilisables et adaptables, i.e. *ouvertes*.

Dans le contexte des environnements répartis, ce cadre réflexif offre un modèle dynamique permettant la réutilisation et l'adaptabilité des mécanismes liés à la répartition. Notre première expérimentation nous a permis « d'ouvrir » les différentes caractéristiques internes du bus logiciel CORBA, comme le mécanisme d'invocation via un *proxy*. Le résultat, OpenCorba, est un ORB ouvert, capable d'adapter dynamiquement les stratégies de représentation et d'exécution du bus logiciel.

Remerciements

Je tiens à remercier l'ensemble de l'équipe objet de l'École des Mines de Nantes pour leurs remarques constructives lors de la rédaction de cet article. Je remercie encore chaleureusement Laure-Anne Lagier, Noury Bouraqadi-Saadani et Fred Rivard pour leurs encouragements lors du développement de OpenCorba.

Bibliographie

- [BER 94] BERGMANS L. — *Composing Concurrent Objects*. PhD thesis, University of Twente, Enschede, Pays-Bas, Juin 1994.
- [BIR 91] BIRMAN K., SCHIPER A., STEPHENSON P. — Lightweight Causal and Atomic Group Multicast. In *ACM Transactions on Computer Systems*, vol.9, n°3, p.272-314, 1991.
- [BLA 97] BLAIR G.S., COULSON G., DAVIES N., ROBIN P., FITZPATRICK T. — Adaptive Middleware for Mobile Multimedia Applications. In *Proceedings of the 8th International Workshop on NOSSDAV'97*, St-Louis, Missouri, Mai 1997.
- [BLA 98] BLAIR G.S., COULSON G., ROBIN P., PAPATHOMAS M. — An Architecture for Next Generation Middleware. In *Proceedings of Middleware'98*, Springer-Verlag, p.191-206, N. Davies, K. Raymond, J. Seitz Eds, Septembre 1998.
- [BRI 96] BRIOT J.P., GUERRAOUI R. — Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances. In *TSI*, Hermès, vol.15, No.6, Paris, Juin 1996.
- [BS 98a] BOURAQADI-SAADANI N., RIVARD F., LEDOUX T. — Composition de métaclasse. In *Actes JFLA'98*, Lac de Côme, Italie, Février 1998.

- [BS 98b] BOURAQADI-SAADANI N., LEDOUX T., RIVARD F. — Safe Metaclass Programming. In *Proceedings of OOPSLA'98*, ACM Sigplan Notices, Vancouver, Octobre 1998.
- [COI 87] COINTE P. — Metaclasses are First Class : the ObjVlisp Model. In *Proceedings of OOPSLA'87*, ACM Sigplan Notices, p.156-167, Orlando, Floride, Octobre 1987.
- [DAN 94] DANFORTH S., FORMAN I.R. — Reflections on Metaclass Programming in SOM. In *Proceedings of OOPSLA'94*, ACM Sigplan Notices, Portland, Oregon, Octobre 1994.
- [FEL 97] FELBER P., GUERRAOUY R., SCHIPER A. — A CORBA Object Group Service. In *Workshop CORBA : Implementation, Use and Evaluation*, ECOOP'97, Jyväskylä, Finlande, Juin 1997.
- [GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES John — *Design Patterns*, Addison-Wesley Reading, Massachusetts, 1995.
- [GOL 89] GOLDBERG A., ROBSON D. — *Smalltalk-80 : The Language*, Addison-Wesley, Reading, Massachusetts, 1989.
- [GOL 97] GOLM M. — *Design and Implementation of a Meta Architecture for Java*. Master's Thesis, University of Erlangen, Janvier 1997.
- [GOW 96] GOWING B., CAHILL V. — Meta-Object Protocols for C++ : The Iguana Approach. In *Proceedings of Reflection'96*, Ed. Kiczales, San Francisco, California, Avril 1996.
- [HAM 97] HAMILTON J. — *Programming with DirectToSOM C++*, John Wiley & Sons Inc., New-York, 1997.
- [HUR 95] HÜRSH W.L., LOPES C.V. — *Separation of Concerns*. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, Février 1995.
- [JUL 88] JUL E., LEVY H., HUTCHINSON N., BLACK A. — Fine-grained mobility in the Emerald system. In *ACM Transactions on Computer Systems*, vol.6(1), p.109-133, Février 1988.
- [KIC 91] KICZALES G., DES RIVIERES J., BOBROW D.G.— *The Art of the Metaobject Protocol*. The MIT Press, 1991.
- [KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.M., IRWIN J. — Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, LNCS 1241, Springer-Verlag, p.220-242, Jyväskylä, Finlande, Juin 1997.
- [LED 96] LEDOUX T., COINTE P. — Explicit Metaclasses as a Tool for Improving the Design of Class Libraries. In *Proceedings of ISOTAS'96*, LNCS 1049, p.38-55, Springer-Verlag, Kanazawa, Japon, Mars 1996.
- [LED 97] LEDOUX T. — Implementing Proxy Objects in a Reflective ORB. In *Workshop CORBA : Implementation, Use and Evaluation*, ECOOP'97, Jyväskylä, Finlande, Juin 1997.
- [LED 98] LEDOUX T. — *Réflexion dans les systèmes répartis : application à CORBA et Smalltalk*. Thèse de doctorat, Université de Nantes, École des Mines de Nantes, Mars 1998.

- [LIE 96] LIEBERHERR K.J. — *Adaptive Object-Oriented Software : The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
- [MAE 87] MAES P. — Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, ACM Sigplan Notices, p.147-155, Orlando, Floride, Octobre 1987.
- [McA 95] MCAFFER J. — Meta-level architecture support for distributed objects. In *Proceedings of IWOOS'95*, p.232-241, Lund, Sweden, 1995.
- [OKA 94] OKAMURA H., ISHIKAWA Y. — Object Location Control Using Meta-level Programming. In *Proceedings of ECOOP'94*, p.299-319, LNCS 821, Springer-Verlag, Juillet 1994.
- [OMG 95] OBJECT MANAGEMENT GROUP — *Object Management Architecture Guide*, Revision 3.0. OMG TC Document ab/97-05-05, Juin 1995.
- [OMG 97] OBJECT MANAGEMENT GROUP — *CORBAservices: Common Object Services Specification*. OMG TC Document formal/98-07-05, Novembre 1997.
- [OMG 98] OBJECT MANAGEMENT GROUP — *The Common Object Request Broker : Architecture and Specification*, Revision 2.2. OMG TC Document formal/98-07-01, Février 1998.
- [PER 97] PERENNOU T. — *Une architecture à métaobjets pour systèmes répartis tolérant les fautes*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Janvier 1997.
- [RIV 97] RIVARD F. — *Évolution du comportement des objets dans les langages à classes réflexifs*. Thèse de doctorat, Université de Nantes, École des Mines de Nantes, Juin 1997.
- [SHA 86] SHAPIRO M. — Structure and Encapsulation in Distributed Systems : The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, p.198-204, Cambridge, MA, Mai 1986.
- [SIN 97] SINGHAI A., SANE A., CAMPBELL R. — Reflective ORBs : Supporting Robust, Time-critical Distribution. In *Workshop Reflective Real-Time Object-Oriented Programming and Systems*, ECOOP'97, Jyväskylä, Finlande, Juin 1997.
- [SMI 82] SMITH B.C. — *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, MIT, Janvier 1982.
- [STR 87] STROUSTRUP B. — What is "Object-Oriented Programming" ?. In *Proceedings of ECOOP'87*, p.57-76, LNCS 276, Special issue of Bigre 54, Paris, 1987.
- [WAT 88] WATANABE T., YONEZAWA A. — Reflection in an Object-Oriented Concurrent Language. In *Proceedings of OOPSLA'88*, ACM Sigplan Notices, p.306-315, San Diego, California, Septembre 1988.