Utilisation de la réflexivité dans SO3

Par Marc ROZIER

1. TABLE DES MATIERES

2. INTRODUCTION

Les applications actuelles exigent de plus en plus de qualité de service, notamment au niveau des systèmes opératoires, programmes complexes et difficiles à mettre en œuvre. De plus, ces programmes doivent être performants et inclure, de manière transparente à l'utilisateur, des services variés : sécurité, répartition, persistance, tolérance aux fautes, etc.

Actuellement, un système opératoire réalise toujours certains choix : choix d'une politique de scheduling, choix de la gestion mémoire, choix d'un type de système de gestion de fichiers (SGF), etc. De plus, ils doivent s'adapter aux souhaits de l'utilisateur, ce qui implique que certaines propriétés du système doivent être modifiées lors de l'exécution. En programmation « classique », cet aspect est difficile à concevoir : si ces fonctionnalités ont été implantées dès le début, il n'a pas de problème. Par contre, rajouter des propriétés après, relève parfois du défi. De même, la modification de propriétés ou d'attributs pendant le déroulement du programme, s'avère complexe. La notion de réflexivité a pour but de contribuer à cette adaptation.

Dans un premier temps, nous définirons la réflexivité et nous en présenterons certains aspects. Afin d'illustrer cette définition, nous étudierons quelques cas d'applications réflexives pour présenter les différents types de problèmes qu'elle est susceptible de résoudre. Nous détaillerons quelques exemples de méta-langages supportant la réflexivité. En effet, pour implanter de la réflexivité, des langages spécifiques sont nécessaires. Enfin, un exemple de système opératoire orienté objets réflexif sera développé.

Dans un deuxième temps, nous aborderons le projet SO^3 (Système Opératoire Orienté Objets). Ce projet a pour but de concevoir un système opératoire en utilisant au maximum les concepts objets tels que l'héritage et la généricité. Nous présenterons SO^3 dans son intégralité, c'est-à-dire les concepts originaux et les extensions envisagées. Un système opératoire actuel se doit de prendre en compte la répartition, c'est ce que nous analyserons pour conclure cette partie.

Dans certains types d'application (comme un système opératoire réparti, par exemple), la notion de réflexivité peut prendre un autre sens, dans la mesure où elle est considérée comme un outil, une aide au développement. C'est pourquoi dans cette troisième partie, nous nous intéresserons aux méthodes de superposition de la réflexivité sur un système opératoire. Pour commencer, nous situerons le travail concernant la réflexivité par rapport au reste du projet et par rapport à la répartition. Puis, les problèmes apparus lors de l'élaboration du modèle seront décrits, ainsi que les méthodes pour les résoudre.

3. REFLEXIVITE ET META-CLASSES

3.1. Réflexivité

Dans cette partie, nous allons définir les besoins qui ont conduit à la conceptualisation de la réflexivité. Ensuite, nous clarifierons ces notions par la définition des termes spécifiques pour la réflexivité. Nous donnerons un aperçu de la raison pour laquelle les objets et la réflexivité vont de pair. La réflexivité appliquée sur les classes donne le concept de méta-classe, que nous étudierons dans une quatrième partie.

3.1.1. Besoins

Il est indéniable que la programmation répartie a besoin de souplesse et de flexibilité, pendant la conception (au niveau de la réalisation) et pendant l'exécution (au niveau de la transparence).

En effet, au niveau conceptuel, il faut que la répartition s'intègre à l'intérieur des composants du système et ne soit pas une entrave à la programmation. De même, durant l'exécution, la répartition doit être transparente pour l'utilisateur et doit prendre en compte les modifications (voulues ou non voulues) du système et du réseau (cf. chapitre 3.2.3 « Mécanisme de tolérance aux fautes »). Le noyau de répartition doit pouvoir s'adapter le cas échéant.

Dans ce but, le concepteur doit posséder un outil capable d'offrir une certaine simplicité dans la réalisation d'une application, une méthode de programmation qui permette de contrôler les effets de la répartition et tous les problèmes résultants. La majorité des problèmes survenant pendant l'exécution, une stratégie spéciale se montre nécessaire afin de contrôler le programme. C'est la réflexivité. Grâce à cette notion, il devient facile de contrôler le déroulement d'un programme pendant son exécution. Mais tout d'abord, regardons de près la définition de la réflexivité.

3.1.2. Définitions

On peut définir la *réflexivité* par la capacité d'un programme d'observer ou de changer son propre code aussi bien que sa syntaxe, sa sémantique ou son implantation, à la compilation ou à l'exécution. [MJD96]

Souvent est fait appel à la notion de *réification*. On peut définir la reification par la représentation d'un attribut d'un objet comme membre de cet objet, c'est-à-dire la représentation des données et des fonctions comme des données. Le terme employé est réifier.

L'idée phare de la réflexivité est de programmer son application sans se préoccuper des traitements réflexifs ultérieurs. Puis, un méta-source est superposé sur le source initial : il pourra réaliser des modifications sur le source de l'application. Par exemple, si un développeur écrit un programme quelconque, une autre personne pourra alors écrire, au

dessus du source, un méta-source qui se chargera de la gestion de l'objet, indépendamment du rôle original de l'objet.

Définitions nécessaires pour le reste du document :

Termes	Définition
Méta-source	Le texte source qui se trouve dans la partie réflexive.
Méta-code	Le code (exécutable) réflexif.
Méta-objet	Objet utilisant la réflexivité, instanciation d'une méta-classe.
Méta-classe	Classe réflexive, ensemble de méta-méthodes et d'attributs.
Méta-méthode	Le méta-source de la méthode.

3.1.3. Des Objets vers les Méta-Objets

L'approche par objets apporte certains avantages [Strous92] en plus de l'héritage et du polymorphisme : la modularité, la réutilisabilité, l'extensibilité et la portabilité.

La modularité, déjà incluse dans les langages procéduraux, permet de découper un problème complexe en éléments simples et facilement manipulables. Il devient possible pour le programmeur de travailler sur ces éléments de façon indépendante. Ce dernier pourra aussi les tester indépendamment.

La réutilisabilité vient du besoin de récupérer les modèles déjà écrits et testés. Cette notion permet à un programmeur de s'atteler exclusivement à son application, sans perdre du temps et des efforts à écrire du code déjà réalisé par un autre. De plus, il est certain que le programmeur peut trouver des modèles classiques d'objets (gestion de files, de listes, de piles…) afin de les réutiliser pour ses propres programmes.

L'extensibilité est la capacité d'adaptation d'un objet au changement de sa spécification. Un programmeur peut donc modifier ou ajouter des propriétés au modèle, sans avoir tout à réécrire. Cette notion permet de rajouter de nouveaux concepts ou fonctionnalités au programme initial.

La portabilité est la facilité avec laquelle un objet (ou logiciel) peut être adapté à différents environnements logiciels et matériels. A la vue des progrès actuels en informatique, tant au niveau logiciel (systèmes opératoires, en particulier) que matériel (architecture des ordinateurs), la portabilité est une notion importante.

3.1.4. Méta-classe

La classe est la définition d'un modèle d'objet. Elle définit le comportement de cette entité et elle encapsule les définitions des données et des méthodes (fonctions et procédures). Elle hérite parfois du comportement d'une autre classe et possède toutes les méthodes (en fonction du mode d'héritage) de la classe. En terminologie C++, les classes, dont on hérite, sont appelées classes de base.

La classe est donc l'abstraction du contenant de l'information (variables et méthodes). C'est pourquoi, nous pouvons définir la *méta-classe* comme la sémantique du comportement de la classe d'origine. Tout ce qui dépend de la gestion de la classe est décrit dans la méta-classe : c'est son auto-représentation [Maes87]. Mais il faut lier la méta-classe avec une

classe utilisateur sur laquelle elle s'applique, on peut dire que la méta-classe est *superposée* sur la classe utilisateur.

De plus, la méta-classe définit les méthodes d'instanciation et d'initialisation des objets de la classe [Gold83]. Ainsi, nous pouvons redéfinir l'initialisation d'un objet de la classe par une méta-méthode du constructeur. Cette dernière peut prévoir des traitements sur la gestion future de l'objet, indépendamment du propre rôle de l'objet.

Par exemple, avant la création d'un objet à partir de sa classe, un méta-objet est créé, puis une méta-méthode d'initialisation est exécutée. Ensuite, l'objet de bas niveau est exécuté en commençant par son constructeur.

En conclusion, nous pouvons considérer des méta-objets comme des éléments qui peuvent s'auto-vérifier et adopter par la suite un comportement adéquat (auto-modification).

3.2. Applications réflexives

Bien que concept récent, la réflexivité commence à concerner plusieurs types d'applications. Dans un premier temps, nous allons traiter les applications dites élémentaires, c'est-à-dire celles qui utilisent les principes simples et fondamentaux de la réflexivité. Ensuite, nous nous intéresserons au mécanisme de répartition, ou comment la réflexivité peut être efficace pour la distribution d'objets. Enfin, nous nous pencherons sur l'influence de la réflexivité sur les concepts de tolérance aux fautes.

3.2.1. Applications basiques

La plus simple des applications de la réflexivité est la trace d'un programme ou d'un objet. En effet, une méta-méthode est superposée à chaque méthode, afin d'informer l'utilisateur du nom de la fonction appelée ainsi que de ses paramètres. Ainsi l'utilisateur peut avoir la connaissance de tous les appels.

Dans [Kirb96], l'auteur exploite la réflexivité en Java, afin de créer un navigateur d'objet, un « object debugger ». Cela permet d'obtenir toutes les informations souhaitées sur les objets, à savoir les méthodes utilisées, les attributs (avec leurs valeurs respectives), les liens d'héritage avec d'autres objets, etc. Ce concept est très pratique quand une mise au point du programme est nécessaire pendant l'exécution.

Grâce à la réflexivité, il devient très facile d'appliquer des préconditions ou des postconditions aux méthodes. Par exemple, on peut coder une fonction racine carrée, sans se préoccuper des conditions sur les paramètres. Ces dernières se placeront dans le méta-source et autoriseront ou non l'entrée dans la fonction de calcul. Dans ce cas précis, l'entrée dans la méthode (ici, la fonction racine carrée) ne sera autorisée par la méta-méthode, que si le nombre en paramètre de la fonction est supérieur à zéro.

3.2.2. Mécanisme de répartition

Une application majeure de la répartition est le mécanisme des RPC (*Remote Procedure Call* ou Appel de Procédure à Distance). En effet, il permet d'invoquer d'une manière conventionnelle une procédure qui peut se trouver sur un site distant. En général, la transparence de l'appel est géré par le noyau système : transfert de l'appel, passage de paramètres et retour des résultats.

Au travers de la réflexivité, il est possible d'avoir une certaine transparence au niveau de l'appel. Comme une méta-fonction peut s'occuper de l'initialisation préalable de la fonction, elle peut aussi entreprendre la gestion de la répartition.

Un des fonctionnements possibles est le suivant : l'appel d'une fonction est dérouté sur la méta-fonction, qui connaît la localisation réelle de la fonction. L'exécution continue soit sur le même site, soit sur un site différent. Nous exposerons un tel système dans la partie 3.3.1, « Java réflexif ». Puis nous proposerons notre propre système dans la partie 5.

3.2.3. Mécanisme de tolérance aux fautes dans un système réparti

Les méta-objets peuvent contrôler l'exécution d'un programme. Ils sont en mesure de surveiller l'activité du programme en cours d'exécution, et par conséquent de prendre en compte la tolérance aux fautes.

D'une part, [Per97] explique qu'un protocole à méta-objet (MOP) est efficace pour sécuriser une application. Le service le plus utilisé est la possibilité de modifier l'invocation des méthodes et l'accès aux attributs d'un objet pour implanter les mécanismes de tolérance aux fautes au méta-niveau. L'appel est dérouté sur une méta-méthode qui peut faire certaines modifications. Il faut donc pouvoir intégrer les actions à mener avant ou après chaque exécution d'une méthode au niveau de base.

Le rôle des méta-objets est d'assurer qu'un objet répond correctement à toute invocation, en dépit des erreurs qui pourraient entraver son fonctionnement. [Per97] présente deux familles de mécanisme de tolérance aux fautes : les points de reprise sur support stable et la réplication d'objets. Dans les deux cas, les méta-objets doivent gérer les aspects tolérance aux faute et les aspects communication à distance.

De plus, l'utilisation récursive [Per97] des méta-objets permet plus de souplesse quant à la prise en compte de la tolérance aux fautes. Grâce aux différents niveaux de méta-objets, nous pouvons leur attribuer un rôle précis : une couche de méta-objet s'occupera de la tolérance aux fautes et une autre de la répartition. Ce principe rend la programmation utilisateur plus simple, car on s'appuie sur les méta-objets déjà écrits.

D'autre part, [Xu96] a fait des recherches sur la surcoût en temps qu'engendrent les méta-objets assignés à la tolérance aux fautes. Sa conclusion est que le surcoût est acceptable lorsque le réseau est fiable et le temps de communication fini, sinon les temps de réponse sont considérablement majorés.

3.3. Langages réflexifs

Pour coder des méta-objets, nous avons besoin de langages supportant la réflexivité (*méta-langage*). D'un côté, nous allons décrire comment la réflexivité est aisée à implémenter dans un langage interprété. D'un autre côté, nous nous apercevrons que les langages compilés sont moins adaptés. Nous nous intéresserons à un exemple de méta-langage interprété (le Méta-Java) puis à des méta-langages compilés.

3.3.1. Java réflexif

Le principe étant de modifier le code, même à l'exécution, nous constatons que les langages les plus adaptés à la réflexivité, sont les langages interprétés. En effet, ils consistent à évaluer les instructions au moment de l'exécution, donc, changer le code à cet instant,

s'avère plus aisé. Comme Java est un langage objet interprété, il est donc adapté aux besoins de la réflexivité.

Dans [Klein96], une méthode de RMI (Remote Method Invocation) est esquissée. Cela consiste en l'appel d'une méthode vers une autre methode qui se trouve sur un objet distant (cf. Figure I), dans un système réparti.

L'objet A fait un appel à un objet B qui est sur un site différent. L'appel est dérouté sur la méta-méthode correspondante dont le rôle est de trouver l'objet B puis de transférer l'appel avec les paramètres. La méta-méthode reçoit l'appel et transmet la requête et la méthode B. Cette dernière traite l'appel et renvoie les résultats par le même chemin. En résumé, le passage via les méta-méthodes se montre totalement transparent.

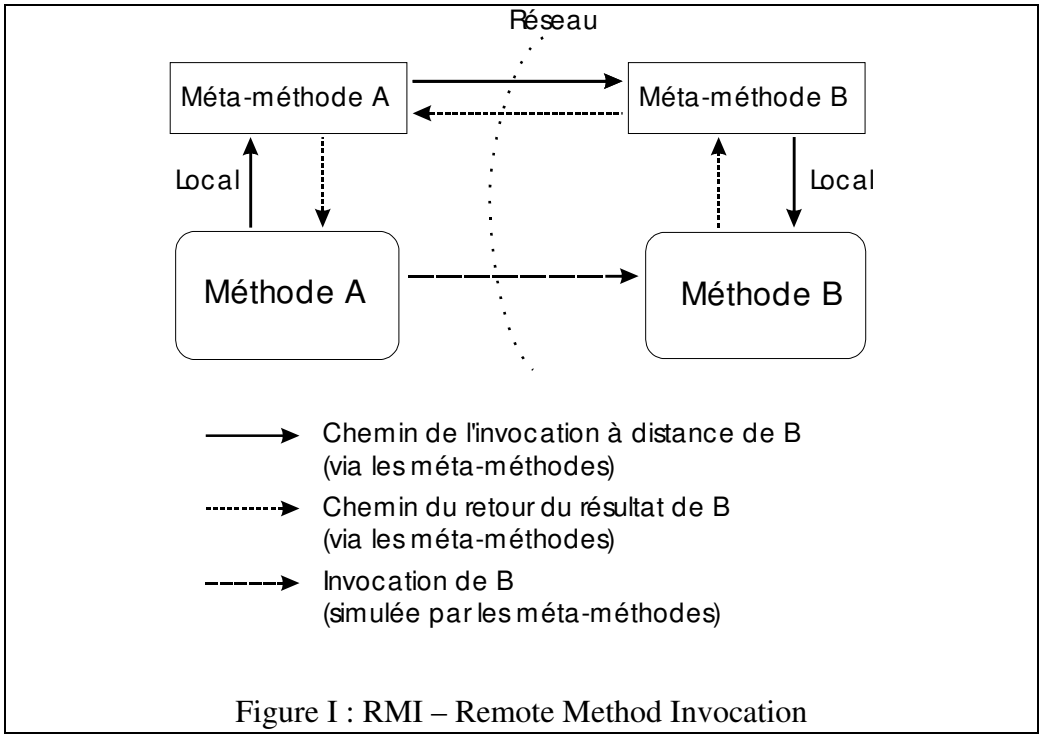

Figure I : RMI – Remote Method Invocation

Comme Méta-Java est interprété, il propose plusieurs possibilités *pendant l'exécution* : le fait d'attacher dynamiquement un méta-objet à un objet de base, la modification de syntaxe ou de sémantique, etc.

Ce langage propose d'autres points d'intérêt et d'autres extensions : contrôle de la migration, sécurité, objets actifs et persistants, synchronisation, tolérance aux fautes, etc. L'idée est de rendre ces concepts totalement transparents. Le concepteur peut ainsi séparer la programmation de ces extensions de la programmation système.

3.3.2. Langages Acteurs

Il y a une dizaine d'années, les langages objets n'offraient pas la possibilité de s'exécuter en parallèle. Avec le développement des machines, cette notion empêchait toute concurrence entre entités. Un nouveau concept compensant ce manque, est apparu : les langages acteurs. Les acteurs sont des entités autonomes concurrentes, qui possèdent leur propres données et

qui peuvent les échanger. Les transferts de messages entre deux acteurs sont la seule façon de communiquer, ils se réalisent de manière asynchrone, unidirectionnelle et point à point.

Toutefois, les acteurs peuvent se modifier pour acquérir une propriété supplémentaire. C'est pourquoi les langages acteurs se sont rapprochés du concept de réflexivité [Mij98]. Ainsi, ils bénéficient de la possibilité de raisonner sur leur propre état et de s'adapter. Cette extension se concrétise au niveau de l'acteur lui-même puis du groupe auquel il appartient.

La réflexivité individuelle permet à un acteur d'avoir connaissance de son état et d'exprimer ses caractéristiques indépendamment des autres acteurs du système. La notion de réflexivité de groupe permet de définir des ensembles d'entité ayant en commun certaines caractéristiques et de représenter les relations entre les acteurs d'un groupe.

La réflexivité représente donc une extension au modèle de base. De plus, elle permet d'aller plus loin dans la transparence des traitements et elle autorise une entité à se gérer elle-même, d'après ses propres connaissances.

3.3.3. C++ réflexif

Il existe plusieurs langages compilés utilisant la réflexivité, mais nous nous intéresserons à la famille C++, car la plupart de systèmes opératoires sont écrits dans ce langage.

- *EC ++*

Le langage EC++ est une extension du C++, au travers de laquelle l'environnement parallèle est transparent [McEw95]. Le principe de EC++ réside dans le fait de cacher les différents détails des architectures matérielles, afin de s'adapter aux diverses machines parallèles. Ce langage est portable, il est fourni sous forme de blibliothèques.

La réflexivité est introduite au 'level 0', c'est-à-dire au niveau des appels. Ce niveau utilise les méta-objets pour gérer les appels aux procédures et pour identifier les classes. Ce langage est intéressant dans l'utilisation de la réflexivité mais pas dans sa réutilisation par un développeur. Il inclut des notions d'attente par nécessité pour la synchronisation, principe qui s'inspire des machines Data-Flow.

- *OPEN C++*

Open C++ [Shi95] est un MOP (Meta-Object Protocol), c'est-à-dire une interface pour développeur qui intègre les notions de la réflexivité. C'est une extension au langage C++. Ce compilateur transforme un source programmé en Open C++ et un source utilisateur en C++, en un programme réflexif.

Les protocoles à méta-objets ne facilitent pas la résolution d'un problème, mais permettent une meilleure organisation entre les classes et leur auto-représentation (les méta-classes).

Nous analyserons plus profondément Open C++ dans le chapitre 5.1.

3.4. Systèmes opératoires orientés objets *réflexifs*

Le principe de la réflexivité est très intéressant dans un système opératoire orienté objets. En effet, ce dernier peut réagir en fonction de son état et d'événements extérieurs, puis peut se

modifier pour s'adapter à l'environnement. Nous allons présenter les principes étudiés pour les systèmes Apertos et Choices.

3.4.1. APERTOS[1]

Comme la réflexivité permet de modifier la sémantique d'un objet, il est donc possible de donner dynamiquement une ou plusieurs propriétés aux objets. C'est l'approche choisie par APERTOS [Yoko92].

Ce système opératoire est entièrement objet : tout est objet. Néanmoins, si on considère cette façon de faire, on risque de rencontrer certaines difficultés. Par exemple, il est difficile d'inspecter l'intérieur d'un objet, à cause de la protection vis-à-vis de l'accès de ce dernier par d'autres objets. C'est pourquoi il faut donner à chaque objet une nouvelle propriété pour permettre son inspection par d'autres. Ce cas survient lors d'une recherche d'erreur dans un objet pendant l'exécution. Il est nécessaire de donner cette propriété à chaque objet.

APERTOS gère ce problème en séparant le niveau classique (appelé *base level*) du niveau réflexif (appelé *meta level*). Ce dernier est considéré comme une couche séparée, appelée *méta-espace* (ou *metaspace*). La règle est la suivante : quand un objet veut acquérir une (ou changer de) propriété, il migre vers un groupe de méta-objets qui donne (ou génère) la propriété demandée (cf. Figure II). Par exemple, un objet A possède une propriété P, fournie par un méta-objet situé dans un méta-espace. Un autre objet, noté B, désirant la même propriété, va migrer dans le méta-espace voulu pour y « adhérer ». Le méta-objet, proposant la propriété P et appartenant au méta-espace, peut donc modifier l'objet B ou exécuter des pré-traitements, pour que ce dernier acquière la propriété recherchée.

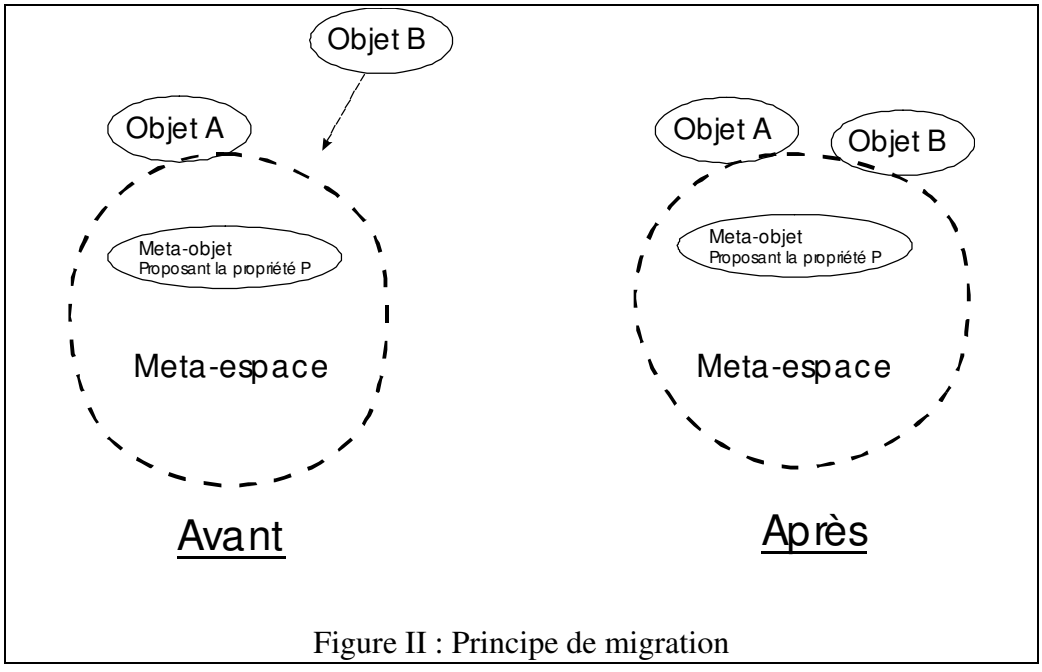

Figure II : Principe de migration

On parlera de « Relocating Code Segment » pour la migration. Toutefois, il faut noter qu'un méta-objet est aussi un objet et qu'il est, par conséquent, susceptible d'avoir lui aussi un méta-objet associé. On observe ici une relation de récurrence entre les méta-objets et l'établissement d'une méta-hiérarchie (cf. Figure III).

[1] Autrefois appelé MUSE

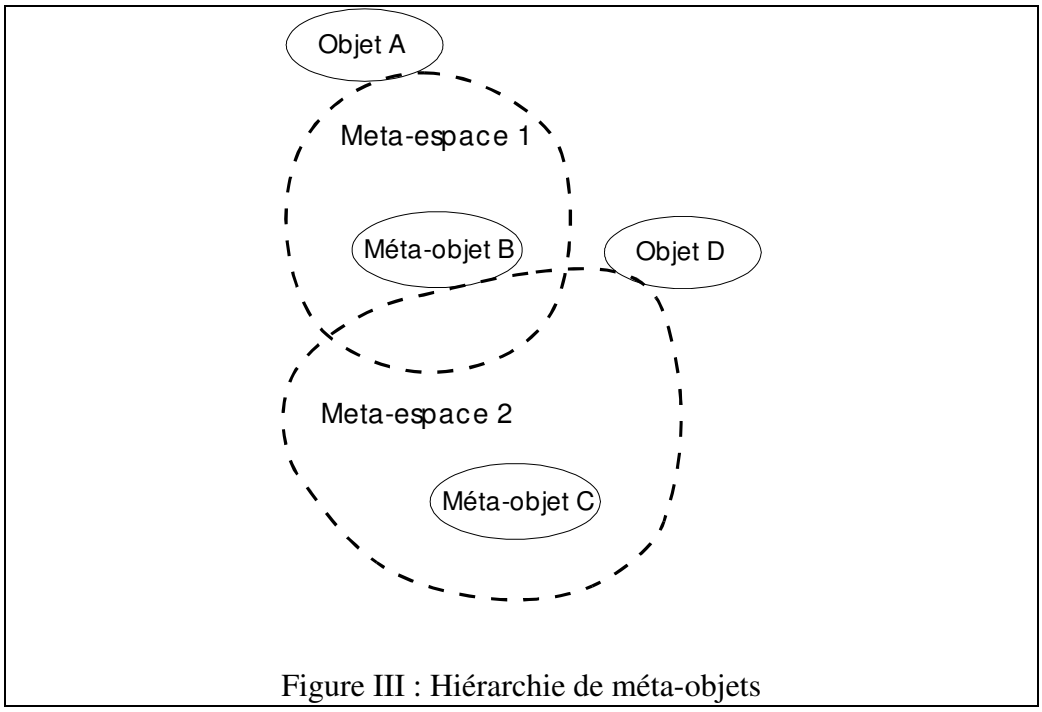

Figure III : Hiérarchie de méta-objets

Par exemple, l'objet A adhère au méta-espace 1, qui contient un méta-objet B. Or, ce dernier est aussi considéré comme un objet. Il peut donc être attaché au méta-espace 2, tout comme l'est l'objet D.

3.4.2. Choices réflexif

Dans cette partie, nous donnerons un aperçu de Choices et plus particulièrement un aperçu de sa version réflexive [Mad96].

Au début, Choices a été défini comme un ensemble de classes permettant d'avoir une base solide pour concevoir un système opératoire. Il considère toute entité comme un objet. Par exemple, les processeurs, les fichiers, l'espace mémoire et les autres machines (serveurs) sont considérés comme des objets. C'est un système opératoire 'totalement' orienté objets.

Choices est composé des sous-systèmes suivants : gestion des processus, de la mémoire, de la persistance, des périphériques et de la communication.

Au fur et à mesure des modifications, les concepteurs ont utilisé le principe d'extensibilité. Alors, Choices a évolué, les auteurs ont prévu d'y incorporer la réflexivité. Ils ont décomposé la problèmatique du système opératoire en classes de base où intervient la réflexivité : stockage, existence, persistance, classe[2], héritage, encapsulation et inspection.

- *Stockage*

Le stockage consiste à surcharger (dérouter), à l'aide de la réflexivité, les fonctions `NEW` et `DELETE` par des `ALLOCATE` et `FREE`. Ces dernières sont des méta-méthodes qui permettent une allocation d'objet plus adaptée en fonction de son type. En effet, l'état du tas (heap) est réifié dans une fonction héritée d'`ALLOCATE`, ce qui permet d'avoir une allocation

[2] Ce terme de classe et les suivants n'ont pas le même sens que celui que nous connaissons. Ce sont juste des redéfinitions de l'auteur de l'application.

plus précise (par exemple placement à l'adresse exacte). De plus, des sous-classes peuvent réifier le tas avec diverses propriétés. On peut alors esquisser des méthodes pour des accès simples ou parallèles (possibilité de gérer les conflits).

- *Existence*

A l'aide d'un compteur de références, le système peut savoir si un objet est ou n'est plus utilisé. Cela indique au « garbage collector » s'il doit détruire l'objet non référencé (d'où libération de mémoire). Dans Choices, il existe non seulement un compteur de références mais aussi une réification des pointeurs sur les objets. La méthode `REFERENCE` est appelée automatiquement quand on assigne l'adresse d'un objet et la méthode `UNREFERENCE` est appelée quand l'adresse est récrite. De même, le compteur de références est mis à jour et l'appel du destructeur est réalisé lorsque le compteur est remis à zéro. L'intérêt de ce concept est d'automatiser la destruction des objets.

- *Persistance*

Dans la même optique que l'existence, un compteur de références de persistance (différent de celui de l'existence) permet de savoir si l'objet a été utilisé récemment. Dans le cas où aucun processus n'accède à l'objet, il est automatiquement stocké sur le disque dur. Bien sûr, nous avons toujours le lien avec l'existence, si l'objet ne devient plus référencé par des pointeurs (aucun lien), il est alors retiré de la mémoire secondaire (du disque dur). Nous pouvons donc établir un lien avec la mémoire virtuelle : ce concept rend transparent l'utilisation du « swap disk ».

- *Classe*

La réflexivité permet à un objet de connaître sa classe. Cela simplifie l'extension dynamique, c'est-à-dire qu'il est possible de changer la forme d'une classe lors de l'exécution. On peut imaginer un système de gestion de fichier (SGF) qui s'adapte aux fichiers qu'il utilise. Par exemple, le concepteur avait prévu une taille des fichiers sur 2 octets (16 bits) mais si, pendant l'exécution, certains fichiers ont une taille plus importante, alors le champ 'longueur de fichier' sera modifié.

- *Héritage*

L'arbre d'héritage est réifié pendant l'exécution. Le système est capable alors de charger du code grâce au `CODE LOADER`. Le code chargé est, en réalité, une sous-classe de l'arbre d'héritage non présente à l'exécution. Le principe est le suivant : le code est localisé par le SGF, il est chargé en mémoire, puis la table des symboles est résolue. Les fonctions membres et les constructeurs sont installés. Il faut noter ici que ce principe existe mais que le C++ (par conséquent Choices) ne le supporte pas.

- *Encapsulation*

L'encapsulation permet au système de protéger des objets en les plaçant dans des zones réservées, des régions protégées physiquement. Pour les utilisateurs, la protection est utile afin qu'ils n'écrivent pas n'importe où. Cette notion supporte un adressage virtuel et une mémoire secondaire. Nous pouvons admettre que l'encapsulation permet d'émuler la protection offerte par UNIX.

- *Inspection*

L'inspection permet le gel du programme afin d'avoir une représentation de l'objet (la réification de l'objet), une trace des appels avec leurs paramètres. Nous pouvons avoir des

informations importantes sur l'état de l'objet : le nombre de références, le temps d'inaction, la protection, l'arbre d'héritage, etc.

3.5. Conclusion

Nous avons vu, au travers de ce chapitre, la définition de la réflexivité ainsi que les différents types d'applications qui y font appel. Bien que des langages comme C++, ne fournissent pas de support spécifique pour la réflexivité, certaines facilités sont apportées par des langages réflexifs, comme Open C++.

Il existe diverses approches de la réflexivité, d'une application élémentaire à un système opératoire réflexif, mais le dénominateur commun à tous ces projets est que la réflexivité ouvre de nouveaux horizons. Cela facilite des concepts complexes, qui peuvent être superposés sur des sources déjà écrits.

4. LE PROJET SO^3

Le projet SO^3 (Système Opératoire Orienté Objets) est une expérience de conception d'un système opératoire en utilisant la programmation objet [Goore95]. Il a pour but de démontrer la faisabilité d'un tel système. Par conséquent, l'approche par objets oblige le concepteur à une abstraction importante du système opératoire, afin de pouvoir définir des hiérarchies de classes abstraites et concrètes les mieux adaptées.

Dans la section suivante, nous allons présenter une brève description de l'approche objets pour un système opératoire. Cette approche doit être structurée afin d'être flexible pour maîtriser la diversité des architectures et les besoins des utilisateurs.

Ensuite, nous mettrons en avant les techniques conçues pour la répartition. L'ajout de cette notion a pour but de démontrer le principe d'extensibilité de l'approche objets, c'est-à-dire un concept non prévu pouvant se greffer au projet de façon à ne pas trop modifier la conception initiale.

4.1. Description de SO^3

Le projet SO^3 vise à élaborer un système opératoire orienté objets, c'est-à-dire il consiste à construire une hiérarchie de classes abstraites et concrètes capables d'offrir un système aussi général que possible [Goore95]. En utilisant les possibilités de la technologie objet, SO^3 se propose de réduire la complexité de la création d'un système opératoire. En effet, l'approche choisie doit être fiable et simple, tout en garantissant les notions élémentaires de la technologie des objets, à savoir la modularité, la réutilisabilité, l'extensibilité et la portabilité.

Pour le projet SO^3, les concepts de base sont donc réduits et se matérialisent en un petit nombre de classes abstraites. A partir de ces classes, et en utilisant intensivement les possibilités d'héritage, simple et multiple, et de généricité, SO^3 fait dériver toute une hiérarchie de classes abstraites de moins en moins générales pour finir par spécifier les classes concrètes spécifiques des objets manipulés. Ceci a le défaut de créer un grand nombre de classes (bien plus qu'il n'y en a d'instanciées), mais cela permet une plus grande abstraction des concepts, et procure une maintenabilité et une extensibilité plus importante.

Dès lors, à chaque étape de dérivation, un nombre minimal de nouveaux concepts sont apportés dans la classe spécialisée. De manière idéale, un seul concept doit être ainsi rajouté, quitte à devoir, si plusieurs notions doivent être intégrées, insérer des classes « intermédiaires » n'ayant pour seule fonctionnalité que de servir de classe de base à une prochaine classe dérivée. Ceci permet de se réserver la possibilité de faire dériver, dans l'avenir, une nouvelle classe d'une des classes intermédiaires, si celle-ci n'a besoin que d'une partie des propriétés des classes de bases.

4.2. La répartition dans SO^3

L'ajout de la notion de répartition sur SO^3 a pour objectif de démontrer l'extensibilité de l'approche objets appliquée aux systèmes d'exploitation. En effet, le but est de montrer que

l'ajout de cette notion peut être réaliser sans modifier foncièrement l'arbre des classes abstraites et concrètes défini dans SO^3.

Dans un premier temps, nous mettrons en évidence les concepts de domaine et de relais développés à base d'objets pour la répartition. Puis dans un deuxième temps, nous présenterons les divers relais ainsi que leur hiérarchie.

4.2.1. Les concepts

Dans un système réparti, il est nécessaire de pouvoir déplacer certaines entités d'un ordinateur à un autre. En technologie objets, l'élément de base étant l'objet lui-même, il devient évident de pouvoir déplacer certains objets d'un site à un autre. En vue d'un équilibrage de charge par exemple, le temps processeur est important, et lorsqu'un site devient saturé, il est souhaitable de transférer un objet actif vers un autre site dont la charge est moindre.

- *Concept de domaine*

Un domaine est une zone homogène d'accès aux données. Par exemple, un processus peut accéder à ses données d'une certaine manière, alors qu'un autre processus ne pourra pas y accéder, du moins pas forcément avec la même adresse.

Deux domaines sont dits compatibles si la valeur d'une référence sur un domaine est valable sur l'autre. Par exemple, sur une machine, si deux objets de deux domaines différents peuvent accéder toute donnée située dans l'un des deux domaines de telle manière que son adresse soit la même pour les deux objets, alors ils sont dans deux domaines compatibles.

Un exemple assez naturel de deux domaines compatibles est donné par deux threads d'un même processus : toute adresse (pointeur) valable dans l'un d'eux est également valable dans l'autre, ce qui permet un partage efficace des données.

Il existe une hiérarchie de domaines abstraits : un objet, un thread, un processus, un site, un réseau, etc. L'invocation ne peut se réaliser qu'avec le domaine père ou le domaine fils, mais en aucune façon, elle ne peut se faire entre deux domaines disjoints.

Le concept de domaine peut être représenté comme une hiérarchie arborescente. Pour fixer les idées, nous pouvons prendre comme image une arborescence de répertoires de disque, dans une SGF.

Un accès entre deux domaines ne peut se faire que par franchissements de frontière successifs vers un domaine père ou un domaine fils, de la même manière que dans le cadre d'une arborescence de répertoires : si l'on veut accéder aux fichiers se trouvant dans le répertoire parent, il faut changer de répertoire. De la même façon, si l'on veut accéder au fichier se trouvant dans un répertoire frère, il faut remonter jusqu'au répertoire père, puis redescendre jusqu'au fils souhaité.

- *Concept de relais*

Pour tout ce qui touche la répartition dans ce document, nous supposerons que le réseau est fiable et que le temps de transfert de données sur le réseau est fini. Le transfert d'objet lui-même ne sera pas évoqué. Nous nous sommes intéressés qu'à l'invocation à distance d'un objet ainsi qu'aux moyens mis en œuvre pour y arriver.

Pour l'invocation, le modèle se base sur le système de relais. Un relais est un *objet*, offrant la même interface que l'objet original, c'est-à-dire qu'il offre l'accès aux mêmes méthodes que l'objet original. Donc, lorsque un objet veut invoquer un autre, il passe par un relais et transmet cet appel au relais.

L'objet concret est toujours référencé par un client via un relais. De ce fait, si l'objet est transféré sur un autre site, cette migration est transparente au client. Lorsqu'un objet est transféré vers un autre site, il laisse derrière lui une sorte de « carte de visite » de ses méthodes : c'est le relais.

Néanmoins, le relais en lui-même n'est qu'un terme générique pour désigner tous les moyens mis en œuvre pour arriver à l'objet : concrètement, le relais est à la base d'une hiérarchie de sous-relais dont il est le père (cf. Figure IV).

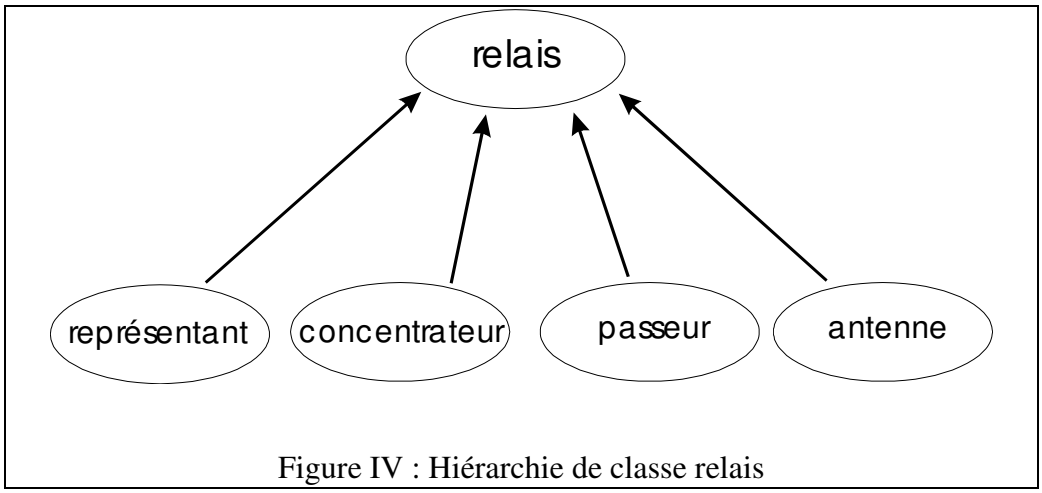

Figure IV : Hiérarchie de classe relais

4.2.2. Les relais en détail

Dans cette partie, nous allons analyser les différents types de relais. A chacun a été assigné un rôle simple et spécifique. C'est par combinaison de ces fonctionnalités élémentaires que l'on obtient des fonctions plus complexes comme pour le RMI (cf. chapitre 3.3.1, « Java réflexif »).

Pour simplifier les explications, nous allons décider que deux objets sont sur des sites différents. L'un d'eux (noté A, objet appelant) veut invoquer une méthode de l'autre objet (noté B, objet appelé). Nous allons donc examiner de près la séquence qui permet à un objet de réaliser l'invocation d'un autre. Nous allons appeler cette séquence : la chaîne de relais d'accès à l'objet.

Le rôle du relais consiste à réceptionner la demande d'invocation et de la transmettre au relais suivant, puis lors de sa réception, de relayer le résultat à l'objet appelant.

- *Le représentant*

Le représentant est le relais appelé directement par l'objet A. L'objet appelant veut invoquer une méthode de B : il s'adresse au relais 'représentant' de B, et réalise son invocation d'une manière normale sur ce dernier, comme s'il s'adressait à l'objet B.

- *Le concentrateur*

Lorsque plusieurs objets d'un même domaine, veulent faire appel au même objet, il faudrait qu'ils aient chacun, une chaîne d'accès de relais vers l'objet appelé. S'ils sont peu nombreux, cela est concevable, mais s'ils sont nombreux les chaînes d'accès vont prendre trop de ressource. A fortiori, si N objets appelants, invoquent chacun M objets, il y aura N x M chaînes d'accès dans ce domaine, ce qui est rédhibitoire.

La solution au problème consiste à créer un nouveau type de relais : le concentrateur. Son rôle est de pouvoir gérer plusieurs appels sur le même objet. Dans l'exemple précédent, il possède donc effectivement N entrées, toutes reliées au représentant respectif des objets appelants.

A défaut d'un relais concentrateur, si la forme de la chaîne locale d'accès à l'objet global doit changer (ajout de moniteurs ou déplacement de l'objet global) et qu'il y a N objets locaux accédant à ce même objet global, alors il faudrait N modifications dans les représentants pour mettre à jour les chaînes locales d'accès.

- *Le moniteur*

Une invocation peut nécessiter un pré-traitement ou un post-traitement pour être réalisée correctement. Par exemple, si l'utilisateur veut crypter ses données, afin d'assurer un certain niveau de protection, il doit ajouter des relais moniteurs à sa chaîne d'accès. Il est évident que leur utilité réside dans l'optique des services : codages (protection), compression, synchronisation, authentification, etc.

Remarques :

- il peut y avoir plusieurs moniteurs à la suite dans une chaîne, afin d'offrir des services plus complexes.
- la présence de ce relais est facultative.

- *Le passeur*

Il arrive un moment où la chaîne d'accès se retrouve en bord de domaine. Le dernier relais va transformer l'invocation d'une méthode en données transmissibles par les différents moyens de transport informatique : l'appel procédural si les domaines sont compatibles sinon les pipes, les sockets, etc. Ce dernier relais est appelé le passeur.

Le relais passeur veille aussi au changement de domaine et fournit tous les renseignements nécessaires aux relais suivant concernant l'historique de l'invocation : passage par des relais moniteur, etc.

- *L'antenne*

Il faut avoir un relais de réception des données transmises, c'est le rôle de l'antenne. Ce dernier relais désencapsule l'appel de méthode et réalise l'invocation sur la chaîne d'accès, dans le domaine d'arrivée.

Voici en résumé le chemin d'accès d'une méthode à distance :

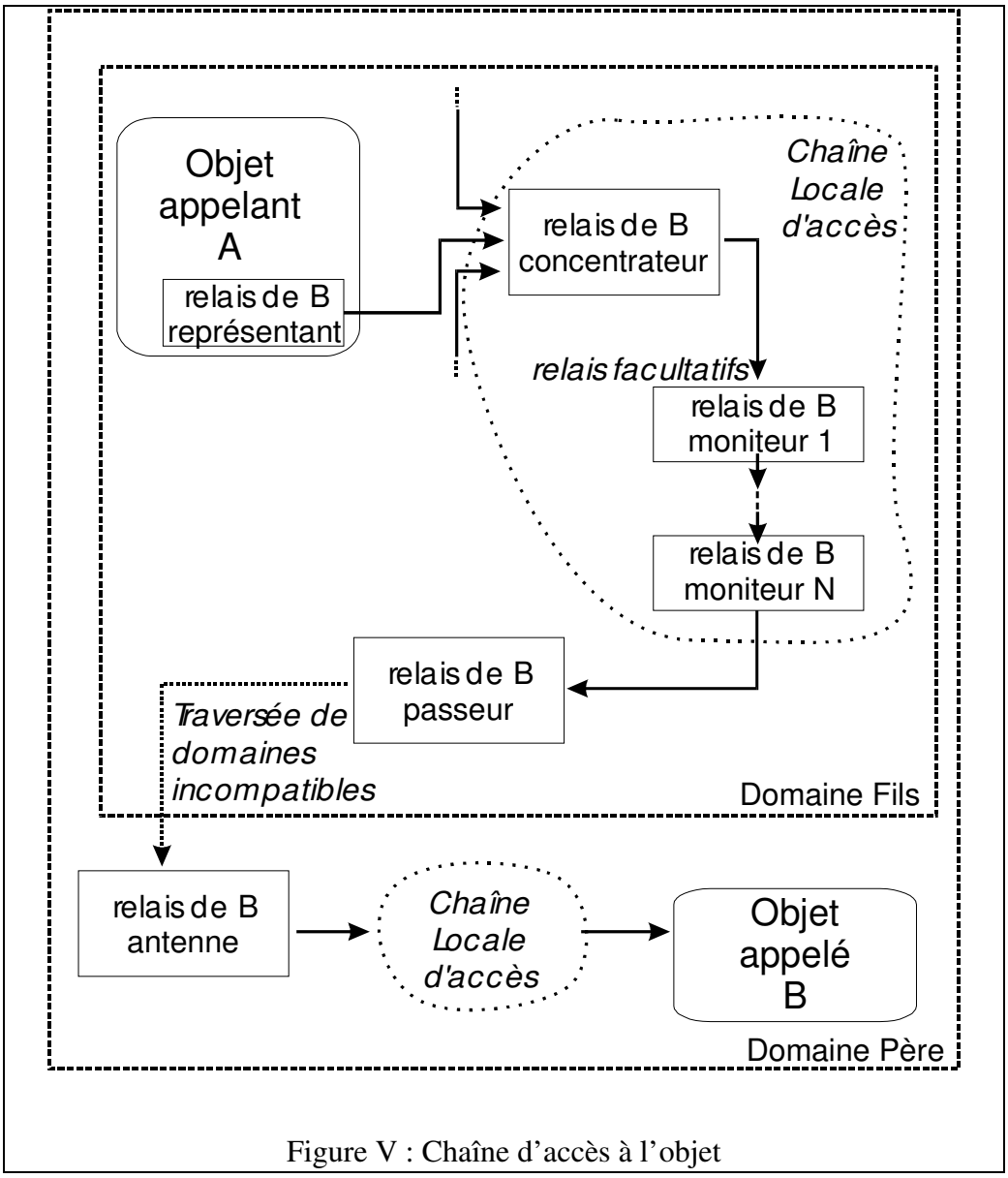

Figure V : Chaîne d'accès à l'objet

5. UTILISATION DE LA REFLEXIVITE POUR LA REPARTITION DANS SO[3]

Préalablement, nous avons détaillé la réflexivité, nous avons proposé des définitions et des exemples d'utilisation. Puis, nous avons brossé un portrait du projet SO[3]. Dans cette partie, nous allons mettre en évidence le travail fait pour utiliser la réflexivité en vue d'introduire la répartition dans le projet SO[3]. Le but est de proposer la répartition comme propriété supplémentaire sur un programme quelconque, d'une manière totalement transparente. La solution envisagée passe par les méta-classes, c'est-à-dire que nous superposerons la répartition sans toucher au source de l'utilisateur.

Dans un premier temps, il a fallu se familiariser avec OPEN C++. Ce langage réflexif étant le premier langage réflexif que nous utilisons, la prise en main a été complexe. C'est pourquoi nous allons essayer d'expliquer comment fonctionne ce langage, au travers de l'étude de sa compilation et au travers d'un exemple simple.

Dans un deuxième temps, nous nous sommes intéressés aux utilisations possibles de la réflexivité dans un système réparti. Nous avons déjà étudié une méthode à base de relais qui permet de concrétiser le répartition, c'est pourquoi nous allons voir comment la réflexivité peut être utile dans ce concept.

5.1. OPEN C++[3]

La réflexivité peut s'avérer être une notion abstraite et confuse. De plus, la distinction entre le niveau méta et le niveau de base semble parfois assez floue. L'étude d'Open C++ [Shi95] a permis de tracer une séparation nette entre ces deux niveaux. Les objectifs de ce langage sont de séparer la programmation d'une application et la modification des opérations de base du modèle objet, grâce à l'utilisation de méta-classes.

Avant d'aborder les explications complexes sur la réflexivité dans SO[3], il est nécessaire d'analyser Open C++ et d'en expliquer la méthode de compilation. C'est pourquoi, nous donnerons un aperçu de l'évolution de ce langage au travers de ses deux méthodes de travail : le RunTime MOP et le Compile-Time MOP. Puis, nous présenterons un modèle caractéristique de programmation, à l'aide d'un exemple.

5.1.1. Méthode de Compilation

Open C++ est une extension au langage C++. Il utilise le C++ comme langage pour les méta-classe comme pour les classes de base. Pour bien cerner comment fonctionne Open C++, nous allons étudier sa méthode de compilation.

En entrée, le compilateur reçoit du code Open C++, c'est-à-dire un ensemble de méta-classes écrites en C++. Ces méta-classes et les classes sur lesquelles elles se superposent,

[3] Pour toute l'étude nous avons travaillé avec les versions 2.5 jusqu'à 2.5.3 d'Open C++. Le programme est en libre accès sur le site du créateur, Shigeru Chiba à http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html

seront compilées au moyen d'Open C++, pour donner un source C++ temporaire. Il ne reste plus qu'à compiler ce dernier avec un compilateur C++ classique, afin d'obtenir un exécutable terminal.

Nous pouvons analyser plus profondément la compilation. En effet, la Figure VI nous montre comment le compilateur génère concrètement un programme réflexif.

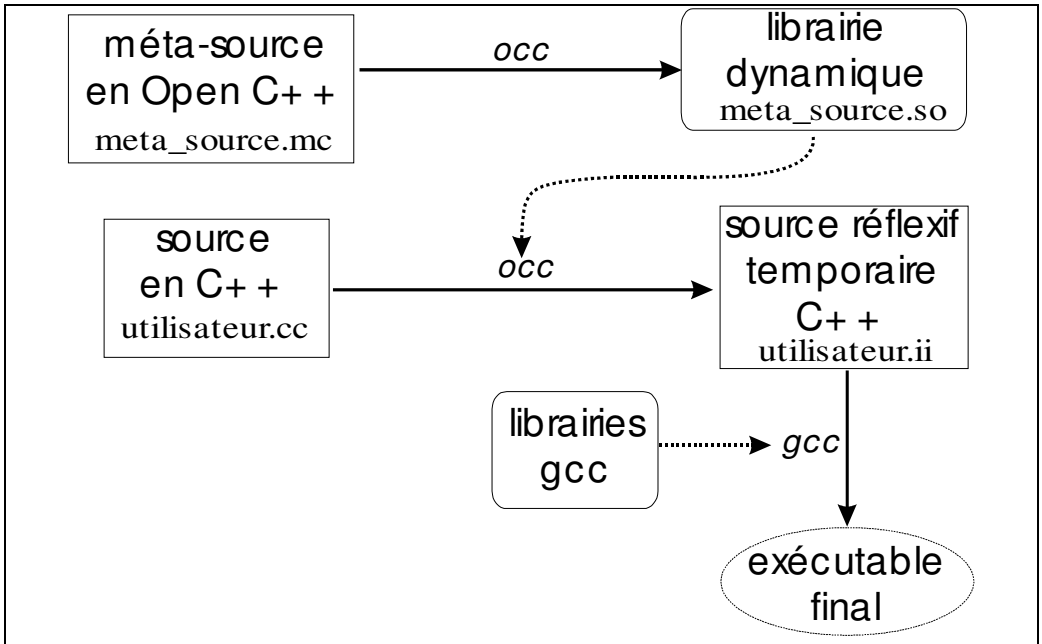

Figure VI : Schéma de la compilation par Open C++

Pour bien comprendre cette figure, le compilateur d'Open C++ est `occ` et le compilateur pour les sources utilisateurs étant `gcc`.

A partir du méta-source, occ génère une librairie dynamique, c'est-à-dire un ensemble d'ajouts[4] pour la compilation future avec le source de bas niveau de l'utilisateur. C'est-à-dire, qu'à partir du méta-source (fichier dont l'extension est '.mc'), `occ` crée une bibliothèque dynamique spéciale (fichier '.so').

Ensuite, nous pouvons lancer la compilation (toujours à l'aide d'`occ`) du source normal (source bas niveau, original), dans lequel un mot-clé réservé `metaclass`, indique au compilateur que la méta-classe qui suit doit superposer une classe désignée. C'est alors, que le méta-compilateur modifie le source à l'aide de la bibliothèque dynamique, et mémorise ce source modifié (fichier '.ii'). Ensuite, la compilation de ce dernier fichier est effectuée par `gcc` afin d'obtenir l'exécutable final.

5.1.2. Evolution

Depuis sa première version, OPEN C++ a évolué. En effet, le compilateur a introduit dans sa version 1.2, la notion de *RunTime Mop*, puis, à partir de la version 2.0, la notion de *Compile-Time Mop*. Nous allons donc étudier les différences entre ces deux concepts.

[4] Plug-in en anglais

- *RunTime MOP*

Dans un *RunTime MOP*, les méta-méthodes sont compilées statiquement. Par contre, l'appel à ces dernières est dynamique, ce qui implique une surcharge de cet appel : un appel à la méthode principale est détourné sur la méta-méthode. Ce principe est intéressant car il correspond à la définition de la réflexivité.

Le déroulement de l'invocation de méthode est le suivant. L'appel d'une méthode de bas niveau[5] est dérouté sur la méta-méthode correspondante, comme sur la Figure VII. De même, l'accès aux variables de l'objet est contrôlé par une méta-méthode. Tout ceci ralentit bien entendu l'exécution.

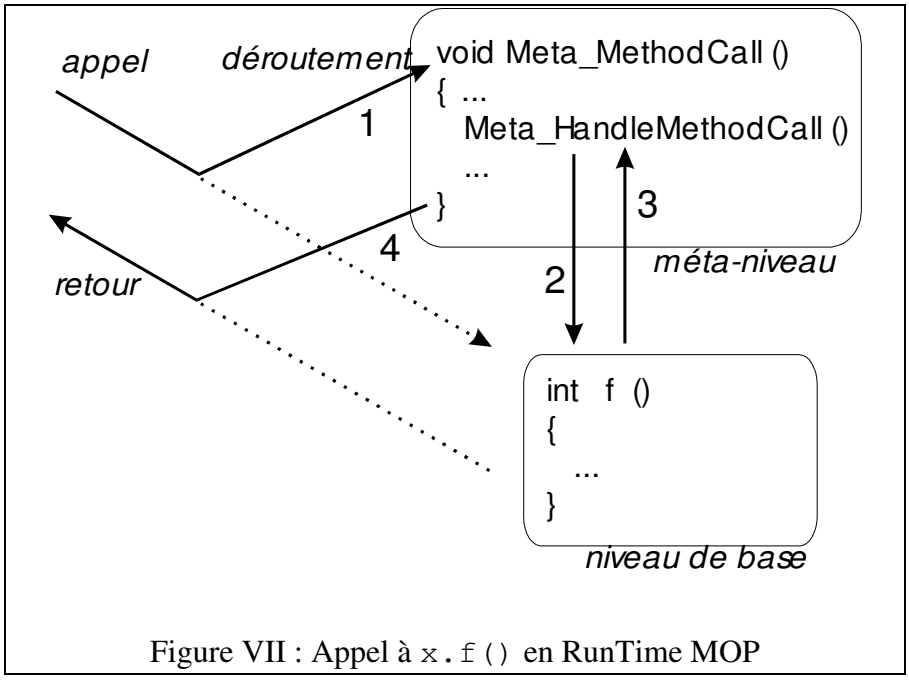

Figure VII : Appel à `x.f()` en RunTime MOP

La création et la destruction de l'objet et du méta-objet se déroule comme suit :

```
1. Création de l'objet
2. Création du méta-objet
3. Liaison méta-objet ↔ objet
4. Appel de Meta_StartUp
5. … [action objet] …
6. Appel de Meta_CleanUp
7. Destruction de la liaison
8. Destruction du méta-objet
9. Destruction de l'objet
```

Malgré une aisance au niveau compilation, cette méthode reste, d'une part limitée, et d'autre part, assez coûteuse en temps UC. C'est pourquoi Open C++ a évolué vers une version Compile-Time.

[5] Le bas niveau est le niveau utilisateur, i.e. tout ce qui correspond au niveau non réflexif. Le contraire du bas niveau est appelé le méta-niveau, meta-level en anglais.

- *Compile-Time MOP*

En *Compile-Time*, la méta-méthode est directement compilée et les modifications qu'elle doit effectuer au sein de la méthode, sont réalisées pendant la compilation. Cela implique un gain important de temps pendant l'exécution. En effet, la méthode ne déroute plus sur la méta-méthode proprement dite. Cette dernière est directement incluse dans la méthode (cf. Figure VIII).

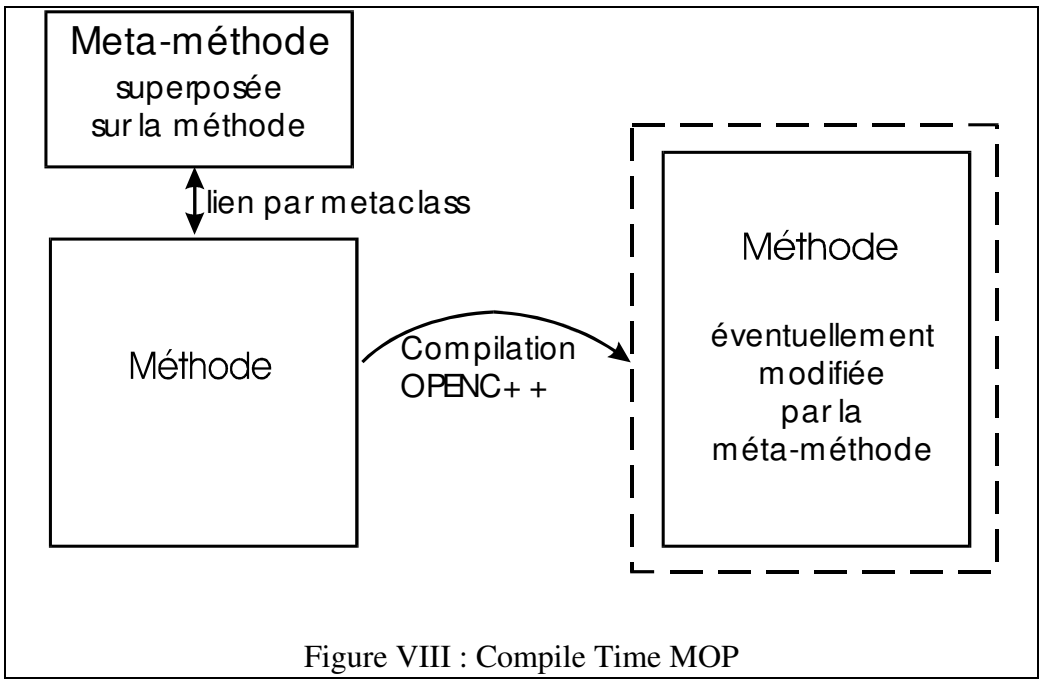

Figure VIII : Compile Time MOP

Dans ce cas-là, nous nous apercevons plus tôt si la méta-méthode est correcte ou non puisqu'elle est compilée au moment de la compilation du source original et les modifications qu'elle doit effectuer au sein de la méthode sont incorporées à ce moment-là.

Il faut noter que, dans les deux cas, ce principe de réflexivité ne modifie pas le texte original du source, c'est-à-dire que le programmeur ne s'aperçoit pas que son source a été modifié. Néanmoins, le compilateur a généré un source réflexif temporaire pendant la compilation.

5.1.3. Premier Test

D'après sa conception, Open C++ permet à l'utilisateur de modifier un source réifié par le compilateur. Il génère automatiquement l'arbre d'analyse (« parsing tree » en anglais) du langage C++, ce qui permet au programmeur de connaître précisément la forme du source et quels sont les fonctions / procédures / attributs de la classe. Il permet entre autres, de changer les noms des fonctions, de rajouter des classes de base en héritage, de connaître les constructeurs / destructeurs, de modifier le corps d'une fonction, etc.

Par conséquent, lors de la compilation du source de bas niveau, le langage réifie le texte de ce source sous forme de `Ptree`, abréviation de « parsing tree ». Le texte devient donc une entité facilement manipulable, à l'aide des fonctions founies par Open C++.

Afin de comprendre comment fonctionne Open C++, nous avons commencé par des manipulations élémentaires, à savoir la compilation des exemples fournis par le concepteur. Mais, tout en restant simple, nous avons créé une méta-classe plus intéressante : la Before-After Class.

La sémantique de cette classe est très simple : pour chaque méthode, elle doit réaliser un traitement *avant* l'exécution et un autre traitement *après*. Les champs d'application sont : la trace d'un programme, les pré et post-conditions, etc.

Pour simplifier, nous avons choisi une trace élémentaire : une méta-méthode associée à chaque méthode. Dans notre exemple, la méta-méthode devra imprimer à l'écran l'entrée et la sortie dans la méthode.

Observons donc un exemple de méta-source Open C++ (extrait de `BAClass.mc`) :

```
void BAClass::InitializeInstance( Ptree* definition,
                                  Ptree* meta_arg)
{
…
}

void BAClass::TranslateClass(Environment* env)
{
  Member member;
  int i = 0;
  while(NthMember(i++, member))  // (1)
  if(member.IsPublic() && member.IsFunction()  // (2)
     && !member.IsConstructor() && !member.IsDestructor()){
      Member wrapper = member;
      Ptree* org_name = NewMemberName(member.Name());
      member.SetName(org_name); // (3)
      ChangeMember(member);
      MakeWrapper(wrapper, org_name);
      AppendMember(wrapper, Class::Public); //(4)
  }
}
```

Ce sont les deux fonctions principales de la méta-classe `BAClass`. Nous allons expliquer ces deux fonctions par ce tableau.

Nom fonction	Rôle
`InitializeInstance`	Est appelé juste avant la création de l'objet. Permet des traitements à ce moment précis. Ensuite, l'objet est créé et initialisé par le constructeur initial.
`TranslateClass`	Elle permet de modifier la classe, méthode par méthode.

Comme `TranslateClass` est une méthode clé, nous allons l'analyser en profondeur.

Une boucle *(1)* passe en revue chaque membre[6] de la classe de bas niveau, afin d'y appliquer un traitement. Si le membre sélectionné est une *fonction* publique, ni constructeur, ni destructeur *(2)*, alors la méta-méthode peut y travailler. Cette dernière va renommer la méthode de bas niveau *(3)*. Mais, quelques problèmes apparaissent à ce moment là.

Problème	Solution
Comment provoquer un traitement *avant* la méthode ?	Inévitablement, la méthode sera appelée, il suffit de placer du code avant le traitement original de la méthode.
Comment provoquer un traitement *après* la méthode ?	Il faut renommer la fonction originale *name* en _org_*name* tout en laissant le code original. Puis, créer une nouvelle fonction *name* qui fera l'appel à _org_*name*. Enfin, après l'appel à _org_*name*, réaliser le post-traitement. C'est utile lorsque la fonction ne se termine pas à la dernière ligne de code (e.g., exit en C++).

Comment est généré le code de la nouvelle fonction `name` et les modifications de la fonction d'origine ?	Attention de bien réaliser l'appel :
	
	Si c'est une *procédure* alors marquer seulement :
	`// pré-traitement`
	`_org_name( [paramètre] ) ;`
	`// post-traitement`
	
	Si c'est une *fonction*, penser au retour et à son type :
	`Type_retour Var ;`
	`// pré-traitement`
	`Var = _org_name( [paramètre] );`
	`// post-traitement`
	`return Var;`
	
	Il faut aussi que la fonction appelante soit correcte (procédure ou fonction) avec le bon type de retour.

Enfin, il ne reste plus qu'à donner l'ordre au compilateur de placer la méta-classe `BAClass` au dessus de la fonction du niveau de base, celle créée par l'utilisateur dans son propre code, notée `Cntr`.

```
metaclass BAClass Cntr(Francais);

class Cntr
{
 private:
      int   cpt;
 public:
      Cntr(int start=0);

      void  incr() { … XXX }
      int   get() { … return YYY }
};
```

De plus, nous remarquons que nous pouvons passer des paramètres à la méta-classe (`Cntr(Francais)`). Bien sûr, ces paramètres sont pris en compte comme des `Ptree`, et par conséquent ils sont traités comme du texte. Donc, nous pouvons paramétrer les méta-

[6] 'Member' en anglais. Un membre est un attribut ou une fonction d'un classe.

classes, c'est utile lorsqu'on veut donner un sens différent à la méta-classe. Par exemple, si l'on veut que la méta-classe ne s'occupe que d'une méthode précise, on peut placer le nom de la méthode de bas niveau dans les paramètres de la méta-classe, alors la méta-classe ne modifiera que la méthode concerné.

Voici le code généré par Open C++ :

```
class Cntr
{
 private:
      int   cpt;
 public:
      Cntr(int start=0);

      void incr() {   // pré-traitement
                      _org_incr();
                      // post-traitement }
      void _org_incr() { … XXX }
  // procédure qui a récupéré le source original de incr()

      int   get() {    // pré-traitement
                       int R = _org_get();
                       // post-traitement
                       return R }

      int   _org_get() { … return YYY }
   // fonction qui a récupéré le source original de get()

};
```

Néanmoins, nous avons rencontré de nombreux problèmes avec Open C++, résolus ou contournés. Par exemple, il existe une fonction permettant de savoir si une fonction est un constructeur ou pas (`bool IsConstructor();`). Si la classe hérite d'autres classes, elle hérite aussi de leurs constructeurs. Ces derniers vont être appelés récursivement, avant d'appeler le constructeur de la classe courante. Ce problème a été soumis à l'auteur et corrigé dans la version suivante d'Open C++.

5.1.4. Conclusion

Un protocole à méta-objets permet donc de séparer la programmation d'une application et la modification des opérations de base du modèle objet, comme l'instanciation de classes et l'invocation de méthodes, que ce soit en RunTime MOP ou en Compile-Time MOP. Un MOP combine à la fois des notions de réflexivité et de programmation orientée objets.

Il existe, par consequent, trois sortes de personne dans une application réflexive :

- Le programmeur en méta-objet : il crée le programme réflexivif (du niveau méta) que vont utiliser les programmeurs/utilisateurs.

- Le programmeur/utilisateur : c'est celui qui crée l'application de base (de bas niveau), en se servant des bilbliothèques dynamiques réflexives (les `.so` de la figure). Il doit néanmoins rajouter un :

```
Metaclass
       <nom classe réflexive>
       <nom classe de base sur laquelle elle s'applique>
       [<paramètres de la méta-classe>]
```

- L'utilisateur final qui utilise l'application réflexive, d'une manière totalement transparente et qui demande un bon niveau de performance.

5.2. Travail sur le modèle de répartition

5.2.1. Présentation de l'exemple utilisé

Afin d'illustrer notre propos, un exemple sera utilisé tout au long de la conception du modèle. Il s'agit d'une hiérarchie simple de deux classes, comportant tous les différents types de définition de méthodes pouvant être rencontrés dans une telle hiérarchie. Afin de rester dans le concret, nous présentons comme exemple, une hiérarchie de classe gérant des imprimantes.

Nom méthode	Virtuelle	Définie dans la classe de base	(Re)-Définie dans la sous-classe
`write`	Oui	Oui	Oui
`formFeed`	Non	Oui	Oui
`latestError`	Non	Oui	Oui
`Send_PS_header`	Non	Non	Oui

De plus, pour pousser l'étude du fonctionnement des méta-classes jusqu'au bout, nous avons rajouté deux attributs, l'un dans la classe de base (`name`), l'autre dans la classe héritée (`PS_session_supported`).

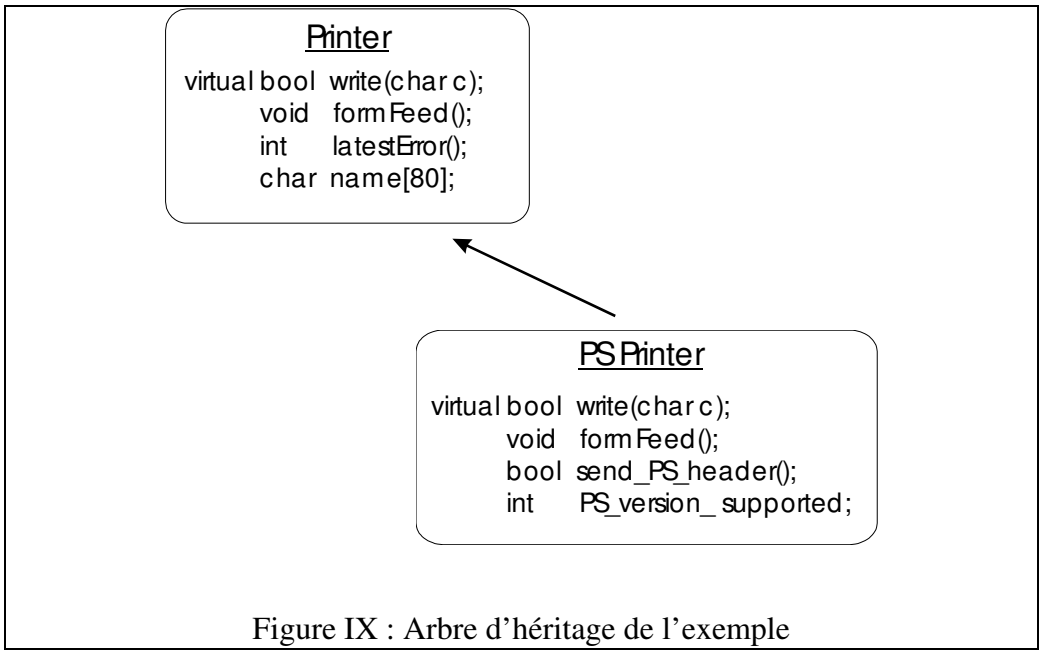

Figure IX : Arbre d'héritage de l'exemple

5.2.2. Problématique

Comme nous l'avons déjà vu, nous avons besoin de relais pour réaliser la répartition d'objets. Ces relais, dans un sens, représentent une abstraction de l'objet appelé, ils symbolisent l'image d'un objet dans un autre. Par conséquent, nous avons besoin de représenter ces relais de manière formelle. La solution adoptée est le concept de signature, une classe dont le relais héritera.

Afin de préciser qu'un objet est réparti, le programmeur doit rajouter un mot-clé devant la création d'objet : `global`.

```
Printer P; // imprimante locale
global Printer P; // imprimante distante
```

Pour le concepteur de la répartition, deux possibilités apparaissent :

- soit *modifier* la classe initiale (de l'utilisateur) par la méta-classe `global`, la rendant globale,
- soit *générer* une seconde classe (appelée Classe Globale) induite de la classe initiale.

Dans la première approche, la classe de l'utilisateur sera toujours globale. C'est la seconde approche qui a été retenue, puisqu'elle permet à l'utilisateur de toujours pouvoir définir des objets non globaux de la classe initiale. L'utilisateur a ainsi la possibilité de définir une classe locale (`Printer`) et une classe globale (`Global_Printer`), et d'avoir une imprimante locale et une distante.

5.2.3. Relation Objet Global[7] et Signature

Le premier choix fondamental nous a semblé être la relation entre l'Objet Global et la Signature de l'objet : soit il y a un lien d'héritage, soit il n'y en a pas.

- *Si le lien d'héritage entre Objet Global et Signature n'existe pas.*

Dans la chaîne d'accès, tous les relais ont un lien par pointeur avec le relais suivant, `Sign_A* Next`. Mais, le dernier relais aura à faire un lien par pointeur avec un `A* Next` (où `A` représente la Classe Globale traitée, puisque cette dernière se trouvera juste après, cf. Figure X), d'où problème de concordance de type. L'objet appelé n'est pas de type attendu par le relais, puisqu'il n'existe aucun lien entre les deux.

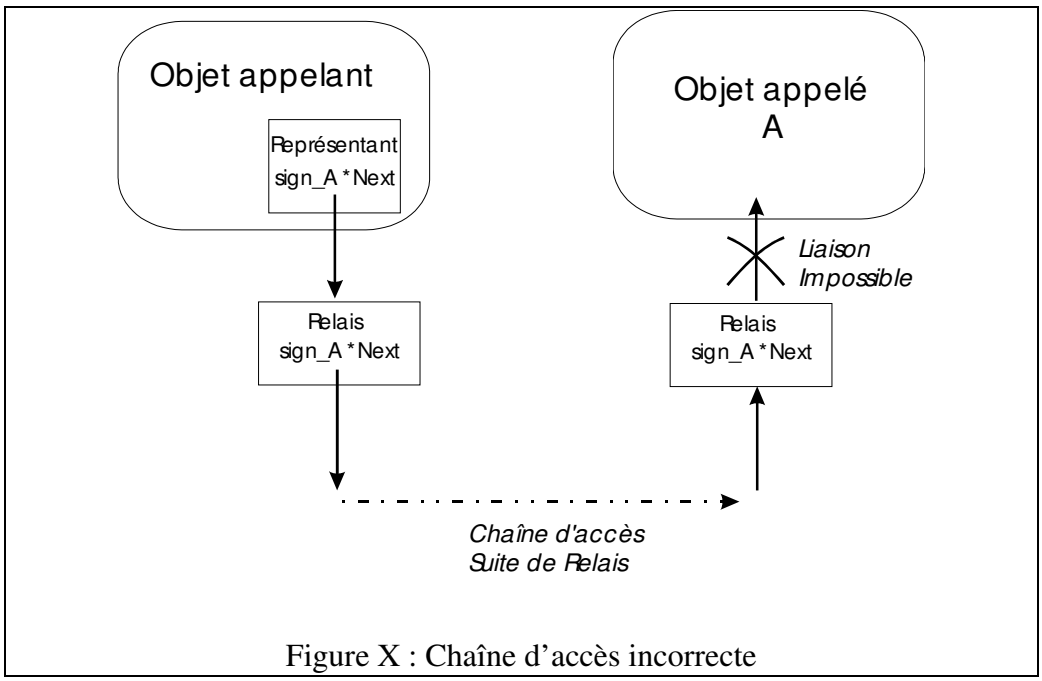

Figure X : Chaîne d'accès incorrecte

Nous nous trouvons obligés dans cette hypothèse de créer un relais dit « terminal ». (cf. Figure XI). Ce relais possède un lien spécial direct vers l'objet A.

[7] L'Objet Global est l'instanciation de la Classe Globale.

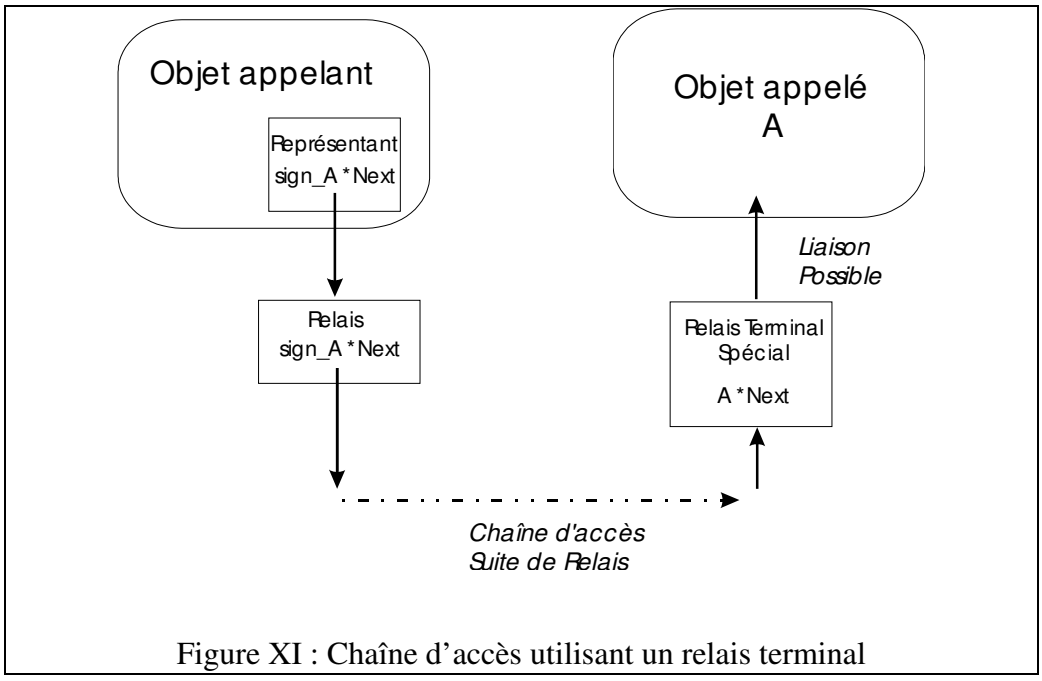

Figure XI : Chaîne d'accès utilisant un relais terminal

Cette solution a été abandonnée, à cause du problème de respect de type qu'il soulève et parce qu'elle distingue artificiellement le dernier relais, qui pourrait très bien pendant l'exécution n'être plus le relais terminal.

- *Si le lien d'héritage, entre Objet Global et Signature, existe*

Le problème du relais « terminal » disparaît, si bien que les relais peuvent faire appel les uns aux autres sans se soucier du type du suivant. Le dernier relais peut, sans problème, être lié avec l'Objet Global par sous-typage. En effet, l'objet appelé est lié par pointeur direct au dernier relais.

C'est pourquoi, pour le reste de l'exemple, nous allons supposer que l'objet global hérite de la signature.

5.2.4. Problème de préfixes

Comme la Classe Globale hérite de la classe virtuelle Signature, nous nous retrouvons avec des doublons dans les définitions des méthode. En effet, il y a les définitions des méthodes par la Signature et les définitions par la Classe Globale.

C'est pourquoi, une solution envigeagée consistait à renommer les méthodes de la Classe Globale en les préfixant (avec `SO3_`) ou considérer les invocations des méthodes publiques de manière virtuelle. En effet, une fonction héritée d'une classe, où elle a été déclarée virtuelle, reste considérée comme virtuelle, même si elle est redéfinie plus tard sans le mot-clé `virtual`.

Nous allons donner un exemple de préfixage. Soit une classe utilisateur :

```
class Printer {
  virtual bool write(char c) { XXX };
          void formFeed();
          int  latestError();
          char name[80];
};
```

Elle pourrait donner lieu à une Signature de la forme :

```
class Signature_Printer {
  virtual bool SO3_write(char c) = 0;
  virtual void SO3_formFeed()    = 0;
  virtual int  SO3_latestError() = 0;
};
```

... et à une Classe Globale telle que :

```
class Global_Printer {
  virtual bool SO3_write(char c) { return write(c); };
          void SO3_formFeed()    { formFeed(); };
          int  SO3_latestError() { return latestError(); };
  virtual bool write(char c) { XXX };
          void formFeed();
          int  latestError();
};
```

Le gros problème du préfixage est le suivant : dans le code utilisant l'Objet Global, nous allons trouver :

```
{
  remote Printer lpt(...);
  lpt.formFeed();
}
```

Ce code devrait être traduit en :

```
{
  Represent_Printer lpt(...);
  lpt.SO3_formFeed();
}
```

...or, l'invocation de la méthode `formFeed()` de l'objet `lpt` s'avère être difficile à traduire avec l'aide d'Open C++.

Une solution pour résoudre ce problème, pourrait utiliser des représentants « traducteurs », c'est-à-dire de la forme :

```
   class Represent_Printer {
     void formFeed() { next->SO3_formFeed(); }
     void SO3_formFeed() { next->SO3_formFeed(); }
     // pour assurer la compatibilité avec la
signature.
     ...
   };
```

Mais cette solution ne nous a pas satisfaits, parce que la « traduction » est plus complexe. C'est pourquoi, nous avons cherché dans la direction de la Classe Globale. Afin qu'elle puisse posséder ses attributs privés, la Classe Globale doit hériter de sa classe originale (à défaut, il faudra faire une recopie du corps de la classe à l'aide d'Open C++).

5.2.5. Les méthodes de recherche

Nous allons décrire les deux méthodes employées pour arriver à un modèle correct.

- *Première méthode*

  Hypothèses :
  Aucune modification de la Classe Utilisateur A.
  `Global_A` hérite à la fois de A et de `Signature_A`.
  Pas de préfixage.
  Aucune recopie du code des méthodes héritées, mais invocation de la méthode de la Classe Utilisateur.

On obtient alors le code traduit suivant :

```
// --- traduction de Printer ---
class Printer
{
  virtual bool write(char c) { XXX1 };
          void formFeed()    { YYY1 };
          int  latestError() { ZZZ1 };
          char name[80];
};

class Signature_Printer
{
  virtual bool write(char c) = 0;
  virtual void formFeed()    = 0;
  virtual int  latestError() = 0;
};

class Global_Printer : public Printer, public Signature_Printer
{
  virtual bool write(char c) { return Printer::write(c); };
          void formFeed()    { Printer::formFeed(); };
          int  latestError() { return Printer::latestError(); };
          char name[80];
};
```

```
// --- traduction de PS_Printer ---
class PS_Printer : public Printer
{
  virtual bool write(char c)    { xxx2 };
          void formFeed()       { yyy2 };
          bool send_PS_header() { ttt2 };
          int  PS_version_supported;
};

class Signature_PS_Printer
{
  virtual bool write(char c)    = 0;
  virtual void formFeed()       = 0;
  virtual bool send_PS_header() = 0;
  virtual int  latestError()    = 0;
};

class Global_PS_Printer : public PS_Printer,
                  public Signature_PS_Printer
{
virtual bool write(char c)    { return PS_Printer::write(c);};
void formFeed()       { PS_Printer::formFeed(); };
bool send_PS_header() { return PS_Printer::send_PS_header(); };
int  latestError()    { return Printer::latestError(); };
};
```

Cette méthode paraît très intéressante. Comme la Classe Globale hérite de la signature et de la Classe Utilisateur, nous nous retrouvons avec le problème d'une méthode héritée de deux classes de bases et non redéfinie dans la classe fille, c'est pourquoi nous la redéfinissons nous-même. Dans le corps de cette fonction, nous faisons un appel correct à la fonction utilisateur.

Ceci ne pose pas de problème en Open C++. En effet, la méthode (issue d'Open C++) `Class::NthMember()` renvoie tous les membres d'une classe, y compris les membres hérités, et la méthode `Member::Provider()` renvoie la classe qui fournit effectivement la méthode membre désignée (ce qui permet d'écrire `Printer::latestError()` dans la méthode `latestError()` de `Global_PS_Printer`).

Un autre avantage de cette technique est que l'on a évité toute recopie de code.

Mais il subsiste un problème d'héritage. En effet, puisque `PS_Printer` est une `Printer`, un traitement s'appliquant sur `PS_Printer`, devrait s'appliquer aussi sur `Printer`, mais dans notre méthode, cela ne marche pas. Tout simplement car le code est refusé par le compilateur, parce que le `Signature_PS_Printer` n'est pas un `Signature_Printer`. C'est pourquoi il faut réaliser une relation d'héritage entre les signatures.

- *Deuxième méthode*

On reprend la méthode précédente avec en plus la mise en relation d'héritage des signatures :

Hypothèses :
Aucune modification de la Classe Utilisateur A.
`Global_A` hérite à la fois de `A` et de `Signature_A`.
Pas de préfixage.
Aucune recopie du code des méthodes héritées, mais invocation de la méthode de la Classe Utilisateur.
Relation d'héritage entre les classes signatures.

Le code devient, par conséquent :

```
// --- traduction de Printer ---
class Printer
{
  virtual bool write(char c) { XXX };
          void formFeed()    { YYY };
          int  latestError() { ZZZ };
          char name[80];
};

class Signature_Printer
{
  virtual bool write(char c) = 0;
  virtual void formFeed()    = 0;
  virtual int  latestError() = 0;
};

class Global_Printer : public Printer,
            public Signature_Printer
{
virtual bool write(char c) { return Printer::write(c); };
void formFeed()    { Printer::formFeed(); };
int  latestError() { return Printer::latestError(); };
char name[80];
};

// --- traduction de PS_Printer ---
class PS_Printer : public Printer
{
  virtual bool write(char c)    { xxx2 };
          void formFeed()       { yyy2 };
          bool send_PS_header() { ttt2 };
          int  PS_version_supported;
};

class Signature_PS_Printer : public Signature_Printer
{
  virtual bool write(char c)    = 0;
  virtual void formFeed()       = 0;
  virtual bool send_PS_header() = 0;
};
```

```
class Global_PS_Printer : public PS_Printer,
            public Signature_PS_Printer
{
virtual bool write(char c){ return PS_Printer::write(c); };
void formFeed()    { PS_Printer::formFeed(); };
bool send_PS_header(){return PS_Printer::send_PS_header();};
int  latestError() { return Printer::latestError(); };
};
```

Nous nous retrouvons avec la hiérarchie suivante :

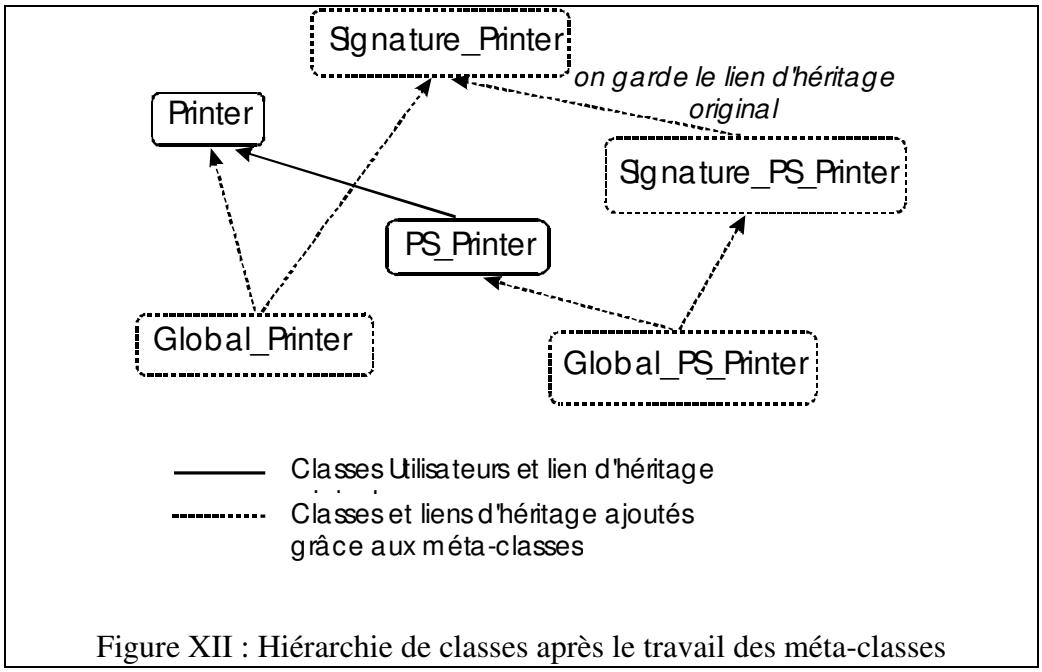

Figure XII : Hiérarchie de classes après le travail des méta-classes

Des problèmes peuvent survenir lors de la destruction des objets globaux. En effet, tous les objets présents dans un domaine sont dans une liste (ici, une liste de `Printer&`), mais il peut y avoir des `Global_Printer&`. Lors de la destruction, cet objet global sera détruit comme un Printer, et non comme un `Global_Printer`, ce qui va court-circuiter toute la gestion SO[3] des Objets Globaux.

Donc, il faut forcer le destructeur de la Classe Utilisateur de `Global_Printer` à être virtuel. Ce problème n'est d'ailleurs en rien spécifique à SO[3] : dans toute hiérarchie de classes, le destructeur devrait être virtuel pour justement éviter que ce genre de problème ne puisse se présenter. Cette virtualisation du destructeur est donc tout à fait logique et souhaitable. Cependant, elle modifie la classe utilisateur à l'insu du programmeur.

5.3. Utilisation des méta-classes

Mon travail au sein du projet a constitué à travailler sur les méta-classes. Dans un premier temps, il a fallu se familiariser avec Open C++ (cf. paragraphe 5.1 « OPEN C++ »), pour bien comprendre comment fonctionne la réflexivité. Puis, le travail s'est centré sur la modélisation des classes `Signature` et `Globale`, leur sens et la préparation à l'élaboration de la répartition avec leur aide.

Nous avons déjà vu que la notion de répartition repose sur les concepts de relais et de classe globale. Grâce à la réflexivité, nous pouvons placer dans les méta-classes tous ces traitements. L'utilisateur va donc informer le méta-compilateur quelles classes il désire « globaliser », afin que ce dernier puisse y appliquer les méta-classes.

Le rôle de la méta-classe `global` consiste à créer les classes `Signature` et `Globale`. La méta-classe agit presque comme un pré-processeur perfectionné, elle analyse les classes de bas niveau (classe programmeur/utilisateur). Elle analyse séquentiellement les classes, en ne s'occupant que des méthodes.

Méthode par méthode, Open C++ permet de sélectionner seulement celles qui répondent à certains critères que nous avons déjà définis : méthodes publiques, non constructeurs, non destructeurs. Ensuite, le méta-programme va créer une nouvelle classe (`class Signature_<name>`) et y placer les en-têtes des méthodes, tout en respectant la nature (procédure ou fonction) et en les rendant virtuelles pures.

Pour chaque nouvelle classe Signature créée, il ne faut pas oublier de réaliser récursivement les liens d'héritage. Ce problème, déjà évoqué, permet de remplacer une classe par une de ses classes de bases. Donc, le méta-compilateur va générer des classes Signatures dont la classe ne sera jamais globale.

Dans le cas de la classe globale, le même système a été utilisé, à la différence que le corps des méthodes doit être créé, comme nous l'avons déjà vu. Bien sûr, la même méta-classe s'occupe de la création des deux classes Signature et Globale.

Nous pouvons affirmer que ce concept est satisfaisant, tant sur le plan conceptuel que sur le plan pratique. Grâce à Open C++, le developpement de la répartition passe par les méta-classes ce qui permet au programmeur de se consacrer au développement du système proprement dit.

6. CONCLUSION

La technologie actuelle apporte beaucoup de souplesse au niveau des applications, grâce aux gains en rapidité d'exécution et aux nouvelles architecture. La transparence au niveau système opératoire est devenue le fer de lance de nombreux concepteurs. Par exemple, dans un réseau de machines, un utilisateur peut accéder à ses fichiers à partir de n'importe quel site. La transparence au niveau programmation est importante elle-aussi car elle permet au programmeur de s'atteler uniquement à l'application qui développe. Mais le grand problème est de s'adapter aux besoins de l'utilisateur pendant l'exécution de l'application.

Nous avons essayé l'apport de la réflexivité pour résoudre ces problèmes. Après avoir présenté la réflexivité en détail dans une première partie, grâce à l'étude des langages réflexifs et des systèmes opératoires réflexifs, puis des applications réflexives (comme la tolérance aux fautes), nous sommes arrivés à un constat de réussite pour la réflexivité. Dans une deuxième partie, après avoir présenter le projet SO^3, nous avons esquissé une méthode pour intégrer la répartition dans SO^3. Dans ce sens, les méta-classes ont donné de bons résultats en ce qui concerne l'extraction des nouvelles classes, pour créer les relais.

Toutefois, bien que performant, le concept de réflexivité doit pouvoir être plus profondément utilisé, notamment dans les systèmes opératoires. La réflexivité est en train de se frayer un chemin intéressant, mais elle peut se trouver des applications encore plus impressionnantes. Néanmoins, la réflexivité utilisée, dans ce rapport, est restreinte à l'observation et à la modification statique des classes, elle n'est pas utilisée pour modifier dynamiquement le comportement de l'objet pendant l'exécution. Mais, comme nous pouvons faire exécuter beaucoup de travail par les méta-classes, il serait intéressant de faire « réfléchir » l'objet, de posséder un objet intelligent, qu'il se gère tout seul grâce à son méta-objet. Ainsi il pourrait faire certains choix politiques, quant à sa gestion interne ou à sa localisation dans un système réparti.

7. TABLE DES ILLUSTRATIONS

8. REFERENCES

[Bouch94] Max Bouché.
''*La démarche Objets : Concept et Outils*''
AFNOR, 94

[Gold83] A.Goldberg et D. Robson
''Smalltalk-80, the Language and its Implementation'',
Addison Wesley, 1983.

[Goore95] Tra Goore Bi.
''*Contribution à la conception par objets des systèmes opératoires*''
Thèse IRIT, 1995

[Kirb96] Graham N.C. Kirby and Ron Morrison.
''*OCB : An Object/Class Browser for Java*''.
PJW2 Workshop 96

[Klein96] Jürgen Kleinöder, Michael Golm.
''*MataJava: An Efficient Run-Time Meta Architecture for Java*''
Proceedings of IWOOS'96 p.54-61

[Mad96] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell.
''*Reification and Reflection in C++ : An Operating System Perspective.*''
Report of Department of Computer Science, University of Illinois

[Maes87] P. Maes
''*Concept and Experimentation in Computional reflection*''
Proceedings OOPSLA'87, p. 21-35

[McEw95] Alistair McEwan.
''*EC++ - Europa Parallel C++*''
Oct 95, Report of the Europa Working Group on Parallel C++

[Mij98] F. Migeon et P. Sallé
'' *Acteurs et Réflexivité* ''
JFLA 98, 2/3 février 1998, Come, Italie

[MJD96] J. Malenfant, M. Jacques, et F.-N. Demers.
''*A Tutorial on Behavioral Reflection and its Implementation*''.
Proceedings of Reflection 96 San Francisco, p. 1-20.

[Per97] T. Pérennou
''*Une Architecture à Méta-objets pour systèmes répartis tolérant les fautes*''
Thése LAAS, 1997

[Shi95] Sigeru Shiba.
''*A Metaobject Protocol for C++*''
OOSPLA'95 proceedings.

[Strous92] B. Stroustrup
''Le langage C++''
Edition Addison-Wesley, 1992.

[Xu96] Jie Xu
''*Implementing Software-Fault Tolerance in C++ and Open C++ : An Object-Oriented and Reflective Approach*'',
1996.

[Yoko92] Y. Yokote
"The Apertos Reflective Operating System: The Concept and Its Implementation",
Sony, CSL technical report SCSL-TR-92-014, also appeared in OOPSLA'92 Proceedings, ACM, pp.414--434, October 1992.