# *OFL* : Une machine virtuelle objet flexible pour la gestion d'objets persistants et mobiles

Robert Chignoli, Pierre Crescenzo et Philippe Lahire

22 mars 1999

# Table des matières

**1 Présentation générale** **4**

**2 Architecture de *OFL/VM*** **6**
2.1 Réification des concepts : Généralités . . . 6
2.2 Entités *built-in* et flexibles : un aspect de la réification . . . 6
2.2.1 Classes *built-in* et modifiables . . . 8
2.2.2 Instances *built-in* et modifiables . . . 10
2.2.3 Les liens : des entités *built-in* fondamentales . . . 11
2.3 Synthèse sur la modélisation des méta-classes . . . 11
2.4 Description des instructions . . . 13
2.4.1 Instructions d'exécution . . . 13
2.4.2 Instructions de déclaration . . . 13
2.5 Autres mécanismes . . . 15
2.5.1 Clauses d'adaptation dans les liens d'importation . . . 15
2.5.2 Modélisation des échanges entre objets . . . 17
2.5.3 Gestion de la visibilité . . . 18
2.6 Services supplémentaires de *OFL/VM* . . . 19
2.6.1 Persistance et mobilité des objets . . . 19

**3 Sémantique opérationnelle d'un langage** **21**
3.1 Les Méta-classes CLASSE . . . 23
3.1.1 La recherche de primitive . . . 28
3.1.2 Les contrôles dynamiques . . . 29
3.1.3 L'exécution de demandes de travaux . . . 29
3.1.4 La création et la destruction d'instances . . . 30
3.1.5 L'attachement d'objets . . . 31
3.1.6 Les opérateurs fondamentaux . . . 31
3.1.7 La gestion de l'ensemble des instances . . . 32
3.2 Les méta-classes LIEN . . . 32
3.2.1 Description des propriétés communes aux *liens entre classes* . . . 32
3.2.2 Description de la méta-classe *lien d'utilisation* . . . 40
3.2.3 Description de la méta-classe *lien d'importation* . . . 40
3.2.4 Description de la méta-classe *lien objet vers classe* . . . 41
3.3 Sémantique d'un langage . . . 43
3.4 Autres aspects . . . 47
3.4.1 Interopérabilité : les éléments de base . . . 47
3.4.2 Sémantique des instructions . . . 48

**4 Principes de mise en œuvre de *OFL/VM*** **50**
4.1 Mise en œuvre d'un langage : méta-programmation . . . 50
4.1.1 Niveaux de méta-programmation . . . 50
4.1.2 Gestion des contrôles . . . 51

4.2 Mise en œuvre d'une application : programmation . . . 52
4.2.1 Écriture d'une application . . . 52
4.2.2 Phase d'exécution d'une application . . . 55
4.3 Position par rapport à d'autres systèmes . . . 59
4.3.1 Objectifs de la réflexivité . . . 59
4.3.2 Expressivité et encapsulation des aspects réflexifs . . . 60
4.3.3 Mise en œuvre et performance des systèmes . . . 62
4.3.4 Méta-programmation et principaux services offerts . . . 63
4.3.5 Bilan . . . 64
4.4 Perspectives et conclusion . . . 64

# Table des figures


2.1 Réification des concepts . . . . . 7
2.2 Classes *built-in* . . . . . 9
2.3 Routines *built-in* . . . . . 9
2.4 Modélisation du concept de classe . . . . . 10
2.5 Instances *built-in* . . . . . 11
2.6 Modélisation des liens . . . . . 12
2.7 Intégration des méta-classes et des aspects sémantiques . . . . . 12
2.8 Intégration des objets d'exécution . . . . . 13
2.9 Modélisation des instructions d'exécution *built-in* . . . . . 14
2.10 Modélisation des instructions d'exécution flexibles . . . . . 14
2.11 Modélisation des instructions de déclaration . . . . . 15
2.12 Différentes clauses d'adaptation . . . . . 16
2.13 Hiérarchie relative aux messages . . . . . 17
2.14 Modélisation des messages . . . . . 18
2.15 Intégration de la persistance . . . . . 19

3.1 Intégration de la modélisation de la sémantique . . . . . 21
3.2 Organisation de la sémantique propre à chaque langage . . . . . 22
3.3 Intégration de la sémantique : la classe `OFL_LANGUAGE` . . . . . 44
3.4 Modélisation des aspects sémantiques . . . . . 48

4.1 Outils pour l'écriture de programmes . . . . . 53
4.2 Architecture du système . . . . . 54
4.3 Comportement à l'exécution . . . . . 55
4.4 Réification d'une expression . . . . . 57
4.5 Exécution d'une instruction . . . . . 58
4.6 Exemple de hiérarchie . . . . . 59

# Chapitre 1

# Présentation générale

Avec des langages comme *Java*, on a à notre disposition un mécanisme nous permettant d'exécuter des logiciels qui se trouvent sur une autre machine n'ayant pas forcément les mêmes caractéristiques (système d'exploitation, processeur, ...). Cette approche, qui apparaît avec l'explosion d'Internet, est intéressante mais elle est un peu limitée par certains aspects :

- La syntaxe de *Java* n'est pas forcément la meilleure et dans tous les cas on peut trouver des catégories d'utilisateurs qui ne sont pas satisfaites par elle.
- Toutes les catégories d'utilisateurs n'ont pas besoin de la même expressivité pour décrire leurs programmes. L'expressivité de *Java* pourra tantôt être trop grande et parfois au contraire, elle sera insuffisante.
- On sait à l'heure actuelle que les mécanismes et concepts que l'on trouve dans la plupart des langages à objets ne sont pas suffisants pour gérer tout à la fois : la persistance des objets et de nouveaux liens sémantiques (vues, lien inverse, lien est-une-sorte-de, lien est-une-version-de, ...).

À travers *OFL* nous cherchons à proposer une plate-forme objet qui permette en particulier :

- d'associer une sémantique à une classe ou à un lien de manière à pouvoir s'adapter aux différents mécanismes mis en œuvre dans les principaux langages et à pouvoir créer différents types de lien comme ceux cités ci-dessus.
- de réifier la plupart des concepts que l'on peut trouver dans les langages de programmation aujourd'hui utilisés dans le monde industriel.
- d'intégrer complètement la notion d'objet persistant de manière à permettre l'exécution d'un tel objet soit dans son contexte d'origine (sémantique du langage utilisé dans le programme qui a créé l'objet), soit dans le contexte de l'application qui l'utilise (sémantique du langage utilisé dans le programme qui consulte ou met à jour un l'objet).
- d'intégrer la notion d'objet mobile ou distribué afin de prendre en compte en particulier les objets persistants qui sont situés *ailleurs*. Pour l'implantation on pourra regarder du coté de *Corba*.

Nous proposons une implémentation de l'approche évoquée ci-dessus qui est basée sur trois ensembles de composants :

**le noyau du modèle** (réification des primitives, des liens, des classes et des messages, ...) On n'y retrouve pas en particulier les composants qui représentent les instructions qui décrivent le corps d'une routine ; l'ensemble de ces instructions est typiquement quelque chose qui peut/doit évoluer au fur et à mesure de la découverte/création de nouvelles structures syntaxiques.

**les composants manipulés par le programmeur** (utilisateur final ou outil). Il sont de deux sortes :

- les composants qui représentent des déclarations de structure (déclaration de primitive, de classe, ...) ; ils vont commander la création ou la mise à jour d'instances des

composants qui forment le noyau du modèle et qui sont donc complètement cachés au programmeur et

– les composants qui décrivent des déclarations d'instructions dans le corps des routines. Celles-ci doivent pouvoir évoluer indépendamment du noyau de notre modèle (rajouter un schéma *case* n'a jamais modifié réellement l'expressivité d'un langage ou d'un moteur d'exécution).

**les bibliothèques de routines** Elles décrivent divers usages des déclarations d'instruction ou de structuration (opération d'affectation ou déclaration d'une nouvelle routine par exemple). Ces bibliothèques permettent de factoriser le code écrit pour représenter les principales situations simples (par exemple la possibilité de décrire une affectation d'entier au lieu de décrire une instruction d'affectation générale mais appliquée à un entier), mais aussi les cas plus complexes. Ces raccourcis doivent assurer une meilleure lisibilité du code (écrit à la main ou généré par un outil). Si les bibliothèques sont complètes, le programmeur ne devrait plus accéder qu'à de très rares occasions aux composants cités dans le point précédent.

Par contre, nous ne proposons pas de syntaxe pour décrire ces programmes, mais simplement une notion de *classe script* permettant de regrouper au *même endroit* toutes les instructions qui décrivent la déclaration et l'exécution des traitements. Naturellement le contenu de cette classe script sera complètement dépendant de celui des composants que l'on veut représenter.

Notre sentiment est que la syntaxe est une affaire d'habitude de programmation et que l'effort doit surtout porter sur les mécanismes proposés par les moteurs de déclaration (structuration des notions mises en œuvre dans le programme) et d'exécution (exécution des traitements déclarés suivant des protocoles d'échanges de message dont la nature dépend de la sémantique des classes et des liens).

Un méta-programmeur pourra toujours rajouter un quatrième niveau à l'architecture proposée en proposant une syntaxe et un traducteur (compilateur, interprète, un mélange des deux, ...) qui génère, à partir de cette syntaxe, des appels aux routines des bibliothèques que nous proposons (voir la classe OFL_LIBRARIES). On pourra aussi dans un premier temps oublier la syntaxe pour décrire à la main le contenu des différentes classes scripts qui composent un programme.

Pour donner un ordre d'idée de l'ampleur du projet, *OFL/VM* correspond à 416 classes (dont 283 spécifiques). Les 283 classes écrites pour *OFL/VM* correspondent à 41 334 lignes de code.

# Chapitre 2

# Architecture de *OFL/VM*

## 2.1 Réification des concepts : Généralités

Dans *OFL*, les principaux concepts présents dans les langages de programmation sont réifiés. Ces concepts sont de plusieurs natures :

- le contenu des classes (description des primitives, des paramètres de généricité, des paramètres formels, ...),
- les instructions représentant le contenu des routines,
- la structuration des types et des classes.

Les deux premiers points sont décrits dans la figure 2.1 alors que le dernier est développé dans la figure 2.7.

On notera que les concepts pris en compte se doivent de répondre à l'objectif de *OFL* qui est de fournir un ensemble de primitives réparties dans une hiérarchie de classes qui doivent permettre la modélisation des langages actuels (*C++*, *Eiffel*, *Java*, ...) et à long terme de permettre la construction de nouveaux langages prenant mieux en compte les différents liens sémantiques qui peuvent unir les classes entre elles. Le langage utilisé pour modéliser *OFL* est le langage *Eiffel* ; la figure 2.1 présente la réification des principaux concepts définis dans le système *OFL*, mais l'étude sur tous les concepts à prendre en compte ou pas, avec leur niveau de complexité n'est pas encore figée.

Pour choisir les concepts importants à mettre en œuvre dans *OFL*, notre principale source d'inspiration a été le langage *Eiffel* lui-même d'autres concepts, qui ne sont pas mis en œuvre dans *Eiffel*, nous ont aussi semblé intéressants :

- le fait que tout objet puisse accéder à l'information de son type,
- la possibilité pour une routine d'accepter une autre routine en paramètre,
- ...

Une étude plus approfondie des autres langages permettra d'affiner les concepts à mettre en œuvre dans *OFL*.

## 2.2 Entités *built-in* et flexibles : un aspect de la réification

On trouve dans *OFL* des entités *built-in* et modifiables ; celles-ci sont de plusieurs natures : classes, instances, instructions, ...

On aurait pu aller jusqu'au bout de la réflexivité et dire que toute classe était une instance de la méta-classe modélisant le concept de classe, mais on a préféré donner à un nombre limité de classe le statut *built-in* dans un souci de simplification et de performance ; ces classes sont :

- les méta-classes modélisant le concept de lien (LINK_M_CLASS et ses descendantes) ;
- les classes décrivant les types simples : booléen, entier, réel, double, caractère (BOOLEAN_M_CLASS, INTEGER_M_CLASS, ...) ;

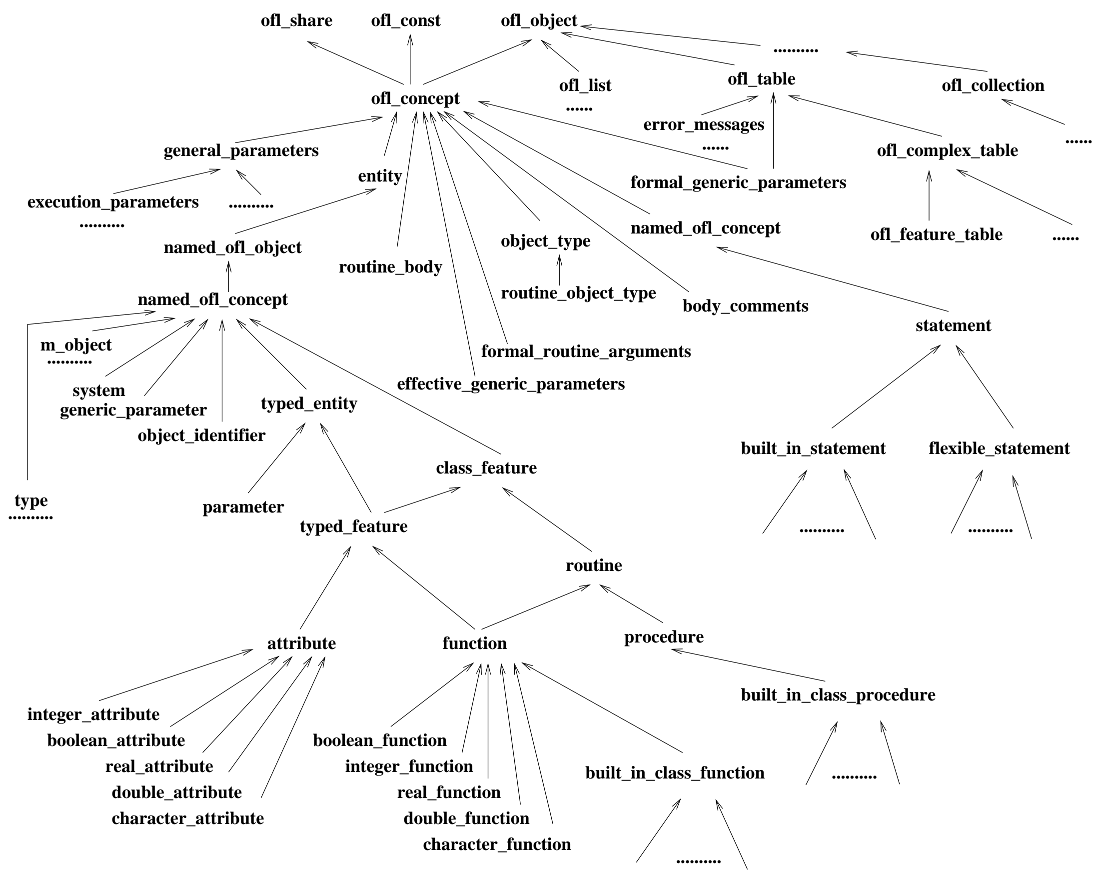


Fig. 2.1 – *Réification des concepts*

– les classes de la bibliothèque de base : chaîne de caractères, fichier, tableau, ...

Les méta-classes modélisant le concept de lien sont considérées comme des classes *built-in*, elles définissent des fonctions qui nous semblent nécessaires et suffisantes pour décrire et manipuler des liens. En cela, elles définissent les limites de flexibilité du système *OFL* en terme de manipulation des liens et fixe un des objectifs principaux de *OFL* : « être ouvert et flexible pour s'adapter à des besoins différents en matière de programmation mais ne pas forcément permettre de tout faire ».

Définir les types de base à partir de classes *built-in*, c'est à dire en fixant au départ les primitives de manipulation de ces types n'est pas vraiment une limitation car leurs fonctionnalités sont en général bien connues. Par contre, faire ce choix permet d'envisager un fonctionnement plus performant.

La troisième catégorie,, les principales classes de la bibliothèque de base, comme les chaînes de caractères ou la notion de tableau, pourraient ne pas être *built-in* ; c'est pour simplifier les choses au départ qu'un tel choix a été fait.

Afin de ne pas exagérer l'importance du choix de donner à une classe le statut *built-in* ou modifiable, il faut préciser les deux points suivants :

- à tout moment il est facile de rajouter statiquement une primitive aux classes *built-in* : le coût est exactement le même que celui de rajouter une routine à une classe de n'importe qu'elle application ;
- on peut a tout moment décider de changer le statut des classes *built-in* en modifiables, sans remettre en cause l'architecture de *OFL* (ni les programmes construit avec *OFL*).

### 2.2.1 Classes *built-in* et modifiables

La notion de classe est décrite par des méta-classes dont on peut changer la sémantique pour modéliser des types de lien et des concepts de classe. Ces méta-classes sont modélisées par deux sous-hiérarchies de classes qui sont les deux parties d'une hiérarchie qui intègre tous les concepts méta ; il s'agit des classes modifiables (décrivant une classe utilisateur) et les classes *built-in* (décrites lors de l'implémentation du système). Une classe est dite *built-in* si elle a des fonctionnalités câblées, c'est à dire s'il n'est pas possible de modifier ou d'ajouter dynamiquement des primitives à cette classe.

Il est important de noter que toute méta-classe *built-in* ne modélise qu'une seule classe. Par exemple la classe `INTEGER_M_CLASS` modélise la classe ENTIER de base (rien n'interdit au programmeur de définir sa propre classe ENTIER au moyen d'une classe modifiable et d'un ou plusieurs liens vers l'instance unique de la méta-classe INTEGER_M_CLASS).

#### Classes *built-in*

Toutes les classes *built-in* sont des descendantes de la classe `BUILT_IN_M_CLASS` (voir figure 2.2).

On remarquera en particulier que les classes *built-in* héritent toutes directement de la classe `BUILT_IN_M_CLASS` et aucune d'entre elles sauf les classes `LINK_M_CLASS`, n'initialise un lien avec d'autres classes. Ceci est obligatoire si on veut laisser le méta-programmeur libre d'implanter les types de lien qu'il souhaite utiliser. Par ailleurs les classes présentées dans le schéma peuvent se diviser en quatre catégories :

- les types de base (entier, réel, double, ...) et le type *racine* (SET 1),
- la gestion des liens (SET 2), voir chapitre 3,
- l'utilisation et la gestion du service de persistance (SET 3), voir section 2.6.1 et
- la gestion de classes de base (chaîne de caractères, tableau, ...).

On notera par ailleurs que si les liens ayant pour source une classe *built-in* sont eux aussi câblés (aucune classe *built-in* n'introduit de lien avec une classe modifiable), il n'en est naturellement pas de même pour les liens qui ont pour cible une classe *built-in*. Par conséquent n'importe qu'elle classe modifiable pourra avoir des liens de n'importe quelle nature vers une classe *built-in*.

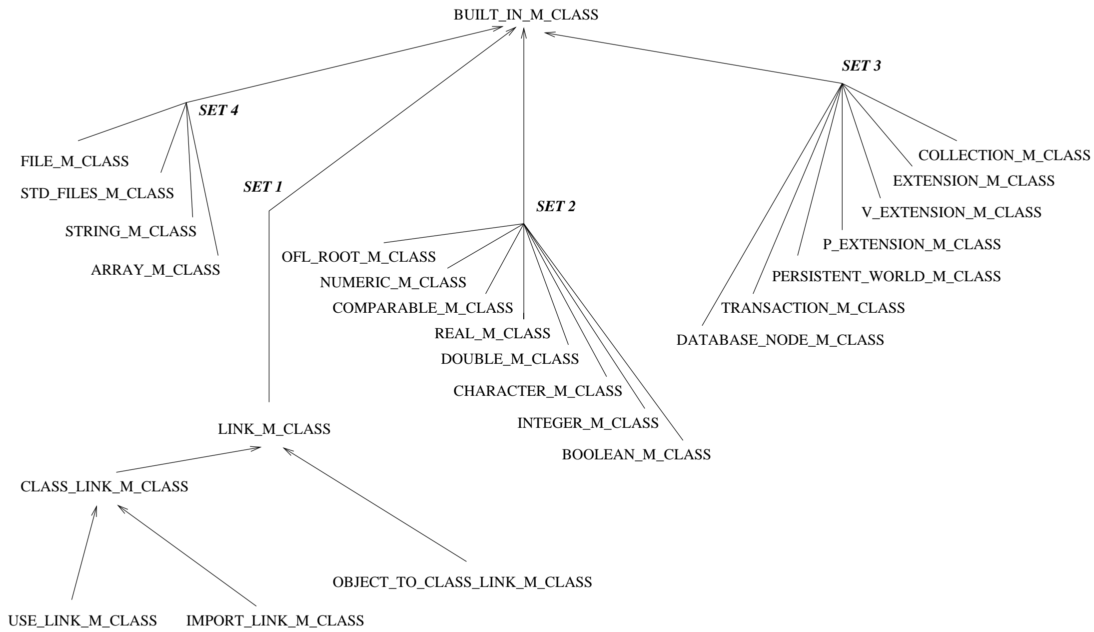


FIG. 2.2 – *Classes built-in*

Les classes *built-in* permettent la gestion d'instances de différentes natures comme on le verra dans la section 2.2.2, cependant il faut préciser que la gestion de la sémantique est globalement identique ; seule la recherche et éventuellement l'exécution des routines ne partagent pas la même sémantique que les classes modifiables. Toute procédure ou fonction d'une classe *built-in* devrait être décrite par une instance *ad hoc* de la classe `ROUTINE` (ex : `COPY_PROCEDURE` ou `EQUAL_CONTENTS`). Une des particularités des routines *built-in* est que le corps de la routine n'est pas réifiée. De même toute classe *built-in* pourra redéfinir les opérateurs génériques tels que la copie ou l'égalité d'instance qui sont pour l'instant fixés dans `BUILT_IN_M_CLASS`.

Pour l'instant seules les routines de la classe `OFL_ROOT_M_CLASS` sont associées à des descriptions spécifiques de la classe `ROUTINE` (voir figure 2.5).

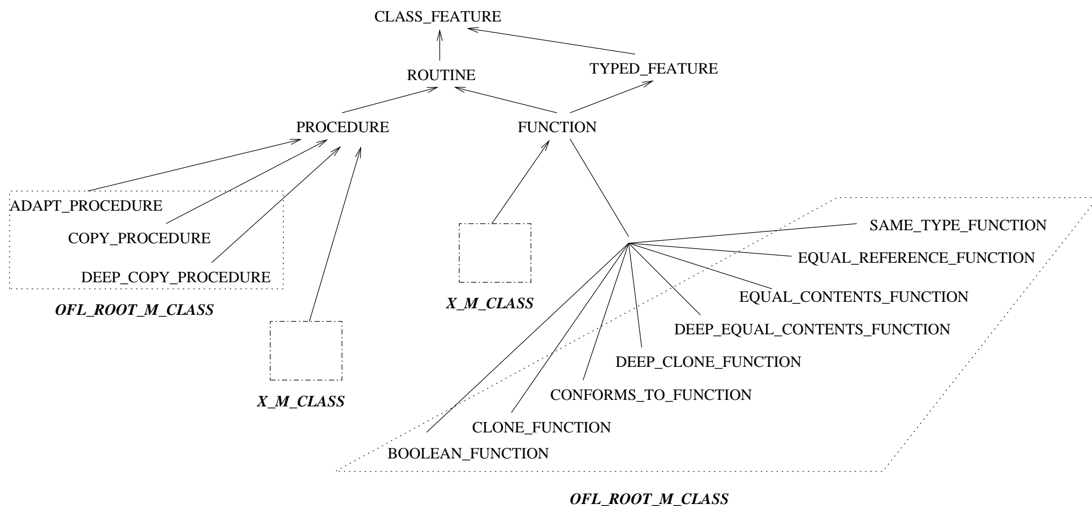


FIG. 2.3 – *Routines built-in*

### Classes Modifiables

L'ensemble des classes modifiables d'un programme est modélisé par la classe UPDATABLE_M_-CLASS.

Une classe permet de créer, de gérer et piloter l'exécution des instances de toutes les classes de l'application, sauf celles qui sont *built-in* (UPDATABLE_M_CLASS).

La figure 2.4 montre la hiérarchie permettant de modéliser la notion de type et de classe modifiable ou non. La classe FEATURE_HANDLING_SEMANTICS cache l'accès aux opérateurs qui paramètrent la sémantique des classes.

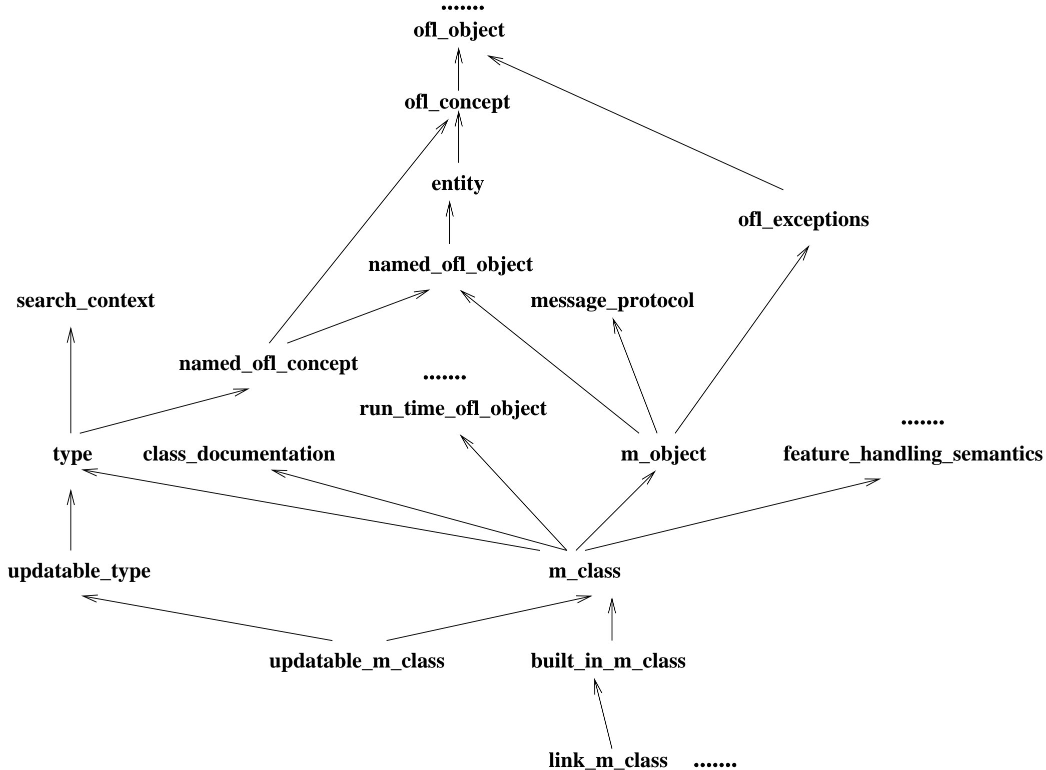


Fig. 2.4 – *Modélisation du concept de classe*

## 2.2.2 Instances *built-in* et modifiables

### Instances modifiables

Toutes les instances de classes modifiables sont modélisées par la classe UPDATABLE_OFL_OBJECT. Cette classe permet en particulier :

- la gestion de plusieurs champs de même nom et donc la prise en compte de toute sorte de surcharge,
- l'accès à la classe qui l'a créé à travers un lien et
- la soumission de demande de travail (de la plus générale à la plus spécialisée).

### Instances *built-in*

Les instances des classes *built-in* appartiennent à une hiérarchie qui débute par la classe BUILT_IN_OFL_OBJECT et qui se divise en plusieurs sous-hiérarchies qui correspondent aux catégories des classes *built-in* décrites ci-dessus (voir figure 2.5).

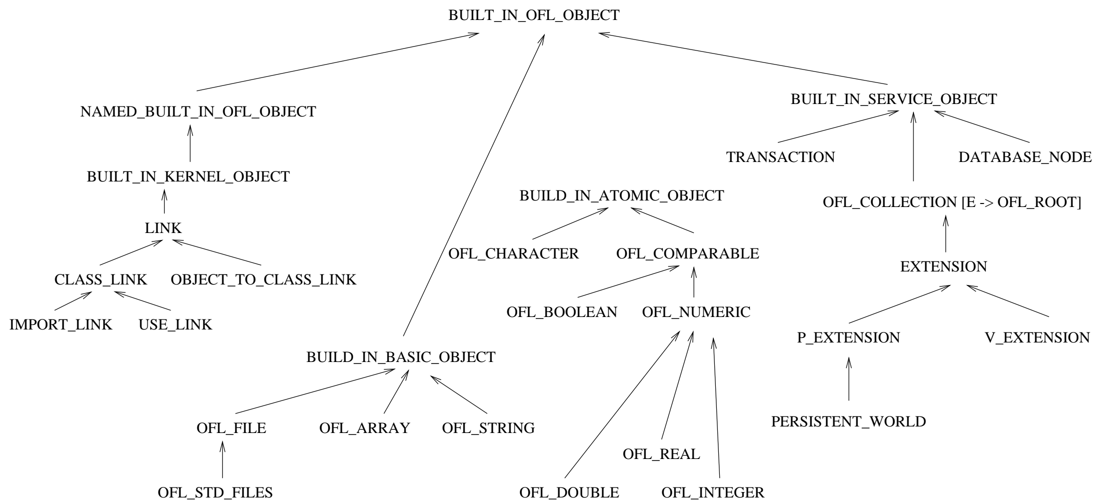


FIG. 2.5 – *Instances built-in*

On remarque que la modélisation de ces instances introduit de nombreuses interactions entre les différentes instances *built-in*. Il ne faut pas confondre l'utilisation du mécanisme d'héritage qui modélise des objets d'exécution avec les liaisons entre classes dont la sémantique doit être décrite par le méta-programmeur.

Pour faire la synthèse sur les classes et instances *built-in* sur un exemple : la classe `INTEGER_M_CLASS` est une méta-classe qui n'engendre qu'une seule classe : la classe `OFL_INTEGER` (cette méta-classe n'aura donc qu'une instance dont le nom est `INTEGER`). Les instances de cette classe sont représentées par des instances de la classe `OFL_INTEGER`.

### 2.2.3 Les liens : des entités *built-in* fondamentales

La hiérarchie de classes ayant pour racine `LINK_M_CLASS` est composée de plusieurs méta-classes dont seules les feuilles peuvent générer des instances de liens. Chacune de ces instances possèdent les opérateurs nécessaires à la description de la sémantique qu'elle véhicule. Il y a les liens entre classes et les liens entre un objet et une classe. Parmi les liens entre classes on distingue les liens d'importation (liens de version, liens d'héritage, ...) et les liens d'utilisation (liens de composition, d'agrégation, de clientèle, ...). On peut naturellement avoir besoin de plusieurs instances des méta-classes de lien (une pour chaque type de liens). La figure 2.6 montre la double hiérarchie des méta-classes et des instances de lien. On rappelle que la sémantique du lien est dans les instances des classes `*_LINK_M_CLASS` (type de lien) et que les propriétés propres à chaque lien sont mémorisées dans les instances des classes `*_LINK`.

On notera que dans notre approche nous n'incluons pas la possibilité de décrire des liens entre objets directement : tout lien entre objets se fait à travers une référence dont la sémantique est décrite par un lien d'utilisation.

## 2.3 Synthèse sur la modélisation des méta-classes

La figure 2.7 montre l'intégration des concepts de classe et d'instance dans la hiérarchie en insistant sur les aspects concernant la modélisation des classes, alors que la figure 2.8 décrit plus précisément les aspects liées aux objets d'exécution.

On peut noter sur ces schémas que les instances des méta-classes `UPDATABLE_M_CLASS` et `BUILT_IN_M_CLASS` sont toutes considérées aussi des objets d'exécution (`RUN_TIME_OFL_OBJECT`), comme les instances des classes qu'elles représentent (`UPDATABLE_OFL_OBJECT` et `BUILT_IN_OFL_-`

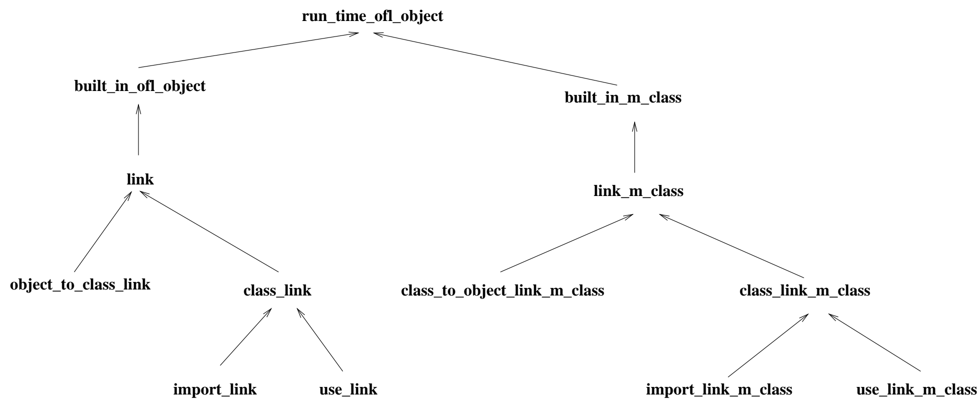


Fig. 2.6 – *Modélisation des liens*

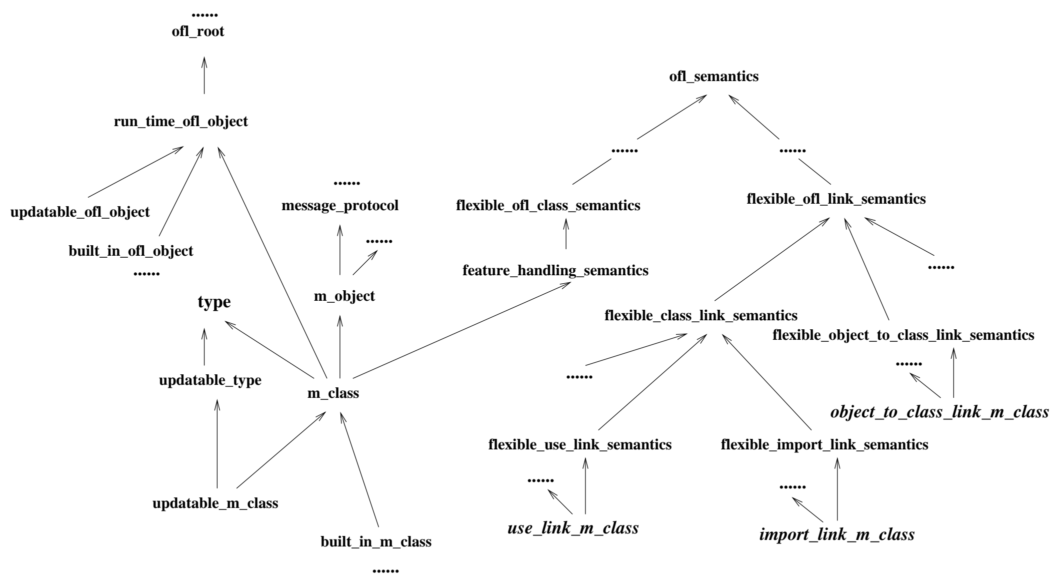


Fig. 2.7 – *Intégration des méta-classes et des aspects sémantiques*

OBJECT). L'intérêt est de permettre à une classe d'avoir le statut d'objet persistant, ce qui laisse de nombreuses ouvertures pour la gestion et l'archivage des classes, de même pour les *chercheurs d'objets* (OBJECT_FINDER) qui permette à un objet d'accéder à des informations sur sa localisation en mémoire persistante ou sur le réseau (voir section 2.6.1). Par ailleurs, une routine est aussi un objet d'exécution ce qui permet de mettre en œuvre la notion d'objet de première classe et de rendre possible la gestion de paramètres de type ROUTINE.

Par ailleurs on peut noter la localisation particulière de l'instance de la classe OFL_ROOT_-M_CLASS qui devrait être liée par un lien d'importation (voir chapitre 3) à toute classe. Cette instance est modélisée par la classe OFL_ROOT qui est placée au dessus de la classe RUN_TIME_OFL_-OBJECT ; elle cache ainsi aux autres objets les détails d'implémentation d'un objet d'exécution. Les opérateurs mis en œuvre au niveau de la classe OFL_ROOT_M_CLASS (copie et clonage d'instance, tests d'égalité, adaptation et conformance d'objets, ...) auront naturellement une implémentation de base adaptée à chaque objet (objets *built-in* ou non, classe, chercheur d'objet, ...) et pourront être paramétrés par méta-programmation (voir chapitre 3).

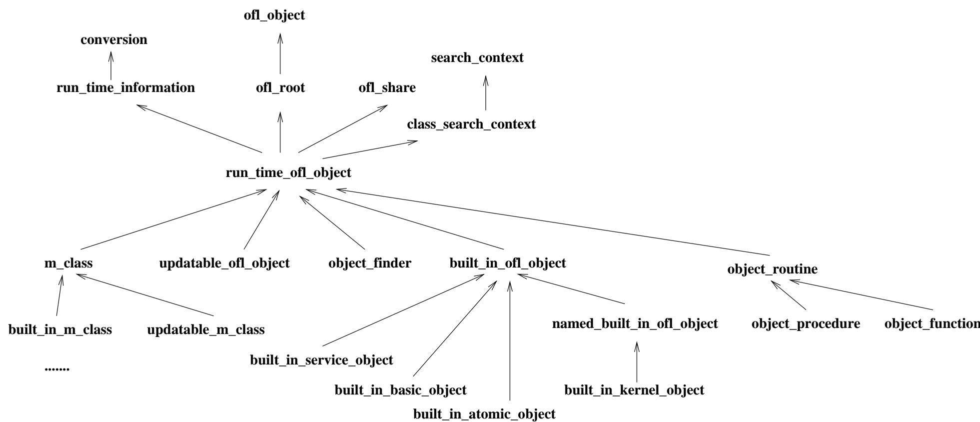


FIG. 2.8 – *Intégration des objets d'exécution*

## 2.4 Description des instructions

### 2.4.1 Instructions d'exécution

Une instruction d'exécution se trouve dans le corps d'une routine, dans une assertion. On distingue deux sortes d'instruction d'exécution : les instructions dont la sémantique est fixé (voir figure 2.9) et les instructions dont la sémantique peut changer (voir figure 2.10).

On notera que les instructions *built-in* représentent le strict nécessaire pour l'écriture de programme. Cet ensemble pourra être étendu en fonction des besoins ; cependant il ne faut pas oublier que l'objectif est uniquement d'avoir un ensemble d'instructions qui permette de modéliser pas trop difficilement tout ce que l'on trouve dans les langages classiques.

### 2.4.2 Instructions de déclaration

Les instructions de déclaration permettent de notifier à la machine virtuelle la déclaration de nouvelles entités (classe, lien, primitive, ...). L'intérêt de ces instructions est de rendre le code des applications indépendant de la structure interne de la machine virtuelle. Ces instructions sont rassemblées dans la figure 2.11.

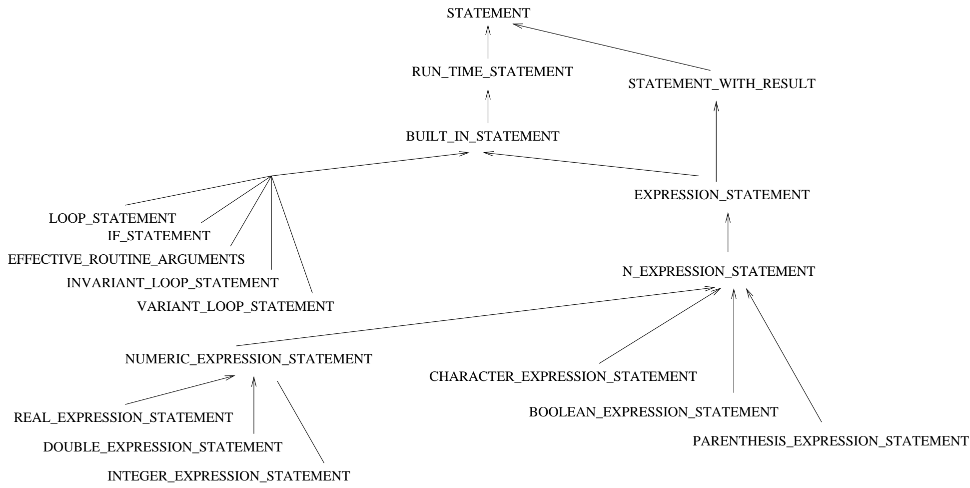


Fig. 2.9 – *Modélisation des instructions d'exécution built-in*

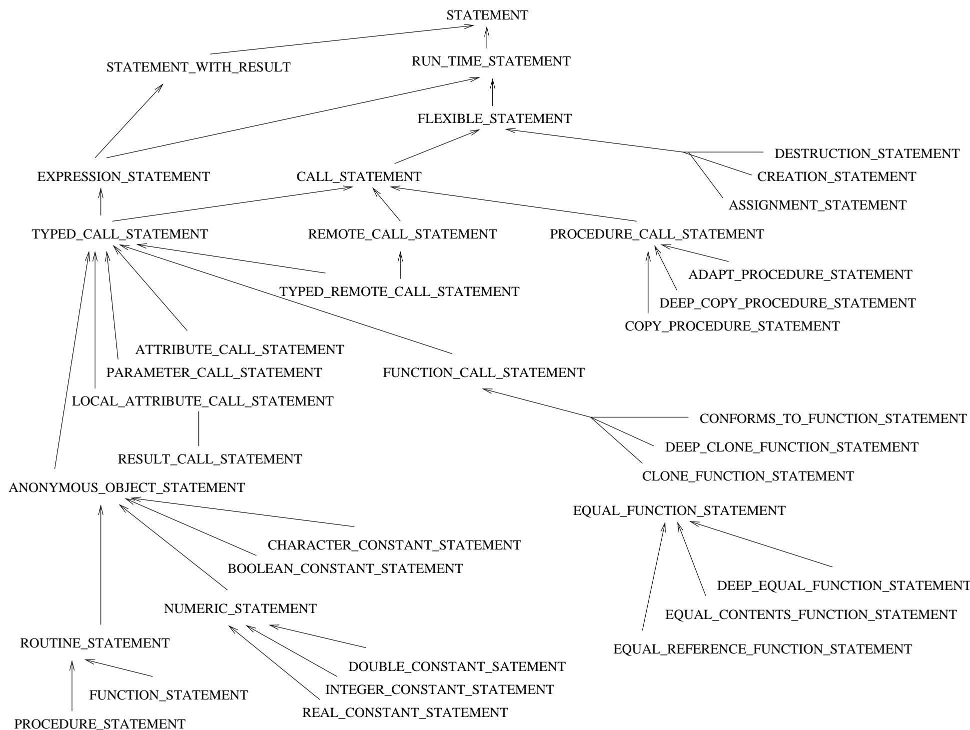


Fig. 2.10 – *Modélisation des instructions d'exécution flexibles*

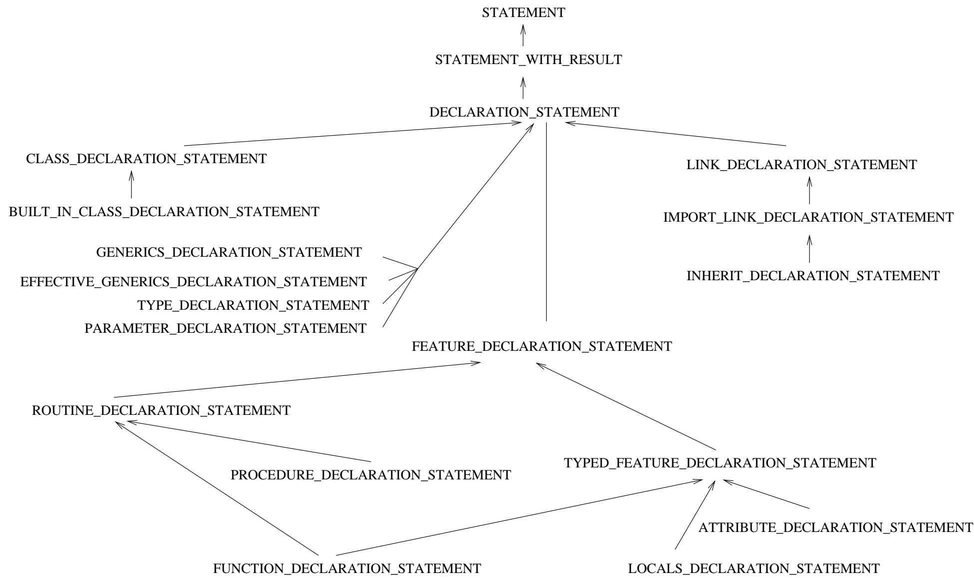


FIG. 2.11 – *Modélisation des instructions de déclaration*

## 2.5 Autres mécanismes

### 2.5.1 Clauses d'adaptation dans les liens d'importation

Cette section présente l'approche générale qui a servi à l'implémentation du système. Naturellement les structures de bas niveau qui permettrait une implantation performante reste a étudier [1].

Les objets qui permettent la gestion de l'adaptation des primitives à travers des liens sont des instances des classes suivantes :

`FEATURE_FILTER` mémorisation des clauses d'adaptation spécifiques à une primitive lorsqu'on traverse un lien ; ces clauses d'adaptations représentent le filtre d'une primitive entre deux classes ;

`CLASS_FEATURE` et ses descendants : mémorisation des informations relatives à une primitive et

`OFL_FEATURE_TABLE` qui est une table spécialisée pour le stockage et la recherche de filtres dans un lien.

Dans le cas ou il n'y a aucune clause d'adaptation pour une primitive lorsqu'on traverse un lien, aucune instance de la classe `FEATURE_FILTER` n'est associée à la primitive. Par contre à toute requête concernant le filtre appliqué à la primitive, le système renverra un filtre par défaut (pas de renommage, pas de redéfinition, pas d'abstraction, pas de masquage).

Une classe contient donc un certain nombre de liens de même type ou non ; on peut par exemple imaginer des liens de sous-typage et d'implémentation. Chaque lien contient les filtres qu'il met en œuvre concernant une primitive ; ces filtres concernent le changement de nom ou de comportement d'une primitive et éventuellement le masquage de cette primitive.

Si une primitive change de nom (`rename`), le nouveau est mémorisé dans le filtre, si la primitive est cachée (`hide`) ou n'a plus de comportement (`undefine`) cette information est aussi mémorisée dans le filtre ; par contre si c'est le comportement ou la signature de la primitive qui est modifié, alors le nouveau comportement ou la nouvelle signature est défini dans la classe initiatrice du lien par la déclaration d'une nouvelle description de primitive (le lien permet de gérer la filiation entre la nouvelle description de la primitive et la description d'origine).

Toute implémentation de la sémantique de la recherche de routine pour un donné (le plus connu étant le lien d'héritage), devra tenir compte de ce d'implantation dans le système mais sera

1. Voir *a general framework for inheritance management and method dispatch in object-oriented languages* dans ECOOP'97.

totalement libre de l'exploitation des informations potentiellement contenues dans les filtres pour déterminer la description de la primitive à retenir.

L'implémentation de la recherche pour un langage de programmation nécessite de pouvoir accéder à un certain nombre d'informations. L'ambition de supporter les principaux langages nécessite d'offrir un ensemble étendu de fonctionnalités pour :

- exécuter une routine avec un type statique différent du type dynamique de l'objet,
- récupérer l'ensemble des filtres d'une primitive à partir d'une classe donnée,
- ...

**Situation 1 - Type statique = type dynamique** Par exemple pour *Eiffel*, si on veut exécuter une primitive sur un objet avec comme point de vue son type dynamique (l'attribut par lequel on accède à l'objet a exactement le même type que l'objet), il suffit de remonter à partir de la classe qui a généré l'objet jusqu'à ce que l'on trouve la primitive, en résolvant éventuellement l'ambiguïté s'il y a plusieurs réponses (voir dans la figure 2.12, les appels à partir d'un attribut de type CARRE qui référence aussi un objet de type `CARRE`).

**Situation 2 - Type statique ≠ type dynamique** Par contre, toujours dans *Eiffel*, si on veut exécuter une primitive sur un objet avec comme point de vue un type différent de son type dynamique (un type parent dans le cas d'*Eiffel*), l'approche est un peu différente (voir figure 2.12) les appels à partir d'un attribut de type `FIGURE` (qui référence un objet de type `CARRE`). On parcourt la hiérarchie en partant de la classe représentant le type statique de l'attribut (le parcours se fait donc en sens inverse par rapport a la première situation). Ceci oblige à maintenir pour une classe une liste de ses descendants et ses ancêtres directs et à associer à chaque lien son type afin de pouvoir gérer l'héritage multiple et l'héritage répété.

On peut dire autrement les choses : Pour chercher la primitive on part toujours de la classe de la hiérarchie qui est associée au type statique de l'objet (type de l'attribut) et on se dirige vers la classe qui est associée au type dynamique de l'objet.

Ce type de recherche pourrait s'appliquer identiquement aux langages mentionnés plus haut qui supportent l'héritage, seuls changeraient les traitements des clauses d'adaptation associées aux liens. Pourtant on préférera laisser le méta-programmeur décider de la façon de rechercher la primitive car cela permet à *OFL* de s'adapter à de nouveaux langages et de pouvoir obtenir des solutions de parcours plus optimisées.

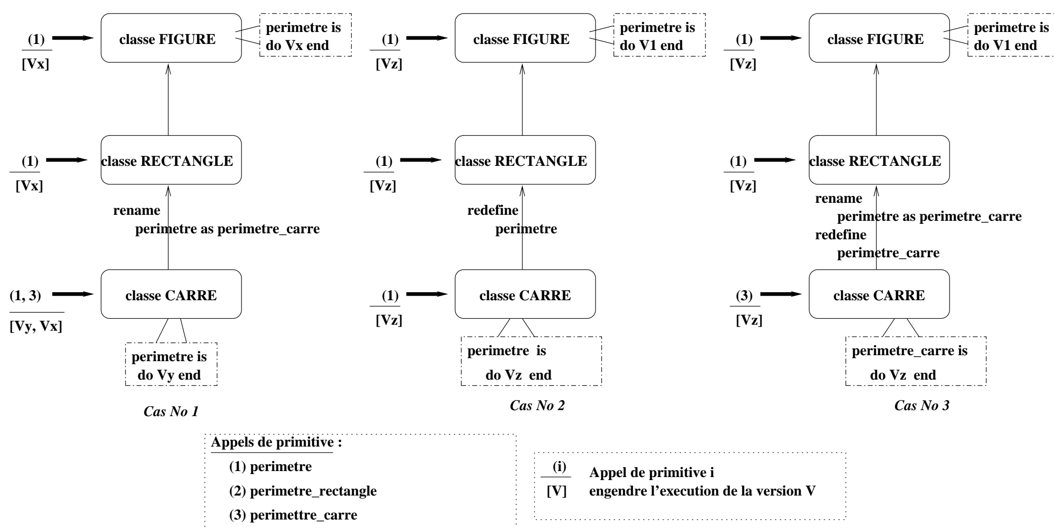


FIG. 2.12 – *Différentes clauses d'adaptation*

### 2.5.2 Modélisation des échanges entre objets

Dans *OFL*, il y a deux types de message : les messages internes à un objet et ceux envoyés à d'autres objets. Ce concept de message permet essentiellement de rassembler les informations nécessaires à l'accomplissement de la tâche à faire : il s'agit en particulier :

- de l'objet expéditeur et des informations nécessaires au calcul de l'objet destinataire ;
- de l'instruction à exécuter : on notera que dans un langage comme *Eiffel* un message externe ne pourrait correspondre qu'à une routine, tandis que pour un message interne ce pourrait être à la fois un attribut ou une routine ;
- du type (statique et dynamique) de l'attribut (ou de la fonction) dans l'objet expéditeur ;
- du type (statique et dynamique) de l'attribut (ou de la fonction) dans l'objet destinataire ;
- de la mémorisation que le message nécessite une réponse ou pas ;
- du caractère synchrone/asynchrone du message et
- du caractère calculé ou non du message.

La figure 2.13 montre la hiérarchie de classes qui permet la modélisation d'un message et le lien avec les instructions qui se trouvent dans le corps d'une routine. On notera en particulier que toute instruction dont la sémantique est paramétrée est considérée comme une demande de travail potentielle et, à ce titre, il est possible de voir cette demande comme une demande interne à un objet (`INTRA_MESSAGE`) ou comme une demande faite à un autre objet *via* éventuellement des objets intermédiaires (`MESSAGE`). Un message est constitué éventuellement de messages intermédiaires (`VIA_MESSAGE`).

Une affectation est une demande de travail interne à l'objet courant tandis que la création ou la destruction est une demande de travail interne à une instance de méta-classe (éventuellement la méta-classe pourra en interne faire appel aux services d'autres instances de méta-classe). Un appel de primitive est une demande de travail interne à l'objet courant alors qu'un appel de primitive distant (utilisation de la notation pointée) n'est pas une demande de travail interne mais le traitement d'un message mettant obligatoirement en jeu plusieurs objets.

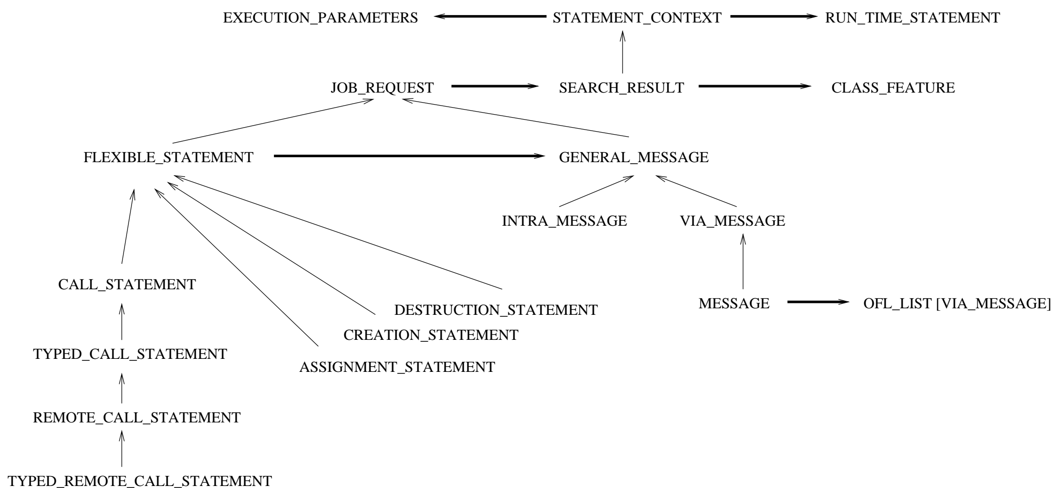


FIG. 2.13 – *Hiérarchie relative aux messages*

La figure 2.14 montre le contenu d'un message (instance de MESSAGE) qui permet de gérer les échanges entre objets. En particuliers un message contient :

- l'expéditeur du message,
- l'instruction (appel de routine en général) qui permet de calculer le destinataire,
- la primitive utilisée pour calculer le destinataire une fois qu'elle a été déduite de l'instruction,

– le lien (lien d'utilisation) à utiliser pour transmettre le message (une fois que le destinataire est calculé),
– des informations d'exécution qui peuvent être ajoutés,
– les paramètres évalués et
– les messages intermédiaires éventuels, lorsqu'il s'agit d'une expression pointée (REMOTE_-CALL_STATEMENT).

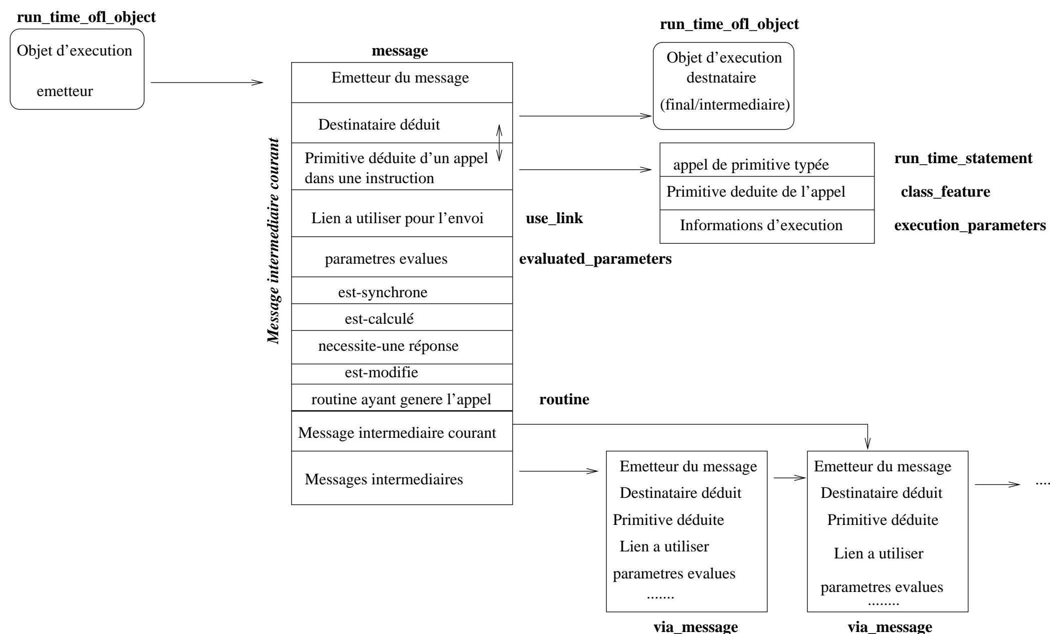


FIG. 2.14 – *Modélisation des messages*

### 2.5.3 Gestion de la visibilité

Notre volonté a été ici d'étendre le mécanisme d'exportation associé au lien de clientèle d'*Eiffel* et présent de manière plus limitée dans d'autres langages. Ce mécanisme s'appuie sur les classes :

FEATURE_GROUP_REACTION qui décrit le comportement d'un méta-objet pour un groupe de primitives lorsque la demande est faite par un autre méta-objet à travers un lien donné (un même comportement pour toutes les primitives) et

MESSAGE_PROTOCOL qui contient l'ensemble des réactions d'un méta-objet. Si aucune réaction n'est spécifiée pour une primitive c'est la réaction par défaut (accepter la demande) qui s'applique. La classe M_OBJECT hérite de MESSAGE_PROTOCOL et son descendant direct est M_CLASS.

Ce mécanisme prend en compte la possibilité pour une classe (ou plus généralement un méta-objet), d'accepter, ignorer, refuser, retarder ou demander une nouvelle fois un message représentant une demande d'utilisation d'une primitive à travers un lien donné. En fonction de la sémantique du langage que l'on veut modéliser on peut naturellement associer à ces types de situation les actions appropriées.

Le comportement de la classe ou du méta-objet peut être le même pour tous ou différer suivant l'identité des méta-objets. Ce comportement dépend aussi du lien qui est utilisé pour transmettre le message, ce qui permet d'étendre le contrôle d'accès à une primitive à d'autres liens que le lien de clientèle.

Par exemple un objet de type BANQUE pourra refuser l'accès à la primitive retirer_du_compte à une personne qui n'est pas cliente (instance de la la classe PERSONNE_CLIENTE), autoriser son utilisation aux employés (instance de la classe EMPLOYE) et retarder l'accès aux interdits de chèque

(instance de la classe PERSONNE_A_SURVEILLER). Naturellement ces possibilités dépendent du langage et donc les actions associées notamment à *retarder* aussi.

## 2.6 Services supplémentaires de *OFL/VM*

### 2.6.1 Persistance et mobilité des objets

L'objectif est de fournir un moteur qui offre des services pour les objets persistants. Cependant ce service doit s'appuyer sur une approche orthogonale à jour avec l'état de l'art du moment. Cette partie est encore pour l'instant peu étudié et nécessite un travail de recherche approfondi.

#### Critères d'intégration de la persistance

Il est nécessaire dans un premier temps d'offrir un mécanisme de persistance qui garantisse la possibilité, pour un langage qui a pour cible notre plate-forme, d'offrir une gestion transparente de la persistance. Il faut par ailleurs dés maintenant envisager une persistance distribuée sur le réseau. La modélisation proposé doit en particulier tenir compte des approches de *Corba*, de l'*ODMGg*, de *Java* persistant et de *FLOO*.

#### Éléments pour l'intégration de la persistance

La persistance est un service important qui doit être offert par la plate-forme *OFL*, c'est pourquoi elle intègre un mécanisme de gestion de la persistance. Attention, notre propos n'est pas de refaire un serveur d'objets selon les critères des bases de données mais d'intégrer la modélisation de la persistance et de proposer un serveur d'objet *light* (voir figure 2.15).

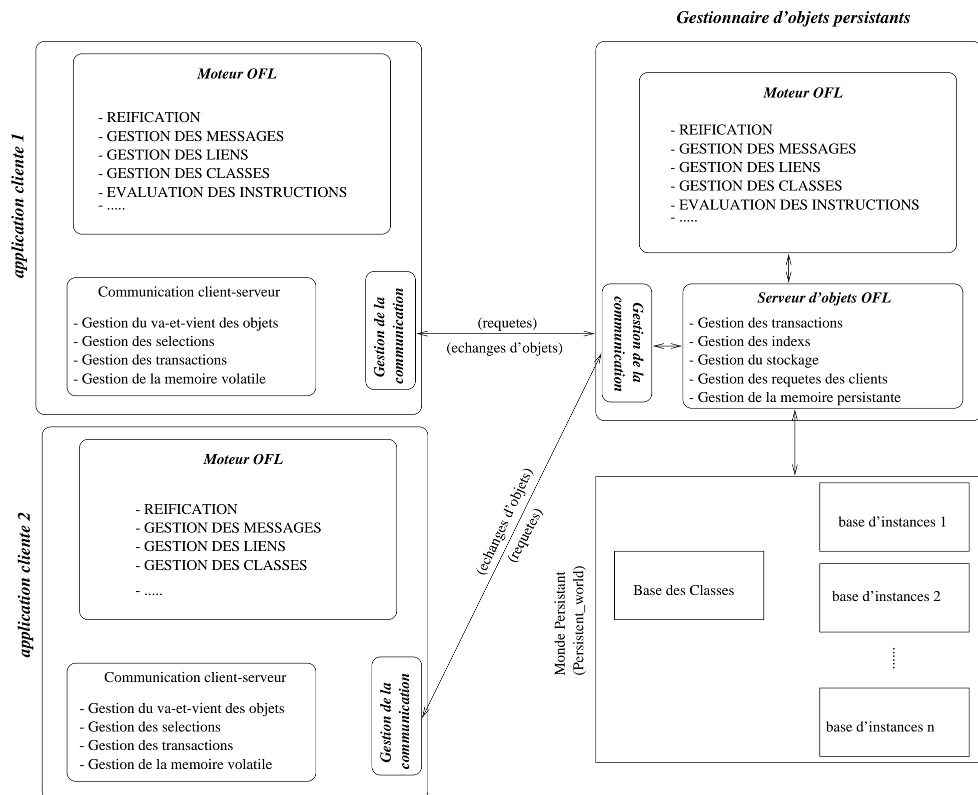


FIG. 2.15 – *Intégration de la persistance*

Cette intégration devrait se faire de manière assez simple au niveau des liens entre classes et les liens d'instanciations. Il est nécessaire de gérer la persistance à la fois au niveau des liens d'importations et de clientèle et au niveau des liens *objet vers classe* pour pouvoir gérer (voir section 3.2.1) :

- à partir d'un opérateur comme `lookup` (lien d'importation) le chargement éventuel et la mise à jour des classes (qui sont typiquement des objets persistants), au moment nécessaire dans la recherche d'une primitive ;
- à partir d'un opérateur `submit` (lien *objet vers classe*) d'une instance de classe (`UPDATABLE_OFL_OBJECT` ou `BUILT_IN_OFL_OBJECT`), le chargement de la classe qui contient sa sémantique et
- à partir d'opérateurs comme `send` (lien d'utilisation), le chargement des instances persistantes de ces classes. C'est à partir d'autres opérateurs comme `assign` et `after_request` que l'on mettra en œuvre la mise à jour et la synchronisation entre l'instance persistante et son image en mémoire volatile.

Cette intégration se fait par l'ajout de séquences d'actions dans les différents opérateurs qui paramètrent la sémantique des liens et des classes. Ceci nous permet de dire que l'usage de la persistance est paramétré dans *OFL/VM* mais pas l'implantation.

Par exemple, cette séquence d'action doit permettre, au moment de l'envoi de message et donc au moment du changement d'objet courant de charger un objet persistant si celui-ci n'est pas déjà en mémoire persistante. Les classes `PERSISTENT_OBJECT_MANAGER`, `PERSISTENT_OBJECT_MANAGER_NODE`, `DATABASE_NODE` et les autres classes fournisseurs doivent implémenter le serveur d'objets et la communication éventuelle avec d'autres serveurs d'objets. Ces classes ne doivent à aucun moment être manipulées par le programmeur (ni par le méta-programmeur, en principe).

Par contre un certains nombre de classes de base permettent la manipulation (consultation, sélection, ...) des objets persistants et, à ce titre, elles peuvent être manipulées par des programmes (ces classes sont en particulier accessibles par la classe `OFL_ROOT_M_CLASS`). Il s'agit des classes `PERSISTENT_WORLD`, `OFL_COLLECTION`, `V_EXTENSION`, `EXTENSION`, `P_EXTENSION` et `TRANSACTION`.

Enfin, la classe `OBJECT_FINDER` est la classe de base pour l'intégration de la persistance ; cette classe permet de localiser l'objet, qu'il soit volatile ou persistant. À terme, cette classe devra pouvoir gérer la recherche d'un objet sur le réseau. Le statut d'objet d'exécution permet à tout moment, et seulement pour les objets *choisis* par l'usage de la persistance qui est défini au niveau des opérateurs sémantiques, de remplacer un objet persistant par son *chercheur d'objet*.

Plus précisément, à la création de tout objet, on peut (en définissant le contenu des classes héritières de la classe `OFL_SEMANTICS`), faire en sorte d'attacher une instance de cette classe à tout objet (qu'il soit volatile ou persistant). De même, dans le code décrivant la sémantique de l'affectation on pourra en outre associer l'instance de la classe `OBJECT_FINDER` au champ approprié dans l'instance de la classe `UPDATABLE_OFL_OBJECT`. Ainsi chaque instance de cette classe, qui par un de ses champs veut référencer une instance de la classe `RUN_TIME_OFL_OBJECT` (`UPDATABLE_OFL_OBJECT` ou `BUILT_IN_OFL_OBJECT`), pourra pointer sur l'instance de la classe `OBJECT_FINDER` associée à cet objet. On notera que dans la classe `UPDATABLE_OFL_OBJECT` le type statique de chaque champ est `RUN_TIME_OFL_OBJECT`, ce qui permet au méta-programmeur qui décrit la sémantique du langage de choisir entre passer toujours par un `OBJECT_FINDER` (qui hérite de `RUN_TIME_OFL_OBJECT`), ou d'optimiser quand c'est possible en référençant directement l'objet volatile.

Les classes `PERSISTENT_OBJECT_IDENTIFIER` et `VOLATIL_OBJECT_IDENTIFIER` permettent de modéliser l'adresse de l'objet volatile, l'adresse de l'image de l'objet persistant en mémoire volatile ou l'adresse de l'objet persistant mémoire persistante. On notera que tout objet persistant est attaché à un seul monde persistant qui peut éventuellement se trouver sur un noeud distant du réseau. Les classes `PERSISTENT_KEY` et `VOLATIL_KEY` sont les classes de bas niveau implémentant ces deux notions. On notera que le gestionnaire d'objets persistants est unique localement, qu'il est directement accessible à partir d'une instance de type `OBJECT_FINDER` et que c'est à lui de transmettre les requêtes à des gestionnaires distants si nécessaire.

# Chapitre 3

# Sémantique opérationnelle d'un langage

On vient de voir qu'il y a deux catégories de méta-classe :

- celles qui modélisent le concept de classe (*built-in* ou modifiable) et
- celles qui modélisent le concept de lien (`LINK_M_CLASS` et descendantes), qui sont des spécialisations du concept de classe (*built-in*) et qui offrent des primitives pour modéliser la sémantique des liens.

La figure 2.9 montre l'intégration de la sémantique et la gestion de la flexibilité de cette sémantique. On reviendra plus tard à la gestion de la flexibilité mais, en gros, on a deux architectures qui cohabitent : l'architecture qui modélise la sémantique selon *OFL* et la hiérarchie qui modélise la flexibilité de la sémantique.

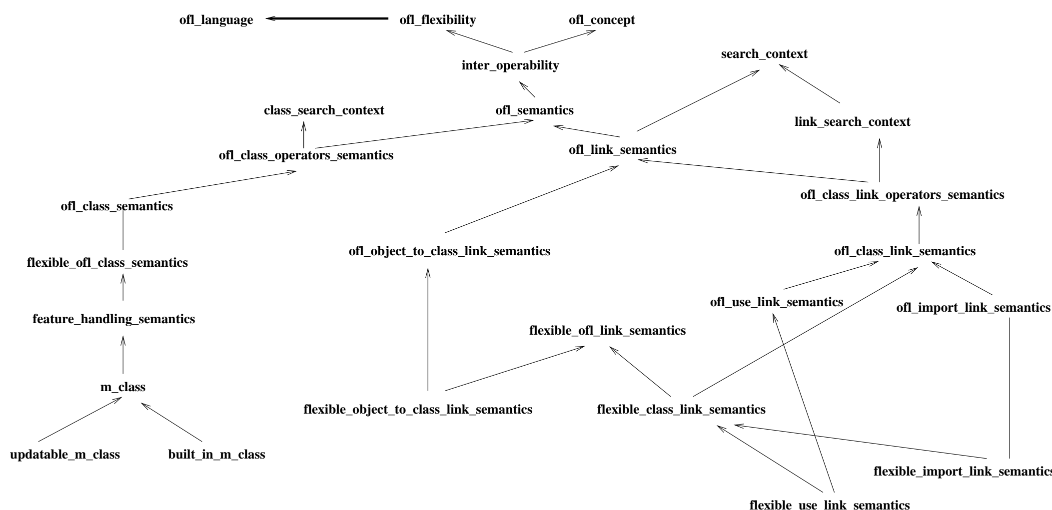


FIG. 3.1 – *Intégration de la modélisation de la sémantique*

La figure 3.2 montre les classes qui vont contenir la sémantique pour les différents langages.

Il y a deux sortes caractéristiques dans les méta-classes, dont le but est :

- la création et la gestion des classes et leur contenu (routines, attributs, liens entre classes, ...). Ces caractéristiques représentent la réification du concept de classe et
- la gestion du comportement des instances de ces classes à l'exécution ; la sémantique associée aux descendants de la classe `M_CLASS` (`UPDATABLE_M_CLASS` et `*_LINK_M_CLASS`) définit la sémantique dynamique du langage.

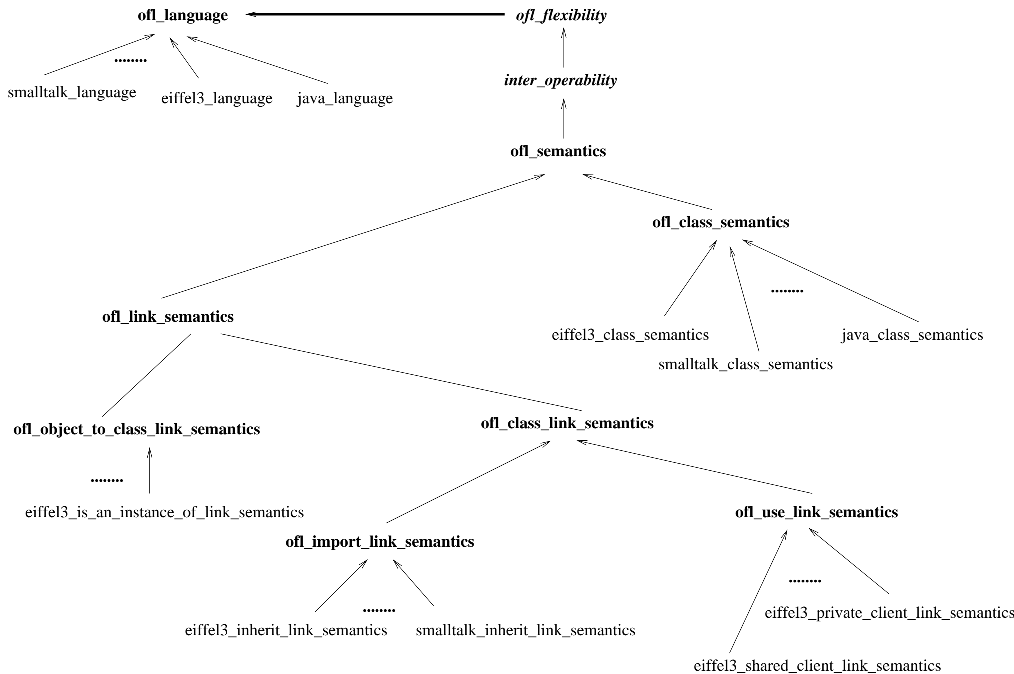


FIG. 3.2 – *Organisation de la sémantique propre à chaque langage*

La conception de *OFL* permet de gérer pour un même langage plusieurs sémantiques de classes (le mot *classe* est à prendre avec précaution : il définit simplement une structure pouvant contenir des liens et des primitives), et naturellement plusieurs types de lien. Par exemple il sera possible de modéliser un langage qui offrira à la fois un concept de classe à la *Eiffel* et un concept de vue à la *VBOOL* et qui intégrerait des liens d'héritage, de clientèle et de version.

Pour bien comprendre ce qui va suivre il faut rappeler les objectifs du système :

- offrir une réification des principaux concepts que l'on peut trouver dans des langages comme *Eiffel*, *Java* ou *C++* et en particulier les concepts de classe et de lien entre classes. Cette réification ne colle pas à un langage donné (bien que nous nous soyons largement inspirés du langage *Eiffel*) mais se veut au contraire un support pour implémenter ces langages et plus généralement, d'autres langages.
- proposer un paramétrage des réactions des objets selon les situations dans lesquelles on les place ; ces réactions dépendront du contenu des primitives qui décrivent la sémantique du ou des types de classe et des types de lien ; il y a deux niveaux de paramétrage :
  - un paramétrage au niveau des méta-classes `LIEN` (`IMPORT_LINK_M_CLASS`, `USE_LINK_M_CLASS` et `OBJECT_TO_CLASS_LINK_M_CLASS`) qui décrivent l'influence d'un lien dans les différentes situations prévues dans le langage et
  - un paramétrage au niveau de la méta-classe `CLASSE` (`M_CLASS` et ses descendants directs) qui met en œuvre la coordination entre les différents liens et types de lien.

Tous les opérateurs prennent en paramètre la demande de travail ce qui permet a chacun d'accéder à toute information nécessaire (voir figures 2.13 et 2.14), et de mettre à jour ces informations au fur et à mesure afin que les opérateurs suivants puissent les utiliser. Par ailleurs les opérateurs se trouvant dans les classes descendant de `OFL_LINK_SEMANTICS` doivent principalement modéliser l'incidence du lien sur une action donnée, décrite par l'opérateur, tandis que les opérateurs de la classe `OFL_CLASS_SEMANTICS` doivent principalement coordonner les actions faites dans les opérateurs de lien.

## 3.1 Les Méta-classes CLASSE

Ces méta-classes sont représentées, comme on l'a dit plus haut par des classes modifiables et *built-in*. La principale différence au niveau des fonctionnalités est que les premières possèdent des primitives permettant d'ajouter, de modifier ou de supprimer des attributs, des fonctions des procédures et des liens, alors que les secondes n'offrent pas ces fonctionnalités.

Les principales situations prises en compte au niveau de la méta-classe `CLASSE` sont :
- la recherche de primitive,
- les contrôles sémantiques (dynamiques),
- l'exécution de primitives,
- l'envoi de message,
- la création et la destruction d'instances,
- l'attachement d'objets et
- les opérateurs qui dépendent des liens (`clone`, `copy`, ...).

#### Description des services offerts par la classe OFL_CLASS_SEMANTICS

```
Deferred class OFL_CLASS_SEMANTICS

feature -- Création / mise à jour de l'extension

 make_extension(c : like class_anchor) is
 -- Déclenchement de la construction de l'extension
 -- Utilise les filtres de construction présents dans les liens
 -- et les extensions des classes "liées"
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void

 update_extension(c : like class_anchor) is
 -- Déclenchement de la mise à jour de l'extension
 -- Utilise les filtres de construction présents dans les liens
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void

-- Gestion de primitives de la classe en général (locales ou non)

 all_features(c : like class_anchor) : OFL_FEATURE_TABLE [like item] is
 -- retourne les routines de la classe
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void

-- Gestion de la recherche d'une primitive a partir d'un appel de primitive

 item(c : like class_anchor ; s : like request) : CLASS_FEATURE is
 -- retourne une primitive correspondant a 's'
 -- Gestion de la recherche dans la classe et dans les liens
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require else
  request_is_a_call : s.call_action /= void
 ensure then
  is_consistent : match(c, Result, s)
  is_updated : s.candidate_feature = Result

 match(c : like class_anchor ; f : like item ; s : like request) : BOOLEAN is
 -- retourne 'True' si 'f' correspond a 's' dans la classe elle-même
 -- Définit ce que veut dire "correspond"
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void and f /= void
  request_is_a_call : s.call_action /= void
  name_is_the_same : f.name = s.call_action.feature_name
 ensure
  is_not_modified : s = old s and f = old f

 local_lookup(c : like class_anchor ; s : like request) : like lookup is
 -- retourne le résultat de la primitive correspondant a 's'
 -- Gestion de la recherche dans la classe en utilisant les liens
 -- d'utilisation associés aux primitives quand elles sont typées
```

```
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void
  request_is_a_call : s.call_action /= void
 ensure
  exists : Result /= void and match(c, Result.candidate, s)

 lookup(c : like class_anchor ; s : like request) : SEARCH_RESULT is
 -- retourne le résultat de la recherche correspondant a 's'
 -- a partir d'une recherche a travers les liens d'importation
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void
  request_is_a_call : s.call_action /= void
 ensure
  exists : Result /= void and match(c, Result.candidate, s)

 is_import_links_valid(c : like class_anchor) : BOOLEAN is
 -- Est-ce que les liens d'importation ayant 'c' pour source sont
 -- valides pour le langage (héritage simple, multiple, répété, ..)
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void

 is_use_links_valid(c : like class_anchor ; f : like item) : BOOLEAN is
 -- Est-ce que les liens d'utilisation partant d'une primitive 'f'
 -- sont valides pour le langage (nombre de liens, ...)
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  feature_is_not_void : f /= void

-- Contrôle et gestion des paramètres associes a un appel de primitive

 is_parameters_compatible(c : like class_anchor ; p : like request) : BOOLEAN
  is
 -- Est-ce que les paramètres effectifs sont compatibles avec
 -- les paramètres formels de la demande de travail de 'p'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  request_is_set : p /= void and then p.contents /= void
  is_a_routine : p.contents.to_routine /= void
  parameters_set : p.parameters /= void

 parameter_evaluation(c : like class_anchor ; p : like request) is
 -- Partie de code qui doit être exécutée lors de
 -- l'evaluation des paramètres effectifs de la demande
 -- de travail de 'p'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  request_is_set : p /= void and then p.contents /= void
  is_a_routine : p.contents.to_routine /= void
  parameters_set : p.parameters /= void
 ensure
  parameters_are_evaluated : p.evaluated_parameters /= void

 effective_to_formal(c : like class_anchor ; p : like request) is
 -- Partie de code qui doit être exécutée lors de
 -- l'association des paramètres effectifs aux paramètres
 -- formels de la primitive associée a la demande de travail 'p'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  request_is_set : p /= void and then p.contents /= void
  is_a_routine : p.contents.to_routine /= void
  parameters_are_evaluated : p.evaluated_parameters /= void
 ensure
  formals_set : p.evaluated_parameters /= old p.evaluated_parameters

 detach_effective(c : like class_anchor ; p : like request) is
 -- Partie de code qui doit être exécutée lors du
 -- détachement des paramètres effectifs des paramètres
 -- formels de la primitive associée a la demande de travail 'p'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
```

```
  request_is_set : p /= void and then p.contents /= void
  is_a_routine : p.contents.to_routine /= void
  parameters_are_evaluated : p.evaluated_parameters /= void
 ensure
  formals_set : p.evaluated_parameters /= p.evaluated_parameters

-- Définition des règles de contrôle de type.

 is_type_conformance(c : like class_anchor ; t1, t2 : like class_to_type) :
  BOOLEAN is
 -- retourne 'True' si le type 't1' est conforme au type 't2'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 -- note : l'ordre est important
 require
  class_is_not_void : c /= void
  is_not_void : t1 /= void and t2 /= void
 ensure
  is_not_modified : t1 = old t1 and t2 = old t2

 is_same_type(c : like class_anchor ; t1, t2 : like class_to_type) : BOOLEAN is
 -- retourne 'True' si le type 't1' est égal au type 't2'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : t1 /= void and t2 /= void
 ensure
  is_not_modified : t1 = old t1 and t2 = old t2

 is_generics_valid(c : like class_anchor ; t : like class_to_type) : BOOLEAN is
 -- Est-ce que l'instanciation des paramètres génériques dans 't' est
 -- compatible avec la définition des paramètres génériques formels
 -- de la classe associée a 't'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  request_is_set : t /= void and t.is_generic

-- Exécution avant / après la primitive : coordination de l'action des liens

 is_valid_request(c : like class_anchor ; r : like request) : BOOLEAN is
 -- est-ce que le traitement associe a la demande de travail designée
 -- par 'r' est autorise a s'exécuter
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  candidate_is_set : r /= void
  is_a_call : r.call_action /= void

 before_request(c : like class_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de l'exécution
 -- du traitement désigné par 'r'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  candidate_is_set : r /= void
  is_a_call : r.call_action /= void

 after_request(c : like class_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de l'exécution
 -- du traitement désigné par 'r'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  candidate_is_set : r /= void
  is_a_call : r.call_action /= void

-- Contrôle de l'exécution appel de primitive

 execute_with_result(c : like class_anchor ; m : INTRA_MESSAGE) : OFL_ROOT is
 -- Exécution du message interne 'm' et retour d'un résultat
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  message_not_void : m /= void
  is_not_void : m.action /= void and m.parameters /= void
  exists : c.has(m)

 execute(c : like class_anchor ; m : INTRA_MESSAGE) is
 -- Exécution du message interne 'm'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
```

```
 require
  class_is_not_void : c /= void
  message_not_void : m /= void
  is_not_void : m.action /= void and m.parameters /= void
  exists : c.has(m)

-- Gestion de l'opérateur d'envoi de message synchrone/asynchrone
-- C'est la nature du message qui dicte la stratégie utilisé

 send(c : like class_anchor ; m : MESSAGE) is
 -- Envoi du message 'm' de manière
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  message_not_void : m /= void
  is_not_void : m.action /= void and m.parameters /= void
  is_in_class : c.has(m)

 send_with_response(c : like class_anchor ; m : MESSAGE) :
  like execute_with_result is
 -- Retourne le résultat de l'envoi du message 'm'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  message_not_void : m /= void
  is_not_void : m.action /= void and m.parameters /= void
  is_in_class : c.has_typed_feature(m)
  is_a_message_with_response : m.is_with_response

-- Gestion de la soumission d'une instruction

 submit(c : like class_anchor ; s : like request) is
 -- Soumission d'une instruction 's'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void

 submit_with_response(c : like class_anchor ; m : like request) :
  like execute_with_result is
 -- Retourne le résultat de l'envoi d'une instruction 'm'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : m /= void

-- Opérateurs d'affectation

 assign(c : like class_anchor ; s : ASSIGNMENT_STATEMENT) is
 -- définition de l'affectation
 -- sur 'current_object'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void
  target_is_set : s.target /= void
 ensure
  c.current_object = old c.current_object
  current_is_modified : c.current_object.is_modified
  object_is_set : is_type_conformance(c, s.target.statement_result.type,
   s.source.statement_result.type) and then c.is_current_updatable implies
   c.current_updatable.any_field(s.target.feature_name).value =
   s.source.statement_result

-- Gestion de la création / destruction d'instances

 create(c : like class_anchor ; s : CREATION_STATEMENT) is
 -- Création d'une instance de la classe
 -- a partir de 's'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void
  target_is_set : s.target /= void
 ensure
  object_is_created : c.is_current_updatable implies
   c.current_updatable.any_field(s.target_name) /= void
  object_is_typed : s.new_type /= void implies
   is_same_type(c, c.current_object.type, s.new_type.declared_type)
```

```
 instance_creation(c : like class_anchor ; s : CREATION_STATEMENT) :
  OFL_ROOT is
 -- Création d'une instance de la classe a partir de 's'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void
  is_not_generic : not c.is_generic

 generic_instance_creation(c : like class_anchor ; s : CREATION_STATEMENT ;
  g : EFFECTIVE_GENERIC_PARAMETERS) : like instance_creation is
 -- Création d'une instance de la classe a partir de 's'
 -- 'g' représente une instanciation des paramètres génériques
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void
  is_generic : c.is_generic

 destroy(c : like class_anchor ; s : DESTRUCTION_STATEMENT) is
 -- destruction d'une instance de la classe a partir de 's'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void
  target_is_set : s.target /= void
 ensure
  object_is_destroyed : c.current_object /= old c.current_object

 instance_destruction(c : like class_anchor ; s : DESTRUCTION_STATEMENT) is
 -- Destruction d'une instance de la classe identifie par 's'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  is_not_void : s /= void
  target_is_set : s.target /= void

-- Exécution de primitives spécifiques

 equal_reference(c : like class_anchor ;
  s : EQUAL_REFERENCE_FUNCTION_STATEMENT) : OFL_BOOLEAN is
 -- Exécution de l'égalité de reference sur 'current_object'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  operator_not_void : s /= void
  is_an_equality_statement : s.feature_name = N_equal_reference
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 equal_contents(c : like class_anchor ; s : EQUAL_CONTENTS_FUNCTION_STATEMENT)
  : OFL_BOOLEAN is
 -- exécution de l'égalité de contenu sur 'current_object'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  operator_not_void : s /= void
  is_an_equality_statement : s.feature_name = N_equal_contents
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 deep_equal_contents(c : like class_anchor ; s : DEEP_EQUAL_FUNCTION_STATEMENT)
  : OFL_BOOLEAN is
 -- exécution de l'égalité profonde de contenu sur 'current_object'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  operator_not_void : s /= void
  is_an_equality_statement : s.feature_name = N_deep_equal_contents
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 copy_instance(c : like class_anchor ; s : COPY_PROCEDURE_STATEMENT) is
 -- exécution de la copie de contenu sur 'current_object'
 -- 'c' représente la classe a laquelle on veut associer la sémantique
 require
  class_is_not_void : c /= void
  operator_not_void : s /= void
  is_a_copy_statement : s.feature_name = N_copy
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 deep_copy_instance(c : like class_anchor ; s : DEEP_COPY_PROCEDURE_STATEMENT)
  is
```

```
-- exécution de la copie profonde de contenu sur 'current_object'
-- 'c' représente la classe a laquelle on veut associer la sémantique
require
 class_is_not_void : c /= void
 operator_not_void : s /= void
 is_a_copy_statement : s.feature_name = N_deep_copy
 is_correctly_set : s.parameters /= void and then s.parameters.count = 1

clone_instance(c : like class_anchor ; s : CLONE_FUNCTION_STATEMENT)
 : RUN_TIME_OFL_OBJECT is
-- exécution d'un clone d'objet sur 'current_object'
-- 'c' représente la classe a laquelle on veut associer la sémantique
require
 class_is_not_void : c /= void
 operator_not_void : s /= void
 is_a_copy_statement : s.feature_name = N_clone
 is_correctly_set : s.parameters /= void and then s.parameters.count = 1

deep_clone_instance(c : like class_anchor ; s : DEEP_CLONE_FUNCTION_STATEMENT)
 : RUN_TIME_OFL_OBJECT is
-- exécution d'un clone profond d'objet sur 'current_object'
-- 'c' représente la classe a laquelle on veut associer la sémantique
require
 class_is_not_void : c /= void
 operator_not_void : s /= void
 is_a_copy_statement : s.feature_name = N_deep_clone
 is_correctly_set : s.parameters /= void and then s.parameters.count = 1

adapt_instance(c : like class_anchor ; s : ADAPT_PROCEDURE_STATEMENT) is
-- Adapte le format de l'objet associé a l'instruction d'adaptation
-- a celui du type de même nom présent dans l'application
-- 'c' représente la classe a laquelle on veut associer la sémantique
require
 class_is_not_void : c /= void
 operator_not_void : s /= void
 is_a_copy_statement : s.feature_name = N_adapt
 is_correctly_set : s.parameters /= void and then s.parameters.count = 1
ensure
 is_converted : s.parameters.first.is_modified implies
  not s.parameters.first.internal_is_equal(old s.parameters.first)

conforms_to_instance(c : like class_anchor ;
 s : CONFORMS_TO_FUNCTION_STATEMENT) : OFL_BOOLEAN is
-- exécution d'un test de conformance d'objet sur 'current_object'
-- 'c' représente la classe a laquelle on veut associer la sémantique
require
 class_is_not_void : c /= void
 operator_not_void : s /= void
 is_a_copy_statement : s.feature_name = N_conforms_to
 is_correctly_set : s.parameters /= void and then s.parameters.count = 1

end -- OFL_CLASS_SEMANTICS
```

### 3.1.1 La recherche de primitive

La première catégorie d'opérateurs sémantiques concerne le paramétrage de la recherche de primitive. Ces opérateurs sont les suivants :

`match` description de la sémantique des règles de surcharge et/ou de compatibilité. Comme cet opérateur a accès à l'instruction complète (signature de la primitive, paramètres effectifs, ...), tout type de surcharge peut être pris en compte.

`local_lookup` recherche de la primitive à l'intérieur de la classe qui fait appel à elle. Normalement cet opérateur appelle `match`. Il retourne une instance de `SEARCH_RESULT` qui contient en particulier la primitive candidate ; une simple modification de la classe `SEARCH_RESULT` ou une spécialisation de cette classe pourrait permettre éventuellement de retourner plusieurs primitives candidates.

**lookup** recherche de la primitive dans le(s) lien(s) d'importation. Cet opérateur fait appel à ceux qui se trouvent dans les liens d'importations et qui paramètrent la recherche de primitive en fonction du type de lien. La principale fonction de l'opérateur `lookup` est de coordonner les actions faites par ces opérateurs. Elle retourne une instance de `SEARCH_RESULT`. On notera que dans le cadre d'une gestion persistante des instances de méta-classes (`UPDATABLE_M_CLASS` et

BUILT_IN_M_CLASS), c'est ici que l'on devra décrire les règles d'utilisation de la persistance et en particulier le chargement des objets (voir section 2.6.1).

`item` Cet opérateur retourne la primitive (instance de la classe `CLASS_FEATURE`) qui a été choisie. Sa fonction est en particulier de décrire l'enchaînement des appels aux opérateurs `local_lookup` et `lookup`.

`all_features` permet de construire l'ensemble des versions de primitives accessibles aux instances de la classe.

Pour conclure sur ces opérateurs, on peut dire qu'ils permettent d'implanter tout mécanisme de liaison dynamique et tout mécanisme de surcharge. Naturellement, on pourra intégrer l'appel aux opérateurs de contrôle dynamique en fonction des vérifications qui sont nécessaires (voir section 4.1.2).

### 3.1.2 Les contrôles dynamiques

La complexité des contrôles dynamiques à décrire dépend en particulier de ceux déjà effectués de manière statique et des services utilisés (mobilité, persistance, ...), voir section 4.1.2.

Le paramétrage qui est proposé permet d'insérer au bon endroit les contrôles que le méta-programmeur veut mettre en œuvre.

`is_import_links_valid` permet de décrire les règles de validité pour l'utilisation des liens d'importation à partir de la classe. Ceci est en particulier important pour le contrôle de l'importation répétée, multiple ou simple. Il est a utiliser principalement dans deux situations : vérification de la validité de la sémantique si le compilateur en amont n'implémente pas ces contrôles (voir section 4.1.2), ou si la classe et l'objet qui font une demande de travail ont été créé avec un autre langage (interopérabilité, voir section 3.4.1).

`is_use_links_valid` mêmes objectifs que `is_import_links_valid` mais cela concerne les liens d'utilisation.

`is_type_conformance` vérification de la compatibilité de type. Celle-ci dépend beaucoup de la sémantique des liens d'importation mis en jeu et cet opérateur devra donc coordonner les routines correspondantes des classes qui descendent de `OFL_CLASS_LINK_SEMANTICS`.

`is_same_type` En fonction du lien d'importation utilisé la notion de *même type* est très différente : par exemple on pourra dire que deux types représentés par les classes `PERSONNE_V1` et `PERSONNE_V2` unis par un lien *est-une-version-de* sont identiques au sens du langage incluant ce lien de version.

`is_generics_valid` vérification de la validité de l'instanciation des paramètres génériques par rapport aux contraintes associées à la définition des paramètres génériques formels.

`is_parameters_compatible` vérification de la compatibilité entre les paramètres effectifs et formels. Cet opérateur sera en particulier appelés par les opérateurs définissant l'exécution (voir section 4.1.1). Lui-même utilise `is_generics_valid` lorsque les types à comparer sont génériques.

`is_valid_request` Cet opérateur a pour objectif de permettre d'exploiter le mécanisme de gestion de la visibilité décrit dans la section 2.5.3. Par exemple, c'est ici que l'on pourra définir les règles d'exportation d'une primitive mais comme le laisse supposer l'expressivité du langage on pourra aller beaucoup plus loin surtout dans le cas d'échanges asynchrones entre les objets (voir section 2.5.2).

### 3.1.3 L'exécution de demandes de travaux

Ces opérateurs assurent la gestion de l'exécution des instructions et des échanges entre objets. Avant toute exécution ces opérateurs devraient normalement utiliser les opérateurs permettant de déterminer les primitives et de contrôler la validité de leur utilisation.

`parameter_evaluation` description de la manière d'évaluer les paramètres effectifs lors de l'exécution d'une primitive (ordre, contrôles et actions préalables), ...

`effective_to_formal` description des actions à faire avant d'attacher les paramètres effectifs aux paramètres formels. Cet opérateur s'attend à ce que les paramètres effectifs soient déjà évalués. Un exemple d'action à faire est de réaliser une copie de certains paramètres évalués. Cet opérateur dépend fortement de la sémantique des liens ; en particulier la sémantique des liens de composition ou d'agrégation n'ont pas le même comportement.

`detach_effective` C'est l'opération inverse de la précédente ; elle doit être réalisée complètement à la fin de l'exécution de la routine, en particulier cet opérateur doit laisser *propre* la routine.

`before_request` C'est la première action faite après l'attachement des paramètres effectifs. Cet opérateur coordonnera les actions *avant* définies dans les liens. Notamment, dans le cas d'un lien d'utilisation, il faudra prévoir le chargement de l'objet attaché à la primitive dans l'objet courant et/ou le déclenchement de transaction (voir section 2.6.1) ; autre traitement à prévoir dans cet opérateur : le changement de la *routine courante*, le déclenchement de la vérification des assertions (invariant, préconditions, ou tout autre opération comme des contrôles spécifiques de cohérence, des actions de déboguage ou de traçage, ...).

`after_request` c'est la dernière action faite avant le détachement des paramètres effectifs. Les objectifs sont les mêmes que pour `before_request` ; dans le cadre d'un environnement persistant c'est par exemple ici que l'on pourra mettre en œuvre la gestion de fin de transaction ou la synchronisation avec l'image de l'objet en mémoire persistante si ce dernier a été modifié.

`execute` cet opérateur coordonne l'exécution d'une primitive dans l'objet courant. À ce titre, il utilise directement ou indirectement la plupart des opérateurs cités ci-dessus (voir section 4.1.1). Les instructions concernées sont celles qui sont décrites par des classes qui descendent de `CALL_STATEMENT`. Dans le cas d'une classe descendante de `REMOTE_CALL_STATEMENT`, cet opérateur n'est utilisé que pour les évaluations dans l'objet courant (les autres actions sont effectuées par les opérateurs `send` ou `submit`).

`execute_with_result` Cet opérateur a les mêmes objectifs que `execute` mais s'applique aux instructions retournant un résultat (`TYPED_CALL_STATEMENT`, ...).

`send` Cet opérateur est utile pour décrire toutes les actions spécifiques à l'envoi de message : évaluation du destinataire du message (`execute_with_result`), prise en compte de la nature de l'échange (synchrone ou asynchrone), gestion de plusieurs destinataires intermédiaires (voir section 4.2.2), ... Cet opérateur appelle la primitive `submit` de l'objet représentant le destinataire intermédiaire ou final.

`send_with_response` mêmes objectifs que `send` mais l'envoi de message nécessite la gestion d'un retour. On notera que les deux opérateurs ne doivent être appelés qu'a partir des opérateurs `submit` et `submit_with_response`.

`submit` C'est l'opérateur qui définit la demande d'un travail. Il est appelé uniquement par l'opérateur `submit` défini dans le lien d'instanciation (lien de type *objet-vers-classe*). Cet opérateur gère en particulier le changement d'objet courant, les contrôles de visibilité et la répartition du travail à faire entre les opérateurs `execute` ou `execute_with_result` (exécution interne au nouvel objet courant, détermination du destinataire), et les opérateurs `send` et `send_with_response` (transmission du travail restant à effectuer au destinataire intermédiaire ou final).

`submit_with_response` mêmes objectifs que `submit` mais attend une réponse à la demande de travail.

### 3.1.4 La création et la destruction d'instances

La gestion de la création d'instance et de la destruction d'instance dépendent de la sémantique des liens d'utilisation. Ces opérateurs décrivent des demandes de travail spécifiques.

`instance_creation` Description de l'allocation des instances, déclenchement de la mise à jour de l'extension de la classe correspondante et éventuellement de celles des classes qui sont liées à elle par des liens d'importation.

`generic_instance_creation` Même objectif que `instance_creation` mais s'applique aux instances de classes génériques.

`create` Cet opérateur appelle la création d'instance adéquate (`instance_creation` ou `generic_instance_creation`) après avoir fait les contrôles nécessaires, puis utilise l'opérateur d'attachement (`assign`).

`instance_destruction` Description du traitement associé à la destruction d'instances : désallocation, traitement adapté aux objets volatiles et persistants, etc.

`destroy` Coordination des traitements, associés à la destruction d'une instance, qui sont réalisés par les liens concernés.

### 3.1.5 L'attachement d'objets

Les actions faites par l'opérateur `assign` traitent de :

- la description de la validité d'une affectation. Par exemple, certains langages ne permettent pas l'attachement quand l'attribut ne se trouve pas dans l'objet courant (cas d'une expression pointée), ou l'application n'a pas le droit de modifier l'objet persistant (gestion des privilèges).
- la gestion de l'attachement d'un objet. Celui-ci se divise en plusieurs opérations :
  - évaluation de la partie droite de l'attachement,
  - détermination de la cible de l'attachement (partie gauche),
  - contrôle de type,
  - attachement de l'objet au champ correspondant dans l'objet ou dans la routine (variable locale),
  - prise en compte de la persistance,
  - toute autre opération définie par le méta-programmeur,
  - ...

Si le besoin s'en fait sentir (pour permettre une description plus aisée de la sémantique), on pourra rajouter des primitives permettant de décrire les actions propres à la classe à faire avant et après l'affectation. Elles pourront s'appeler `before_assign` et `after_assign`. On pourra aussi encapsuler les contrôles cités ci-dessus dans un opérateur `is_assign_valid`.

### 3.1.6 Les opérateurs fondamentaux

Un certain nombre d'opérateurs fondamentaux ont un comportement différent en fonction des liens mis en œuvre, c'est pourquoi il a semblé intéressant de les paramétrer en fonction du concept de classe et des liens entre classes utilisés.

Par ailleurs on notera que :

- tous ces opérateurs sont définis dans la classe racine (`OFL_ROOT`, `OFL_ROOT_M_CLASS`)
- chaque objet d'exécution contient la définition de base adéquate de ces opérateurs (voir les différents objets d'exécution dans la figure 2.7). Cette définition de base pourra dans la majorité des cas être utilisée par le méta-programmeur pour décrire la partie fastidieuse de ces opérateurs.
- la description de la sémantique (qui fait appel au besoin à l'implémentation se trouvant dans les différentes catégories d'objets d'exécution), est centralisée dans les instances de `M_CLASS` (plus précisément dans les classes descendantes de `OFL_CLASS_SEMANTICS`).

Ces opérateurs sont :

`equal_reference` description de l'égalité de référence entre l'objet passé en paramètre et l'objet courant ;

`equal_contents` description de l'égalité de contenu superficielle entre l'objet passé en paramètre et l'objet courant ;

`deep_equal_contents` description de l'égalité de contenu profonde entre l'objet passé en paramètre et l'objet courant ;

`copy_instance` description de la copie superficielle de l'objet passé en paramètre dans l'objet courant ;

`deep_copy_instance` : description de la copie profonde de l'objet passé en paramètre dans l'objet courant ;

`clone_instance` description de la duplication superficielle de l'objet passé en paramètre ;

`deep_clone_instance` : description de la duplication profonde de l'objet passé en paramètre ;

`adapt_instance` description de l'adaptation de l'objet passé en paramètre au type de même nom (utile pour les objets persistants) et

`conforms_to_instance` description de la compatibilité de type entre l'objet passé en paramètre et l'objet courant.

### 3.1.7 La gestion de l'ensemble des instances

Pour mieux définir la sémantique des liens et des classes, il est nécessaire de pouvoir paramétrer le fonctionnement de l'extension d'une classe par rapport aux liens d'importation mis en œuvre. C'est en particulier utile pour différencier les différents usages du mécanisme d'héritage.

`make_extension` description de la création de l'extension d'une classe et

`update_extension` description des actions de mise à jour de l'extension.

On notera en particulier que des appels de ces opérateurs devront être placés dans les opérateurs associés à la création et à la destruction d'instances.

## 3.2 Les méta-classes LIEN

Ces méta-classes sont représentées par les classe *Eiffel* `USE_LINK_M_CLASS`, `IMPORT_LINK_M_CLASS` et `OBJECT_TO_CLASS_LINK_M_CLASS` qui sont des classes *built-in*, c'est à dire des classes ayant une description câblée. En d'autres mots, il n'est pas possible de modifier ou d'ajouter des primitives à cette classe sans générer une nouvelle plate-forme (un nouvel *OFL*). Ces méta-classes génèrent des liens : chaque lien est représenté selon sa nature par une instance d'une des classes `USE_LINK`, `IMPORT_LINK` ou `OBJECT_TO_CLASS_LINK` qui offre les raccourcis directs appropriés pour la mise en œuvre des différents types de lien.

On notera que la sémantique d'un lien est centralisée dans une instance des classes `*_LINK_M_CLASS` (une seule sémantique pour un type de lien), mais que les clauses d'adaptations sont mémorisées dans chaque instance des classes `*_LINK` ; ces clauses sont liées à l'utilisation du lien et dépendent donc des classes ou des objets à lier (par exemple si A hérite de B et que C hérite de B, leurs clauses d'adaptations doivent pouvoir être différentes).

La sémantique des liens entre classes est définie dans la classe `OFL_CLASS_LINK_SEMANTICS`. Cette classe ne peut donner lieu directement à la création d'un nouveau type de lien mais permet de centraliser les propriétés communes aux liens d'importation et aux liens d'utilisation.

### 3.2.1 Description des propriétés communes aux *liens entre classes*

Nos remarques lors de la description des opérateurs définis dans `OFL_CLASS_SEMANTICS` montrent que les opérateurs qui sont décrits ci-dessous sont appelés par ceux de `OFL_CLASS_SEMANTICS`.

```
Deferred class OFL_CLASS_LINK_SEMANTICS

feature -- Gestion de l'exécution des routines

 is_feature_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que la primitive designée par 'r' et appelée à travers
 -- le lien 'l' remplit les conditions d'une exécution
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  is_a_call : r.call_action /= void
```

```
 before_feature(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- designée par 'r' et appelée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  is_a_call : r.call_action /= void

 after_feature(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
 -- designée par 'r' et appelée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  is_a_call : r.call_action /= void

-- Gestion de l'envoi de message

 is_send_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que l'envoi de message 'r' envoyé à travers
 -- le lien 'l' remplit les conditions d'une exécution
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  is_a_call : r.remote_call_action /= void

 before_send(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de l'envoi
 -- du message 'r' à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  is_a_call : r.remote_call_action /= void

 after_send(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de l'envoi
 -- de message 'r' à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  is_a_call : r.remote_call_action /= void

-- Gestion spécifique des actions concernant contrôle des paramètres

 is_parameters_control_valid(l : like link_anchor ; r : like request) :
  BOOLEAN is
 -- Est-ce que les paramètres effectifs sont compatibles avec
 -- les paramètres formels pour la demande de travail correspondant
 -- a 'r' à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

 before_parameters_control(l : like link_anchor ; r : like request) is
 -- Partie de code a exécuter avant la vérification de compatibilité
 -- entre les paramètres formels et les paramètres effectifs de la
 -- demande de travail correspondant a 'r' à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

 after_parameters_control(l : like link_anchor ; r : like request) is
 -- Partie de code a exécuter après la vérification de compatibilité
 -- entre les paramètres formels et les paramètres effectifs de la
 -- demande de travail correspondant a 'r' à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
```

```
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

-- Gestion spécifique des actions concernant l'evaluation des paramètres

 is_parameter_evaluation_valid(l : like link_anchor ; r : like request) :
  BOOLEAN is
 -- Est-ce que l'evaluation des paramètres effectifs est possible
 -- pour la demande de travail correspondant a 'r' à travers le
 -- lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

 before_parameter_evaluation(l : like link_anchor ; r : like request) is
 -- Partie de code a exécuter avant l'evaluation
 -- des paramètres effectifs de la demande de travail
 -- correspondant a 'r' à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

 after_parameter_evaluation(l : like link_anchor ; r : like request) is
 -- Partie de code a exécuter après l'evaluation
 -- des paramètres effectifs de la demande de travail
 -- correspondant a 'r' à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

-- Gestion spécifique des actions sur les paramètres une fois évalués

 is_effective_to_formal_valid(l : like link_anchor ; r : like request) :
  BOOLEAN is
 -- Est-ce que l'association des paramètres effectifs aux paramètres
 -- formels de la primitive correspondant a 'r' et appelée à travers
 -- le lien 'l' est valide
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

 before_effective_to_formal(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée avant
 -- l'association des paramètres effectifs aux paramètres
 -- formels de la primitive designée par 'r' et appelée
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

 after_effective_to_formal(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée après
 -- l'association des paramètres effectifs aux paramètres
 -- formels de la primitive designée par 'r' et appelée
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  contents_is_set : r /= void and then r.contents /= void
  candidate_is_set : r.candidate_feature /= void
  request_is_a_call : r.call_action /= void
  candidate_is_a_routine : r.contents.to_routine /= void

-- Gestion de l'affectation de primitives modifiables
```

```
 is_assign_valid(l : like link_anchor ; r : STATEMENT_CONTEXT) : BOOLEAN is
 -- Est-ce que l'opération d'attachement designée par 'r',
 -- et demandée à travers le lien 'l' remplit les conditions
 -- d'un attachement
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_an_assign : r.to_assign /= void

 before_assign(l : like link_anchor ; r : STATEMENT_CONTEXT) is
 -- Partie de code qui doit être exécutée avant l'opération
 -- d'attachement designée par 'r', et demandée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_an_assign : r.to_assign /= void

 after_assign(l : like link_anchor ; r : STATEMENT_CONTEXT) is
 -- Partie de code qui doit être exécutée après l'opération
 -- d'attachement designée par 'r', et demandée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_an_assign : r.to_assign /= void

-- Gestion de la création

 is_creation_valid(l : like link_anchor ; r : SEARCH_RESULT) : BOOLEAN is
 -- Est-ce que l'opération de creation designée par 'r',
 -- demandée à travers le lien 'l' remplit les conditions
 -- d'une creation
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_a_creation : r.to_creation /= void

 before_creation(l : like link_anchor ; r : SEARCH_RESULT) is
 -- Partie de code qui doit être exécutée avant l'opération de creation
 -- designée par 'r', et demandée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_a_creation : r.to_creation /= void

 after_creation(l : like link_anchor ; r : SEARCH_RESULT) is
 -- Partie de code qui doit être exécutée avant l'opération de creation
 -- designée par 'r', et demandée à travers le lien 'l'
  require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_a_creation : r.to_creation /= void

-- Gestion de la destruction

 is_destruction_valid(l : like link_anchor ; r : SEARCH_RESULT) : BOOLEAN is
 -- Est-ce que l'opération de destruction designée par 'r',
 -- demandée à travers le lien 'l' remplit les conditions
 -- d'une destruction
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_a_destruction : r.to_destruction /= void

 before_destruction(l : like link_anchor ; r : SEARCH_RESULT) is
 -- Partie de code qui doit être exécutée avant l'opération de destruction
 -- designée par 'r', et demandée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_a_destruction : r.to_destruction /= void

 after_destruction(l : like link_anchor ; r : SEARCH_RESULT) is
 -- Partie de code qui doit être exécutée avant l'opération de destruction
 -- designée par 'r', et demandée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void
  is_a_destruction : r.to_destruction /= void

-- Gestion d'une primitive particulière : Égalité de référence
```

```
 is_equal_reference_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que la primitive traitant l'égalité de référence
 -- et appelée par 'r' à travers le lien 'l' remplit les conditions
 -- d'une exécution
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_equal_reference_call : r.contents.to_equal_reference /= void

 before_equal_reference(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- traitant l'égalité par référence et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_equal_reference_call : r.contents.to_equal_reference /= void

 after_equal_reference(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
 -- traitant l'égalité par référence et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_equal_reference_call : r.contents.to_equal_reference /= void

-- Gestion d'une primitive particulière : Égalité superficielle de contenu

 is_equal_contents_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que la primitive traitant l'égalité de contenu
 -- et appelée par 'r' à travers le lien 'l' remplit les conditions
 -- d'une exécution
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_equal_contents_call : r.contents.to_equal_contents /= void

 before_equal_contents(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- traitant l'égalité par contenu et appelée par 't'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_equal_contents_call : r.contents.to_equal_contents /= void

 after_equal_contents(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
 -- traitant l'égalité par contenu et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_equal_contents_call : r.contents.to_equal_contents /= void

-- Gestion d'une primitive particulière : Égalité profonde de contenu

 is_deep_equal_contents_valid(l : like link_anchor ; r : like request)
   BOOLEAN is
 -- Est-ce que la primitive traitant l'égalité profonde de contenu
 -- et appelée par 'r' à travers le lien 'l' remplit les conditions
 -- d'une exécution
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_equal_call : r.contents.to_deep_equal_contents /= void

 before_deep_equal_contents(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- traitant l'égalité profonde par contenu et appelée par 't'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_equal_call : r.contents.to_deep_equal_contents /= void

 after_deep_equal_contents(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
```

```
 -- traitant l'égalité profonde par contenu et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_equal_call : r.contents.to_deep_equal_contents /= void

-- Gestion d'une primitive particulière : clonage superficiel d'un objet

 is_clone_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que la primitive traitant le clonage d'objets
 -- et appelée par 'r' à travers le lien 'l' remplit les conditions
 -- d'une exécution
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_clone_call : r.contents.to_clone /= void

 before_clone(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- traitant le clonage d'objets et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_clone_call : r.contents.to_clone /= void

 after_clone(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
 -- traitant le clonage des objets et appelée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_clone_call : r.contents.to_clone /= void

-- Gestion d'une primitive particulière : clonage profond d'un objet

 is_deep_clone_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que la primitive traitant le clonage profond d'objets
 -- et appelée par 'r' à travers le lien 'l' remplit les conditions
 -- d'une exécution
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_clone_call : r.contents.to_deep_clone /= void

 before_deep_clone(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- traitant le clonage profond d'objets et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_clone_call : r.contents.to_deep_clone /= void

 after_deep_clone(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
 -- traitant le clonage profond des objets et appelée à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_clone_call : r.contents.to_deep_clone /= void

-- Gestion d'une primitive particulière : copie superficielle du contenu d'un
-- objet

 is_copy_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que la primitive traitant la copie du contenu d'objets
 -- et appelée par 'r' à travers le lien 'l' remplit les conditions
 -- d'une exécution
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_copy_call : r.contents.to_copy /= void

 before_copy(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- traitant la copie du contenu d'objets et appelée par 't'
 -- à travers le lien 'l'
 require
```

```
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_copy_call : r.contents.to_copy /= void

 after_copy(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
 -- traitant la copie du contenu et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_copy_call : r.contents.to_copy /= void

-- Gestion d'une primitive particulière : copie profonde du contenu d'un objet

 is_deep_copy_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que la primitive traitant la copie profonde du contenu d'objets
 -- et appelée par 'r' à travers le lien 'l' remplit les conditions
 -- d'une exécution
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_copy_call : r.contents.to_deep_copy /= void

 before_deep_copy(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- traitant la copie profonde du contenu d'objets et appelée par 't'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_copy_call : r.contents.to_deep_copy /= void

 after_deep_copy(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
 -- traitant la copie profonde du contenu et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_copy_call : r.contents.to_deep_copy /= void

-- Conformance et adaptation de l'objet

 is_adapt_instance_valid(l : like link_anchor ; r : like request) : BOOLEAN is
 -- Est-ce que la primitive traitant l'adaptation d'objet a un type
 -- et appelée par 'r' à travers le lien 'l' remplit les conditions
 -- d'une exécution
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_copy_call : r.contents.to_adapt /= void

 before_adapt_instance(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée au début de la primitive
 -- traitant l'adaptation d'objet a un type et appelée par 't'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_copy_call : r.contents.to_adapt /= void

 after_adapt_instance(l : like link_anchor ; r : like request) is
 -- Partie de code qui doit être exécutée a la fin de la primitive
 -- traitant l'adaptation d'objet a un type et appelée par 'r'
 -- à travers le lien 'l'
 require
  link_is_not_void : l /= void
  candidate_is_set : r /= void and then r.contents /= void
  is_a_deep_copy_call : r.contents.to_adapt /= void

end -- class OFL_CLASS_LINK_SEMANTICS
```

L'organisation de la gestion des opérateurs mentionnés ci-dessus est définie dans la classe `M_CLASS` (voir ci-dessus l'énumération des services offerts pour une classe). Les principales situations prises en compte au niveau des méta-classes `LIEN` sont naturellement très corrélées avec celles de la méta-classe `CLASSE` ; elles traitent des aspects suivants :

- contrôle sémantiques,

– exécution de demandes de travaux,
– création et destruction d'instances,
– attachement d'objets et
– opérateurs fondamentaux.

Le méta-programmeur est libre d'appeler ces primitives de n'importe quel opérateur de `OFL_CLASS_SEMANTICS` mais d'une manière générale il est préférable de respecter les correspondances mises en évidences ci-dessous.

#### Contrôle sémantiques

Les contrôles sémantiques faits concernent seulement le contrôle des paramètres (voir section 3.1.2). `is_parameters_compatible` de OFL_CLASS_SEMANTICS correspond ici à :

`is_parameters_control_valid` contrôle de validité de l'utilisation de l'opérateur ;

`before_parameters_control` une action à faire avant la réalisation du travail (par exemple le test de la validité des paramètres) et

`after_parameters_control` une action à faire après.

#### Exécution de demandes de travaux

Les contrôles sémantiques faits concernent certains des contrôles cités dans la section 3.1.3. Pour chacun on a un contrôle de validité de l'utilisation de l'opérateur, une action à faire avant la réalisation du travail (par exemple l'attachement des paramètres effectifs aux paramètres formels) et une action à faire après.

`parameters_evaluation` de `OFL_CLASS_SEMANTICS` correspond ici à `is_parameter_evaluation_valid`, `before_parameter_evaluation` et `after_parameter_evaluation` ;

`effective_to_formal` de `OFL_CLASS_SEMANTICS` correspond ici à `is_effective_to_formal_valid`, `before_effective_to_formal` et `after_effective_to_formal` et

`detach_effective` de `OFL_CLASS_SEMANTICS` correspond ici à `is_detach_effective_valid`, `before_detach_effective` t `after_detach_effective`.

De même certains opérateurs de `OFL_CLASS_SEMANTICS` ne correspondent qu'à un opérateur :

`is_valid_request` de `OFL_CLASS_SEMANTICS` correspond ici à `is_feature_valid` ;

`before_request` de `OFL_CLASS_SEMANTICS` correspond ici à `before_feature` et

`after_request` de `OFL_CLASS_SEMANTICS` correspond ici à `after_feature`.

#### Création et destruction d'instances

Les créations et destructions d'instances concernent certains opérateurs cités dans la section 3.1.4. Pour chacun, on a un contrôle de validité de l'utilisation de l'opérateur, une action à faire avant la réalisation du travail (par exemple l'attachement des paramètres effectifs aux paramètres formels) et une action à faire après.

`create` de `OFL_CLASS_SEMANTICS` correspond ici à `is_creation_valid`, srcbefore_creation et `after_creation` et

`destroy` de `OFL_CLASS_SEMANTICS` correspond ici à `is_destruction_valid`, `before_destruction` et `after_destruction`.

#### Attachement d'objets

L'attachement d'objets présenté ici est lié directement à l'opérateur `assign` (voir section 3.1.5). Lui correspond un contrôle de validité de l'utilisation de l'opérateur (`is_assign_valid`), une action à faire avant la réalisation du travail (`before_assign`) et une action à faire après (`after_assign`).

**Opérateurs fondamentaux**

On trouve dans chaque lien entre classes, pour chaque opérateur fondamental cité dans la section 3.1.6, un contrôle de validité de l'utilisation de l'opérateur, une action à faire avant la réalisation du travail (par exemple la duplication de l'objet), et une action à faire après.

### 3.2.2 Description de la méta-classe *lien d'utilisation*

Au stade de notre étude, le lien d'utilisation ne définit pas d'opérateur supplémentaire par rapport à la description d'un lien entre classes. et doncla classe `OFL_USE_LINK_SEMANTICS` contient les mêmes opérateurs que `OFL_CLASS_LINK_SEMANTICS`.

Il serait intéressant d'étudier si la place des opérateurs concernant l'exécution de primitives (`before_feature`, ...), et l'envoi de message (`before_send`, ...), ne seraient pas plutôt dans la classe `OFL_USE_LINK_SEMANTICS`. L'étude des liens d'importation comme les liens de versions devrait nous aider à décider de l'emplacement de ces opérateurs.

### 3.2.3 Description de la méta-classe *lien d'importation*

La sémantique d'un lien d'importation est décrit par la classe `OFL_IMPORT_LINK_SEMANTICS`. En plus des opérateurs définis dans la classe `OFL_CLASS_LINK_SEMANTICS`, celle-ci introduit la possibilité de définir des filtres.

Parmi les différentes possibilités offertes pour filtrer ce qui passe par un lien on retrouve toutes les clauses d'adaptation présentes dans *Eiffel* :

- renommage,
- redéfinition,
- abstraction,

Plus une clause permettant de cacher des primitives à la source du lien (généralisation de la clause d'exportation propre au lien de clientèle), ce qui permet de modéliser des liens moins spécifiques que les liens d'héritage ; par exemple des liens de versions où une primitive `P` présente dans la version `V1` n'est pas forcément présente dans la version suivante `V2`.

```
Deferred class OFL_IMPORT_LINK_SEMANTICS

feature -- Gestion de la recherche d'une primitive

 is_match(l : like link_anchor ; s : like context ; f : like filter_anchor) :
  BOOLEAN is
 -- Retourne "True" si la primitive 'f' correspond a la primitive
 -- appelée dans l'instruction 's' à travers le lien 'l'
 -- (définition de la sémantique de la surcharge pour le lien 'l')
 require
  link_is_not_void : l /= void
  statement_exists : s /= void
  feature_is_set : f /= void

 lookup(l : like link_anchor ; s : like context) : SEARCH_RESULT is
 -- Recherche dans la cible du lien 'l', la primitive qui
 -- correspond a l'appel fait dans l'instruction 's'
 -- (définition de la sémantique de la recherche de primitive
 -- dans un lien d'importation)
 require
  link_is_not_void : l /= void
  statement_exists : s /= void
 ensure
  Result /= void implies is_match(l, s, Result.candidate)

 lookdown(l : like link_anchor ; s : like context) : SEARCH_RESULT is
 -- Recherche dans la source du lien 'l', la primitive qui
 -- correspond a l'appel fait dans l'instruction 's'
 -- (définition de la sémantique du polymorphisme dans un lien
 -- d'importation)
 require
  link_is_not_void : l /= void
  statement_exists : s /= void
 ensure then
  Result /= void implies is_match(l, s, Result.candidate)
```

```
-- Description de la mise à jour de l'extension

 update_extension_of_from_c(l : like link_anchor) is
 -- Déclenchement de la mise à jour de l'extension
 -- de la classe 'l'.'from_c' à partir de l'extension
 -- de la classe 'l'.'to_c'
 require
  link_is_not_void : l /= void

 update_extension_of_to_c(l : like link_anchor) is
 -- Déclenchement de la mise à jour de l'extension
 -- de la classe 'l'.'to_c' à partir de l'extension
 -- de la classe 'l'.'from_c'
 require
  link_is_not_void : l /= void

 update_extension(l : like link_anchor ; to_update : M_CLASS) is
 -- Mise à jour de l'extension de la classe 'to_update'
 -- à partir de 'l'.'from_c' et de 'l'.'to_c' quand c'est utile
 require
  link_is_not_void : l /= void
  class_is_not_void : to_update /= void

end -- class OFL_IMPORT_LINK_SEMANTICS
```

Ces opérateurs traitent des aspects suivants (voir section 3.1.7), ils sont appelés par l'opérateur `lookup`, `make_extension` et `update_extension`:

- la recherche de la primitive à travers un lien (peut inclure des tests de compatibilité de type et la gestion de la surcharge) ;
- la construction/la mise à jour de l'ensemble des occurrences associées à une classe.

On notera que l'on propose au méta-programmeur de définir une sémantique différente pour faire une recherche avec la vision d'une classe descendante à celle qui a créée l'objet ou d'une classe parente. On pourrait aussi définir une primitive qui coordonne les deux et prend pour paramètre la vision que l'on veut avoir, c'est à dire le type de la primitive à travers laquelle on accède à l'objet.

### 3.2.4 Description de la méta-classe *lien objet vers classe*

La sémantique du lien qui unit un objet à une classe (le cas qui nous préoccupe en particulier est le lien d'instanciation) est modélisé par la classe `OFL_OBJECT_TO_CLASS_LINK_SEMANTICS`.

```
Deferred class OFL_OBJECT_TO_CLASS_LINK_SEMANTICS

feature -- Gestion de la soumission d'une instruction

 submit(l : like link_anchor ; s : like request) is
 -- Soumission d'une instruction 's' a travers le lien 'l'
 require
  link_not_void : l /= void
  request_not_void : s /= void
  request_set : s.action_feature_name /= void and s.parameters /= void
  exists : s.call_action /= void implies l.to_c.has(s)

 submit_with_response(l : like link_anchor ; s : like request) :
  RUN_TIME_OFL_OBJECT is
 -- Retourne le résultat de l'envoi du message 's' a travers le lien 'l'
 require
  link_not_void : l /= void
  request_not_void : s /= void
  request_set : s.action_feature_name /= void and s.parameters /= void
  exists : s.call_action /= void implies l.to_c.has(s)

-- Gestion de l'affectation

 assign(l : like link_anchor ; s : ASSIGNMENT_STATEMENT) is
 -- Réalise l'affectation modélisée par 's' a travers le lien 'l'
 require
  link_not_void : l /= void
  statement_not_void : s /= void
  there_is_a_left_part : s.target /= void
  there_is_a_rigth_part : s.source /= void
 ensure
```

```
 has_new_value : current_object.fields(target) = obj

-- Exécution de primitives spécifiques

 equal_reference(l : like link_anchor ; s : EQUAL_REFERENCE_FUNCTION_STATEMENT)
  : OFL_BOOLEAN is
 -- Exécution de l'égalité de reference sur 'current_object'
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  link_is_not_void : l /= void
  operator_not_void : s /= void
  is_an_equality_statement : s.feature_name = N_equal_reference
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 equal_contents(l : like link_anchor ; s : EQUAL_CONTENTS_FUNCTION_STATEMENT) :
  OFL_BOOLEAN is
 -- exécution de l'égalité de contenu sur 'current_object'
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  link_is_not_void : l /= void
  operator_not_void : s /= void
  is_an_equality_statement : s.feature_name = N_equal_contents
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 deep_equal_contents(l : like link_anchor ; s : DEEP_EQUAL_FUNCTION_STATEMENT)
  : OFL_BOOLEAN is
 -- exécution de l'égalité profonde de contenu sur 'current_object'
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  link_is_not_void : l /= void
  operator_not_void : s /= void
  is_an_equality_statement : s.feature_name = N_deep_equal_contents
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 copy_instance(l : like link_anchor ; s : COPY_PROCEDURE_STATEMENT) is
 -- exécution de la copie de contenu sur 'current_object'
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  link_is_not_void : l /= void
  operator_not_void : s /= void
  is_a_copy_statement : s.feature_name = N_copy
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 deep_copy_instance(l : like link_anchor ; s : DEEP_COPY_PROCEDURE_STATEMENT)
  is
 -- exécution de la copie profonde de contenu sur 'current_object'
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  link_is_not_void : l /= void
  operator_not_void : s /= void
  is_a_copy_statement : s.feature_name = N_deep_copy
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 clone_instance(l : like link_anchor ; s : CLONE_FUNCTION_STATEMENT) :
  RUN_TIME_OFL_OBJECT is
 -- exécution d'un clonage d'objet sur 'current_object'
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  link_is_not_void : l /= void
  operator_not_void : s /= void
  is_a_copy_statement : s.feature_name = N_clone
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 deep_clone_instance(l : like link_anchor ; s : DEEP_CLONE_FUNCTION_STATEMENT)
  : RUN_TIME_OFL_OBJECT is
 -- exécution d'un clone profond d'objet sur 'current_object'
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  link_is_not_void : l /= void
  operator_not_void : s /= void
  is_a_copy_statement : s.feature_name = N_deep_clone
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 conforms_to_instance(l : like link_anchor ;
  s : CONFORMS_TO_FUNCTION_STATEMENT) : OFL_BOOLEAN is
 -- exécution d'un test de conformance d'objet sur 'current_object'
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  link_is_not_void : l /= void
  operator_not_void : s /= void
```

```
  is_a_copy_statement : s.feature_name = N_conforms_to
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1

 adapt_instance(l : like link_anchor ; s : ADAPT_PROCEDURE_STATEMENT) is
 -- Adapte le format de l'objet associé a l'instruction d'adaptation
 -- a celui du type de même nom présent dans l'application
 -- 'l' représente le lien auquel on veut associer la sémantique
 require
  class_is_not_void : l /= void
  operator_not_void : s /= void
  is_a_copy_statement : s.feature_name = N_adapt
  is_correctly_set : s.parameters /= void and then s.parameters.count = 1
 ensure
  is_converted :
   not s.parameters.first.internal_is_equal(old s.parameters.first) and
   s.parameters.first.is_modified

end -- class OFL_OBJECT_TO_CLASS_LINK_SEMANTICS
```

Normalement (sauf pour la création et peut être la destruction), les instructions dites flexibles évoque l'opérateur `submit` ou `assign` ou l'opérateur correspondant (CLONE_FUNCTION_STATEMENT appelle `clone_instance`), sur l'objet courant, de même les opérateurs `send` et `send_with_response` de OFL_CLASS_SEMANTICS appellent l'opérateur `submit`.

## 3.3 Sémantique d'un langage

La description du comportement des objets dans les différentes situations indiquées ci-dessus est regroupée dans deux hiérarchies de classes dont les racines sont :

- la classe OFL_CLASS_SEMANTICS et
- la classe OFL_LINK_SEMANTICS.

L'implantation de ces services pour un langage donné est réalisée dans des classes descendantes.

La sémantique dynamique d'un langage est définie par :

- une instance de chaque classe (descendante de la classe OFL_CLASS_SEMANTICS), qui décrit une notion de classe pour le langage et
- par une instance de chaque classe (descendante de la classe OFL_LINK_SEMANTICS), qui décrit un type de lien du langage.

La classe OFL_LANGUAGE regroupe l'accès à l'ensemble de ces informations (voir figure 3.3). Chaque langage est représenté par un descendant de la classe OFL_LANGUAGE qui regroupe l'ensemble des instances des descendants de OFL_CLASS_SEMANTICS et OFL_LINK_SEMANTICS associées à ce langage.

Par exemple la classe EIFFEL3_LANGUAGE contiendra une instance de la classe EIFFEL3_OFL_LANGUAGE_SEMANTICS (dans OFL_CLASS_SEMANTICS) et une instance de chacune des classes EIFFEL3_INHERIT_FROM_LINK, EIFFEL3_IS_PRIVATE_CLIENT_LINK ainsi que EIFFEL3_IS_SHARED_CLIENT_LINK. Ces quatre classes décrivent respectivement la sémantique d'une classe en *Eiffel* et la sémantique *Eiffel* des liens de clientèles (expansées ou non) et du lien d'héritage.

```
Deferred class OFL_LANGUAGE

feature {OFL_FLEXIBILITY} -- Utilisation des choix standards

 is_free_implementation : BOOLEAN
 -- est-ce que le méta-programmeur définit toute la sémantique librement

 is_default_handling : BOOLEAN is
 -- Est-ce que l'on applique le traitement par défaut
 -- s'il existe ?
 ensure
  consistent : not is_free_implementation

 set_default_handling is
 -- Active le traitement par défaut
 ensure
  is_activated : is_default_handling and not is_free_implementation

 unset_default_handling is
 -- Désactive le traitement par défaut
```

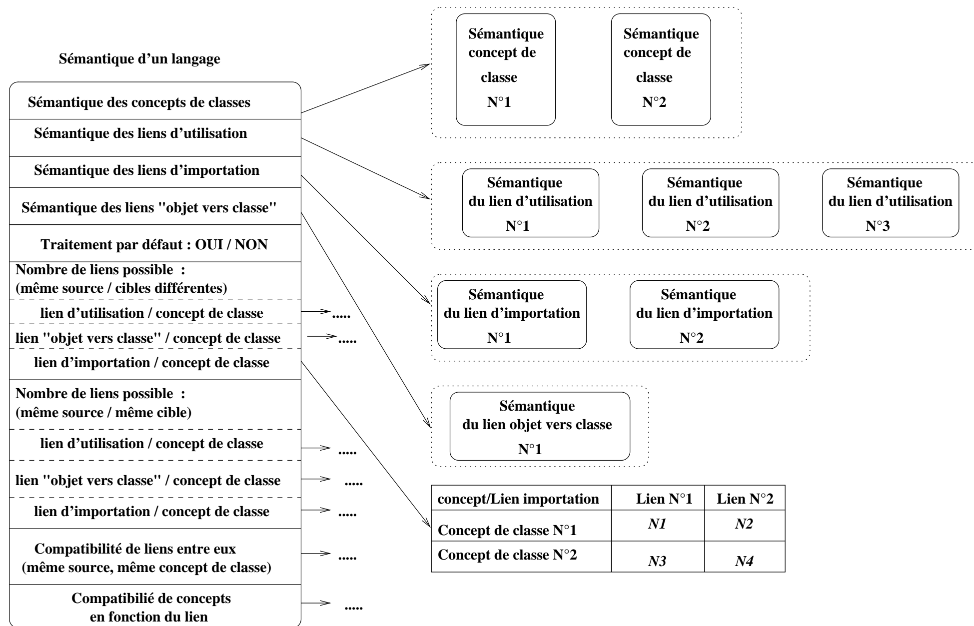


FIG. 3.3 – *Intégration de la sémantique : la classe OFL_LANGUAGE*

```
 ensure
  is_not_activated : not is_default_handling and is_free_implementation

-- Compatibilité entre langages

 equivalent_class_semantics(l : OFL_LANGUAGE ; s : OFL_NAME) : like s is
 -- Retourne le nom du concept de classe equivalent dans le langage 'l'
 -- L'équivalence est plus ou moins "lâche" en fonction de
 -- la distance entre les langages
 require
  language_is_set : l /= void
  concept_name_is_set : s /= void
  concept_name_is_consistent : has_class_semantics(s)

 equivalent_link_semantics(l : OFL_LANGUAGE ; s : OFL_NAME) : like s is
 -- Retourne le nom du concept de lien equivalent dans le langage 'l'
 -- L'équivalence est plus ou moins "lâche" en fonction de
 -- la distance entre les langages
 require
  language_is_set : l /= void
  concept_name_is_set : s /= void
  concept_name_is_consistent : has_link_semantics(s)

-- Contrôle des liens d'importation

 nb_same_source_import_link(sc : OFL_NAME ; s : OFL_NAME) : INTEGER is
 -- Retourne le nombre de liens d'importation de type 's'
 -- qu'il est possible d'initier a partir d'une même classe
 -- source correspondant au concept 'sc'
 require
  concept_names_are_set : s /= void and sc /= void
 ensure
  is_consistent : not has_import_link_semantics(s) implies Result = 0
 -- Result = -1 implies pas de limitations

 nb_same_import_link(sc : OFL_NAME ; s : OFL_NAME) : INTEGER is
 -- Retourne le nombre de liens d'importation de type 's'
 -- qu'il est possible d'initier a partir d'une même classe source
 -- correspondant au concept 'sc' et vers un même type cible
 require
  concept_namesare_set : s /= void and sc /= void
 ensure
  is_consistent : not has_import_link_semantics(s) implies Result = 0
  Result = 0 implies nb_same_source_import_link(sc, s) <= 1
```

```
 -- Result = -1 implies pas de limitations

 is_compatible_import_links(sc : OFL_NAME ; s : ARRAY [OFL_NAME]) : BOOLEAN is
 -- Est-ce que les types de lien d'importation définis dans 's'
 -- peuvent être utilises simultanément a partir d'une même
 -- classe source correspondant au concept 'sc'
 -- Réglementation de la composition de liens d'importation
 require
  concept_names_are_set : s /= void and sc /= void and then s.count > 1

-- Contrôle des liens d'utilisation

 is_compatible_use_links(sc : OFL_NAME ; s : ARRAY [OFL_NAME]) : BOOLEAN is
 -- Est-ce que les types de lien d'utilisation définis dans 's'
 -- peuvent être utilises simultanément a partir d'une même
 -- entité typée définie dans une classe correspondant au
 -- concept de classe 'sc'
 -- Réglementation de la composition de liens d'utilisation
 require
  concept_names_are_set : s /= void and sc /= void and then s.count > 1

-- Contrôle des liens d'objet vers classe

 nb_same_source_object_to_class_link(s : OFL_NAME) : INTEGER is
 -- Retourne le nombre de liens "objet vers classe" de type 's'
 -- qu'il est possible d'initier a partir d'un même objet
 require
  concept_names_are_set : s /= void
 ensure
  is_consistent : not has_object_to_class_link_semantics(s) implies Result = 0
 -- Result = -1 implies pas de limitations

 nb_same_object_to_class_link(s : OFL_NAME) : INTEGER is
 -- Retourne le nombre de liens "objet vers classe" de type 's'
 -- qu'il est possible d'initier a partir d'un même objet
 -- et vers un même type cible
 require
  concept_names_are_set : s /= void
 ensure
  is_consistent : not has_object_to_class_link_semantics(s) implies Result = 0
  Result = 0 implies nb_same_source_object_to_class_link(s) <= 1
 -- Result = -1 implies pas de limitations

 is_compatible_object_to_class_links(s : ARRAY [OFL_NAME]) : BOOLEAN is
 -- Est-ce que les types de lien "objet vers classe" définis dans 's'
 -- peuvent être utilises simultanément a partir d'un même objet
 -- Réglementation de la composition de liens "objet vers classe"
 require
  concept_names_are_set : s /= void and then s.count > 1

-- Contrôle des concepts de classe

 is_compatible_class(s : OFL_NAME ; t : OFL_NAME ; l : OFL_NAME) : BOOLEAN is
 -- Est-ce que le lien de type 'l' d'une classe de concept 's'
 -- vers une classe de concept 't' est possible ?
 -- Réglementation de la composition des concepts de classe
 require
  concept_names_are_set : s /= void and t /= void and then l /= void
  link_semantics_exist : has_class_link_semantics(l)
  source_class_semantics_exists : has_class_semantics(s)
  target_class_semantics_exists : has_class_semantics(t)

-- Sémantique d'une classe pour le langage

 OFL_class_semantics : OFL_TABLE [OFL_CLASS_SEMANTICS] is
 -- Table des différentes sémantiques de classe
 -- Chaque sémantique correspond à un genre de classe
 -- (on peut éventuellement avoir dans un langage plusieurs
 -- concepts de classe)

 class_semantics(n : OFL_NAME) : OFL_CLASS_SEMANTICS is
 -- Accès a l'instance décrivant la sémantique d'une classe
 -- le genre de classe est déterminé par 'n'

 has_class_semantics, is_class_supported(n : OFL_NAME) : BOOLEAN is
 -- est-ce qu'il existe une sémantique pour le type de classe 'n'
 -- dans le langage

-- Sémantique de tous les liens pour le langage
```

```
 OFL_link_semantics : OFL_TABLE [OFL_LINK_SEMANTICS] is
 -- Table des différentes sémantiques de lien
 -- Chaque sémantique correspond à un type de lien

 link_semantics(n : OFL_NAME) : OFL_LINK_SEMANTICS is
 -- Sémantique associée au type de lien 'n'
 require
  link_semantics_exists : has_link_semantics(n)

 has_link_semantics, is_link_supported(n : OFL_NAME) : BOOLEAN is
 -- est-ce qu'il existe dans le langage un type de lien 'n'

-- Sémantique de tous les liens entre classes pour le langage

 OFL_class_link_semantics : OFL_TABLE [OFL_CLASS_LINK_SEMANTICS] is
 -- Table des différentes sémantiques de lien entre classes
 -- Chaque sémantique correspond à un type de lien

 class_link_semantics(n : OFL_NAME) : OFL_CLASS_LINK_SEMANTICS is
 -- Sémantique associée au type de lien entre classes 'n'
 require
  link_semantics_exists : has_class_link_semantics(n)

 has_class_link_semantics, is_class_link_supported(n : OFL_NAME) : BOOLEAN is
 -- est-ce qu'il existe dans le langage un type de lien entre classes 'n'

-- Sémantique des liens d'importation pour le langage

 OFL_import_link_semantics : OFL_TABLE [like import_link_semantics] is
 -- Table des différentes sémantiques de lien d'importation
 -- Chaque sémantique correspond à un type de lien d'importation

 import_link_semantics(n : OFL_NAME) : OFL_IMPORT_LINK_SEMANTICS is
 -- Sémantique associée au type de lien d'importation 'n'
 require
  link_semantics_exists : has_import_link_semantics(n)

 has_import_link_semantics, is_import_link_supported(n : OFL_NAME) : BOOLEAN is
 -- est-ce qu'il existe dans le langage un type de lien
 -- d'importation 'n'

-- Sémantique des liens d'utilisation pour le langage

 OFL_use_link_semantics : OFL_TABLE [like use_link_semantics] is
 -- Table des différentes sémantiques de lien d'utilisation
 -- Chaque sémantique correspond à un type de lien d'utilisation

 use_link_semantics(n : OFL_NAME) : OFL_USE_LINK_SEMANTICS is
 -- Sémantique associée au type de lien d'utilisation 'n'
 require
  link_semantics_exists : has_use_link_semantics(n)

 has_use_link_semantics, is_use_link_supported(n : OFL_NAME) : BOOLEAN is
 -- est-ce qu'il existe dans le langage un type de lien
 -- d'utilisation 'n'

-- Sémantique des liens objet->classe pour le langage

 OFL_object_to_class_link_semantics :
  OFL_TABLE [like object_to_class_link_semantics] is
 -- Table des différentes sémantiques de lien entre un objet et une classe
 -- Chaque sémantique correspond à un type de lien d'utilisation

 object_to_class_link_semantics(n : OFL_NAME) :
  OFL_OBJECT_TO_CLASS_LINK_SEMANTICS is
 -- Sémantique associée au type de lien "objet vers classe" 'n'
 require
  link_semantics_exists : has_object_to_class_link_semantics(n)

 has_object_to_class_link_semantics,
  is_object_to_class_link_supported(n : OFL_NAME) : BOOLEAN is
 -- est-ce qu'il existe dans le langage un type de lien
 -- "objet vers classe" 'n'

-- Gestion des erreurs sémantiques

 OFL_semantics_errors : SEMANTICS_ERRORS is
 -- Table des messages des différentes erreurs sémantiques du langage

 semantics_error(n : STRING) : MESSAGE_TEXT is
```

```
 -- texte du message d'erreur Sémantique ayant 'n' pour nom
 require
  semantics_error_exists : has_semantics_error(n)

 has_semantics_error, is_error_supported(n : STRING) : BOOLEAN is
 -- est-ce qu'il existe une erreur sémantique de nom 'n'
 -- dans le langage

 set_semantics_error(key : STRING) is
 -- mémorise l'erreur 'key'
 require
  is_present : key /= void and then has_semantics_error(key)

 reset_semantics_error is
 -- annule la dernière erreur sémantique

 is_semantics_error : BOOLEAN is
 -- est-ce qu'une erreur sémantique est mémorisée

-- classes de lien

 meta_links : OFL_TABLE [LINK_M_CLASS] is
 -- description des types de lien

 meta_link(n : OFL_NAME) : LINK_M_CLASS is
 -- Retourne la classe qui décrit le type du lien 'n'
 require
  name_is_set : n /= void
 ensure
  meta_links_updated : meta_links.has(n)

end -- class OFL_LANGUAGE
```

Parmi les opérateurs cités ci-dessous peu sont paramétrables ; ils concernent essentiellement :

- la compatibilité des langages entre eux (`equivalent_class_semantics`, `equivalent_link_semantics`),
- la compatibilité des types de lien entre eux (en fonction du concept de classe concerné) (`is_compatible_import_links`, `is_compatible_use_links`),
- la compatibilité des concepts de classe entre eux (`is_compatible_class`),
- le nombre de liens possible pour une classe en fonction du concept de classe et du type de lien (`nb_source_import_link`, `nb_same_import_link`),
- le contrôle du niveau de méta-programmation (`is_default_handling`),
- l'association de la sémantique des concepts de classe et des types de lien utilisés (`OFL_class_semantics`, `OFL_use_link_semantics`, `OFL_import_link_semantics` et `OFL_object_to_class_link_semantics`) et
- l'association des erreurs sémantiques propres au langage (`OFL_semantics_errors`).

## 3.4 Autres aspects

On étudie ici un certains nombre de mécanismes qui font partie des aspects importants de *OFL/VM*.

### 3.4.1 Interopérabilité : les éléments de base

Le système *OFL* propose une solution pour permettre à un objet de réagir, soit suivant la sémantique du langage qui l'a créé, ou qui lui a été associé à un instant T, soit suivant la sémantique du langage qui a servi pour l'écriture de l'application.

La mise en œuvre de cette solution se fait en deux temps :

- le choix dynamique de l'implémentation de la sémantique d'une classe ou d'un type de lien et
- le choix du langage associé à l'objet (le langage dans lequel est écrite l'application ou celui de l'application qui a créé l'objet : c'est intéressant pour le partage d'objets persistants ou

pour l'évolution de ces objets lorsque la sémantique du langage qui a servi pour leur création a elle-même évoluée)

La figure 3.4 décrit la mise en œuvre de cette solution. Chaque instance de méta-classe (type de lien et concept de classe), et chaque objet d'exécution (`UPDATABLE_OBJECT`) choisissent au moment de l'exécution d'une demande de travail quel langage elles utilisent en fonction de la valeur du *sélecteur de langage*. Naturellement l'interopérabilité est possible suivant un certain nombre de contraintes qui sont encore à approfondir. Plus simplement on dira que l'interopérabilité sera d'autant plus possible que les langages seront proches les uns des autres. En particulier deux langages qui ont les mêmes types de lien mais avec différentes implémentation pourront s'appliquer indifféremment à un même objet ; de même si on peut déterminer une équivalence entre type de liens (la classe OFL_LANGUAGE possède des opérateurs pour déterminer cette équivalence).

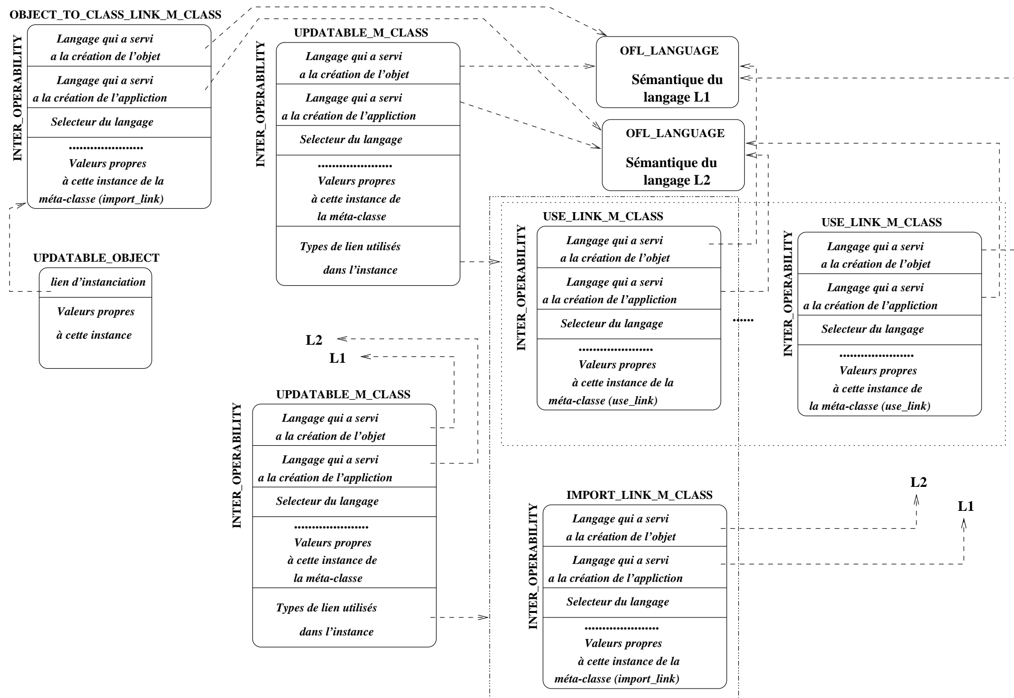


FIG. 3.4 – *Modélisation des aspects sémantiques*

Les classes préfixées par `FLEXIBLE_` (`FLEXIBLE_OFL_CLASS_SEMANTICS` et `FLEXIBLE_OFL_LINK_SEMANTICS`) permettent de choisir au moment de l'exécution l'instance de type `OFL_CLASS_SEMANTICS` (par exemple `EIFFEL3_OFL_CLASS_SEMANTICS`) ou de type `OFL_LINK_SEMANTICS` (exemple : `EIFFEL3_INHERIT_FROM_LINK_SEMANTICS`), qui est adéquate.

L'instance adéquate est choisie à partir des deux langages (éventuellement le même) qui sont accessibles à partir de la classe `INTER_OPERABILITY`. On notera enfin la présence des primitives `equivalent_class_semantics` et `equivalent_link_semantics` qui permettent d'envisager la mise en œuvre d'une gestion des types de lien ou des concepts de classe équivalents. L'équivalence entre des langages pourra être déduites d'informations présentes dans le monde persistant (éventuellement ailleurs sur le réseau).

### 3.4.2 Sémantique des instructions

Il y a des instructions qui contienne entièrement leur sémantique (celle-ci est fixée), c'est le cas de toutes les structures de base d'un langage : schéma conditionnel, schéma itératif, ...

D'autres instructions qui correspondent à des aspects des langages que l'on veut paramétrer sous-traitent les aspects sémantique à l'objet courant.

Dans notre machine virtuelle, on fait cohabiter deux approches différentes suivant les composants : la localisation dans les composants de leur sémantique et la déportation de leur sémantique dans les méta-classes qui centralisent la sémantique de plusieurs composants, ce qui permet une modification plus facile pour le méta-programmeur.

# Chapitre 4

# Principes de mise en œuvre de *OFL/V*

## 4.1 Mise en œuvre d'un langage : méta-programmation

### 4.1.1 Niveaux de méta-programmation

La présence d'une option `default_handling` permet d'alléger la tâche du méta-programmeur. Le principal avantage de la définition de deux niveaux de méta-programmation est que le méta-programmeur peut

- réécrire complètement le contenu de toutes les routines décrivant la sémantique des classes et des liens pour le langage choisi. Ceci nécessite alors une excellente connaissance du fonctionnement de la machine virtuelle.
- bénéficier d'une aide méthodologique pour ajouter simplement les traitements spécifiques au langage et utiliser les enchaînements prédéfinis qui sont adaptés à la plupart de langages industriels. Le méta-programmeur n'a alors besoin que d'une connaissance superficielle de la machine virtuelle.

Par exemple on peut examiner l'enchaînement par défaut fourni pour l'exécution d'une routine :

```
execute(m : INTRA_MESSAGE) is
-- Exécution du message interne 'm'
require
 message_not_void : m /= void
 is_not_void : m.action /= void and m.parameters /= void
 exists : attached.has(m)
local
 f_tmp : like item
 b_tmp : BOOLEAN
do
 if is_default_handling then
 -- Recherche de la primitive
  f_tmp : = item(m)
  if not is_error(m) then
  -- positionnement de la primitive sélectionnée
   m.set_candidate_feature(f_tmp)
   -- Evaluation des paramètres
   parameter_evaluation(m)
   if not is_error(m) then
   -- Contrôle de paramètres
    b_tmp : = is_parameters_compatible(m)
    if not is_error(m) then
     if b_tmp then
      -- attachement : paramètres effectifs -> formels
      effective_to_formal(m)
     else
      -- Erreur sémantique
      -- Traitement de l'erreur de compatibilité
     end -- if
     -- Préparation et exécution de la primitive
     before_request(m)
```

```
      -- Note : partie paramètre de l'exécution
      execute_semantics(attached, m)
      -- exécution d'un code commun après l'exécution
      after_request(m)
      -- détachement des paramètres effectifs
      detach_effective(m)
     else
      -- Traitement de l'erreur
     end -- if
    else
     -- Traitement de l'erreur
    end -- if
   end -- if
  else
   -- sémantique complètement libre
   execute_semantics(attached, m)
  end -- if
 ensure
  end -- execute
```

### 4.1.2 Gestion des contrôles

Dans la deuxième édition de *Object-oriented software construction*, Bertrand Meyer souligne les avantages du typage dynamique en particulier pour garantir la possibilité d'obtenir des exécutifs performants. Sans nul doute le point crucial concerne la recherche de la version des primitives dans la hiérarchie des classes (voir la section 2.5.1).

On peut considérer *OFL* comme un exécutif évolué capable d'associer dynamiquement sa sémantique à un objet, même quand celui-ci est persistant et que la classe qui l'a engendré n'est pas initialement dans le système de classes correspondant à l'application. Éventuellement le langage ayant généré la classe peut ne pas être le même que celui qui a servi à écrire l'application. En ce qui concerne la mise en œuvre, on veut pouvoir garantir un compromis performance/extensibilité acceptable pour l'écriture d'applications industrielles.

Comme *OFL* ne représente qu'un exécutif, il doit normalement être associé a une syntaxe, c'est à dire à un langage (à définir ou déjà existant sur le marché).

**Préparation de la plate-forme *OFL*** Les opérations suivantes doivent être faite :

- remplissage du corps des routines qui décrivent la sémantique du langage (essentiellement une pour chaque concept de classe et une pour chaque type de lien) ;
- génération d'une classe `X_OFL` (ou `X` est le nom du langage) qui définit/*customise* l'exécutif pour le langage (classe dérivée de `OFL_PROGRAMMING`) et qui stocke la sémantique dans le monde persistant et
- compilation de l'exécutif `X_OFL`.

**Traduction des programmes *OFL*** Les programmes écrits avec un de ces langages devront être analysés (interprétation et/ou compilation) afin de [1] :

- vérifier un ensemble de règles de validité dont nous allons discuter le contenu,
- générer des instructions de déclaration (déclaration des structures et des traitements) dans des classes scripts et
- exécuter ces instructions de déclaration et stocker la réification des classes dans le monde persistant [2].

La création d'une instance de la classe racine et l'exécution d'un de ses constructeurs déclenchent, le moment venu, l'exécution de l'application (voir par exemple la classe `APPLICATION01`).

La sémantique des classes et des liens est naturellement contenu dans la description du langage. Toute classe peut être rendue persistante et le chargement d'une instance persistante d'un

1. Éventuellement c'est le programme `X_LOOP` qui pourra directement déclencher la traduction.
2. À l'heure actuelle il n'y a pas encore de gestion persistante et cette phase est immédiatement suivie de l'exécution de l'application.

descendant de M_CLASS est géré par les liens. les situations qui peuvent conduire au chargement d'une classe sont :

- l'accès d'un objet à ses caractéristiques à partir du lien d'instanciation (lien de type OBJECT_-TO_CLASS_LINK) qui unit l'objet à sa classe et
- l'accès d'une classe à un de ses descendants ou ascendants à travers un lien d'importation (lien de type IMPORT_LINK). Ceci est en particulier utile dans la phase de recherche d'une primitive dans une hiérarchie de classe.

Le lien d'utilisation permet la modélisation de la gestion du chargement et de la mise à jour des objets persistants et mobiles de l'application [3]. La gestion du chargement et de la mise à jour des classes et plus généralement des objets persistants et mobiles à travers les liens permet d'adapter l'utilisation de la persistance (et de la mobilité) des objets selon les besoins du langage [4] ; cependant l'implémentation de la persistance et de la mobilité des objets est câblée dans le moteur afin de garantir la performance et la robustesse du système.

Dans l'avenir, la plate-forme *OFL* pourra être récrite en utilisant *OFL* ce qui devrait permettre de considérer la description du langage comme un objet persistant identifiable par un nom (racine de persistance) [5]. L'intérêt serait d'être complètement orthogonal (au sens de la persistance le système *OFL*).

**Discussion sur les tâches du traducteur** La définition de l'ensemble des tâches d'analyse et de vérification que doit réaliser le traducteur influe directement sur le niveau plutôt statique ou plutôt dynamique de *OFL*, et par là de la capacité de *OFL* à gérer des objets mobiles [6].

## 4.2 Mise en œuvre d'une application : programmation

### 4.2.1 Écriture d'une application

Le système *OFL* n'est pas un langage mais plutôt un protocole méta-objet qui doit servir de support à la mise en œuvre de langages de programmation. Un programme en *OFL* sera donc une séquence d'appels aux routines de *OFL*, qui pourra être générée automatiquement par un compilateur ou un interprète ou écrit directement par un programmeur (bien que ce ne soit pas l'approche souhaitée). Pour mieux structurer la séquence d'exécution de ces routines on propose un mécanisme basé sur la classe SCRIPT_CLASS qui permet d'encapsuler dans une classe la description de chaque composant (déclarationdes classes, des primitives et du corps des routines) de manière à utiliser l'approche objet même pendant la phase de descriptions de l'application (voir figure 4.1).

D'une manière générale l'architecture du système permet de bien séparer chacune des phases de la description d'un programme :

- Toute classe qui décrit un composant d'application (exemple : une classe RECTANGLE) hérite de la classe SCRIPT_CLASS (voir A sur la figure). Ces classes utilisent exclusivement les primitives offertes par les bibliothèques (classes qui héritent de OFL_LIBRARY) (voir B sur la figure).
- Les bibliothèques (par exemple la classe FEATURE_DECLARATION_LIBRARY), sont implémentées de telle façon à n'utiliser que des classes qui héritent directement ou indirectement de la classe STATEMENT. Il y a essentiellement deux types d'instructions : celles qui permettent de faire une déclaration (héritières de DECLARATION_STATEMENT) et celles qui permettent de décrire le corps des routines (héritières de RUN_TIME_STATEMENT) (voir C sur la figure).

3. On pourra étudier le meilleur endroit ou placer les contrôles d'accès aux primitives d'une classe lorsque ces primitives sont appelées à travers un lien de clientèle.

4. Le moteur d'exécution gère des références volatiles, persistantes et mobiles dont la sémantique est définie par des liens.

5. Pour l'instant la sémantique d'un langage est seulement accessible à travers une classe.

6. *OFL* pourra se définir comme une plate-forme orientée objet associée à un méta-protocole pour objets mobiles et persistants (voir le titre du document

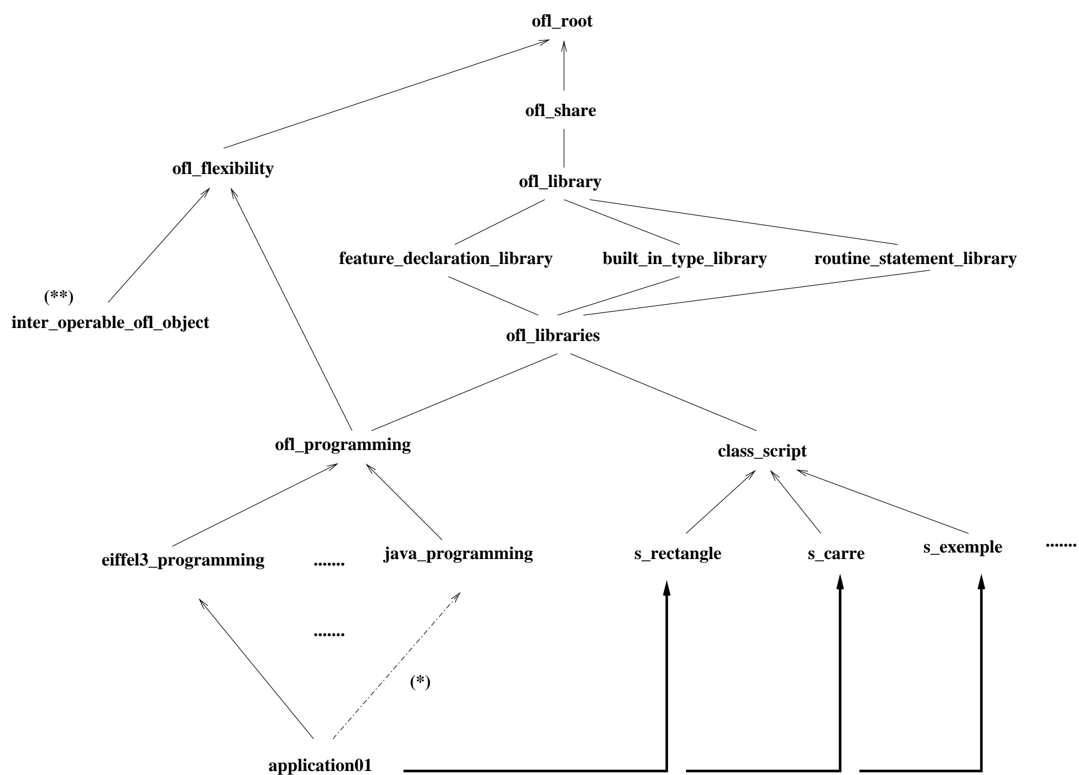


FIG. 4.1 – *Outils pour l'écriture de programmes*

– Les composants qui décrivent des instructions créent et mettent à jour des instances des classes du noyau de système. Ces objets sont destinés à la modélisation des différents concepts atomiques de *OFL* indépendamment de la représentation des instructions de déclaration des composants et des corps de routines (voir D sur la figure).

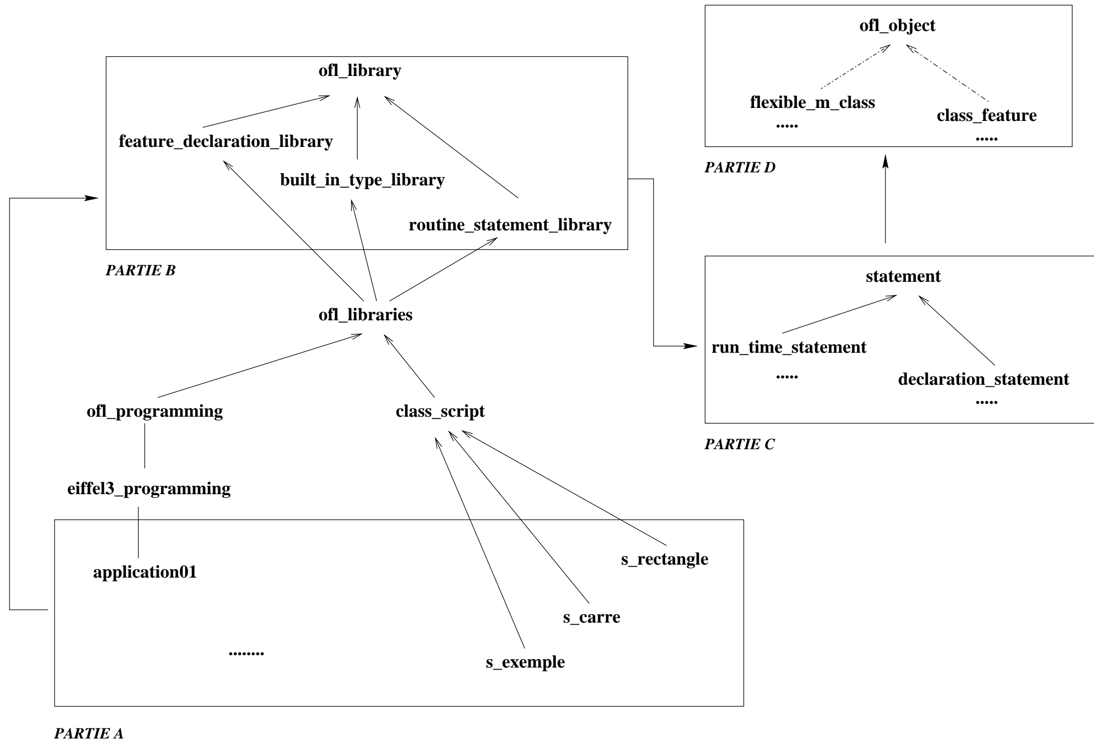


FIG. 4.2 – *Architecture du système*

La séparation entre description de l'instruction et modélisation de l'effet de cette instruction permet de garantir :

– une meilleure capacité de notre système à évoluer (l'évolution de l'architecture des instructions peut se faire indépendamment des modifications du comportement associé aux instructions) ;
– la possibilité de centraliser dans un objet la gestion d'une instruction et de faire correspondre la modélisation de l'instruction à plusieurs entités ou concepts élémentaires, ce qui devrait augmenter la simplicité et l'expressivité du système.

On notera que la description des instructions que l'on trouve dans le corps d'une routine est modélisée à partir des descendants de la classe `RUN_TIME_STATEMENT`. C'est le seul cas ou des objets qui représentent des concepts élémentaires du modèle référencent directement des descendants de la classe `OFL_STATEMENT` ; ce n'est pas le cas pour les autres (déclaration de routines, de classes, de liens, . . . ). Ce choix est basé sur les réflexions suivantes :

– une classe héritière de `DECLARATION_STATEMENT` génère les objets sur lesquels sera basée la sémantique de l'exécution.
– une classe héritière de `RUN_TIME_STATEMENT` ne fait qu'exécuter des traitements sur des objets créés par les classes héritières de `DECLARATION_STATEMENT` ; une modification de la forme de ces instructions n'a pas d'effet sur la structure du programme.

On notera que, dans la bibliothèque relative aux liens, on trouvera la possibilité de créer directement les principaux types de lien d'importation (pour l'instant on trouve seulement le lien d'héritage) et une création de lien d'importation générale dans lequel on devra spécifier le nom du

type de lien d'importation. Ce nom devra être présent dans la liste des noms de lien associé au langage (un contrôle pourra être fait en ce sens).

En ce qui concerne les liens d'utilisation il doivent être spécifiés à la creation de la fonction, du paramètre ou de l'attribut local ou non. La bibliothèque de création des primitives contient des primitives pour créer les liens les plus utilisés sans avoir à spécifier la sémantique (celle si est retrouvée par le `statement` concerné, à travers les informations présentes dans la sémantique du langage associé à la classe qui contient la primitive). Pour les autres liens (dont la sémantique doit être connue par la machine virtuelle), il y a des primitives qui ont cette fois besoin du nom du type de liens.

### 4.2.2 Phase d'exécution d'une application

#### Description générale des objets à l'exécution

La figure 4.3 décrit de manière détaillée le comportement des objets à l'exécution (accès à leur sémantique, envoi de message, ...). L'approche envisagée est de partager un objet d'exécution en trois sous-objets :

- le gestionnaire de demandes de travail et
- l'exécutif chargé de répondre à une demande de travail.

Le premier reçoit éventuellement des demandes de l'extérieur. Après analyse, il propage cette demande à un autre objet ou il envoie une note interne à l'exécutif pour qu'il fasse le travail demandé et rende une réponse qui est alors transmise à l'objet ayant fait la demande de travail.

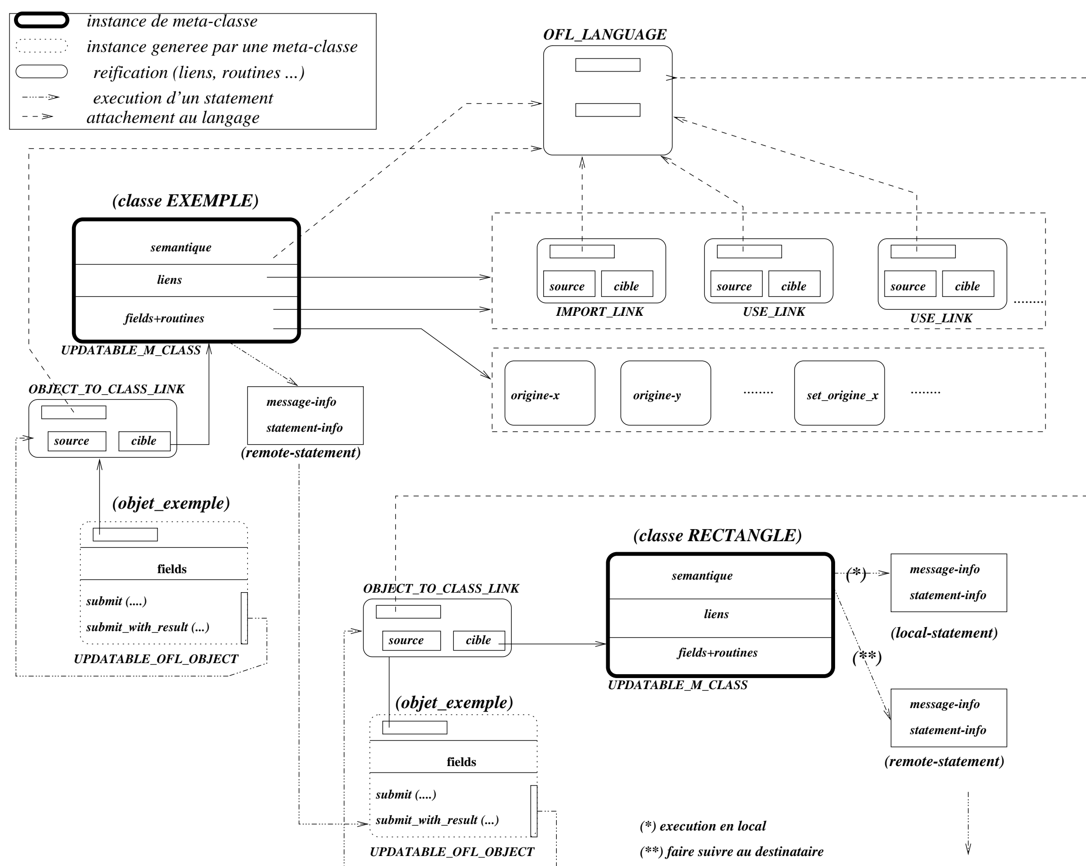


FIG. 4.3 – *Comportement à l'exécution*

On remarque en particulier qu'un objet à l'exécution (l'instance d'une classe de l'application) est de type RUN_TIME_OFL_OBJECT. Il peut soit être représenté par une instance de la classe UPDATABLE_OFL_OBJECT, soit par une instance d'un descendant de BUILT_IN_OFL_OBJECT. Chaque objet représentant l'instance d'une classe de l'application référence son type et donc sa classe (une instance d'un descendant de la classe M_CLASS : UPDATABLE_M_CLASS ou BUILT_IN_M_CLASS). On rappelle que ces dernières contiennent toutes les informations décrivant la classe (primitives, liens, ...), et la sémantique de la classe (coordination entre les liens et traitement spécifique pour tous les concepts paramétrés). Par exemple parmi les traitements paramétrés on trouve :

- submit et submit_with_result,
- execute et execute_with_result,
- send et send_with_response.

**Description intuitive d'une exécution** Une instance de classe reçoit une demande de travail par l'intermédiaire d'un appel de sa routine submit ou submit_with_result. Cette soumission va nécessiter un dialogue entre l'instance et sa classe puisque c'est cette dernière qui contient la sémantique de la réaction à la soumission d'un travail. D'une manière générale (on ira plus dans le détail dans la section suivante), une soumission de travail déclenche une analyse de la demande : soit celle-ci est interne (c'est l'objet lui-même qui demande l'exécution d'une de ses routines à partir d'une autre de ses routines), soit celle-ci provient d'une autre instance de classe (ou éventuellement d'une instance de méta-classe). Dans le premier cas, c'est un appel à execute ou execute_with_result qui est effectué, dans le second cas cela provoque un envoi de message (send ou send_with_response, selon le cas) qui engendre une soumission de travail vers une autre instance de classe ; ceci se répète jusqu'à ce que l'on atteigne le destinataire final.

On rappelle que chaque lien est une instance de la classe LINK et modélise un lien entre deux classes ou entre un objet et une classe ; cette instance est elle-même associée à une instance de la méta-classe LINK_M_CLASS qui décrit les caractéristiques et la sémantique d'un type de liens. C'est à ce niveau que sont mémorisés tous les concepts paramétrables d'un lien.

On rappelle aussi que la possibilité d'associer dynamiquement une nouvelle sémantique à une classe et par voie de conséquence à un objet est réalisée au moyen d'une indirection à travers un objet de type OFL_LANGUAGE. Toutes les informations permettant de décrire la sémantique des liens et des classes sont centralisés dans cette classe mais implémentées dans ses descendants.

### Modélisation de l'exécution d'une instruction

Pour expliquer notre modélisation de l'exécution des instructions d'une routine, on va s'appuyer sur les instructions mettant en avant le plus de problèmes ; il s'agit des expressions pointées (*remote call*).

Soit l'instruction I1 représentant l'expression f1(exp1).f2(exp2).proc1(exp3), soit la classe C1 représentant le type de f1, C2 représentant le type de f2, et soit la classe C représentant le type de l'objet courant.

Du fait de la réification, chaque instruction d'une routine est représenté par un objet dont la classe est un descendant de la classe STATEMENT (voir figure 4.4).

Supposons que l'on veuille exécuter l'instruction I1 (I1.execute(...)), le contenu de la primitive execute va construire une demande de travail (*job-request*) et va la soumettre à l'objet courant. Celui-ci, à partir des informations contenues dans l'instance de la méta-classe auquel il est rattaché et de la sémantique associée à cette méta-classe (et donc aux liens qui lui sont associés), va analyser la demande de travail et donc s'apercevoir que cette demande de travail nécessite la participation d'autres objets que l'objet courant et nécessite donc l'envoi d'un message.

L'objet courant va d'abord demander l'évaluation des paramètres exp1, exp2 et exp3, puis il va déterminer le premier destinataire D1 du message par l'émission d'un message interne (f1(exp1-évalué)), puis il va envoyer le message f2(exp2).proc1(exp3) à D1. L'envoi de ce message va

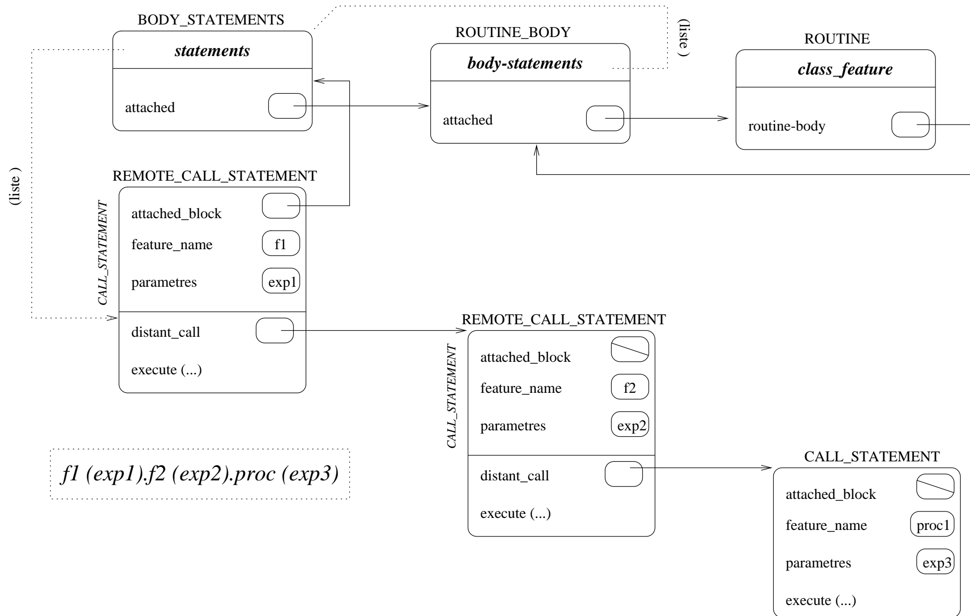


FIG. 4.4 – *Réification d'une expression*

provoquer deux choses [7] :

- le changement de l'objet courant qui devient le destinataire `D1` et
- la soumission d'une demande de travail `f2(exp2).proc1(exp3)` au nouvel objet courant.

À partir d'ici le processus qui vient d'être décrit se reproduit à l'identique : l'objet courant, après analyse du message, se rend compte que la demande de travail qu'il a reçu nécessite l'envoi d'un message. Il va donc rechercher le deuxième destinataire `D2` du message en émettant un message interne (`f2(exp2-évalué)`). et il va envoyer le message `proc1(exp3)` à `D1`. L'envoi de ce message va provoquer deux choses :

- le changement de l'objet courant qui devient le destinataire `D2` et
- la soumission d'une demande de travail `proc1(exp3)` au nouvel objet courant.

Le nouvel objet courant, après analyse du message, se rend compte que la demande de travail ne nécessite plus le concours d'un autre objet et il va simplement émettre un dernier message en interne demandant l'exécution de `proc1(exp3)`.

Lorsque la dernière demande de travail n'est pas une procédure mais une fonction, le résultat est retourné successivement aux expéditeurs du message jusqu'à l'expéditeur initial.

La figure 4.5 montre un exemple de modélisation de l'exécution d'une instruction plus simple que celle présenté dans la figure 4.3.

Cette instruction repose sur la hiérarchie de classes définie dans la figure 4.5, qui correspond à la représentation *OFL* de ce cas d'école (voir aussi la section 4.2.1).

L'instruction qui est modélisée ici est `carre.perimetre`. à partir d'une instance de la classe `EXEMPLE`. Cet exemple est plus simple que le précédent dans le sens ou le premier destinataire du message est le destinataire final et que la fonction `perimetre` n'a pas de paramètre à évaluer. Voici un résumé des différentes phases de l'exécution de cette instruction :

- L'objet courant déclenche l'exécution de la routine `execute` de l'instance de la classe `REMOTE_CALL_STATEMENT` qui représente l'instruction `carre.perimetre` ; celle-ci se trouve dans la routine de création `make` de la classe `EXEMPLE`.

7. Naturellement, le changement de l'objet courant et la soumission de travaux ne sont que les aspects fondamentaux : les sémantiques d'un lien ou d'une classe peuvent engendrer nombre de contrôles et de mises à jour.

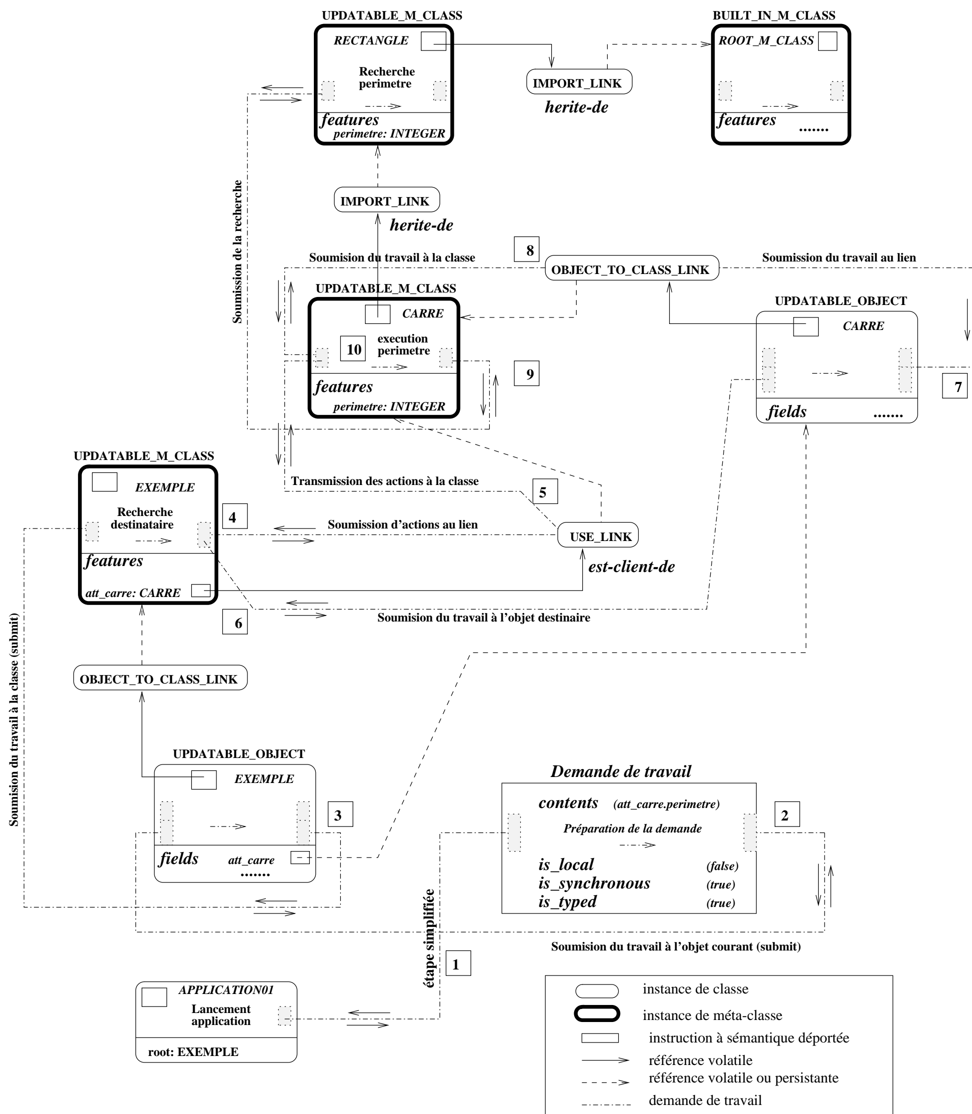


FIG. 4.5 – *Exécution d'une instruction*

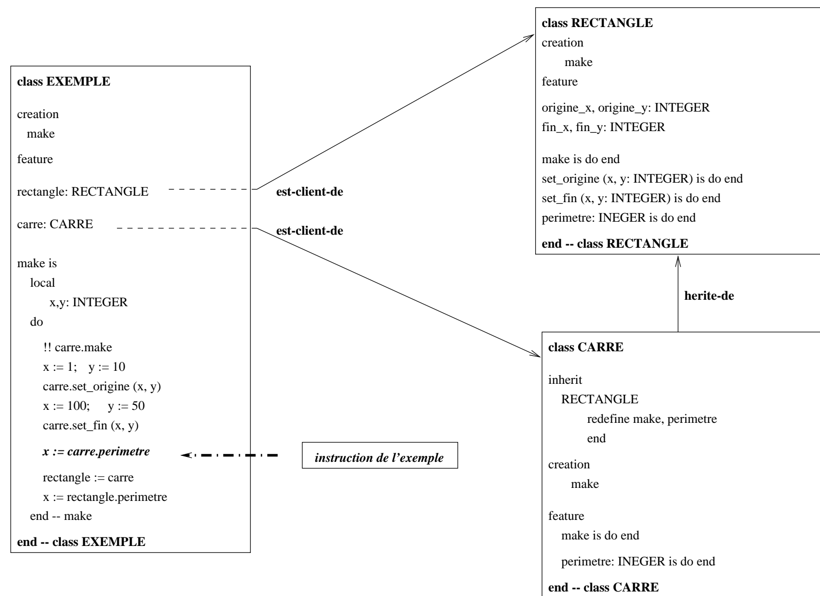


FIG. 4.6 – *Exemple de hiérarchie*

- L'évaluation de la routine `execute` provoque la génération d'une demande de travail (*job-request*) qui sera soumis à l'objet courant (instance de la classe `EXEMPLE`). Comme cette demande de travail est issue d'une instance de la classe `REMOTE_CALL_STATEMENT`, elle s'exprime à travers un message qui doit être envoyé par l'objet courant.
- Celui-ci, avec l'aide de sa classe, analyse le message et recherche/calcule son destinataire (qui est ici le destinataire final).
- Une fois que le destinataire est connu (l'instance de la classe `CARRE` qui est référencé par l'attribut `carre` de la classe `EXEMPLE`), le message lui est envoyé et l'objet destinataire reçoit une demande de travail émanant de l'instance de la classe `EXEMPLE`, en même temps qu'il devient le nouvel objet courant du système (celui qui est en train de *travailler*). Cette demande de travail est l'exécution de la routine `perimetre` sur l'objet courant ; elle est matérialisée dans *OFL* par une instance de la classe `INTRA_MESSAGE` (message interne).
- L'objet courant, après avoir analysé le message, toujours avec l'aide de sa classe, va demander l'exécution de la routine `perimetre` sur lui-même et il va retourner le résultat à l'expéditeur (fin de l'exécution de la routine `submit_with_result`), qui redevient l'objet courant.

On notera pour finir qu'un message mémorise au fur et à mesure des transmissions intermédiaires le chemin qu'il a suivi ainsi que les primitives qui ont été retournées par un `lookup`.

## 4.3 Position par rapport à d'autres systèmes

Nous proposons ici de situer notre travail par rapport à l'état de l'art. Pour cela nous proposons un certains nombre de critères de comparaison. Nous nous sommes intéressés à des systèmes réflexifs construits autour de *Java*, *C++*, *Smalltalk* et *Lisp* :

### 4.3.1 Objectifs de la réflexivité

Nous proposons ici de présenter les objectifs de chaque langage réflexif (prototypage, interopérabilité, optimisation, simplification des traitements, etc.).

**Systèmes autour de *Lisp*** *CLOS* (*Common Lisp Object System*) [GGL93] est certainement un des premiers protocoles méta-objet.

**Systèmes autour de *Smalltalk*** Dans *NeoClassTalk*, F. Rivard [F.97] s'intéresse à l'évolution du comportement des objets dans le cadre de langages réflexifs et dynamiquement typés. Il introduit en particulier la possibilité de modifier la classe associée à un objet (la classe qui modélise les primitives auquel il a accès) ; sa motivation est d'offrir un support pour diminuer la distance qui existe entre la conception d'une application et son implémentation dans un langage de classe, lorsque l'on est confronté à des objets dont la structure et le comportement évolue au cours du temps. L'évolution peut venir de l'objet mais aussi d'une évolution de la classe qui est vouée à être modifiée au cours du temps (dans ce dernier cas, on aura une nouvelle classe et c'est elle qui sera attachée à l'objet en remplacement de la classe qui a évolué et la mise à jour de la structure de l'objet se fera de manière paresseuse, variable d'instance par variable d'instance).

*OpenCorba* [Led99] est basé sur *NeoClassTalk* [F.97], il a pour objectif d'étendre le modèle initial de l'*OMG* [Gro95] avec des bibliothèques de méta-protocoles spécialisant les mécanismes de programmation répartie afin d'introduire de nouvelles sémantiques (concurrence, réplication, sécurité, ...).

**Systèmes autour de *C++*** *OpenC++* [Chi95, Chi98a, Chi98b] a été conçu pour permettre le développement d'outils autour du langage *C++* et dispenser le programmeur de tâches fastidieuses comme la programmation d'un analyseur ou la modélisation du système de types. Certaines utilisations d'*OpenC++* sont le développement d'extensions de *C++* ou l'optimisation de la compilation des bibliothèques de classes.

Un autre projet *DART* [RLVG98] repose sur une extension de la syntaxe *C++* et sur le *MOP* d'*OpenC++* ; ses objectifs sont proches de ceux de *LEAD++* [AW98] mais l'expérimentation de ce système est faite dans le cadre d'environnements 3D.

*Iguana est une extension de C++ qui offre une réification de concepts et une possibilité de modifier l'implémentation. C'est C++ qui a été choisi à cause de sa capacité à permettre le développement d'applications système.*

**Systèmes autour de *Java*** *Java* offre un nombre limité de possibilités de réflexion ; elles concernent la sécurité (accès aux classes `socket` et `file`), la transformation d'un tableau d'octets en classe mais pas l'inverse (pas de réification et donc pas de possibilité de changer la sémantique d'une classe) à travers le chargeur de classe. L'API pour la réflexion offre seulement une réification des structures et a peu de fonctionnalités ; elle est surtout utilisée pour l'implémentation des *beans*. *Object serialization* est un méta-système pour la gestion de la persistance.

L'objectif de *Reflexive Java* [Col97] est d'étendre *Java* dans le but d'offrir de plus grandes capacités d'adaptation pour l'implémentation de *middleware* et à ce titre s'oriente sensiblement vers les problèmes liés à la gestion de transaction dans le cadre d'Internet (sécurité et concurrence) ; il est construit sur *javabeans*. Un autre système comme *dalang* [Z98] est construit à partir de la bibliothèque de classes `java.lang.reflect` et sa conception repose sur une volonté de maintenir sa portabilité sur toutes les *JVM* (même chose pour *Reflexive Java*).

Le système *metaXa* [GK98] est un système réflexif non ciblé vers une utilisation particulière.

Il existe d'autres travaux autour de *Java* comme le langage réflexif *LEAD++* [AW98]. Ce dernier est dédié aux environnements distribués avec des objets mobiles et à pour objectif d'adapter le logiciel à l'environnement d'exécution.

### 4.3.2 Expressivité et encapsulation des aspects réflexifs

Nous étudions ici plus précisément les possibilités offertes par chaque système en matière réflexivité (introspection et intercession) et l'encapsulation de la sémantique, c'est à dire les facilités offertes au méta-programmeur pour le guider dans les modifications sémantiques.

**Systèmes autour de *Lisp*** *CLOS* (Common Lisp Object System) définit la notion de méta-classe comme des sous-classes d'une des méta-classes de base, à savoir `class`, `slot-definiton`, `generic-function`, `method` et `method-combination`. Le protocole méta-objet de *CLOS* repose fortement sur la notion de fonction générique. Ces fonctions génériques permettent de paramétrer la sémantique de *CLOS* et de l'adapter à un besoin particulier. Une fonction générique s'appliquera en particulier à une méta-classe et donc toute classe qui veut être gérée selon la sémantique d'un groupe de fonctions devra *descendre* de la méta-classe a laquelle elles sont associées . Une fonction générique est constituée en particulier de : $n$ méthodes (`before`, `around` et `after`), $m$ méthodes primaires et une méthode de composition qui gère leur exécution.

**Systèmes autour de Java** *Reflexive Java* [Col97] et *dalang* [Z98] paramètrent uniquement l'invocation de méthode. *dalang* entoure l'appel à une routine avec des routines `before` et `after` et offre la possibilité de modéliser l'exécution d'une routine sur un objet à travers plusieurs méta-objets qui lui sont associés par chaînage.

Le système *metaXa* [GK98] fourni des facilités pour réifier les structures et la sémantique ; le système prend en compte l'invocation de méthode, l'accès aux variables, les opérations de gestion, la création d'objet et le chargement de classe. Il y a plusieurs opérateurs qui prenne l'événement en paramètre pour traiter l'exécution d'une routine ou l'accès à une variable : un pour activer le code du méta-objet (`doExecute`, `doReadVariable`), un autre pour chaîner l'exécution avec un autre méta-objet (`continueExecution`, `continueReadVarible`) et un autre enfin pour exécuter la routine sur l'objet sans réaliser d'autres opérations méta (`nonRefletiveExecute`, `non-ReflectiveReadVariable`). Pour les autres mécanismes, l'attachement de méta-objets permet d'avoir différentes stratégies pour le chargement de classes, pour le blocage ou l'allocation des objets ou pour la gestion des événements dans les *threads*. La gestion des méta-objets basée sur la règle « en cas d'attachement à un objet O, le méta objet associé à O est propagé au nouvel objet attaché et dissocier de l'ancien ». permet entre autre de paramétrer la gestion de la persistance. Les informations nécessaires (*annotations*), doivent être communiquées au méta-objet à l'instanciation ou à l'activation des opérateurs ; parmi les informations intéressantes on notera : le mode de passage des paramètres, le protocole de duplication d'instance, de gestion de la persistance, d'envoi de message, etc.

Dans *Guaraná* [OB98b, OGB98], c'est un méta-objet principal (*primary*) qui règle les échanges entre l'application et le noyau du système : exécution de l'opération par le méta-objet lui-même, transmission d'une opération de remplacement au noyau et transmission de l'opération originale au noyau (dans les deux derniers cas, le méta-objet se réserve le droit, sur demande préalable, de modifier le résultat retourné par le noyau). Si plusieurs méta-objets sont associés à un même objet un méta-objet particulier appelé *composer* est chargé de regrouper, coordonner voire de filtrer l'action des autres méta-objets associés à l'objet de base. Plusieurs sortes de *composer* sont particulièrement intéressants : *sequential composer*, *concurrent composer*, mais d'autres plus spécifiques pourront aussi être implémentés. Dans ce système, la création d'un objet de l'application et d'un méta-objet est initiée par l'application au fur et à mesure de leur exécution à partir de demandes de reconfiguration. Si cette demande est initiée par un objet qui a déjà une méta-configuration (le méta-objet primaire est traité de manière transparente) le système va traiter cette demande (remplacement, ajout, rejet, création), sinon celle-ci sera transmise aux méta-objets associés à sa classe puis à ceux des classes parentes, ces derniers pouvant modifier le contenu de la demande. On notera que c'est à sa création, et sous le contrôle du méta-objet primaire de l'objet qui l'a créé, qu'un objet se voit associé éventuellement son méta-objet primaire. Après cette opération de reconfiguration un message spécifique est envoyé aux méta-objets de la classe qui peut ainsi demander elle aussi des reconfigurations.

Le modèle de *LEAD++* [AW98] est basé sur le concept de *procédure adaptable* (l'idée est d'adapter l'exécution de la procédure en fonction de l'environnement d'exécution). Ces procédures sont associées à des *méthodes adaptables* et la *bonne* méthode est choisie en s'appuyant sur l'état de l'environnement d'exécution (*base-level dispatching*), sur des informations méta associés à ce même environnement (*meta-level dispatching*), et en fonction d'une *stratégie d'adaptation*. Le

déclenchement de ces procédures est initié par des événements qui correspondent à des changements d'états de l'environnement d'exécution.

**Systèmes autour de *C++*** Dans *OpenC++* [Chi95, Chi98a, Chi98b], pour introduire une extension du langage *C++*, il faut écrire un programme méta, qui une fois compilé fournit les informations relatives à ces extensions au compilateur *OpenC++*. Les aspects méta sont représentés par des instances de classes prédéfinies (et des classes dérivées éventuelles décrites par le méta programmeur) qui sont appelés méta-objets. Ces méta-objets permettent de faire de l'introspection et d'agir sur le comportement des objets et sur le code généré.

L'approche suivie par *Iguana* [GC96] est assez similaire à la notre, en particulier il est possible de faire cohabiter plusieurs *instances* du *MOP* (pour nous plusieurs instances de langage) mais bien sûr autour des possibilités de *C++*. Les principaux concepts réifiés sont : classe, arbre d'héritage, type (contrôle dynamique ou statique), méthode (notion de *before* et *after*, adresse, ...), identité d'objets, autres aspects spécifiques de *C++* (table des symboles, fonctions virtuelles, membres, ...), activation de méthodes, création et suppression d'objets, envoi et réception de messages au niveau de l'objet et de la classe, recherche de primitive. Ce système offre la possibilité, lors de l'instanciation du *MOP*, de définir les entités que l'on veut réifier afin d'offrir un système plus performant.

**Systèmes autour de *Smalltalk*** *NeoClassTalk* [F.97] est basé sur la syntaxe et la philosophie réflexive de *Smalltalk*, le noyau des classes de *ClassTalk* [Coi87], la réification de l'application des envois de message de *Moostrap* [Mul95]. Ses principales fonctionnalités offertes par *NeoClassTalk* sont regroupées par catégories : objets classes, structure et comportement (contrôle et initialisation paresseuse des variables d'instances, faculté d'un objet à s'adapter à la sémantique d'une autre classe ou modification du lien d'instanciation), héritage (introduction d'un héritage multiple avec une intégration limitée de l'héritage répété avec partage des variables d'instances, modification à l'exécution du lien d'héritage), envoi de message, syntaxe et compilation (extension du `lookup` câblé de *Smalltalk* pour prendre en compte l'héritage multiple, réification de l'invocation de méthode, et contrôle de celle-ci par la classe du receveur), propriétés de classe (manipulation explicite des classes et des méta-classes sans différenciation, composition dynamique par héritage simple de propriétés élémentaires des classes, possibilité d'appliquer la sémantique de *Smalltalk* et donc possibilité de programmation par propriétés de classes implicites et explicites).

### 4.3.3 Mise en œuvre et performance des systèmes

Comme cela a été évoqué plus haut, la réification et les contrôles dynamiques posent des problèmes de performance. Nous examinons ici comment ce problème est pris en compte par les différents systèmes.

**Systèmes autour de *C++*** *Iguana* [GC96] est implémenté comme un pré-processeur de *C++* et le code *donné* au compilateur *C++* sera d'autant plus différent du code source que l'usage des possibilités de réflexion sera plus important. Le système permet de décider de la réification des concepts cités ci-dessus indépendamment. La modification de la sémantique associée par défaut est réalisée en héritant de la classe modélisant la réification et en spécialisant les méthodes qui s'y trouve (il est possible de donner un nom a l'instance de la méta-classe pour distinguer plusieurs versions qui coexistent). L'ensemble des méta déclarations est encapsulé dans la notion de *protocol* ; il est possible de spécifier qu'un protocole utilise d'autres protocoles existant. C'est au moment de la déclaration d'une classe que l'on décide du ou des protocoles utilisés. De même il est possible d'associer à une variable le protocole utilisé pour l'instance associée à celle-ci (une méta-classe correspondante est ainsi générée par le système).

Il semble que ce système permette la définition et l'utilisation simultanée de plusieurs protocoles mais, dans ce cas, ces protocoles implémentent la sémantique de catégories disjointes.

**Systèmes autour de Java** Dans *dalang* [Z98], la mise en œuvre est basée sur la notion de *wrapper class*. Chaque classe dont on veut paramétrer l'invocation de méthode est remplacée par une classe générée par le système ayant le même nom et qui hérite d'une méta-classe implantant les routines `before` et `after` ; cette classe est ensuite compilée par un compilateur *Java* standard. On notera que l'utilisation de *wrappers* entraine des problèmes avec l'utilisation de l'héritage (pas de partage des super-classes avec la classe d'origine) et de l'accès à l'objet courant `self` (*delegation vs forwarding*). La performance ne pose de problème que dans le cas du chargement dynamique de classe qui nécessite l'utilisation d'un chargeur de classe spécifique (modification du code source) et, la première fois, une opération de compilation préalable. On notera qu'une approche par modification du byte-code résoudrait une grande partie des problèmes mentionnés dans ce paragraphe.

Dans *Reflexive Java* [Col97] un méta-objet se compose de deux couches : la première intercepte l'invocation de méthode de la part d'un client et la fait suivre à une deuxième couche composée de plusieurs méta-objets implémentant chacun la gestion d'une fonctionnalité (persistance, ...) ; l'exécution de ces méta-objets est coordonnée par la première couche ce qui permet aux méta-objets d'être réutilisés dans d'autres systèmes. Afin de ne pas imposer un surcoût aux objets qui n'ont pas besoin d'une invocation spécifique des méthodes, la réflexion ne s'applique qu'aux objets pour lesquels le méta-programmeur en fait la demande.

La mise en œuvre de *metaXa* est basée sur une extension de la *JVM* et s'applique à fournir une solution performante et sûre du point de vue des types. Cette extension de *JVM* est basée sur une séparation des codes de base et méta, et sur un transfert du contrôle des objets de base vers les méta-objets à travers des événements traitant chacun un des mécanismes paramétrés. Le système permet l'introduction progressive de méta-objets, autorise la liaison de $n$ classes avec un méta-objet et *vice versa* et permet un contrôle complet de l'invocation d'une routine (paramètre, valeur retournée, acceptation, rejet ou retardement de l'exécution).

*Guaraná* [OB98b, OGB98] est une plate-forme qui doit être placée au dessus de n'importe qu'elle plate-forme logicielle ; cependant sa mise en œuvre est basée sur la modification d'une implémentation de la *JVM* appelée *Kaffe Open VM* et la syntaxe du langage *Java* n'a pas été touchée. L'association entre un objet d'application et le(s) méta-objet(s) qui assure(nt) sa gestion est dynamique et réalisée par un opérateur global au système (*reconfigure*), il n'y a aucun lien entre l'objet et son méta-objet. Un des objectifs de cette approche est la possibilité de décrire des bibliothèques de méta-objets comme *MOLDS* [OB98c] qui est destinée à des systèmes distribués. Les aspects relatifs à la performance tels que des *benchmarks* sont abordés dans [OB98a].

La mise en œuvre de *LEAD++* [AW98] est réalisée à partir d'un pré-processeur de *Java*. Ce système est associé à un traducteur, des bibliothèques de classes et de fonctions de bas niveaux. La syntaxe du langage est une extension de *Java* et le code méta est séparé du code de l'application.

### 4.3.4 Méta-programmation et principaux services offerts

Nous étudions ici les langages ou les systèmes en focalisant sur les services offerts en matière de persistance, de distribution ou de mobilité des objets sur le réseau.

**Systèmes autour de Java** Dans *dalang* [Z98], on envisage d'introduire une gestion de la persistance par héritage de méta-objets afin de prendre en compte ce service lors de l'invocation de méthode. *Reflexive Java* [Col97] offre des méta-objets pour prendre en compte la gestion de la persistance, de la concurrence et de la récupération d'erreur. Dans les deux systèmes, on offre des outils permettant de définir les aspects méta, soit graphiquement[Col97], soit par un fichier de configuration [Z98], soit par une interface de méta-programmation [GK98] qui demande, pour le méta-programmeur la manipulation d'entités en invoquant un nom (risque d'erreurs). Par ailleurs *metaXa* offre des facilités pour gérer la persistance (méta-objet *MetaPersistence*), et propose une solution pour utiliser la réflexion pour améliorer les performances du système.

### 4.3.5 Bilan

Comme *CLOS*, le modèle que nous proposons est indépendant des langages et l'utilisation de la réflexivité est orientée vers le paramétrage d'un langage de programmation. Par rapport à *CLOS* nous offrons un cadre plus directif pour le méta-programmeur, nous allons dans ce sens travailler sur les aspects méthodologiques.

**Les points positifs de notre approche** Nous pensons comme [OGB98] qu'il est important de fournir au méta-programmeur une approche permettant de réutiliser les composants méta, de coordonner l'action de ces composants et d'une manière générale de diminuer la complexité de la tâche du méta-programmeur, ou du moins de faire en sorte que la complexité soit proportionnelle aux besoins du méta-programmeur.

Par ailleurs si dans [GK98] on propose une liste de méta-informations classées par services (persistance, sécurité, etc), ces informations ne sont, d'une manière générale, pas très structurées alors que nous proposons une collection d'opérateurs et une localisation de ceux-ci dans trois types d'entités (liens, classe, langage) en fonction de leur nature.

**Les points à étudier de manière approfondie** La performance de la réflexivité est un point qui revient souvent dans la littérature ; elle est prise en compte principalement par l'utilisation d'outils permettant une génération des objets méta avant l'exécution et par l'absence d'une réification systématique. Notre approche orientée vers une réification complète adresse de manière insuffisante les problèmes de réflexivité ; il pourra être intéressant d'adapter notre approche à une réification par nécessité.

## 4.4 Perspectives et conclusion

Nous partageons les propos des auteurs de [OGB98] quand il disent que l'introduction de réflexivité structurelle ou sémantique est intéressante dès que l'on veut adresser des problèmes comme la gestion de la persistance, de la distribution des objets, de la tolérance aux pannes, de la duplication d'information.

Dans cet article nous avons proposé un modèle réflexif qui permet de paramétrer la sémantique d'un langage au sens large (*reflective behaviour*). Ce modèle apporte une contribution aux problèmes suivants :

- modélisation de la sémantique opérationnelle d'un langage,
- séparation du code d'une application des informations et du code méta,
- adaptation de la tâche du méta-programmeur à la complexité des changements à opérer,
- composition et réutilisation de méta-objets
- gestion et contrôle de l'interopérabilité entre objets et
- intégration de services comme la persistance ou la mobilité des objets.

Notre objectif de réaliser une plate-forme unifiée permettant l'implantation de langages intégrant la distribution d'objets persistants et mobiles sur le réseaux est un projet ambitieux et ne peut être envisagé que sur le long terme. La preuve en est que nous abordons très peu les problèmes de performance et ne proposons pas d'outils pour aider le méta-programmeur qui pour l'instant doit décrire sa sémantique à partir de routines en *Eiffel*. Ce travail devra être réalisé sous peine de limiter notre apport à des aspects de modélisation. Nous sommes actuellement en train de finaliser un prototype (40 000 lignes et environ 300 classes *Eiffel* spécifiques au projet) et nous travaillons sur le modèle d'intégration de la persistance auquel une implémentation basée sur un système de gestion de bases de données existant fera suite.

# Bibliographie


[AB91] Abiteboul S. and Bonner A. “Objects and Views”. In *?*, pages 238–247. 1991.

[ACP98] Ayache L., Crescenzo P. and Paquelin J.-L. *From Software Ingeneering to Linguistic Ingeneering.* 1st International Workshop on Applications of Constraint-Based Programming to Computational Linguistics, Blaubeuren, May 1998, 25 pages.

[ADC96] Aponte M.-V. and Di Cosmo R. “Type Isomorphisms for Module Signatures”. In *Programming Languages: Implementations, Logics and Programs (PLILP'96)*, pages 334–346. September 1996.

[AG96] Arnold K. and Gosling J. *The Java Programming Language.* The Java Series. Sun Microsystems, 0-201-63455-4, 1996.

[AHG95] (Al) Haddad H. M. and George K. M. “A survey of method binding and implementation selection in object-oriented programming languages”. *Journal of Object Oriented Programming*, vol. 8, num. 6 (October 1995), pages 28–41.

[Ala91] Alagic S. “Object-Oriented Databases”. In Bezivin [Bez91], pages? Tutorial.

[Arn98] Arnaud F. “Eiffel: Frequently Asked Questions”. ftp://rtfm.mit.edu/pub/usenet/-comp.lang.eiffel/comp.lang.eiffel_Frequently_Asked_Questions_(FAQ), 9 October 1998.

[AW98] Amano N. and Watanabe T. “*LEAD++*: An objet-oriented reflective language for dynamically adaptable software”. In Fabre and Chiba [FC98]. UTCCP Report 98-4; ISSN: 1344-3135.

[Ban91] Bancilhon F. “Object-Oriented Database Systems”. In Bezivin [Bez91], pages? Tutorial.

[BBCF98] Balke C., Bos K., Candlin D. and Fisher S. “Eiffel Style Guide”. http://hepunx.rl.ac.uk/atlas/oo/eiffel/style_guide.html, 1998.

[BC96] Boyland J. and Castagna G. “Type-Safe Compilation of Covariant Specialization: A Practical Case”. In Cointe [Coi96], pages 3–25.

[BD91] Bancilhon F. and Delobel C. “Recent Advances in O-O DBMS”. In Bezivin [Bez91], pages? Tutorial.

[BD93] Bosch C. and David R. “An Environment for Managing the Evolution of Object-oriented Systems”. In Ege [Ege93], pages 163–172.

[Bel96] Bellahsene Z. ´Modifications Virtuelles d'une Hiérarchie de Classesˇ. In Dennebouy [Den96], pages 92–107.

[Ber91] Bergstein P. L. “Object-Preserving Class Transformations”. In Paepcke [Pae91], pages 299–313. ACM SIGPLAN Notices, vol. 26, num. 11.

[Ber92] Bertino E. “A View Mechanism for Object-Oriented Databases”. In *?*, pages 136–151. 1992.

[Ber97] Berger L. ´Coordination entre objets distants par l'expression de dépendances dans un modèle objet asynchroneˇ. Rapport tech., Université de Nice – Sophia Antipolis, Juin 1997.

[Bez91] Bezivin J. (ed.) . *International Conference on Technology of Object-Oriented Languages and Systems (Tools 4, Europe'91)*, Paris. Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-923160-9, 4–8 March 1991, 415 pages.

[BG95] Bertino E. and Guerrini G. “Objects with Multiple Most Specific Classes”. In Olthoff [Olt95], pages 102–126.

[BK93] Barclay P. J. and Kennedy J. B. “Viewing Objects”. In *Advances in Databases*, num. 696 in Lecture Notes in Computer Science, pages 93–110. Springer-Verlag (Berlin), July 1993.

[BK96] Brandt S. and Knudsen J. L. "Generalising the BETA Type System". In Cointe [Coi96], pages 421–448.

[Bor84] Borne I. ´Micro-Smalltalkˇ. In *BIGRE 41*, pages 30–51. 1984.

[Bor96] Borron H. J. "Colored-Object Programming: Color Graphs, a Visual Formalism for Synthesizing the Behaviour of Objects". Tech. report 2876, Inriathèque, April 1996.

[Bou94] Bourdin C. ´Points de vue et représentation multiple et évolutive dans les langages à objetsˇ. Rapport tech.599587, Université de Montpellier II, 28 Juin 1994.

[Bou96] Bourgeois W. *W-Icon : un langage de prototypage pour interfaces graphiques*. Thèse de doctorat, spécialité Informatique, Université de Nice – Sophia Antipolis, 8 Novembre 1996, 220 pages.

[BW89] Barsalou T. and Wiederhold G. "Knowledge-directed Mediation between Application Objects and Base Data". In *?* 1989.

[Cav97] Cavarroc J. ´Utilisation d'un méta-protocole objet pour une bibliothèque de framesˇ. Rapport tech., Université de Nice – Sophia Antipolis, Juin 1997.

[CCL96] Chignoli R., Crescenzo P. and Lahire P. "An Extensible Environment for Views in Eiffel". Tech. report 96-53, I3S – UPRESA CNRS 6070 – Université de Nice – Sophia Antipolis, November 1996, 15 pages.

[CCL97] Chignoli R., Crescenzo P. et Lahire P. ´Liens entre classes dans les langages à objetsˇ. Rapport tech.97-22, I3S – UPRESA CNRS 6070 – Université de Nice – Sophia Antipolis, Décembre 1997, 26 pages.

[CDGL95] Calvanese D., De Giacomo G. and Lenzerini M. "Structured Objects: Modeling and Reasoning". In *Deductive and Object-Oriented Databases (DOOD'95)*, vol. 1013 of *Lecture Notes in Computer Science*, pages 229–246. Springer-Verlag (Berlin), 1995.

[Chi95] Chiba S. "A Metaobject Protocol for *C++*". In ACM [WB95], pages 285–299. ACM SIGPLAN Notices, vol. 30, num. 10.

[Chi98a] Chiba S. *OpenC++ 2.5 Reference Manual*. Institute of Information Science and Electronics, University of Tsukuba, 1998.

[Chi98b] Chiba S. "OpenC++ tutorial". Tech. report , Institute of Information Science and Electronics, University of Tsukuba, 1998.

[CL92] Capretz L. F. and Lee P. A. "Reusability and Life Cycle Issues Within an Object-Oriented Methodology". In Ege [Ege92], pages 139–150.

[CL94] Chen J. and Lin X. "Clustering Classes Thorough Graph Transformation". In *?*, pages 1–14. 1994.

[CL96] Chen J.-b. and Lee S. C. "Generation and reorganization of subtype hierarchies". *Journal of Object Oriented Programming*, vol. 8, num. 8 (January 1996), pages 26–35.

[CMPS97] Clavel G., Mirouze N., Pichon E. et Soukal M. *Java : La synthèse*. (Paris) I. (ed.) . Masson (Paris), 2-7296-0656-4, Juillet 1997, 219 pages.

[Coi87] Cointe P. "The *ClassTalk* System: a Laboratory to Study Reflection in *Smalltalk*". In Nardi P. M. . D. (ed.) , *Informal Proceedings of the First Workshop on Reflection*, pages 155–176. 1987.

[Coi96] Cointe P. (ed.) . *European Conference on Object-Oriented Programming (ECOOP'96)*, vol. 1098 of *Lecture Notes in Computer Science*, Linz, Austria. Springer-Verlag (Berlin), 9–12 July 1996.

[Col97] Collet P. *Un modèle fondé sur les assertions pour le génie logiciel et les bases de données : application au langage OQUAL, une extension d'Eiffel*. Thèse de doctorat, spécialité Informatique, Université de Nice – Sophia Antipolis, 15/12/1997, 242 pages.

[Coo91] Cook S. "Object-Oriented Languages Compared". In Bezivin [Bez91], pages? Tutorial.

[DBFPD94] Ducasse S., Blay-Fornarino M. and Pinna-Dery A.-M. "Embedding behavioral relationships between objects using computationnal reflection". Tech. report RR 94-60, I3S – URA CNRS 1376 – Université de Nice – Sophia Antipolis, December 1994.

[ddCdlL93] ÉDITIONS DU CERCLE DE LA LIBRAIRIE (ed.) . *Mémento Typographique.* Hachette, 2-7654-0447-X, Janvier 1993, 121 pages.

[DDF95] DERY A.-M., DUCASSE S. AND FORNARINO M. "Control and PAC model". Tech. report RR 95-03, I3S – URA CNRS 1376 – Université de Nice – Sophia Antipolis, February 1995.

[Deb94a] DEBRAUWER L. ´L'héritage multipleˇ. http://www.netinfo.fr/objectland/Langages/-n6.94/HeritageMultiple.html, Juin 1994.

[Deb94b] DEBRAUWER L. ´La méthode de modélisation objet OMTˇ. http://www.netinfo.-fr/objectland/Langages/n9.94/OMT.html, Septembre 1994.

[Den96] DENNEBOUY Y. (ed.) . *Langages et Modèles à objets (LMO'96)*, Leysin (Suisse), 16–18 October 1996. EPFL, Laboratoire de Bases de Données, 1996, 274 pages.

[DF93] DUCASSE S. ET FORNARINO M. ´Protocole pour la gestion des dépendances entre objets grâce au contrôle des fonctions génériquesˇ. Rapport tech.RR 93-62, I3S – URA CNRS 1376 – Université de Nice – Sophia Antipolis, Octobre 1993.

[DHH+95] DUCOURNAU R., HABIB M., HUCHARD M., MARIE-LAURE M. ET NAPOLI A. ´Le point sur l'héritage multipleˇ. *Technique et science informatiques*, vol. 14, n° 3 (1995), pages 309–345.

[dlC93] DE LA COMMUNICATION F. C. (ed.) . *Code Typographique.* Fédération C.G.C. de la Communication, 2-9507157-0-2, 17, Février 1993, 120 pages.

[dT93] DES TYPOGRAPHES A. S. (ed.) . *Guide du Typographe Romand.* AST, 5, 1993, 213 pages.

[DT95] DOBBIE G. AND TOPOR R. "Resolving Ambiguities caused by Multiple Inheritance". In *Deductive and Object-Oriented Databases (DOOD'95)*, vol. 1013 of *Lecture Notes in Computer Science*, pages 265–280. Springer-Verlag (Berlin), 1995.

[Duc93] DUCASSE S. ´Méta Protocole pour l'expression de dépendances entre objetsˇ. Rapport tech., Université de Nice – Sophia Antipolis, Juin 1993.

[Duc95a] DUCASSE S. "Introduction of a Reified Inheritance Mechanism by means of First Class Object Dependencies in a Reflective Language". Tech. report RR 95-28, I3S – URA CNRS 1376 – Université de Nice – Sophia Antipolis, July 1995.

[Duc95b] DUCASSE S. "Inheritance Mechanism Reification by Means of First Class Object". Tech. report RR 95-12, I3S – URA CNRS 1376 – Université de Nice – Sophia Antipolis, March 1995.

[Duc96] DUCASSE S. ´De la variété du traitement des relations dans les langages à objetsˇ. Rapport tech., I3S – URA CNRS 1376 – Université de Nice – Sophia Antipolis, 1996.

[dVdV93] DE VIVO G. O. AND DE VIVO M. "A Pragmatic Approach to C++, Eiffel and Ada 9X Programming.". In *?* 1993.

[dVdVG95] DE VIVO M., DE VIVO G. AND GONZALES L. "A Brief Essay on Capabilities". In *?* 1995.

[Ege92] EGE R. (ed.) . *International Conference on Technology of Object-Oriented Languages and Systems (Tools 8, USA'92)*, Santa Barbara (Cal.). Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-042441-2, 3–5 August 1992, 316 pages.

[Ege93] EGE R. (ed.) . *International Conference on Technology of Object-Oriented Languages and Systems (Tools 11, USA'93)*, Santa Barbara (Cal.). Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-103979-2, 2–5– August 1993, 639 pages.

[Ewi97] EWING G. "Eiffel Liberty". http://www.elj.com/elj/v1/n1/gew/, November 1997.

[F.97] F. R. *Évolution du comportement des objets dans les langages à classes réflexifs.* Thèse de Doctorat en Informatique, Université de Nantes, 1997.

[FC98] FABRE J.-C. AND CHIBA S. (eds.) . *Workshop on Reflective Programming in C++ and Java.* Center for Computational Physics, University of Tsukuba, Japan, 1998. UTCCP Report 98-4; ISSN: 1344-3135.

[Fir94] FIRESMITH D. G. "Inheritance diagrams: Which way is up?". *Journal of Object Oriented Programming*, vol. 7, num. 1 (April 1994), pages 10–16.

[Foo93] FOOTE B. "Workshop on Object-Oriented Reflection and Metalevel Architectures". In Paepcke [Pae93], pages 123–126. ACM SIGPLAN Notices, vol. 28, num. 10.

[GC96] Gowing B. and Cahill V. "Meta-Object Protocols for *C++*: The *Iguana* Approach". In Kiczales [Kic96].

[GGL93] G. G. R., G. B. D. and L. W. J. "Clos in context - The shape of the design space". *Journal of Object Oriented Programming*, (1993).

[GK98] Golm M. and Kleinöder J. "*metaXa* and the Future of Reflection". In Fabre and Chiba [FC98]. UTCCP Report 98-4; ISSN: 1344-3135.

[GM96] Gawecki A. and Matthes F. "Integrating Subtyping, Matching and Type Quantification: A practical Perspective". In Cointe [Coi96], pages 26–47.

[Gro95] Group O. M. "The Common Object Request Broker: Architecture and Specification", 1995. revision 2.0.

[GSV+96] Gomes B., Stoutamire D., Vaysman B., Klawitter H. and Feldman J. "A Language Manual for Sather 1.1 and pSather 1.1". Tech. report , International Computer Science Institute, Berkeley, 1996.

[Gué97] Guégan R. ´La modélisation objet avec UMLˇ. `http://stm.tj/objet/modelisation_avec_uml.html`, 1997.

[Kic96] Kiczales (ed.) . *Reflexion'96*, San Francisco, California. 4/1996.

[KM91] Kitchel S. W. and Martin N. L. "Using Descriptor Classes in Object-Oriented Systems". In Korson and Vaishnavi [KV91], pages 167–178.

[KR97] Kernighan B. et Ritchie D. *Le langage C Norme ANSI.* (Paris) M. (ed.) . Masson (Paris), 2225830355, deuxième édition, 1997, 296 pages.

[KV91] Korson T. and Vaishnavi V. (eds.) . *International Conference on Technology of Object-Oriented Languages and Systems (Tools 5, USA'91)*, Santa Barbara (Cal.). Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-923178-1, 29 July – 1 August 1991, 485 pages.

[Lai97] Lai M. *UML : La notation de modélisation objet : Applications en Java.* (Paris) I. (ed.) . Masson (Paris), 2-225-82844-X, Juillet 1997, 250 pages.

[Led99] Ledoux T. ´OpenCorba: un bus réflexif adaptableˇ. In Malenfant et Rousseau [MR99], pages 9–23.

[LHQ94] Lawson T., Hollinshead C. and Qutaishat M. "The Potential For Reverse Type Inheritance in Eiffel". `http://www.cs.cf.ac.uk/CLE/Abstracts.html`, March 1994.

[Lie86] Lieberman H. "Delegation and Inheritance : Two Mechnisms for sharing Knowledge in Object-Oriented Systems". In *BIGRE + GLOBULE 1986*, pages 79–89. January 1986.

[LM95] Lehrmann Madsen O. "Open Issues in Object-oriented Programming - A Scandinavian Perspective". *Software, Practice & Experience*, vol. 25, num. S4 (December 1995), pages 3–43.

[Loo91] Loomis M. "Object-Oriented Databases". In Bezivin [Bez91], pages? Tutorial.

[LR96] Louvet P. and Ridoux O. "Parametric Polymorphism for Typed Prolog and λProlog". In *Programming Languages: Implementations, Logics and Programs (PLILP'96)*, pages 47–61. September 1996.

[Mag94] Magnusson B. (ed.) . *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 13, Europe'94)*, Versailles (France). Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-350539-1, 7–11 March 1994, 550 pages.

[Men95] Menzies T. "ISA Object PARTOF Data Modeling?". In *?*, pages 1–9. 27 April 1995.

[Mey91] Meyer B. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-247925-7, 1991, 596 pages. Second revised printing, 1992; Traduction française par Bruno Terseur et Chantal Saint-Cast "Eiffel, le langage", InterÉdition (Paris), 1994, 2-7296-0525-8, 560 p.

[Mey95] Meyer B. "Static Typing and Other Mysteries of Life". In *Web-adapted Keynote Lectures at OOPSLA'95 and TOOLS PACIFIC'95*, pages 1–17. December 1995. from `http://www.eiffel.com`, and also in "Object Currents", Vol. 1, No 1, at `http://www.sigs.com/objectcurrents/oc9601.toc.html`.

[Mey96] MEYER B. "Draft fragments from *Object-Oriented Software construction*, second edition", 1996. from `http://www.eiffel.com/`.

[Mey97] MEYER B. *Object-Oriented Software construction.* The O-O series. Prentice Hall Inc. (Englewood Cliffs, NJ), 0-13-629155-4, 2nd, 1997, 1254 pages.

[MLM96] MATEOS-LAGO J. AND MARIO R.-A. "GOTA Algebras: A Specification Formalism for Inheritance and Object Hierarchies". In *Programming Languages: Implementations, Logics and Programs (PLILP'96)*, pages 62–76. September 1996.

[MR99] MALENFANT J. ET ROUSSEAU R. (eds.) . *Langages et Modèles à objets (LMO'99)*, Villefranche-sur-Mer (France), 27–29 Janvier 1999. I3S, OCL, Hermes (Paris), 1999, 296 pages.

[Mul95] MULET P. *Reflection and Prototypes.* Thèse de Doctorat, Université de Nantes, 1995.

[Mur97] MURRAY K. E. "Generalization and Specialization". *Journal of Object Oriented Programming*, vol. 10, num. 4 (August 1997), pages 91–96.

[Nat93] NATIONALE I. (ed.) . *Lexique des Règles typographiques en usage à l'Imprimerie Nationale.* Imprimerie Nationale, 2-11-081075-0, 5, Février 1993, 197 pages.

[NIC98] NICE. "Eiffel Object Model". `http://www.objs.com/x3h7/eiffel.htm`, 1998.

[NM94] NAJA H. ET MOUADDIB N. ´Un modèle pour la représentation multiple dans les bases de données orientées-objet.ˇ. In *?*, pages 173–189. 1994.

[OB98a] OLIVA A., AND BUZATO L. "The Implementation of *Guaraná* on *Java*". Tech. report IC-98-32, Instituo des Computaçaos, Universidade Estadual de Campinas, 1998.

[OB98b] OLIVA A. AND BUZATO L. "Composition of meta-objects in *Guaraná*". In Fabre and Chiba [FC98]. UTCCP Report 98-4; ISSN: 1344-3135.

[OB98c] OLIVA A. AND BUZATO L. "An Overview of *MOLDS*: a Meta-Object Library for Distributed Systems". Tech. report IC-98-15, Instituo des Computaçaos, Universidade Estadual, 1998.

[OGB98] OLIVA A., GARCIA I. AND BUZATO L. "The Reflective Architecture of *Guaraná*". Tech. report IC-98-14, Instituo des Computaçaos, Universidade Estadual de Campinas, 1998.

[Olt95] OLTHOFF W. (ed.) . *European Conference on Object-Oriented Programming(ECOOP'95)*, vol. 952 of *Lecture Notes in Computer Science*, Aarhus, Danemark. Springer-Verlag (Berlin), 7–11 August 1995.

[Omo93] OMOHUNDRO S. "The Sather Language: Efficient, Interactive, Object-Oriented Programming". *Dr. Dobbs Journal*, (1993).

[Pae91] PAEPCKE A. (ed.) . *6th Annual ACM Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'91)*, Phoenix, Arizona, 6–11 October 1991. 0-201-55417-8, November 1991, 365 pages. ACM SIGPLAN Notices, vol. 26, num. 11.

[Pae93] PAEPCKE A. (ed.) . *8th Annual ACM Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'93)*, Washington, (DC, USA), 26 September – 1 October 1993. October 1993. ACM SIGPLAN Notices, vol. 28, num. 10.

[Per95] PERROUSSEAUX Y. *Manuel de Typographie Française Élémentaire.* PERROUSSEAUX A. (ed.) . Yves Perrousseaux, 2-911220-00-5, 2, Octobre 1995, 126 pages.

[PJP95] PAZZAGLIA J.-C. ET JEAN-PIERRE R. ´Vers un outils d'étude dynamique des programmes acteursˇ. Rapport tech.RR 95-07, I3S – URA CNRS 1376 – Université de Nice – Sophia Antipolis, Mars 1995.

[Poi98] POISSON M.-A. ´Conception et développement orientés objet (OO)ˇ. `http://pages.-infinit.net/map/oo.html`, 1998.

[QKB96] QIAN Z. AND KRIEG-BRÜCKNER B. "Types Object-Oriented Functional Programming with Late Binding". In Cointe [Coi96], pages 48–72.

[Rif93] RIFFLET J.-M. *La programmation sous Unix.* Ediscience international, 2-84074-013-3, 3e , 1993, 630 pages.

[RJFW96] Rossie Jr. J. G., Friedman D. P. and Wand M. "Modeling Subobject-Based Inheritance". In Cointe [Coi96], pages 248–274.

[RLVG98] Raverdy P. G. and Le Van Gong H. anf Lea R. "*DART*: A reflective middleware for adaptative applications". In Fabre and Chiba [FC98]. UTCCP Report 98-4; ISSN: 1344-3135.

[RM96] Rinat R. and Magidor M. "Metaphoric Polymorphism: Taking Code Reuse One Step Further". In Cointe [Coi96], pages 449–471.

[Rol95] Rolland C. *LaTeX: guide pratique*. Addison-Wesley France S. P. (ed.) . Addison-Wesley France, SA (Paris), 2-87908-104-1, Juin 1995, 345 pages.

[SdSSD94] Souza dos Santos C., Serge A. and Delobel C. "Virtual Schemas and Bases". In *?*, pages 81–94. 1994.

[Sei96] Seidewitz E. "Controlling inheritance". *Journal of Object Oriented Programming*, vol. 8, num. 8 (January 1996), pages 36–42.

[Ser85] Serlet B. "Object-Oriented Programming in Cedar". In *Journées Langages Orientés Objet*, pages 64–68. 1985.

[SK95] Stoutamire D. and Kennel M. "Sather Revisited: A High-Performance Free Alternative to C++". *Commputers in Physics*, vol. 9, num. 5 (September – October 1995), pages 519–524.

[SKM94] Saji N., Kageyama T. and Motohide T. "C++ Metaobject on CLOS MOP". In Wilpolt [Wil94]. ACM SIGPLAN Notices, vol. 29, num. 11.

[SLJ94] Suppiah A., Lawson T. and Jones A. "Expressing and Processing Constraints in Eiffel". In Magnusson [Mag94], pages 337–347.

[SLT91] Scholl M. C., Laasch C. and Tresch M. "Updatable Views in Object-Oriented Databases". In *?*, pages 189–207. 1991.

[SO91] Schmidt H. W. and Omohundro S. M. "CLOS, Eiffel, and Sather: A Comparison". `http://www.icsi.berkeley.edu/ sather/Publications/tr-91-047.html`, September 1991.

[SO96] Stoutamire D. and Omohundro S. "The Sather 1.1 Specification". Tech. report , International Computer Science Institute, Berkeley, August 1996, 93 pages.

[Swi93] Switzer R. *Eiffel, an introduction*. Prentice Hall Inc. (Englewood Cliffs, NJ), 1993, 176 pages.

[TN95] Tsukamoto M. and Nishio S. "Inheritance Reasoning by Regular Sets in Knowledge-bases with Dot Notation". In *Deductive and Object-Oriented Databases (DOOD'95)*, vol. 1013 of *Lecture Notes in Computer Science*, pages 247–264. Springer-Verlag (Berlin), 1995.

[TO96] Talens G. et Oussalah C. ´Version d'objets pour l'ingénierieˇ. *Technique et science informatiques*, vol. 15, n° 2 (1996), pages 145–178.

[WB95] Wirfs-Brock R. (ed.) . *10th Annual ACM Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'95)*, Austin (Texas, USA), 15–19 October 1995. November 1995. ACM SIGPLAN Notices, vol. 30, num. 10.

[Wie93] Wierczerzycki W. "Software Reusability through Versions". *Software, Practice & Experience*, vol. 26, num. 8 (August 1993), pages 911–927.

[Wie94a] Wiener R. S. "C++ and Eiffel (is it C++ and Eiffel?) : A Clash of Culture". In Magnusson [Mag94], pages 1–53. Tutorial TA3.

[Wie94b] Wiener R. S. "Comparison of O-O languages". In Magnusson [Mag94], pages? Tutorial MM1.

[Wil94] Wilpolt C. (ed.) . *9th Annual ACM Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'94)*, Portland, (OR, USA), 23–27 October 1994. November 1994. ACM SIGPLAN Notices, vol. 29, num. 11.

[Z98] Z W. "*Reflexive Java* and a Reflexive Component-Based Transaction Architecture". In Fabre and Chiba [FC98]. UTCCP Report 98-4; ISSN: 1344-3135.