DyReCT :
un C++ dynamique et réflexif pour les bases de données

Olivier Jautzy

Janvier 1997

Rapport de Recherche CERMICS N^o 97-89

DyReCT : un C++ dynamique et réflexif pour les bases de données

Olivier Jautzy
Équipe « Base de données » - CERMICS
INRIA, 2004 route des Lucioles 06902 Sophia Antipolis

e-mail : Olivier.Jautzy@sophia.inria.fr
web : http://www.inria.fr/cermics/dbteam/Olivier.Jautzy/

Résumé

Dans le cadre de la réalisation d'un serveur multimodèles de bases de données hétérogènes, nous nous intéressons à la définition de protocoles à métaobjets capables de faciliter la migration d'instances, la distribution et la persistance d'un graphe d'objets. Dans cet article, nous montrons dans quelle mesure il est possible de réaliser un tel protocole pour *C++*, l'un des langages de programmation les plus utilisés actuellement. Notre approche a été de définir de nouvelles modélisations permettant de spécialiser l'envoi de messages, de faire migrer les instances d'une classe à une autre et de modifier les classes dynamiquement à l'exécution. Ces modélisations se sont concrétisées par la réalisation d'un précompilateur, *DyReCT*. Celui-ci doit permettre de distribuer ou de rendre persistant un graphe d'objets de façon transparente pour l'utilisateur.

Abstract

As part of the realization of a multimodel server for heterogeneous databases, we need to define metaobject protocols that enable instance migration, distribution or persistence of a graph of objects. In this paper,we present a methodology to design such a protocol for *C++*, one of the most widely used programming language. Our approach is to define new modelizations that allow instances migration, method call specialisation and run-time class modification. As a sample product of these modelisations, this paper presents a precompiler named *DyReCT*. It enables the programmer to transparently distribute or make persistent a graph of *C++* objects.

1 Introduction

La prolifération des systèmes de gestion de bases de données, basés sur des modèles de données et des langages de requêtes hétérogènes, empêche les utilisateurs, familiers de l'un des systèmes, d'utiliser les données stockées dans d'autres systèmes. La définition d'outils proposant une interface uniforme, permettant de manipuler dans le modèle de l'utilisateur des données issues de bases de données hétérogènes, peut être simplifiée par l'utilisation de langages de programmation autorisant la migration d'instances, la persistance et la distribution d'un graphe d'objets. De tels langages incluent le plus souvent des protocoles à métaobjets (M.O.P.) [Dem96, DL95]. Or, très peu de M.O.P sont actuellement proposés pour *C++* [Str92] , le standard de fait des langages de développement.

Dans cet article, nous proposons un tel protocole pour *C++*. Pour ce faire, nous commencerons par étudier les différentes approches permettant de définir un protocole à métaobjets au dessus de *C++*. Puis, nous présenterons notre système *DyReCT* [Jau96] rendant *C++* réflexif et dynamique et nous verrons comment cet outil peut répondre à nos besoins en termes de bases de données (migration d'instances, persistance et distribution d'un graphe d'objets). Enfin, nous le comparerons à d'autres outils analogues.

2 Un protocole à métaobjets pour *C++* : trois approches possibles

Avant de décrire les différentes approches possibles pour ajouter un M.O.P. à *C++*, commençons par définir la notion de protocole à métaobjets.

Les métaclasses sont apparues pour la première fois dans *Smalltalk* [GR80].Elles ont été introduites dans le monde de la programmation à objets pour résoudre les problèmes induits par la différence de comportement entre classes et objets. Elles sont des classes de classes. En d'autres termes, les classes sont instances des métaclasses. Cette nouvelle "couche" d'entités induit une uniformisation du langage. En effet, les classes deviennent à leur tour des instances, et les métaclasses sont aussi instances d'autres métaclasses. Les langages ainsi définis sont donc totalement objets car toutes les entités manipulées sont des instances.Mis à part l'intérêt théorique de cette uniformité, l'ajout de métaclasses permet à l'utilisateur l'introspection (i.e. interrogation, visualisation) et/ou la spécialisation du comportement des classes à l'exécution du programme. Dans le dernier cas, la distribution [GV90, OV91] et la persistance [ABC+83] d'un graphe d'objets sont grandement facilitées.
Les langages incluant des métaclasses sont appelés langages à métaobjets. De plus, l'ajout de métaclasses implique la définition d'un modèle structurant ces nouvelles entités, ce qui définit un protocole. Ainsi parle-t-on de Protocoles à MétaObjets (M.O.P.).

Aujourd'hui, les protocoles à métaobjets ont principalement été définis sur des langages tels que *LISP*, qui sont aisément spécialisables, ce qui facilite ainsi leur implantation. Des réalisations tels qu'*ObjVLisp* [Coi87] ou *CLOS* [KdRB91] forment la référence en matière de protocoles à métaobjets. A l'inverse, peu de M.O.P ont été définis pour *C++*, ceci étant dû principalement à la nature statique intrinsèque de ce langage. Cependant, celui-ci étant l'un des langages de programmation les plus utilisés actuellement, il est intéressant de vouloir lui ajouter un M.O.P..

Pour ce faire, il est possible de distinguer trois approches:

- La première est l'implantation de bout en bout d'un compilateur [LGLR92]. Cette solution permet de structurer notre modèle comme nous le souhaitons (par exemple, comme celui d'*ObjVLisp*). Or, le succès de *C++* étant dû en grande partie à l'excellente gestion des erreurs et à l'optimisation du code, ces outils doivent donc être implantés dans le compilateur sous peine que cette solution n'ait que peu d'intérêt.
- La seconde est la réutilisation d'un outil existant (par exemple *CLOS*) et l'ajout d'une passerelle entre *C++* et le langage de base de l'outil (par exemple *Common LISP*) [SKT94]. De la sorte, on obtient indirectement un protocole à métaobjets pour *C++*. Cependant, cette approche induit la perte de l'optimisation du code, efficacement réalisée par les compilateurs *C++*, puisqu'on ne définit qu'une passerelle entre *C++* et *LISP*. Enfin, la structure du est ici contrainte par le choix de l'outil existant.
- Enfin, la dernière est la réalisation d'un précompilateur [Chi93a, Chi95], c'est à dire la création d'un outil transformant un *C++* étendu, contenant un M.O.P., en un *C++ standard*. Ceci permet d'utiliser un compilateur *C++ usuel* comportant une gestion puissante des erreurs et l'optimisation du code. Cependant, après la pré-compilation, plus aucun M.O.P. n'existe puisque l'on génère un code *C++* classique sans métaclasse. On dit alors qu'on ne dispose plus que d'un "pseudo-M.O.P." avec des "pseudo-métaclasses". Pour la même raison, cette approche nous oblige à conserver la structure des classes *C++*, contraignant par la même la structure de notre modèle.

Ces trois approches ont des avantages et des inconvénients. Pour notre part, nous avons choisi la dernière car elle nous permet de réutiliser au maximum les outils disponibles pour *C++*, tout en y ajoutant des fonctionnalités de méta-modélisation.

3 DyReCT : une exTension Dynamique et Réflexive de C++

Avant de présenter *DyReCT*, nous décrivons, dans les prochains paragraphes, *OpenC++* [Chi93a, Chi93b], une extension de *C++* permettant de spécialiser l'envoi de messages, puis des techniques de programmation rendant *C++* plus dynamique. Ces outils forment les briques de base de notre modélisation.

3.1 OpenC++

OpenC++ offre une extension de l'appel de méthode standard de *C++*. Il introduit de nouvelles classes - les pseudo-métaclasses - qui jouent le rôle de métaclasses. Lorsqu'un message est envoyé sur un objet (dans le but d'exécuter une méthode), ce message est transmis à une pseudo-metaclasse avant l'exécution de la méthode. En d'autres termes, les pseudo-métaclasses contrôlent l'envoi de message. De plus, comme le comportement des pseudo-métaclasses peut être modifié, l'utilisateur a la possibilité de spécialiser l'envoi de message.

Lorsque l'envoi de message sur une classe `MyClass` est contrôlé par une pseudo-métaclasse, cette classe est appelée classe réflexive (nous appellerons désormais le modèle d'*OpenC++*, modèle réflexif) et, dans le code généré par le précompilateur, son nom sera pré-fixé par `refl_` (cf figure 1).

Le programmeur spécifie la réflexivité d'une classe par la syntaxe :

```
//MOP reflect class MyClass : MyMetaclass;
```

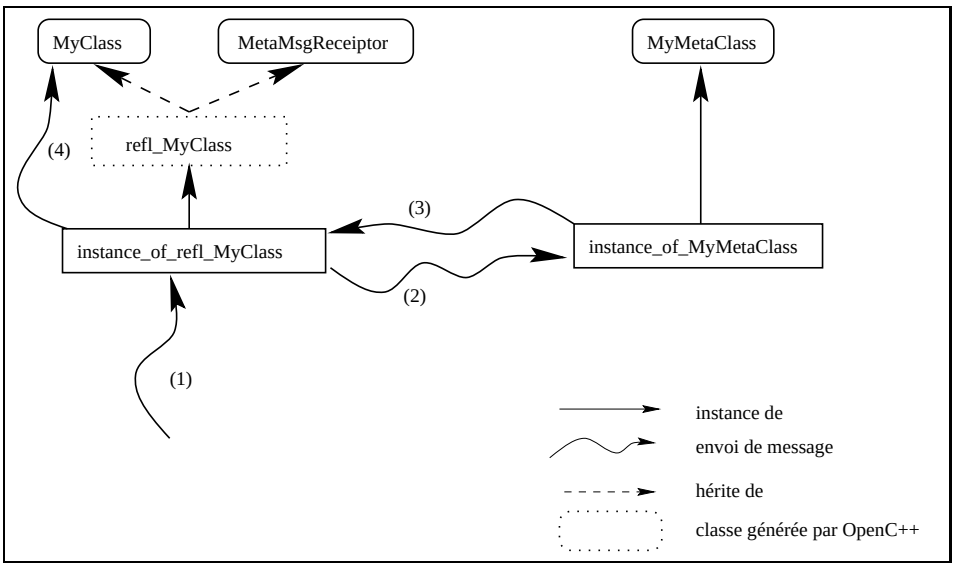

FIG. 1 – *Modèle d'OpenC++*

Une classe `refl_MyClass` est alors générée et chaque envoi de message sur cette classe est transmis à la pseudo-métaclasse `MyMetaclass`. L'utilisateur a alors le choix :

- soit utiliser sa propre classe (`MyClass`) sans contrôle de l'envoi de message,
- soit utiliser la classe générée (`refl_MyClass`), qui possède les même fonctionnalités que sa classe plus le contrôle de l'envoi de message.

OpenC++ laisse ce choix à l'utilisateur dans le but de limiter les baisses de performances dues à l'ajout de la gestion de l'envoi de message par les pseudo-métaclasses. Ainsi, le programmeur n'utilise les fonctionnalités étendues d'*OpenC++* que lorsque c'est nécessaire, c'est-à-dire seulement lorsqu'il y a nécessité de spécialiser l'envoi de message (par exemple, pour distribuer les instances d'une classe sur un réseau).

3.2 Dynamisme en C++

J.O. Coplien [Cop94] décrit différentes techniques de programmation permettant de rendre *C++* plus dynamique et plus proche des langages tels que *LISP*. Nous allons rapidement les passer en revue.

L'analogie Lettre/Enveloppe : Cette technique est basée sur l'utilisation conjointe de deux classes *C++* pour la représentation d'une seule classe. En d'autres termes, chaque classe décrite par l'utilisateur doit, par cette technique, être effectivement implantée sous forme de deux classes distinctes :

- Une classe externe, la classe enveloppe, en est la partie visible, l'interface externe associée à la classe que l'on veut créer. Cette classe est la seule à être manipulée par l'utilisateur. Elle ne contient aucune fonctionnalité. Elle contient la redéfinition de l'opérateur `->` de *C++* et chacune de ses instances contient un pointeur sur un objet de la classe lettre décrite ci-dessous. Cette structure permet de transmettre chaque message, envoyé à une instance de la classe enveloppe, sur l'objet associé, instance de la classe lettre.

– Une classe interne, la classe lettre, cachée aux yeux de l'utilisateur, contient toute les fonctionnalités de la paire classe enveloppe / classe lettre.

De fait, l'utilisateur manipule cette paire de classes comme une seule et même classe ayant les fonctionnalités décrites dans la classe lettre et une indirection décrite dans la classe enveloppe. Ceci permet notamment de faire migrer les instances. En effet, il est facile de changer l'objet pointé par la classe enveloppe sans modifier l'objet enveloppe, c'est à dire sans modifier l'identifiant d'objet.

Les exemplaires : Dans certains langages, les exemplaires sont des entités pouvant être clonées et modifiées pour créer des instances. Coplien introduit cette notion en *C++* de la manière suivante :

```
class Employe : public Class {
public :
    Employe();
    virtual Employe* Make(const char* name, EmployeId id);
private :
    Dollars salary;
    string name;
    EmployeId id;
    ....
}

static Employe employeExemplar ();
extern Employe *employe = &employeExemplar;
```

La méthode `make` joue le rôle du constructeur de sorte qu'il est possible de cloner de nouveaux employés à partir de l'employé statique `employe` qui représente en fait l'exemplaire associé à la classe `Employe`. Ceci se réalise par la syntaxe suivante :

```
Employe* Dupond = employe->Make("dupond","123456789");
```

Le principal avantage d'une telle technique est que, désormais, les constructeurs *C++* sont remplacés par des méthodes virtuelles. On peut donc utiliser tous les avantages des méthodes virtuelles pour construire des objet (principalement, discrimination à l'exécution et non à la compilation de l'objet sur lequel porte la méthode).

Modification des classes : Si un éditeur de lien incrémental est disponible, il est possible de définir une fonction qui permette d'ajouter du code à l'exécution du programme. Supposons qu'une telle fonction se nomme `load`, le programme suivant exécute la fonction dont le code se trouve dans un fichier nommé `incr.o` :

```
int main() {
   typedef void (*PF) (...);     // pointeur de fonction.
   PF anewfunc = (PF) load("incr.o");
   (*anewfunc)();
   return 0;
}
```

Par cette technique, il est donc possible de charger puis exécuter une fonction C. Pour charger une méthode C++, il faut quelques étapes supplémentaires : Chaque objet contient une table des méthodes virtuelles. Voici la structure d'une entrée de cette table, telle que définie dans le compilateur *C++* de la société Sun :

```
struct vt{
   short d;       // utilisé pour la gestion de l'héritage.
   short i;       // idem
   int (*f) ();   // pointeur sur une méthode virtuelle.
}
```

En conséquence, if faut, pour modifier une méthode virtuelle :

- trouver l'entrée de cette méthode dans la table des méthodes virtuelles,
- charger la méthode avec la fonction `load`,
- et modifier le champ `f` de la structure `vt` de sorte qu'il corresponde à la nouvelle méthode chargée.

En résumé, cette technique nous permet de modifier dynamiquement les méthodes virtuelles.

3.3 Deux nouvelles modélisations

Dans les sections précédentes, nous avons vu comment ajouter du dynamisme à *C++*, au prix d'un style de programmation très strict. Notre but est d'obtenir le même dynamisme en tant que fonctionnalités du langage. Pour ce faire, nous allons combiner *OpenC++* et les différentes techniques de programmation décrites ci-dessus.

A l'instar d'*OpenC++*, il est nécessaire, dans le but de limiter les baisses de performances, de laisser le choix à l'utilisateur des fonctionnalités dont il disposera pour chaque classe. Dans cette optique, *DyReCT* [Jau96] gère différents modèles :

- le **modèle de base** de *C++*;
- le modèle défini dans *OpenC++* ou **modèle réflexif**. Il ne permet que la spécialisation de l'envoi de message;
- un **modèle à classe simple** qui rajoute au modèle réflexif les exemplaires;
- Un **modèle à classes multiples** qui rajoute au modèle à classe simple la migration d'objets.

Le choix entre ces différents modèles s'effectue comme en *OpenC++* par la syntaxe :

```
//MOP <Keyword> class <class1, class2, ...> : <metaclass>, <optional name>
```

Cela signifie que la pseudo-métaclasse `metaclass` gère les classes `class1, class2, ...` par le modèle identifié suivant les règles suivantes :

- `Keyword = reflect` : **modèle réflexif**. Rappelons que seule les classes générées (i.e. `refl_classxx`) sont contrôlées par la pseudo-métaclasse.

– `Keyword` = `dynamic` : soit le **modèle à classe simple**, soit le **modèle à classes multiples**. Si `<optional name>` est spécifié, alors le modèle à classes multiples est choisi. Comme en *OpenC++*, seules les classes générées sont contrôlées par la pseudo-métaclasse. Dans le cas du modèle à classe simple, les classes générées sont préfixées de `dyn_` (et non plus de `refl_` comme dans le modèle réflexif). Enfin, dans le cas du modèle à classes multiples, une seule classe est générée. Elle est nommée `optional_name`.

3.3.1 Modèle à classe simple

Dans cette section, nous présentons, à travers un exemple, une modélisation qui rajoute au modèle réflexif la notion d'exemplaire.

Soit la classe suivante représentant un enfant :

```
class Child{
public:
  Child(int _age);
  virtual int GetAge();
  virtual void SetAge(int _age);
private:
  int age;
};
```

En ajoutant la ligne suivante, nous spécifions que la classe `Child` est gérée par la pseudo-métaclasse `Class` disponible dans les librairies fournies avec *DyReCT*.

```
//MOP dynamic class Child : Class;
```

La particularité de ce modèle est l'ajout des notions de lettre/enveloppe et d'exemplaire au dessus du modèle réflexif. La figure 2 nous montre les classes générées par *DyReCT* et comment elles sont liées les unes aux autres. La classe enveloppe est nommée `dyn_Child`, la classe lettre `letter_Child`. Cette dernière correspond en fait à la classe `refl_Child` générée par *OpenC++*. Pour utiliser les fonctionnalités de *DyReCT*, le programmeur doit utiliser la classe `dyn_Child` et, pour ce faire, il doit tout d'abord créer un exemplaire (`metaclasse` étant instance d'une métaclasse) :

```
class_dyn_Child exemplar=metaclasse->Make(``Child'');
```

La création d'objet se fait alors de la manière suivante :

```
dyn_Child child=exemplar->Make(10);
```

Et, finalement, l'appel de méthode se programme classiquement par :

```
child->SetAge(11);
```

Un tel appel de méthode suit alors le chemin indiqué par les numéros (1) à (5) sur la figure 2.

Ce modèle permet d'une part de bénéficier des "constructeurs virtuels" puisque il contient la notion d'exemplaire. D'autre part, une méthode capable de charger dynamiquement une méthode est implantée dans chaque classe enveloppe. Ce modèle introduit donc, de façon transparente pour

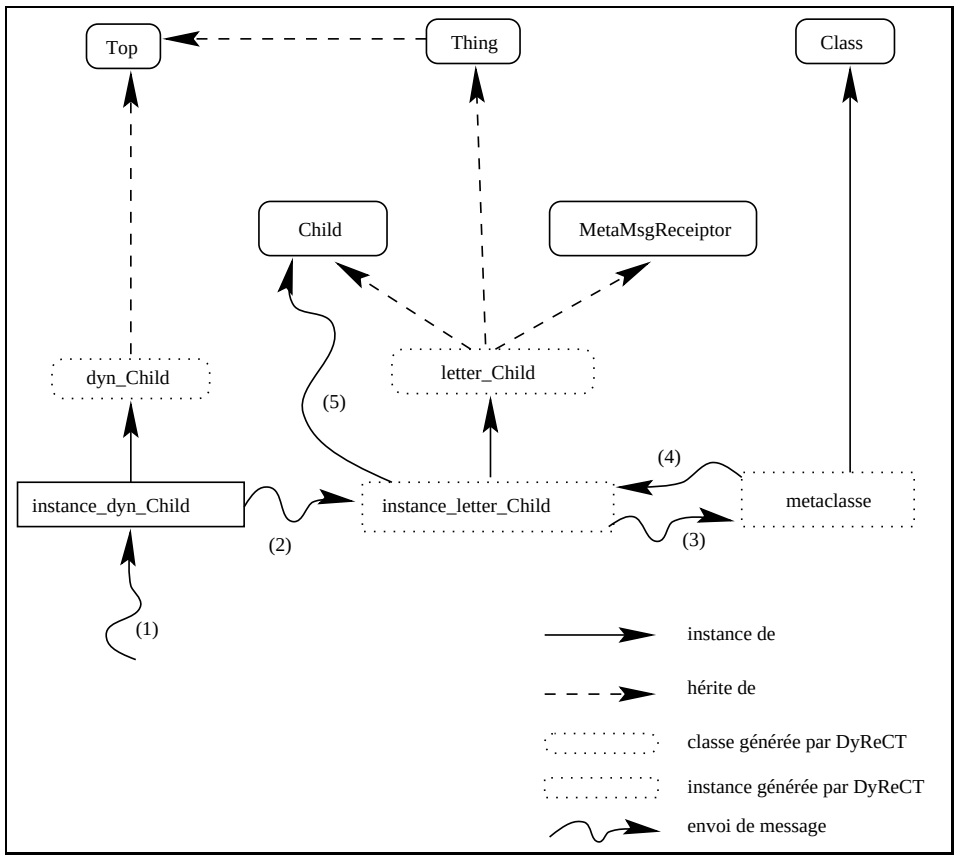

FIG. 2 – *Modèle à classe simple*

l'utilisateur, du dynamisme à *C++*.

Ce modèle est suffisant pour la majorité de nos besoins en termes de bases de données. En effet, le contrôle de l'envoi de message et les exemplaires permettent de réaliser efficacement distribution et persistance. Par exemple, une pseudo-métaclasse de distribution [Jau96] est inclue dans les librairies de *DyReCT*; l'utilisateur n'a qu'à spécifier le contrôle de ses classes par cette pseudo-métaclasse pour ensuite distribuer ses instances sur différents processus.

3.3.2 Modèle à classes multiples

Pour pouvoir faire migrer des instances, il nous faut cependant une étape supplémentaire. En effet, supposons que nous ayons décrit la classe suivante :

```
class Adult {
public:
//MOP dynamic:
    Adult(int _age,char* _job);
    Adult(Child *);
    virtual int GetAge();
    virtual int SetAge(int _age);
    virtual char* GetJob();
```

```
private:
    int age;
    char* job;
}
```

Et supposons que nous voulions transformer l'objet enfant en adulte. (par exemple si notre programme modélise l'évolution d'une personne au cours de sa vie). En *C++* standard, nous devrions programmer comme suit :

```
Adult adult(&child);
delete child;
// et dès lors utiliser l'objet adult;
```

Ceci n'est pas vraiment une bonne solution du fait que l'identifiant d'objet (OID) n'est pas conservé lors de l'évolution de l'objet.

Afin de réaliser une "véritable" migration d'objets, nous proposons d'utiliser *DyReCT*. Pour cela, nous devons tout d'abord déclarer que les classes `child` et `adult` sont gérées par le modèle à classes multiples de *DyReCT* :

```
//MOP dynamic class Child,Adult : Class, Person;
```

La figure 3 décrit les classes générées par *DyReCT* pour ce modèle. `MyClass` représente, dans notre schéma, soit la classe `Child`, soit la classe `Adult`. De plus, une classe lettre est générée pour chacune de ces deux classes (`letter_Child` et `letter_Adult`), elles sont ici représentées par la classe `letter_myClass`. Enfin, la classe `MainLetter_Person` est une classe de base pour `letter_Child` et `letter_Adult` et contient la définition de toutes les méthodes définies dans ces deux classes. Ainsi, par le mécanisme des méthodes virtuelles, un appel de méthode sur un objet de `MainLetter_Person` est automatiquement transmis à l'objet réellement instancié de la classe `letter_Child` ou `letter_Adult` (chemin 3).

Comme décrit précédemment, le programmeur utilise désormais seulement la classe `Person` comme interface externe, et ce pour les deux classes `Child` et `Adult`. Donc, après avoir créé un exemplaire de la classe `Person`, il pourra soit créer une instance pour la classe `Child`, soit une instance pour la classe `Adult` :

```
Person child=exemplar->Make(10);// construit un enfant
Person adult=exemplar->Make(23,"manager");// construit un adulte
```

DyReCT construit une méthode `Make` pour chaque constructeur défini dans les classes `Adult` et `Child`. Ceci implique que les signatures de chaque constructeur de la classe `Adult` et `Child` soient différentes afin de ne pas faire naître d'ambiguïtés.

Enfin, supposons que nous ayons créé une instance pour `Child` et que nous voulions la faire migrer vers `Adult`. La syntaxe est alors la suivante :

```
metaclass->Migrate(&child,"Person","Child","Adult");
```

De cette manière, la migration d'instance conserve l'identifiant d'objet.

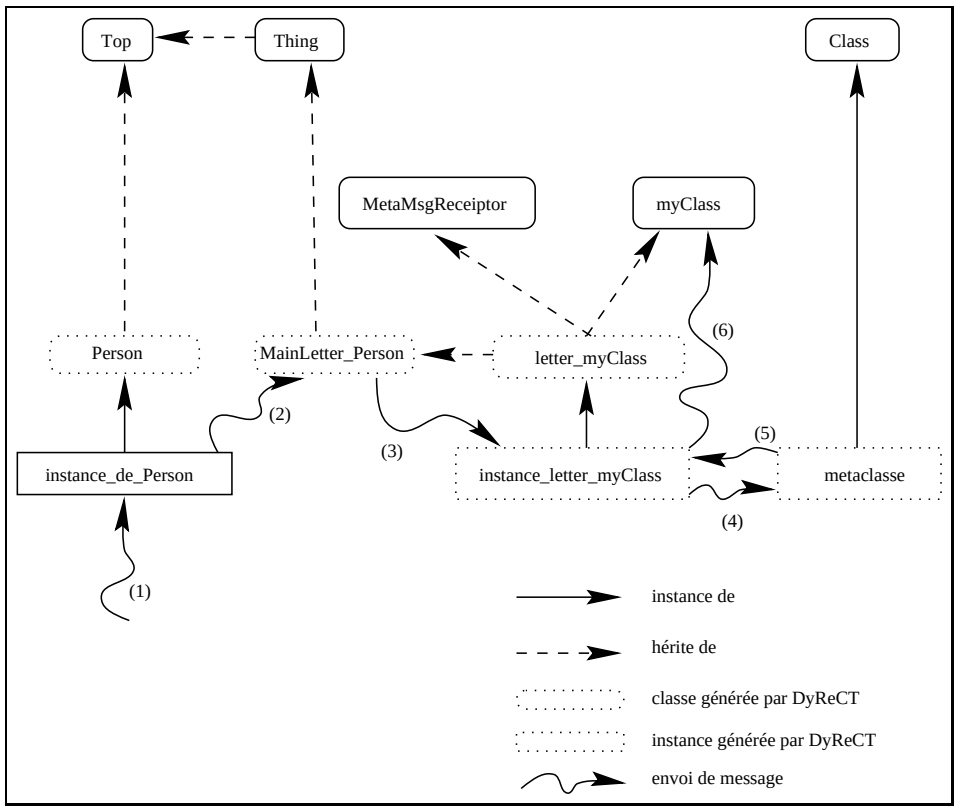

FIG. 3 – *Modèle à classes multiples*

4 Travaux connexes

Une nouvelle version d'*OpenC++* [Chi95] est en cours de développement. Elle n'implique plus aucune perte de performances car les pseudo-métaclasses n'existent plus qu'à la compilation. En fait, les pseudo-métaclasses ne contrôlent plus l'exécution du programme, mais la compilation. Dès lors, *OpenC++* facilite la personnalisation du langage. Cependant, le dynamisme que nous proposons dans *DyReCT* ne pourrait pas être obtenu dans cette nouvelle version car, comme il n'y a plus de méta-niveau, le comportement des classes ne peut plus être modifié à l'exécution.

Dans cet article, nous avons décrit trois approches distinctes permettant l'ajout d'un M.O.P. à *C++*. Nous avons choisi la troisième. Pour leur part, les concepteurs de *C++ Meta* [SKT94] ont choisi de travailler sur la seconde approche en réalisant une intégration de *C++* au sein de *CLOS*. Dans ce travail, le MOP de *CLOS* est étendu pour lui permettre de gérer des classes *C++*. De nouvelles métaclasses ont été définies, elles permettent la transformation de classes *C++* en classes *CLOS*. En conséquence, seule la réutilisation des classes *C++* est possible. On ne peut pas utiliser *C++ Meta* comme un *C++* auquel on aurait rajouter un protocole à métaobjets.

Enfin, une étude récente sur la parallélisation de *C++*, *C++//* [BFC+96] se base sur un protocole à métaobjets pour intégrer *SCHOONER*, un langage parallèle, à *C++*. A l'instar d'*OpenC++*, le MOP défini dans *C++//* ne permet que la spécialisation de l'envoi de message. Il ne peut donc réaliser ni de migrations d'instances ni de modifications de classes à l'exécution.

5 Conclusion

Dans cet article, nous avons montré dans quelle mesure il est possible de définir un protocole à métaobjets pour *C++*. Nous avons modélisé puis implanté *DyReCT* qui permet :

- de rendre persistants objets, champs et méthodes. En effet, le chargement dynamique des méthodes associé aux exemplaires sont les briques de base pour la réalisation d'une persistance en *C++* transparente pour l'utilisateur.
- de distribuer un graphe d'instances sur un réseau. En effet notre modélisation étant réflexive, elle permet de manipuler l'envoi de message facilitant une gestion masquée de la distribution.
- la migration d'instances "propre", c'est-à-dire sans modification de l'identifiant d'objet.

Il est certain que les nouvelles fonctionnalités de *DyReCT* induisent un surcoût en termes de performances. Néanmoins, une bonne gestion par le programmeur de ces diverses fonctionnalités, permet de rendre ce coût négligeable. Notamment, dans le domaine des bases de données distribuées, la perte de performance due à *DyReCT* est inférieure au temps de latence du réseau ou au temps d'accès aux données. Ainsi, si le programmeur n'utilise les fonctionnalités de *DyReCT* que pour gérer la persistance ou la distribution des données, le coût dû à *DyReCT* sera négligeable.

Perspectives : Nous travaillons actuellement sur plusieurs axes de recherche complémentaires. Tout d'abord, nous voulons permettre l'appel de méthodes à distance entre divers processus pouvant s'exécuter en parallèle et ceci de façon complètement transparente pour l'utilisateur. Pour cela, nous associons *DyReCT* à *PVM* [GBD+94], un outil de parallélisation de *C*. De plus, nous voulons ajouter les notions de dépendances à *C++*. Un modèle de dépendances réflexives a été défini par S. Ducasse [Duc97]. Nous étudions dans quelle mesure il est possible de porter cette modélisation dans *DyReCT*. Ce travail devrait pouvoir servir de base à l'inclusion de réflexes spécialisables pour les systèmes de gestion de bases de données à objets.

Références

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, 1983.

[BFC+96] F. Baude, N. Furmento, D. Caromel, R. Namyst, J.M. Geib, and J.F. Méhaut. C++// on top of PM2 via SCHOONER. In *STRATAGEM'96*, Sophia-Antipolis (France), Juillet 1996.

[Chi93a] S. Chiba. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *ECOOP 93 proceedings*, Kaiserslautern (Allemagne), juillet 1993.

[Chi93b] S. Chiba. Open C++ release 1.2 Programmer's guide. Technical report, Department of Science, university of Tokyo, 1993.

[Chi95] S. Chiba. A Metaobject protocol for C++. In *OOPSLA 95 proceedings*, Austin (Texas), October 1995.

[Coi87] P. Cointe. Metaclasses are First Class : the ObjVlisp model. In *OOPSLA'87 proceedings*, 1987.

[Cop94] J.O. Coplien, editor. *Advanced C++ : Programming Style and Idioms.* Addison-Wesley Publishing Company, New York (N.Y., U.S.A.), 1994. 520 pages.

[Dem96] S. Demphlous. Databases Evolution: An Approach by Metaobjects. In *Proceedings of the Third International Workshop on Advances in Databases and Information Systems ADBIS'96*, volume 1, pages 31–37, September 1996.

[DL95] S. Demphlous and F. Lebastard. Persistence of multiple object models. In *OOPSLA'95 Workshop on Metamodeling in Object-Orientation*, Austin (Texas, U.S.A.), Octobre 1995.

[Duc97] S. Ducasse. *Intégration réflexive de dépendances dans un modèle à classes.* Thèse de doctorat, Université de Nice-Sophia Antipolis, Janvier 1997. 245 pages.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, editors. *PVM: Parallel Vitual Machine - A users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, London (England), 1994.

[GR80] A. Goldberg and D. Robson, editors. *SMALLTALK-80, The language and its Implementation.* Addison Wesley, Reading (Massachusetts), 1980.

[GV90] G. Gardarin and P. Valduriez, editors. *SGBD Avancés.* Eyrolles, Paris (France), 1990. pages 1-23.

[Jau96] O. Jautzy. DyReCT : une extension dynamique et réflexive de C++. Mémoire de DEA, Université de Saint-Étienne, Saint-Étienne (France), Septembre 1996. 116 pages.

[KdRB91] G. Kiczales, J. des Rivières, and D. Bobrow, editors. *The Art of the Metaobject Protocol.* M.I.T Press, 1991. 200 pages.

[LGLR92] J. Lamping, G.Kickales, L.Rodriguez, and E. Ruf. An Architecture for an open Compiler. In *IMSA'92 Proccedings*, 1992.

[OV91] T. Ozsu and P. Valduriez, editors. *Principles of distibuted database systems.* Prentice-Hall International Editions, Englewood Cliffs (New-Jersey), 1991. pages 1-11.

[SKT94] N. Saji, T. Kageyama, and M Tajiri. C++ Metaobject on CLOS MOP. In *OOPSLA 94 Workshop on Multi-Language Object Models*, Août 1994.

[Str92] B. Stroustrup, editor. *Le langage C++: 2ème édition.* Addison Wesley, New York (New York, U.S.A.), 1992. 667 pages.

Liste des derniers rapports de recherche du CERMICS

List of previous CERMICS research reports

96-76	C. Buisson., J. P.Lebacque, J. B.Lesort	*Travel Times Computation for Dynamic Assignment Modelling.* 15 pages, novembre 1996, Noisy-Le-Grand
96-77	E. Cancès, C.Le Bris	*On the perturbation methods for some nonlinear quantum chemistry models.* 45 pages, novembre 1996, Noisy-Le-Grand
96-78	S. Depeyre	*Une méthode couplée pour la simulation d'écoulements disphasiques dispersés.* 57 pages, octobre 1996, Sophia-Antipolis
96-79	N.Glinsky-Olivier J.F.Gerbeau B.Larrouturou	*Semi-implicit Roe-type fluxes for low-mach number flows.* ...pages, octobre 1996, Sophia-Antipolis
96-80	*B. Jourdain*	*Propagation du chaos trajectorielle pour les lois de conservation scalaire.* 14 pages, décembre 1996, Noisy-Le-Grand
96-81	*B. Jourdain*	*Diffusions with a nonlinear drift coefficient and probabilistic interpretation of generalized Burger's equations.* 16 pages, décembre 1996, Noisy-Le-Grand
96-82	*D. Hirschkoff*	*Up-to context proofs for the π-calculus in the coq system.* 18 pages, janvier 1997, Noisy-Le-Grand

96-83	*E. V Abrarova,* *A. V. Karapetyan*	*Sur la stabilité et les bifurcations des mouvements stationnaires d'un corps solide dans un champ de gravitation central.* 22 pages, janvier 1997, Noisy-Le-Grand
96-84	*E. V Abrarova,*	*Sur les mouvements stationnaires en orbite d'un système de deux corps avec liaison élastique.* 20 pages, janvier 1997, Noisy-Le-Grand
97-85	J.P. Cioni	*Parallelisation of Maxwell and 3D simulations in electromagnetism using clusters of workstations.* 18 pages, janvier 1997, Sophia-Antipolis
97-86	B. Neveu G. Trombettoni	*Computational complexity of Multi-way, Dataflow constraint problems.* 13 pages, janvier 1997, Sophia-Antipolis
97-87	F. Lebastard	*Driver II (Lisp) : Manuel de référence.* 51 pages, janvier 1997, Sophia-Antipolis
97-88	E. Abrarova	*Non-integrability of the restricted three-body problem and geometrical obstacles to integrability.* 12 pages, janvier 1997, Noisy-Le-Grand
97-89	O. Jautzy	*Direct : un C++ dynamique et réflexif pour les bases de données.* 18 pages, janvier 1997, Sophia-Antipolis

Ces rapports peuvent être obtenus en s'adressant au secrétariat du CERMICS :
The reports can be asked from:

Imane HAMADE
ENPC-CERMICS
6 et 8 Avenue Blaise Pascal
Cité Descartes Champs-sur-Marne
77455-Marne-La-Vallée CEDEX 2
Tél : (33) 01 - 64-15-35-71
email: `hamade@newaphro.enpc.fr`