ObjeXion Software

**ObjeXion *Software* …**

**Prototyping made easy**

SA au capital de 500 000 F
Siret 421 565 565 00015
APE 722Z
Téléphone : 03 89 35 70 75
Télécopie : 03 89 35 70 76

**L'embarcadère**
**5, rue Gutemberg**
**68 800 Vieux-Thann, France**

**www.objexion.com**

# Développement d'un interpréteur *OCL* pour une machine virtuelle *UML*.

Frédéric FONDEMENT
Elève ingénieur - ESSAIM 3ème année
DRT GEII – 1ère année

Michel HASSENFORDER
Responsable DRT

Philippe STUDER
Directeur Technique - ObjeXion Software
Maître de Stage -

Pierre-Alain MULLER
Président Directeur Général – ObjeXion Software
Responsable de stage

# Sommaire



Frédéric FONDEMENT - Stagiaire essaim 3ème année – DRT GEII 1ère année
Philippe STUDER - Directeur Technique ObjeXion Software
Pierre-Alain MULLER - PDG ObjEXion SoftWare – Enseignant essaim responsable
Michel HASSENFORDER - Enseignant essaim – Responsable DRT

# Introduction

Avertissement : Etant intégré à un produit commercial, ce rapport ne présente que peu des solutions techniques adoptées pour des raisons évidentes de confidentialité. Cependant il s'agit aussi d'un travail de recherche, dont toutes les solutions seront présentées.

Créée en janvier 1999, ObjeXion Software SA est une entreprise informatique de haute technologie, dont l'objectif est de développer et de commercialiser une ligne de produits dans le domaine de la modélisation objet des systèmes d'information. Ses fondateurs, Pierre-Alain MULLER et Philippe STUDER, sont deux chercheurs ayant collaboré pendant une dizaine d'années au sein des laboratoires de recherche de l'Université de Haute-Alsace.

Prototyping Suite, le premier produit de l'entreprise consiste en un outil de prototypage des modèles objets, réelle attente des modeleurs de systèmes d'information. Ces modèles objets sont en fait représentés par des diagrammes de classes du langage de modélisation UML, défini par l'Object Management Group.

Récemment, l'OMG a intégré un langage de contraintes au sein d'UML. Ce langage permet notamment d'explorer intégralement un modèle de classes UML. Ma mission consistait donc à intégrer à Prototyping Suite un interpréteur OCL.

Nous verrons successivement chacune de ces parties :

- Description de Prototyping Suite
- Description d'OCL
- La réalisation et l'Intégration dans le logiciel

UML sera supposé connu. Je vous suggère donc de vous renseigner sur le site de l'OMG (http://www.omg.org), ou de lire un traité tel que *Modélisation objet avec UML* de Pierre-Alain MULLER et Nathalie GAERTNER.

# Prototyping Suite

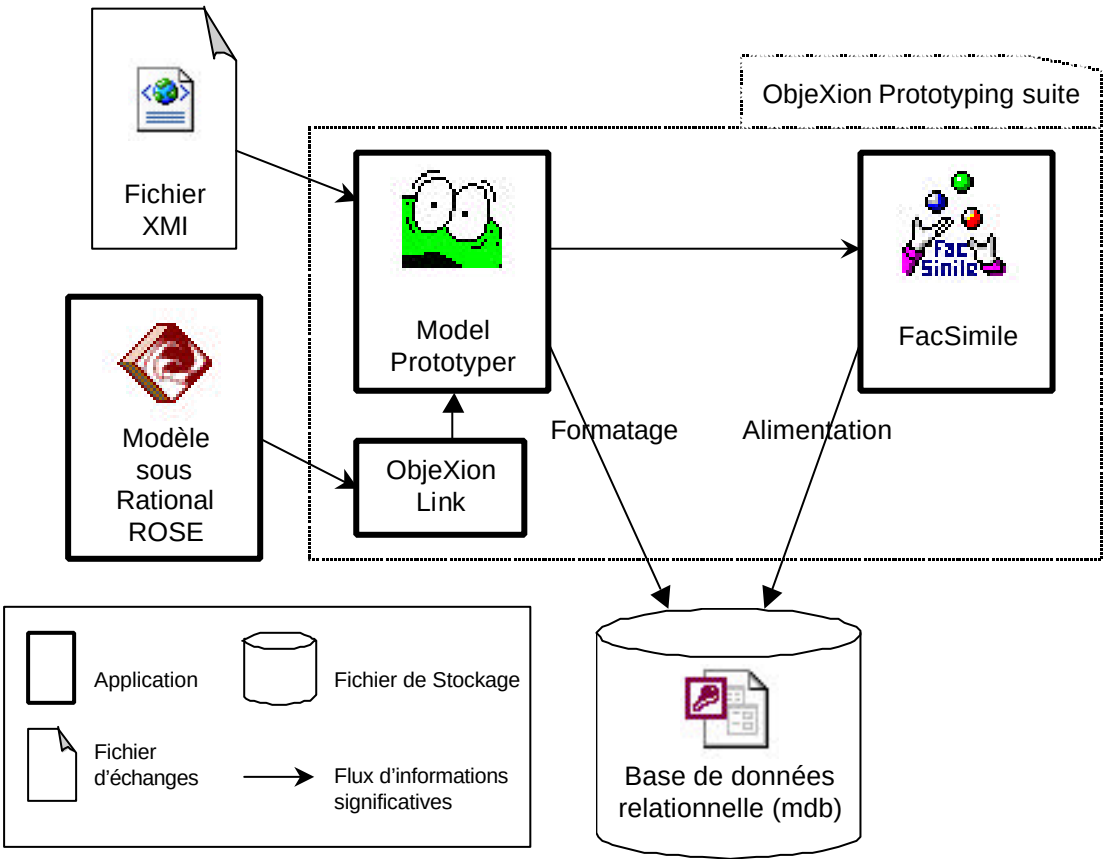
Le but de Prototyping Suite est la validation d'un modèle par prototypage en phase d'analyse. Ceci permet notamment d'accorder un client et un fournisseur de services par exemple, quant à la définition d'un cahier des charges.

Cette validation utilise la technique du *round trip engineering*, c'est à dire d'allers-retours entre le modèle et le prototype : réalisation d'un modèle, analyse et mise en place d'un prototype, découverte d'erreur, changements du modèle, mise à jour du prototype, découverte d'erreurs...

Prototyping Suite comprend trois modules :

- Model Prototyper qui analyse les modèles
- ObjeXion Link qui fait le lien entre une application Rational Rose et Model Prototyper
- FacSimile qui permet de faire vivre le modèle.

L'annexe I p.2 donne un extrait du mode d'emploi de chacun de ces composants.




Frédéric FONDEMENT - Stagiaire essaim 3ème année – DRT GEII 1ère année
Philippe STUDER - Directeur Technique ObjeXion Software
Pierre-Alain MULLER - PDG ObjeXion SoftWare – Enseignant essaim responsable
Michel HASSENFORDER - Enseignant essaim – Responsable DRT

# Model Prototyper

Model Prototyper est l'analyseur de modèles. Il lit un modèle défini dans un fichier XMI (standard de stockage des modèles UML défini par l'OMG, utilisant les mécanismes d'XML) ou à l'aide de ObjeXion Link (voir ci-après). On remarquera que la plupart des éditeurs UML sont capables de générer des fichiers XMI.

L'analyse simplifie le modèle en ne gardant que le diagramme des classes et en éliminant les composants non statiques. Elle l'enrichit aussi de façon à en améliorer l'utilisation, que ce soit en termes de performances (accès aux bases de données) ou en termes d'ergonomie (attributs à présenter, utilisation de clés...)

Le résultat est le formatage d'une base de données relationnelle, destinée à décrire toutes les instances de chaque type (ou classe) défini dans le modèle, ainsi qu'une description interne destinée à faire vivre le modèle (cf. FacSimile).

Model Prototyper est aussi capable de réaliser une analyse incrémentale. C'est cette spécificité qui permet le *round-trip engineering*. La base de données précédemment générée est non pas détruite mais reformatée, ce qui permet aux informations qui y sont stockées de ne pas être détruites.

Cf. annexe I p.3.

# ObjeXion Link

ObjeXion Link est un composant destiné à Rational Rose, qui permet à Model Prototyper de s'y connecter afin d'y lire les informations du modèle contenu.

Rappelons que Rational Rose est un logiciel permettant l'édition d'un diagramme UML, donc d'un diagramme de classe. Depuis la version 98, il est possible d'y ajouter des composants externes et c'est ce mécanisme qui est utilisé pour la lecture des modèles.

Cf. annexe I p.10.

# FacSimile

FacSimile est la partie de Prototyping Suite qui fait vivre le modèle. C'est elle qui va alimenter la base de données sus-dite, c'est à dire le prototype avec les désirs de l'utilisateur :

- Créer un objet (une instance de classe) ;
- Détruire un objet ;
- Modifier un objet (la valeur de ses attributs, ainsi que les liens vers d'autres objets)

Tout ceci est fait en même temps qu'une vérification de bonne validité, tels que les types des attributs (*toto* ne peut représenter un entier), le nombre de liens (si aucun lien n'est défini pour une association 1..*), une unicité des clés (plusieurs classe liées sous le même qualificateur dans une relation qualifiée de multiplicité 1)...

Pour faire un parallèle avec le langage Java, Model Prototyper serait le compilateur (javac), et FacSimile la machine virtuelle (java).

C'est donc bien le cœur de ce qu'on peut appeler une **machine virtuelle UML**.

Cf. annexe I p.12.

# Présentation d'OCL

OCL (Object Constraint Language) est un langage formel pour l'expression de contraintes, standardisé par l'OMG. L'extrait de la définition d'UML arrêté par l'OMG traitant d' OCL est donné en annexe II, p.25. Nous ne nous contenterons ici de brosser qu' un portait rapide de ce langage.

## Motivations

Certaines contraintes d'un système ne sont pas définissables par les éléments de modélisation décrits par UML. Elles sont souvent exprimées par des notes en langage naturel ce qui peut induire des ambiguïtés quant à leurs interprétations. OCL se veut un langage formel permettant de décrire ces contraintes de façon déterministe. De plus, il se veut simple à écrire ainsi qu'à comprendre.

OCL est purement un langage interrogatif : en aucun cas il ne peut modifier le modèle auquel il se rapporte. On parle de langage sans effet de bord, ou *side-effect free*. Ce n'est donc pas un langage de programmation... Pour cette raison, seules les méthodes de requêtes (marquées *isQuery* par le modeleur), sont appelables par une expression en OCL.

Il a trois rôles principaux :

- La vérification de contraintes invariantes dans le temps, par exemple un congélateur ne doit pas avoir une température supérieure à –18°C;
- La vérification de contraintes préalables à une opération, par exemple une tronçonneuse thermique doit avoir de l'essence avant de couper un arbre ;
- La vérification de contraintes post opératoires, par exemple une tronçonneuse doit avoir plus d'essence avant qu'après avoir coupé un arbre.

On remarquera qu'il s'agit là du seul langage normalisé incluant toutes les spécificités des diagrammes de classe UML, par exemple les classes-association ou les liens qualifiés. Cependant, il reste de nombreux points flous. De plus, la grammaire donnée (annexe II p.71) est incompatible avec le reste de la norme. OCL pèche donc toujours par un manque de rigueur dû à sa jeunesse et sa confidentialité car peu de logiciels l'intègrent.

# Contexte

Chaque contrainte OCL s'exprime dans un contexte bien précis. Ce contexte défini soit une classe, soit une méthode de classe, ainsi que le type de la contrainte : invariante, pré opératoire ou post opératoire.

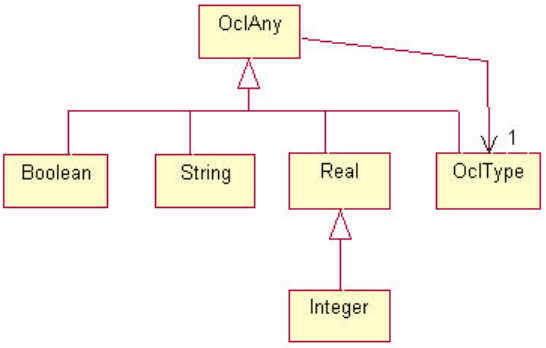
Dans le premier cas, la contrainte se vérifie sur chacune des instances de la classe définie, ainsi que sur chacune des instances de ses sous-classes. Il va de soit que seules les contraintes invariantes sont permises.

Dans le second cas, la contrainte se vérifie à chaque invocation de la méthode spécifiée. Un contexte pré opératoire avant sa réalisation, un contexte post opératoire après. En cas de contrainte invariante, la vérification s'opère avant et après l'invocation ; la contrainte peut donc être transitoirement violée.

# Types

Une contrainte OCL est une expression en OCL définie dans un contexte, renvoyant un résultat de type booléen (vrai ou faux) : la contrainte est validée (pour un objet) si son résultat retourné est *vrai*, ou violée si son résultat est *faux*.

En effet, OCL est un langage fortement typé, et ce à la manière de SmallTalk. Les types de bases, fournis en standard dans le langage, sont donc des classes, toutes héritant de la classe abstraite OclAny – sauf elle-même - y compris la classe représentant les types OclType, et les classes définies dans le modèle. Les compatibilités sont donc déterminées par l'arbre d'héritage. Voici un diagramme de classes résumant la situation :




Il faut, outre les classes du modèle, ajouter à ceci les collections. Elles sont au nombre de trois : le Bag, la Sequence où les éléments sont ordonnés, et le Set où les doublons sont interdits. Tous héritent du type abstrait Collection. Les collections sont compatibles si le genre de collection l'est (deux Set entre eux ou un Set vers une Collection), et si leurs contenus sont compatibles. Un Set d'Integer et une Collection de Real sont donc compatibles. La norme indique qu'un collection n'hérite pas de OclAny, ce qui me semble navrant.

L'annexe II p.51 décrit l'intégralité des types et des opération qui y sont applicables.

# Travail Réalisé

FacSimile est donc une machine virtuelle UML jouant sur des interactions entre objets. Il est donc possible d'y intégrer un interpréteur de contraintes OCL. Ce fut là mon sujet de stage. A l'heure actuelle, ce produit est testé par des utilisateurs.

# Résultat

A priori, deux utilisations d'un interpréteur OCL sont intéressantes au sein de FacSimile : la vérification de validité des objets créés, soit une utilisation classique d'OCL, ainsi que le filtrage des objets suivant un critère exprimé par une contrainte.

Dans les faits, ces deux caractéristiques sont très semblables et ont été réalisées de la même façon. La seule différence est que dans le premier cas on s'efforcera de traquer les objets ne vérifiant pas la contrainte, et dans l'autre cas, il s'agira plutôt d'extraire ceux qui la vérifient. Les caractéristiques de ce fonctionnement sont décrites en annexe III, à la page 95 (*Query Language*).

Cependant, hors d'une contrainte, et grâce à une sémantique complètement orientée vers les diagrammes de classes UML, une expression OCL peut facilement extraire une information au sein d'un objet.

Cet aspect a aussi été développé. Il est possible de demander à une instance le résultat d'une expression OCL, sans être limité par un résultat booléen, et en se bornant à un seul objet, et non pas à toute une collection. C'est ce que décrit l'annexe III en page 96 (*Evaluation Language*).

Enfin, restait un dernier point noir de cette implémentation : les méthodes, dans un modèle, et donc dans l'analyse de Model Prototyper, n'implémentent pas de comportement.

Par le même mécanisme que celui de l'extraction d'informations, il est maintenant possible de définir les méthodes. Par extension, on peut surcharger les méthodes de OclAny, y compris les opérateurs, dont hérite toute classe du modèle.

# Etapes

La première étape a consisté à utiliser le résultat de l'analyse de Model Prototyper. Ceci dans le but d'intégrer les mécanismes de recherche statique (dans le modèle) et d'extraction dynamique d'informations (sur une instance), en y intégrant les mécanismes définis par UML (héritage, liens qualifiés, classes-associations...).

Ensuite il fallut ajouter au modèle les classes définies par OCL, en y incluant la description des opérations qui peuvent s'y appliquer. Ce sont là les opérations de bases sans lesquelles aucun comportement n'est possible.

Puis vint la réalisation de l'interpréteur. En fait ont été réalisés deux interpréteurs en deux étapes. La première étape vérifie la conformité à la grammaire (*parsing*), et la compatibilité des types, la seconde exécutant le code. Le premier interpréteur est chargé des expressions, et le second, qui utilise le premier, permet de vérifier des contraintes.

Une fois cette étape réalisée, il a été possible d'enrichir la première avec l'interprétation de code OCL dans les méthodes.

Enfin vint l'intégration, graphiquement parlant, à FacSimile, dont tous les détails se trouvent à l'annexe III page 95.

# Problèmes majeurs

Les problèmes majeurs viennent principalement de la légèreté de la norme définissant OCL. Faire un interpréteur concret et complet nécessitait de lever tous ces points de flous et contradictions. Il a par exemple été nécessaire de réécrire la grammaire (laquelle ne supportait pas entre autre la fonction *iterate*). De plus, je me suis permis quelques modifications mineures.

La vérification des types au préalable a fait partie intégrante du sujet du stage. Elle ne fait donc pas partie d'un choix technologique personnel, mais d'un souci en amont d'évolutivité. Ceci a posé un problème pour l'instant partiellement résolu de vérification statique. En effet un type est un objet, donc est dynamique. La redéfinition de type, tout comme l'appel de toutes les instances d'un type, pose donc le problème du type de retour... De plus deux types non défini par OCL (Date et Time), ont dû être ajouté, car la base de données les utilisent.

La définition du corps d'une méthode est faite par une expression en OCL. Ceci présente un défaut majeur : OCL est un langage sans effet de bord, donc des opérations telles que l'affectation sont interdites. Ce mécanisme est décrit en annexe III page 97 (*Defining Methods*). Il faut remarquer alors deux choses :

- Afin de le rester lui-même, OCL ne peut appeler que des opérations sans effet de bord ; toute méthode implémentée est donc visible dans une expression en OCL.
- Les contextes post opératoires n'ont pas de sens, tout comme l'appel à des valeur antérieures à une opération.

Toutes les solutions adoptées sont décrites en annexe III (p.74).

# Conclusion et Remerciements

Il existe donc maintenant un interpréteur de code compatible (à peu de choses près) OCL 1.3, intégré à Prototyping Suite. Cet interpréteur est capable de vérifier des contraintes sur des objets, de sélectionner des objets vérifiant ou ne vérifiant pas des contraintes, d'extraire une demande d'informations et de réaliser des appels à des méthodes.

Le futur consistera à créer un langage de programmation, basé sur OCL, permettant une description complète des méthodes. Un projet au sein de l'OMG visant à normer un langage d'action (*Action Language*) est en cours de réflexion mais semble au point mort...

La réalisation de ceci me fut hautement profitable et ce sur beaucoup points.

Tout d'abord, il m'a fait constater la réelle utilité d'UML, et la nécessité de programmer de façon méthodique.

Il m'a donné l'expérience de la mise en place d'un langage objet. Une norme laxiste me donnant de grandes latitudes d'expression, j'ai pu ainsi réfléchir aux qualités intrinsèques d'un langage de programmation dit objet, ainsi que l'utilité réelle d'une norme bien faite.

Il m'a montré la manière de travailler en groupe, que ce soit au niveau du partage du code source, ou au niveau du partage de l'information.

Pour finir, je tiens donc à remercier tout ceux qui m'ont permis de faire ce stage dans des conditions favorables :

- Philippe STUDER, pour de nombreux conseils et une grande patience et une grande disponibilité,
- Pierre-Alain MULLER, pour permettre une grande liberté et son écoute,
- Michel HASSENFORDER, pour avoir trouvé ce stage,
- Laurent MENTEK, pour une aide à la réalisation des présentations,
- Olivier BURGARD, pour son dictionnaire d'anglais,

Et bien sûr l'équipe complète d'ObjeXion et de l'Embarcadère pour son accueil.

Je ne remercie pas plus Dominique GRIESINGER.

# Contributions

Norme UML de l'OMG – http://www.omg.org/
Modélisation objet avec UML – Pierre-Alain MULLER et Nathalie GAERTNER
Compilateur ANTLR – http://www.antlr.org/
« UML »'99 – The Unified Modeling language – Robert France Bernhard Rumpe
Dictionnaire Anglais-Français - Harrap's shorter