Université de Versailles Saint-Quentin-en-Yvelines

Institut National des Télécommunications

Rapport de Stage
DEA Méthodes Informatiques des Systèmes Industriels

(M.I.S.I)

ENREGISTREMENT ET PROPRIÉTÉS NON FONCTIONNELLES

Nabiha BELHANAFI

Responsable de DEA : Ahmed MEHAOUA
Responsable de stage : Chantal TACONET

Septembre 2003

Ce stage de DEA a été réalisé au sein du laboratoire **Systèmes Répartis** du département **Informatique** de l'**Institut National des Télécommunications**

Remerciements

Je tiens à remercier, Guy Bernard pour m'avoir accueilli dans l'équipe Systèmes Répartis de l'Institut National des Télécommunications.

Je tiens à exprimer mes plus sincères remerciements à Chantal Taconet pour son rôle d'encadrante, ses conseils et pour sa disponibilité.
Merci à Nabil Kouici et Douha Ayed pour le partage de leurs connaissances informatique.
Merci à toute l'équipe systèmes répartis de l'INT.

Merci aux membres de ma famille, et à tous ceux qui sont trés chères à mon coeur.

Un grand merci à ceux avec qui j'ai partagée des moments inoubliables : fazou et lynda, en espérant qu'on partagera beaucoup d'autres moments agréables dans le futur.

Une pensée chaleureuse a tous mes ami(e)s.

Enfin une pensée trés émue à ma chère et tendre mère ainsi qu'à mon père.

Table des matières

Table des figures

Chapitre 1

Introduction

En quelques décennies, la production logicielle a évoluée de manière très importante, rendant son processus de plus en plus complexe.
Il devient extrêmement difficile pour une même personne d'être acteur de la conception à l'utilisation d'un logiciel. Cette constatation rend nécessaire le découpage du processus de production en rôles distincts et interopérables.

Dans ce contexte il y a eu un courant de recherche très actif portant sur l'utilisation de modèles et langages à objets pour la construction d'applications réparties. Ces modèles permettaient d'une part, d'exprimer les interfaces des entités logiciels manipulées, résultat de la conception, et d'autre part, de bien séparer la définition de ces interfaces de l'implantation des entités logiciels. Les principaux objectifs des modèles à objets étaient d'améliorer la modélisation d'une application, d'optimiser la réutilisation du code produit et de prendre en compte l'ensemble du cycle de production des applications. Cependant l'intégration d'entités logiciels existantes s'avère difficile car les aspects de communication sont mixés dans les aspects fonctionnels rendant le maintien et l'évolution complexes[MMG99].

Pour pallier les défauts de l'approche objet et viser des notions non prévues initialement, l'approche composant est apparue, cette approche est fondée sur des techniques et des langages de construction d'applications qui intègrent d'une manière homogène des entités logicielle provenant de diverses sources.

Dans le cadre de la norme CORBA 2 [GGM97] [CCM02], l'*Object Management Group* (OMG) offrait une technologie permettant l'exécution répartie d'objets hétérogènes. Pour cela elle offrait un cycle de développement basé sur l'expression des interfaces des objets en OMG IDL (*Interface Description Langage*), la projection de ces interfaces vers un langage de programmation et l'implantation des objets dans ce langage .
Cette approche répondait bien au besoin de spécifier les interfaces fournies par un objet, mais rien ne permettait de spécifier les interfaces requises, de plus les aspects sécurité ou persistance des objets devaient être mise en oeuvre difficilement dans l'implantation des objets.
Afin de faciliter le développement d'applications, la spécification CORBA 3 tend à offrir des moyens pour exprimer les dépendances et besoins d'une entité logicielle en fonction de son environnement sous la forme d'une description plutôt que par programmation, ceci a permis de décharger le programmeur de l'écriture du code non directement lié au métier du composant, en le considérant comme étant une propriété non fonctionnelle.

Une fois les composants instanciés, et pour qu'ils puissent être utilisés par les clients, leurs exportation ou enregistrement, auprès de divers annuaires de recherche sur propriétés est nécessaire. Ils doivent également systématiquement se dé-enregistrer. L'enregistrement bien que systématique est toujours à la charge du programmeur d'objets.

La question posée par ce stage est la suivante : le mode d'enregistrement des composants peut-il être décrit par des propriétés non fonctionnelles ? si c'est le cas, les middlewares pourront fournir des entités qui réalisent des enregistrements intelligent de ces composants.
Le travail proposé dans ce stage comporte différentes facettes, en outre l'étude détaillée du modèle de composant corba (CCM), proposition de modèle pour intégrer ces nouvelles propriétés non fonctionnelles dans CCM et implantation de la proposition dans OpenCCM.

Ce document présente l'ensemble du travail et des résultats obtenus au cours de ce stage. Il s'articule autour du plan suivant. Dans le chapitre 2, nous présentons la spécification du modèle de composant CORBA et nous parlons de l'implantation OpenCCM de ce modèle. Dans le chapitre 3 nous examinons l'enregistrement dans la spécification CCM, ensuite nous présentons dans le chapitre 4 l'approche d'enregistrement automatique que nous avons intégrée dans OpenCCM. Enfin, pour démontrer la faisabilité de notre approche, le chapitre 5 présente la réalisation de la proposition.

Chapitre 2

Spécification CCM

Comme il a été discuté dans [MMG99] [MP02], les modèles à objets ont progressivement montré leurs limites. Certains industriels, comme Microsoft, Sun et l'OMG, ont fait évoluer leurs modèles, vers les composants, dans le but de simplifier le développement d'applications.

La réponse de l'OMG est le *CORBA Component Model* (CCM), modèle de base de la norme CORBA 3. Le CCM est un *framework* de composants, il se décompose en deux niveaux, un niveau basique qui permet de mettre sous forme de composants des objets CORBA, et un niveau étendu qui sera présenté dans ce chapitre .

Tout comme EJB de Sun [VM99], auquel correspond la version basique du CCM en java, la technologie CCM repose sur l'utilisation de conteneurs pour héberger les instances de composants et faciliter leurs déploiement .
La spécification CCM [CCM02] est découpée en quatre modèles et un méta modèle :

- **Le modèle abstrait** : offre aux concepteurs le moyen d'exprimer les interfaces et les propriétés des composants (voir section 2.2). Pour cela le langage IDL a été étendu pour prendre en compte les nouveaux concepts introduits dans le CCM.
- **Le modèle d'implantation** : décrit la structure de l'implantation d'un composant ainsi que certains de ses aspects non fonctionnels à travers le langage CIDL (*Component Implementation Définition Langage*). L'utilisation de ce langage est associée au CIF (*Component Implementation Framework*) qui décrit l'interaction entre les parties fonctionnelles et non fonctionnelles.
- **Le modèle de déploiement** : permet d'automatiser la diffusion et la mise en place d'une application distribuée, et contribue à accroître la réutilisabilité d'entités logicielles en facilitant l'utilisation et l'intégration de composants existants.
- **Le modèle d'exécution** : définit l'environnement d'exécution des instances de composants. Le conteneur a pour fonction principale l'interception des requêtes invoquées sur les composants pour interposer la gestion non fonctionnelle d'une manière totalement transparente.
- **Le méta modèle de composant** : basé sur UML et dispose de projection vers le MOF (*Méta Object Facility*)

Dans ce chapitre nous allons étudier chacun de ces modèles, puisque nous devons automatiser la génération de code d'enregistrement d'un composant auprès des services de recherche sur propriétés, faire une description non fonctionelles de ce dernier et automatiser l'enregistrement lui même, cette étude nous permettra de comprendre la chaine de production de la spécification CCM afin de désigner et de choisir le/les niveaux où se fera cette description et les entités qui rentrent en jeux afin de réaliser un enregistrement non fonctionnelle et automatique.

2.1 Propriétés foncionnelles et non fonctionnelles

Avant d'aborder les différents modèles de la spécification CCM, nous allons donner une définition des propriétés fonctionnelles et non fonctionnelles.
Nous désignons sous le terme de *propriétés fonctionnelles* les services et fonctionnalités qu'un composant offre, alors qu'une *propriété fonctionnelle* définit les services dont un composant à besoin pour s'exécuter.
Les propriétés non fonctionnelles sont de deux types, celles qui nécessitent la génération de code, dans cette catégorie nous pouvons citer la persistance des objets(voir section 2.3.2), dans cette catégorie la description de cette propriété se fait au début de la chaine de production de CCM pour générer du code lors de la compilation, et les propriétés non fonctionnelles qui ne nécessitent pas de génération de code comme par exemple le service de transation, le service de cycle de vie ..., dans ce cas ces propriétés sont décritent dans le descripteur de composants (voir section2.5.1).

2.2 Modèle abstrait

Le modèle abstrait des composants CORBA vise essentiellement deux choses. Premièrement, il vise à décrire un type de composant comme étant une boite noire dont on ne sait rien de l'implantation à l'aide du langage IDL3. Deuxièmement, seule l'implantation fonctionnelle doit être programmé tout ce qui concerne le non fonctionnel doit être simplement décrit. L'implantation non fonctionnelle se fait lors de la compilation.
Le premier point met en avant la volonté de rendre explicite toute interaction avec ou à partir d'un type de composant, dans ce sens, un type de composant ne se limite pas à définir les services qu'il fournit, mais aussi ceux qu'il utilise.
Le second point tend à faciliter la production et à accroître la réutilisabilité des implantations de composants. La description sert de base à la génération automatique de la prise en charge des aspects non fonctionnels.
Dans les sous sections suivantes une étude plus détaillée du modèle abstrait est présentée afin de savoir s'il peut être exploité pour la génération automatique du code relatif à l'enregistrement des composants.

2.2.1 Type de composant

Le type « composant »est un méta-type qui représente une extension et une spécialisation du méta-type « interface »défini dans la spécification CORBA2. Ce méta-type est exprimé en IDL, il peut être projeté vers plusieurs implantations qui devront se comporter de manière similaire.
Un type de composant regroupe la définition d'attributs et de ports. Les attributs représentent les propriétés configurables du type de composant, alors qu'un port représente une interface soit fournie soit utilisée par le type de composant.

La déclaration d'un type de composant se fait à l'aide du nouveau mot clé *component*, cette définition introduit implicitement une interface qui supporte les caractéristiques déclarées dans la définition du corps du composant.
L'identité d'un composant est exprimée premièrement par la référence de base du composant, en plus de cette référence de base d'autres éléments peuvent servir à l'identification du composant.
Les composants apportent au monde CORBA la possibilité d'avoir plusieurs interfaces associées avec une unique référence.
La spécification CCM regroupe sous le nom de port tout mode d'interaction qui existe pour un

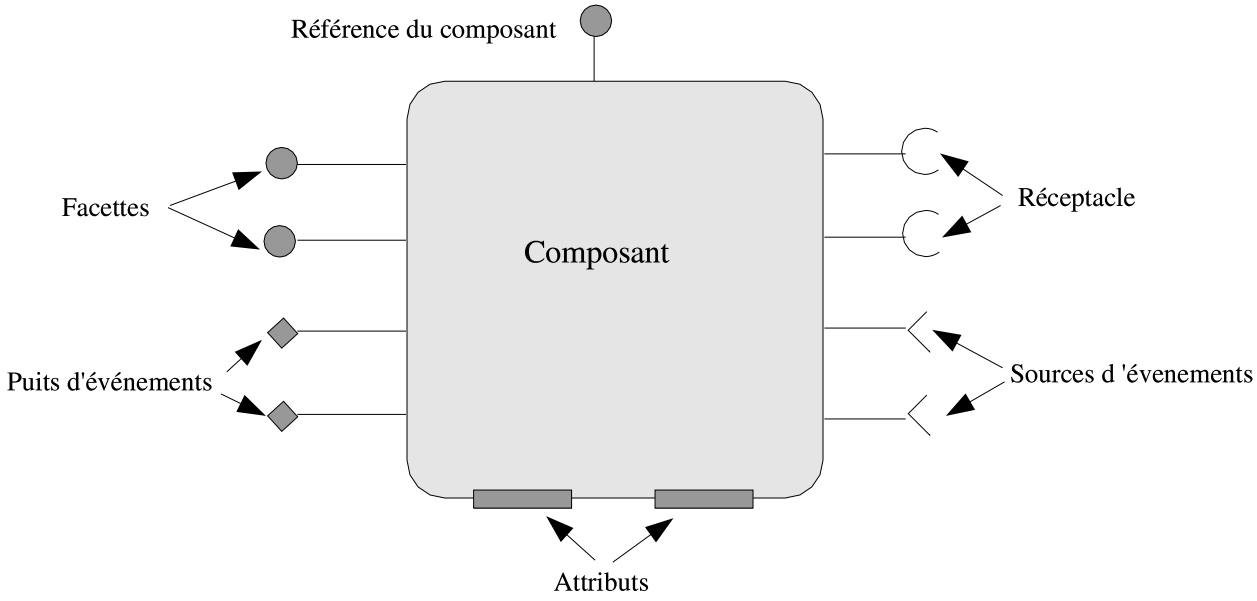

FIG. 2.1 – Le modèle abstrait du composant CORBA

composant. Un port peut être une *facette, un réceptacle, une source* ou *un puit d'événements*.

2.2.2 Facettes

Les facettes correspondent aux interfaces fonctionnelles par lesquelles les instances de composant sont sollicitées. Un composant CCM peut implémenter plusieurs fois la même interface, c'est à dire qu'une interface peut être implémentée deux fois par un composant via deux facettes différentes. Comme tout élément d'une application CORBA, une facette dispose d'une interface OMG IDL. Cette interface est le seul élément disponible pour un client de composant.
La déclaration IDL3 d'une facette se fait à l'aide de la clause *provides*.

```
interface FacadeClient {...};
interface FacadeFournisseur {...};
interface FacadeDepanneur {...};
Component Imprimante{
        provides FacadeClient client;
        provides FacadeFournisseur fournisseur;
        provides facadeDepanneur depanneur;
};
```

FIG. 2.2 – Définition d'une facette en IDL3

En plus d'offrir la possibilité de proposer plusieurs facettes, un composant offre un moyen de naviguer entre celles-ci.Un client peut donc changer son point de vue sur le composant au cours du temps.

2.2.3 Réceptacles

Un réceptacle exprime le besoin de ce composant à utiliser une interface particulière. Cette interface peut être une interface offerte par un autre composant. Ce qui permet d'établir une relation

(appelant/appelé) se traduisant par un point de connexion synchrone entre le composant ayant le réceptacle et un autre composant offrant la facette requise, cette relation est appelée connexion.

Un réceptacle permet donc d'assembler des instances de composants et potentiellement des objets. La composition permet de concevoir plus facilement des applications. Elle permet aussi aux instances de composants de faire de la délégation de traitement. Lorsqu'un composant a besoin d'une interface pour fonctionner, son IDL3 déclare un réceptacle représentant cette interface. Tout comme les facettes, deux réceptacles peuvent représenter une même interface. Le réceptacle est le port dual de la facette, ce qui permet d'effectuer des connexions entre les composants. Dans l'IDL3, un réceptacle est déclaré par le mot clé *uses*, il peut être "multiple", pour se connecter à plusieurs instances de facettes en même temps.

```
Interface BacPapier{...};
component Imprimante {
        uses BacPapier bac_papier;
};
```

FIG. 2.3 – Définition de réceptacles en IDL3

2.2.4 Événements

Le modèle d'événement est du type producteur/consommateur. Pour recevoir des événements un client doit souscrire à ce type d'événement, ensuite le mode *push* est utilisé. Il y a deux types de ports qui traitent les événements : les sources et les puits d'événements.

2.2.4.1 Sources d'événements

Un composant peut produire des événements à travers un port de type source d'événements, chaque composant peut avoir zéro ou plusieurs sources d'événements. Les sources d'événements sont de deux types :

- **Source d'émission(*Emitter*) :** c'est la catégorie de source qui n'accepte qu'un seul consommateur. La connexion producteur-consommateur est directe, elle se fait à l'initialisation du composant.
- **Source de diffusion (*Publisher*) :** c'est la catégorie de source d'événements qui accepte plusieurs consommateurs en même temps. Dans ce cas l'abonnement d'un consommateur à un type d'événement est délégué à un canal d'événements fourni par la structure d'accueil.

Pour définir une source d'événement pour un composant, il faut déclarer la source d'événement elle-même ainsi que le type d'événement qu'elle émet. Une source d'émission est déclarée à l'aide du mot clé *emits*, et la source de diffusion est déclarée à l'aide du mot clé *publishes*.

L'événement est déclaré à l'aide du mot clé *eventtype*. Ce dernier définit un bloc qui contient des champs représentants les données de l'événement et éventuellement des opérations qui permettent de manipuler ces données.

2.2.4.2 Puits d'événements

Un puit d'événements permet à un composant de recevoir des événements d'un certain type. C'est un port dont le type est un consommateur d'événements. Un puit d'événements est déclaré à

l'aide du mot clé *consumes*, suivi du type de l'événement à consommer et d'un identifiant du puit. L'événement est déclaré de la même manière que pour les sources d'événements.

2.2.5 Attributs

Les attributs représentent les propriétés configurables du type de composant, c'est à dire que la configuration d'un composant repose principalement sur le positionnement des valeurs de ses attributs.
La déclaration d'un attribut est faite en utilisant le mot clé *attribute*.

```
component Imprimante
{
   attribute string the_name;
   attribute int vitesse;
   attribute int job;
   atribute boolean marche;
};


home ImprimanteHome manages Imprimante
{
};
```

FIG. 2.4 – Declaration d'attributs

2.2.6 Maisons de composants

Tout comme ***component***, ***home*** est un nouveau méta-type défini pour désigner la maison d'un composant. Le rôle principale d'une maison est de gérer le cycle de vie des composants, c'est à dire d'instancier, de détruire et de retrouver les instances de composants. La maison crée une instance de composant en lui associant une référence CORBA. La maison peut avoir recours à une méthode d'indexation des instances créées à base de clés pour les retrouver. Une clés est une structure de données qui identifie une instance de composant de manière unique à l'intérieur d'une même maison. Un client doit donc fournir à la maison la clé de l'instance dont il recherche la référence.
Un type de composant est défini d'une manière indépendante des types de maison, par contre, un type de maison doit spécifier précisément le type de composant qu'elle va héberger. Plusieurs types de maisons peuvent gérer un même type de composant, mais pas les mêmes instances. A l'exécution, une instance de composant est gérée par une unique instance de maison.

2.2.7 Synthèse

Le modèle abstrait permet au développeur de définir les composants et leurs ports d'une manière très simple, en utilisant le langage IDL3. Pour préparer à l'implantation des composants et leurs implantation dans une plate-forme CCM, la description IDL3 doit être transformée en IDL2 et définir ainsi la vue du client (figure 2.5). Un client peut être aussi bien un client système, qui est la plate-forme CCM elle-même, qu'un client applicatif. Les règles de transformations sont clairement définies par la spécification CCM [CCM02] pour pouvoir automatiser la transformation elle même.

Par ailleurs, la description IDL2 est non seulement automatiquement générée, mais aussi automatiquement codée d'une manière complètement transparente au développeur. Les différentes entités générées sont décrites ci dessous.

- **L'interface principale** du composant qui offre les opérations suivantes :
  - les opérations de l'interface *CCMObject*. Celles-ci sont des opérations d'introspection et de connectique génériques,
  - les opérations des éventuelles interfaces supportées. Celles-ci sont spécifiques à l'application. Les interfaces supportées sont particulièrement intéressantes pour les composants basique qui, par définition, n'offrent pas de port et ne peuvent donc pas offrir d'opérations qu'en déclarant les interfaces supportée,
  - les opérations d'introspections et de connectiques spécifiques à l'application.
- **Le squelette de composant** qui prend en charge les opérations suivantes :
  - les opérations de connectique qui permettent généralement de connecter/déconnecter en synchrone ou asynchrone les composants entre eux, générées à partir des déclarations des réceptacles et des sources d'événements,
  - les opérations d'introspection qui permettent de fournir des informations sur le composant, sa maison et ses ports générées à partir des déclarations de facettes, des puits d'événement et de la maison.
- **Le squelette de la maison du composant** qui prend en charge :
  - les opérations de création, destruction et de recherche de composants, générées à partir des déclarations des maisons.

Le développeur se limite à réaliser les aspects purement métier de son application, à savoir :

- les opérations des éventuelles facettes offertes,
- les opérations des éventuelles interfaces supportées,
- les opérations de consommation d'événement *push*, si le composant possède un puits d'événements.

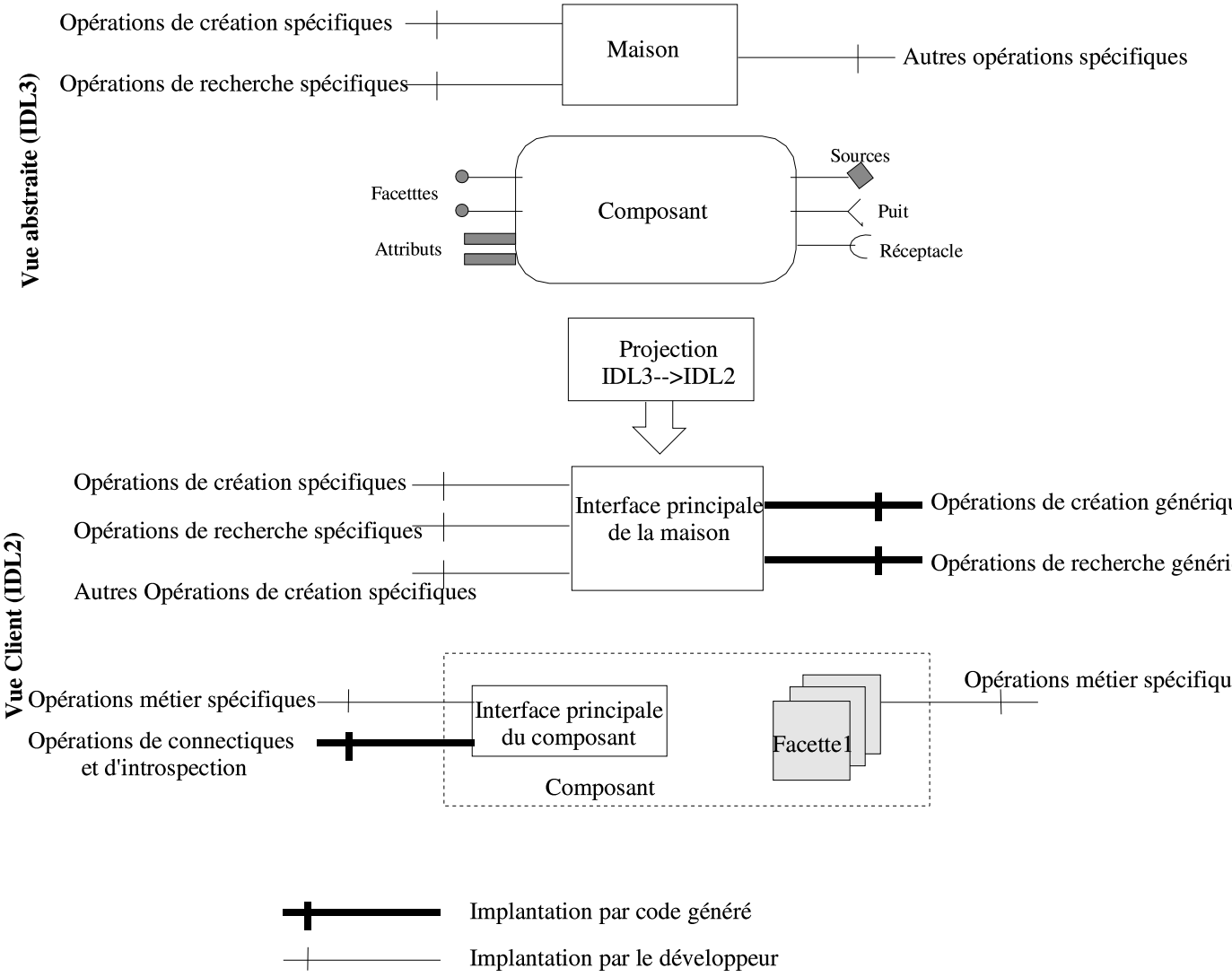

FIG. 2.5 – Projection de l'IDL3 à l'IDL2

2.3 Modèle d'implantation

Le modèle d'implantation offre aux fournisseurs de composants logiciels le moyen de produire l'implantation de certaines parties d'un composant. L'objectif de ce modèle est de décrire les aspects non fonctionnels des composants CORBA. Le modèle CCM sépare ces aspects des aspects purement fonctionnels pour en permettre la gestion automatiquement. Cet automatisation est possible moyennant certaines déclarations supplémentaires que le développeur doit faire en plus des déclarations IDL3 précédemment décrites.

La spécification CCM a défini un modèle d'implantation de composants (CIF : *Component Implementation Framework*) qui permet de faire coopérer l'implantation fonctionnelle avec l'implantation non fonctionnelle, c'est à dire, comment l'implantation fonctionnelle du composant s'intègre dans la partie non fonctionnelle.

Le CIF utilise le CIDL(*Component Implementation Definition Langage*) pour générer des squelettes et des souches. Cela automatise l'implantation de nombreux comportement de base (navigation, activation, gestion de l'état, du cycle de vie ....). En plus du langage CIDL, le CIF repose sur le *framework* défini pour le POA (*Portable Object Adapter*) tout en masquant la complexité de celui ci.

Dans les sections suivantes ce modèle va être étudié plus en détail afin de savoir s'il peut être utilisé pour que l'enregistrement des composants puisse être décrit par des propriétés non fonctionnelles.

2.3.1 CIDL

La description de l'implantation d'un composant repose sur le langage CIDL . Ce langage est une extension de la combinaison du langage IDL2 et du langage PSDL (*Persistent State Description State*) du service de persistance de CORBA. Pour cela, un composant est considéré comme étant un ensemble d'éléments comportementaux fourni par des exécuteurs. Deux types d'exécuteurs sont définis : les exécuteurs de composants, qui implantent les types de composant, et les exécuteurs de maison, qui implantent les types de maison de composant.

La déclaration CIDL permet de décrire :

- la catégorie du composant (voir section 2.3.1.1),
- la structure de l'implantation du composant. Ceci permettra de lier les squelettes générées à l'implantation fonctionnelle du composant pour qu'ils puissent coopérer,
- un éventuel état persistant du composant, si celui ci est persistant. Ceci permettra aux squelettes de prendre en charge sa persistance.

Ces informations sont déclarées en utilisant le terme *composition*. Le développeur doit spécifier autant de composition que d'implémentation de composant.

2.3.1.1 Catégories de composant

Pour pouvoir prendre en charge leurs aspects non fonctionnels, le modèle CCM à classé les composants CORBA en quatre catégories possible : *SERVICE*, *ENTITY*, *PROCESS* et *SESSION*. La catégorie du composant permet de définir la nature persistante ou pas du composant, sa durée de vie et la visibilité de son identité au client.

- Un composant de catégorie *service* est un composant sans état et sans identité. La durée de vie de ce type de composant correspond donc à la durée de l'exécution d'une opération.
- Un composant de catégorie *session* est un composant ayant un état volatile et une identité qui n'est pas persistante. La durée de vie d'un tel composant est une suite d'invocations avec un

client.

– Un composant de catégorie *entité* est un composant ayant un état et une identité persistante. leur durée de vie couvre plusieurs interactions avec le client. Le client connait l'identité du composant via une clé primaire qui lui est associée et qui l'identifie d'une manière unique.
– Un composant de catégorie *process* est un composant semblable à un composant de catégorie entité en terme de persistance et de durée de vie, seulement leurs clients ne connaissent pas leurs identité. L'état persisant du composant et son identité ne sont pas visible au client à moins qu'ils soient fournis explicitement par des opérations définies par le composant lui-même.

2.3.1.2 Compilation CIDL

La figure 2.6 illustre la chaîne de compilation qui met en oeuvre le modèle d'implantation du composant. Le résultat de cette chaîne est la génération des squelettes de composants et de maisons de composants. Le développeur doit étendre ces squelettes pour implanter la partie fonctionnelle et produire une implantation complète.

Pour résumer, un squelette de composant implante :

– La description IDL2 générée, c'est à dire la découverte des ports ainsi que la gestion de la connectique entre composants.
– Les opérations d'activation/passivation, de restauration/sauvegarde de l'état persistant du composant.

Un squelette de maison implante :

– Les opérations de création, destruction et de recherche de composants.

La compilation CIDL génère également un descripteur XML regroupant les contraintes techniques à respecter au moment du déploiement. Ce fichier reprend la description de tous les ports de chaque composant. Le développeur doit étendre ce descripteur pour compléter la description des aspects non fonctionnels des composants.

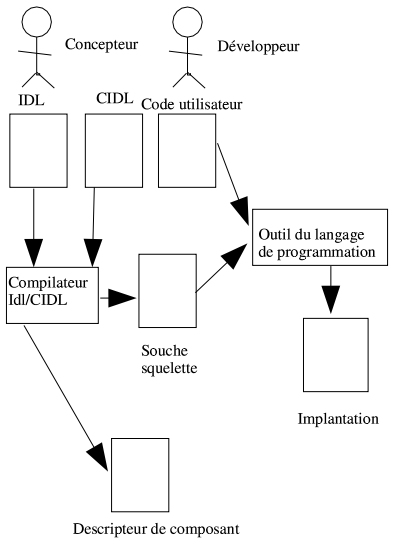

FIG. 2.6 – Chaîne de compilation IDL et CIDL

2.3.2 PSDL

La description de la persistance d'un composant repose sur le langage PSDL. La gestion de la persistance permet à l'état d'un composant de devenir durable, c'est à dire devant être sauvegardé sur un support de stockage comme les bases de données. L'état d'un composant est défini par l'ensemble de ses attributs. Le principe de la gestion de la persistance automatique est de générer le code de deux opération : une opération de restauration et une opération de sauvegarde de l'état du composant dans le squelette du composant d'une manière transparente au développeur.

Le modèle d'implantation répond bien aux soucis de produire rapidement des implantations de composants : décrire des besoins est plus simple qu'écrire du code. En second lieu, il répond bien aussi au problème de produire des composants logiciels de qualité : du code produit automatiquement réduit le risque de bug.

2.4 Modèle d'exécution

Le conteneur est la structure d'accueil des composants qui supporte l'exécution des composants et assure une gestion automatique des aspects non fonctionnels, dans l'objectif de décharger le développeur de la gestion de ses aspects. Pour réaliser ses fonctions, le conteneur est amené à interagir avec l'environnement CORBA de la manière suivante :

- utilisation du service de transaction CORBA pour gérer les transactions.
- utilisation du service de persistance CORBA (PSS : *Persistant State Service*), pour gérer la persistance de l'état des composants et cela en coopérant avec les squelettes des composants. Le développeur d'un composant peut choisir deux politiques de persistances :
  - La persistance est gérée par le conteneur. Si le protocole de connexion à la base de données et l'état persistant du composant sont précisés, alors les implémentations des méthodes *ccm_store* et *ccm_load* sont générées automatiquement . Ces méthodes sont invoquées par le conteneur pour sauvegarder et restaurer l'état du composant.
  - La persistance est gérée par le composant lui-même : Si ni l'état persistant, ni le protocole de connexion à la base de donnée ne sont précisés, alors le programmeur du composant doit fournir une implémentation de méthodes *ccm_load* et *ccm_store*.
- gestion de la présence en mémoire des diverses instances de composants. En effet le conteneur gère l'activation [1] et la passivation [2] des servants[3] de composants, et cela en utilisant le mécanisme d'activation dynamique du POA.
- création des références de composants, en utilisant le mécanisme de création de références d'objets du POA.

2.4.1 Type de conteneur

Il existe plusieurs types de conteneurs classés dans deux grandes catégories :

- **conteneurs orientés services :** ils accueillent des composants dont la fonctionalité correspond à un service. Ces composants n'ont pas un état persistant, leur état n'est valide que durant une session avec un client. Cette catégorie de conteneur est séparée en deux sous catégories :
  - Les conteneurs de type Service sont caractérisés par le fait que les composants qu'ils contiennent ne possèdent pas d'état interne. Les instances de ces composants peuvent être manipulées par plusieurs clients sans risque d'effets de bord. Le conteneur peut alors garder

[1] association d'un servant à une instance de composant
[2] dissocier le servant du composant auquel il a été associé
[3] unité d'exécution du composant

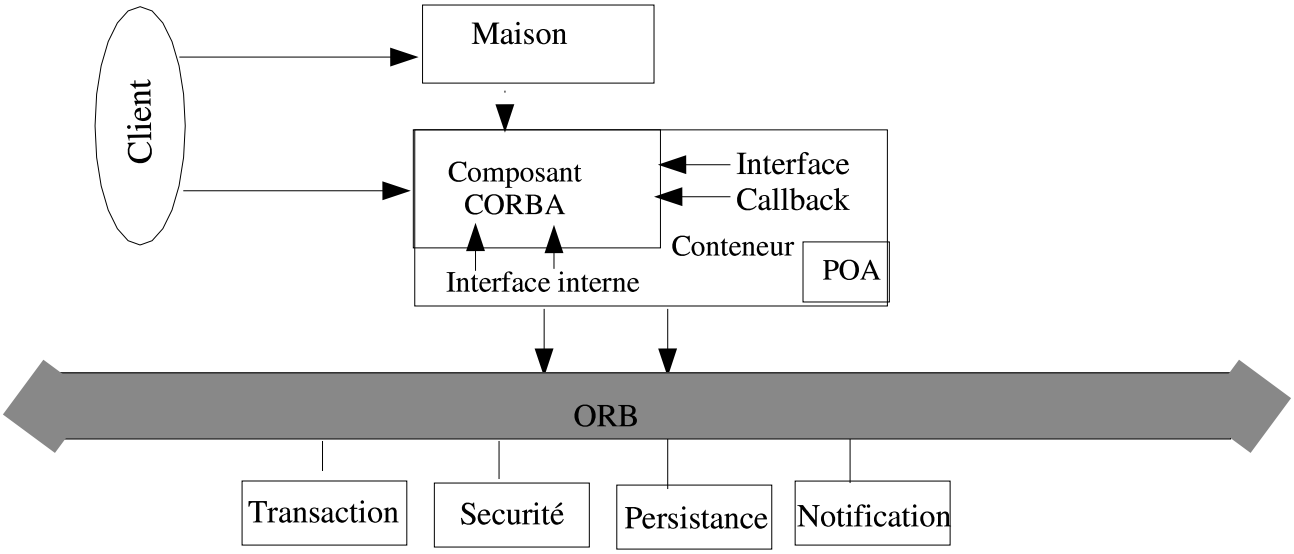

FIG. 2.7 – L'architecture du modèle de programmation des conteneurs

plusieurs instances de ce composant et les fournir successivement aux divers clients sans avoir à les réinitialiser.

- Les conteneurs de type Session sont caractérisés par le fait que les composants qu'ils contiennent possèdent un état interne, ce qui implique qu'une instance de ce genre de composant est exclusive à un client pendant la durée d'une session. Ainsi, contrairement aux conteneurs de type Service, les services fournis peuvent être le résultat d'un ensemble d'interactions entre le client et le composant.

– **Les conteneurs de type Entity et Process :** accueillent des composants gérant des données persistantes. Les composants dans un conteneur de type Process représentent des données utiles pour le processus et cachées au client, alors que les composants de type entité représentent des données non cachées au client. Les instances de composants de ces types de conteneurs représentent des données persistantes, elles peuvent ainsi être partagées entre plusieurs clients à condition de respecter les protocoles de gestion de base de données telles que les transactions et les accès concurrents.

Lors de la programmation d'un composant, il faut lui choisir un type de conteneur. Ce choix dépend de la structure et de la fonctionnalité du composant, il implique aussi l'utilisation de services transparents à sa programmation. Ces services ont été mentionnés dans la section précédente.

2.4.2 Interfaces du conteneur

La figure 2.7 illustre l'architecture d'un serveur CCM qui montre l'intégration d'un composant, sa maison, son conteneur et l'environnement CORBA.

Les éléments de cette architecture interagissent selon des contrats standardisés par CCM. Ces contrats sont définis par un ensemble d'interfaces classées en trois catégories :

- **Interfaces externes** définissent l'interaction du client avec le composant.
- **Interfaces internes** utilisées par le développeur et fournies par le conteneur pour l'implantation du comportement du composant.
- **Interfaces callback** utilisées par le conteneur et implantées par le composant (par génération ou directement) pour le déploiement du composant au sein du conteneur.

2.4.2.1 Interfaces externes

L'interaction des composants avec le client se fait à travers les interfaces externes qui sont définies par :

- les interfaces de maison de composants qui permettent au client de créer, chercher et détruire des instances de composants.
- les interfaces des composants qui représentent les interfaces IDL2 générées à partir de la spécification IDL3 des composants. Parmi ces interfaces il y a des interfaces applicatives telles que les facettes, les interfaces supportées, des interfaces de management qui offrent les opérations de connectiques et d'introspection.

Le client peut être l'application cliente finale ou bien l'outil de déploiement, ou un autre composant.

2.4.2.2 Interfaces internes et de callback

Les interfaces internes sont des interfaces fournies par le conteneur pour l'instance du composant. Ainsi, le composant a des point d'accès aux différents services systèmes ainsi qu'à différentes informations concernant le composant lui-même. Les composants disposent d'un ensemble d'interfaces de *callback* permettant de coopérer avec eux dans le cadre de la gestion des aspects non fonctionnelles. Les interfaces internes et *callback* font partie du contrat entre le composant et son conteneur.

2.5 Modèle d'assemblage et de déploiement

Le modèle de déploiement est le second apport majeur des composants CORBA. Il offre des moyens pour automatiser la mise en place d'une application distribuée. Il contribue à accroître la réutilisabilité d'entités logicielles en facilitant l'utilisation et l'intégration de composants existants.

Le modèle CCM définit un processus de déploiement :

- guidé par un ensemble de descripteurs XML,
- exécuté par un ensemble d'objets spécifiques,
- suivant des chaînes d'exécution.

Les descripteurs sont fournis par le développeur. Ils spécifient les fichiers à installer, les plates formes sur lesquelles déployer les composants, les connexions à établir entre composants, et leurs paramètres de configuration. Grâce à ces descripteurs, les objets spécifiques permettent :

- d'installer le conteneur et les maisons de composants,
- de créer et configurer les composants,
- d'établir les connexions entre composants.

Enfin la chaîne d'exécution décrit la séquence d'opérations à invoquer sur les objets spécifiques pour le déploiement des composants.

2.5.1 Paquetage des composants

Un paquetage d'entité logicielle est l'unité de déploiement. Il est défini comme étant le groupement d'un ensemble de fichiers d'implantation (code) avec un ou plusieurs déscripteurs qui décrivent le contenu de ce paquetage.

Le modèle CCM définit deux types de paquetage, illustrés par la figure 2.8

- **Le paquetage de composant** , associé à un seul composant contient :

  - les implantations du composant et de sa maison qui sont fournies par le développeur du composant,
  - les souches et squelettes du composant. Ceux-ci sont ceux générés automatiquement par les compilations IDL3/CIDL,
  - le descripteur du composant pour publier les propriétés non fonctionneles et configurer le conteneur sur la base de ses propriétés,
  - le descripteur de logiciel du composant qui fournit une vue logicielle du composant.
- **Le paquetage d'assemblage**, associé à un assemblage de composants, contient :
  - un ensemble de paquetage de composant pour chaque composant constituant l'assemblage.
  - un descripteur d'assemblage de composant, décrit dans la section suivante.

Les descripteurs contenus dans le paquetage de composant et dans le paquetage d'assemblage sont appelés déscripteurs de déploiement. Ceux ci sont écrits dans un langage inspiré du langage *Open Software Descriptor* (OSD), un langage basé sur XML, il permet de décrire des paquetages logiciels et leurs dépendances. L'OMG a repris ce langage et l'a adapté aux besoins de description du déploiement des composants[CCM02].

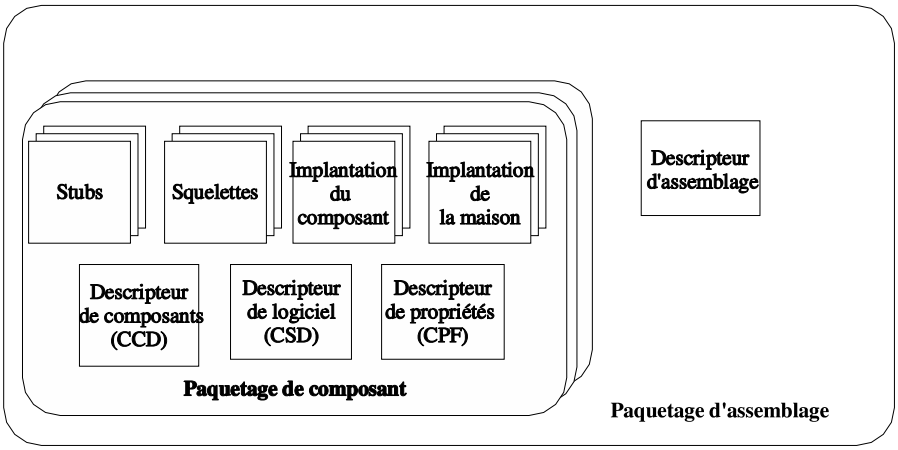

FIG. 2.8 – Paquetages de déploiement

La section suivante présente les différents déscripteurs de déploiement.

2.5.1.1 Descripteur d'assemblage de composants

Il d'écrit un assemblage des composants c'est à dire les éléments de description des composants, de leurs connexions et de leurs partitionnement. Le CAD (*Component Assembly Decriptor*) est un fichier XML décrivant :

- les paquetages de composants constituant l'assemblage, désignés par leur nom (nom de fichier),
- des informations de distribution. Les composants peuvent être déployés sur différentes machines et dans différents processus. Les informations de distribution décrivent quels paquetages de composants instancier, dans quelle machine et dans quel processus,
- les maisons de composants à installer,
- les composants à instancier et le nombre d'instances pour chaque composants,
- les connexions synchrone et /ou asynchrones à faire entre ces instances de composants, ci celles-ci existent.

2.5.1.2 Descripteur de composant CORBA

Le contenu du Descripteur de Composant CORBA (CCD) indique :
- les propriétés fonctionnelles du composant, son type, ses ports et sa maison,
- les propriétés non fonctionnelles :
  - catégorie du composant,
  - politique de cycle de vie,
  - politique et informations de persistance,
  - politique transactionnelle des opérations et des ports d'événements,
  - politique de threading,
  - politiques de sécurité.

Le CCD est en partie généré par le compilateur CIDL. Le développeur doit le compléter par les proprietés non fonctionnelles du composant. Les informations du CCD permettent, d'une part, de publier les fonctionnalités du composant en question, et d'autre part, de configurer le conteneur sur la base des propriétés non fonctionnelles, puisque c'est lui qui en prend la charge.

2.5.1.3 Descripteur logiciel du composant

Le Descripteur Logiciel de Composant (CSD) fournit une vue logicielle du composant . Le développeur doit y décrire les informations suivantes :
- des informations sur les fournisseurs du composant tel que l'auteur, la compagnie, la licence,..,
- le fichier IDL qui contient la déclaration du composant,
- le descripteur CCD du composant,
- des informations sur l'implantation du composant telle que le système d'exploitation supporté, le type de processeur supporté, le langage de programmation et sa version,
- la classe d'implantation de la maison, ainsi que le point d'entrée de la maison. Le point d'entrée de la maison est l'opération qui permet de la créer.

Les informations du Descripteur Logiciel de Composant concernent l'implantation du composant doivent permettre l'installation dynamique et automatique de l'environnement de déploiement et d'exécution.

2.5.1.4 Descripteur de propriétés du composant

Le descripteur de propriété du composant (CPF) décrit le paramétrage des composants et des maisons de composants. Il est utilisé pour configurer un composant ou une maison : positionnement des attributs. Il fournit des propriétés par défaut qui peuvent être modifiées à l'exécution. Les informations contenues dans ce descripteur seront utilisées par les objets de configuration [CCM02].

2.5.2 Déploiement d'applications

Le déploiement se fait à l'aide d'un outil de déploiement fourni par le fournisseur de l'ORB. Cet outil déploie les composants et fait leurs assemblages sur les sites cibles. Cette application de déploiement est une application cliente qui utilise les services d'objets présents sur les sites pour installer, instancier, puis configurer les composants et les connexions. Les quatre étapes de base pour le déploiement sont :
- la définition et le choix des sites d'installation,
- l'installation de ses implantations où cela est nécessaire,
- la création et configuration du conteneur et instanciation des maisons et des composants,
- la connection des composants .

Ces étapes sont réalisées par l'outil de déploiement fourni par une plate-forme CCM.

2.5.2.1 Séquences de déploiement

La séquence de déploiement est mise en oeuvre par la coopération des objets de déploiement voir figure 2.9, et par l'utilisation des différents descripteurs de déploiement.

le déploiement se fait en différentes étapes.

1. L'application de déploiement interragit avec l'opérateur pour connaitre les machines sur lesquelles déployer chaque paquetage de composant, puis l'application de déploiement lance l'installation physique des paquetages de composant sur les sites cibles, et invoque *ComponentInstallation : :install* sur chaque site afin de leur fournir le nom du paquetage de composants installé (étape 1).
2. Les implantations de composants sont installées sous forme d'archives sur les sites où elles sont requises. L'outil de déploiement utilise pour cela les instances de **ComponentInstallation** disponibles sur les sites concernés.
3. Une fabrique d'assemblage est utilisée pour créer une instance d'**Assembly** sur un site (une seule pour toute l'application) en utilisant sa fabrique *AssemblyFactory* (étapes2-3). Lors de la création, l'URL du descripteur d'assemblage est fournie.
4. Le descripteur d'assemblage est utilisé comme une recette pour déployer l'application. En se basant sur ce descripteur, l'objet *Assembly* créé est considéré comme un coordinateur d'assemblage à qui l'application de déploiement va demander de lancer le processus d'assemblage en invoquant *Assembly : :build*.
5. L'objet *Assembly* crée un serveur en invoquant *ServerActivtor : :create_component_server* (étape5), puis crée une instance de conteneur par l'opération *Component_Server : : Create_container* (étape 7).
6. Une fois le conteneur crée, l'objet Assembly crée les maisons de composant à travers l'opération *container : :install_home* (étape 9).
7. L'objet *Assembly* instancie les composants (étapes 11-12) en utilisant leurs maisons.
8. Configuration des attributs des composants en leurs affectant les valeurs spécifiées dans le CPF, puis l'objet *Assembly* connecte les instances de composants et les enregistrent auprès des services de recherche sur propriété spécifiés dans le CAD.
9. L'achèvement du processus de déploiement est traduit par l'appel à *configuration_complete()* (étape13).

2.6 Implémentation OpenCCM de la spécification CCM

Il existe actuellement différentes plates-formes qui implémentent la spécification CCM. Aucune de ces plates-formes ne supporte la totalité de la spécification. OpenCCM fut la première implémentation publique de la spécification CCM, c'est une plate-forme Opensource basée sur Java, elle a été implémentée par une équipe du LIFL[4] .

Le schéma de la figure 2.10 illustre les chaînes de production, d'exécution et de déploiement d'OpenCCM [Fli03].

La chaîne de production contient le modèle abstrait et le modèle d'implantation, alors que la chaîne d'exécution et de déploiement contiennent respectivement le modèle d'exécution, le modèle

[4]Laboratoire d'Informatique Fondamentale de Lille

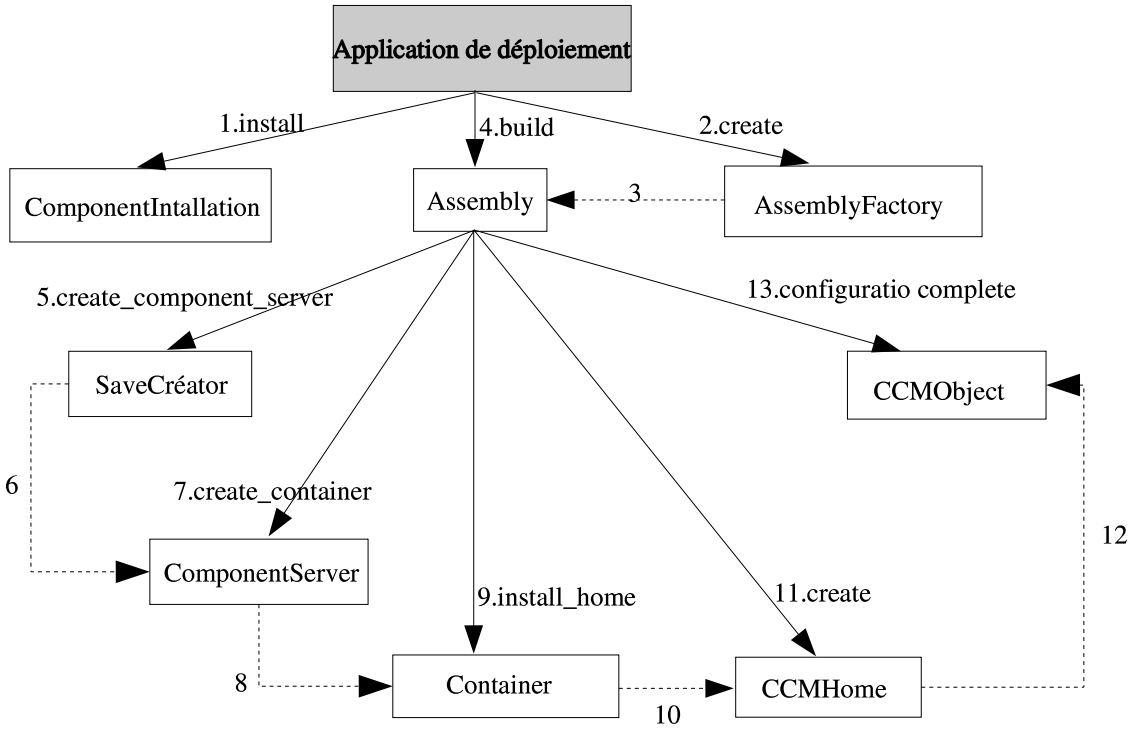

FIG. 2.9 – Le processus de déploiement guidé par les objets de déploiement

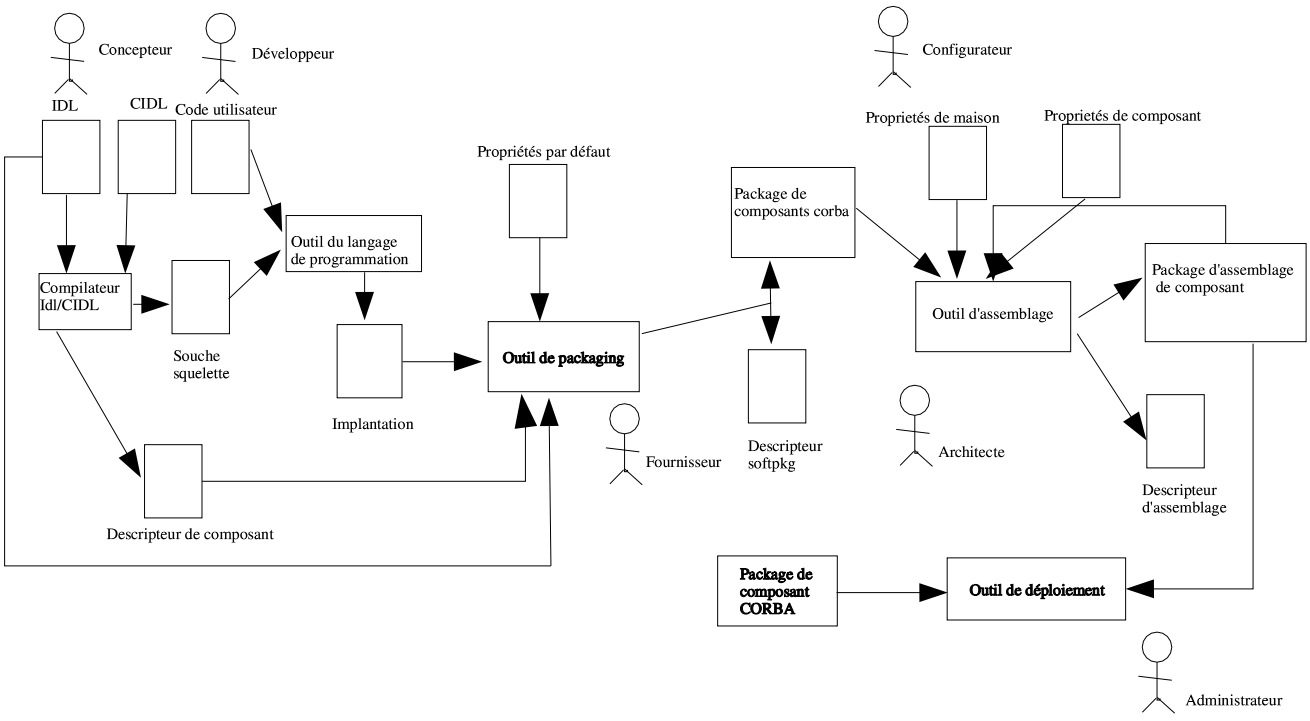

FIG. 2.10 – Étapes de production et de déploiement des applications

d'assemblage et de déploiement.
Dans ce qui suit nous allons reprendre chacune de ces chaînes de production pour voir ce qui est pris en charge automatiquement par la plate forme OpenCCM et ce qui est écrit par le développeur.

2.6.1 Chaîne de production

Cette section explique la chaîne de production d'OpenCCM.
A l'entrée de cette chaîne de production nous avons plusieurs fichiers de description :

- fichier IDL3 contenant la description de composants CORBA,
- fichier PSDL, contenant l'état de persistance des composants (la persistance n'est pas encore prise en compte par OpenCCM),
- fichier CIDL, contenant la structure d'implementation des composants.

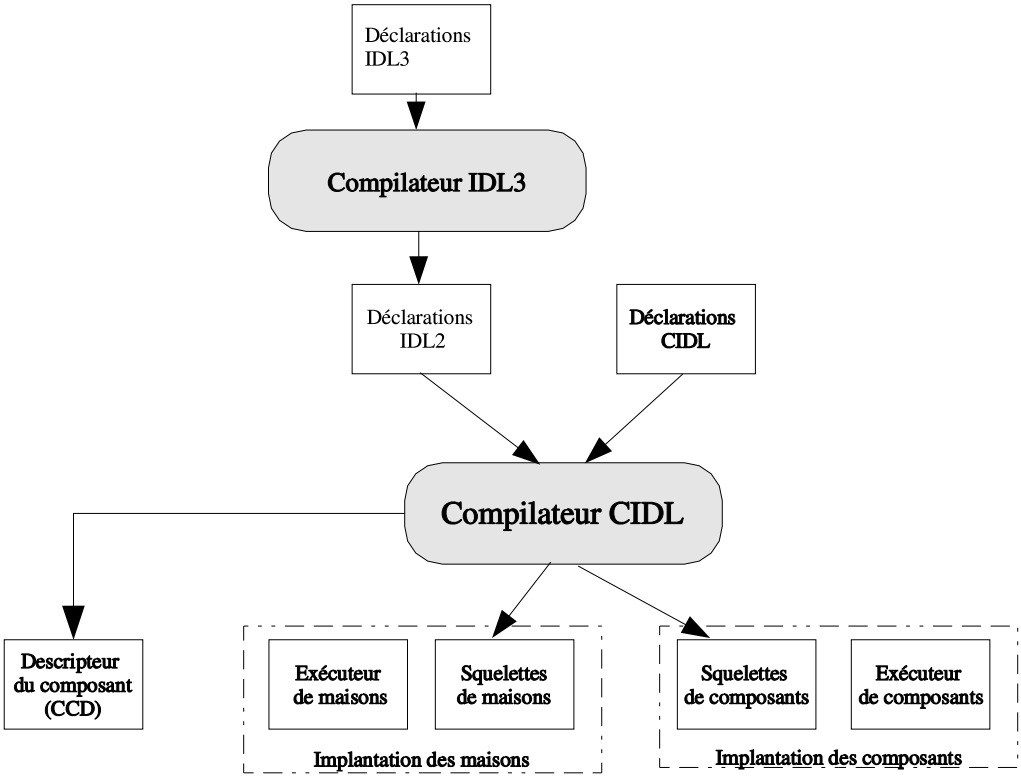

FIG. 2.11 – Chaîne de production d'OpenCCM

la compilation du fichier IDL3 permet en premier lieu de générer une projection vers l'IDL2 de ce fichier, par la suite ce dernier sera compilé avec le fichier CIDL afin de générer :

- les squelettes, qui prennent en compte les proprietés non fonctionnelles,
- les souches qui font appel au code fonctionnel,
- le descripteur de composant CORBA.

2.6.2 Chaîne d'exécution

La chaîne d'exécution d'OpenCCM comprend le modèle de conteneur de la spécification CCM. Comme il a été décrit dans la section 2.3, le conteneur représente l'environnement d'exécution pour une instance de composants CORBA. Cet environnement est implanté comme un serveur d'application et/ou une plate-forme de développement. Toutes les instances de composants, quel que soit

leurs type, sont crées et gérées par un conteneur, de plus le conteneur offre un ensemble standard de services aux instances de composants qu'il héberge.

2.6.3 Chaîne d'assemblage et de déploiement

Comme il a été dit dans la section 2.5 l'assemblage permet de décrire comment des composants simples sont assemblés pour former une application, alors que le déploiement décrit comment installer un composant dans un conteneur et pouvoir exploiter ce dernier ainsi que l'installation des différents paquetages dans les sites cibles. Cette chaine est constituée de plusieurs scripts permettant le déploiement des composants dans les sites dédiés.

La commande CCM_install constitue la première étape de cette chaine, elle permet d'installer l'infrastructure de déploiement par la création du repertoire OpenCCM_CONFIG_DIR et un sous repertoire ComponentServer qui contiendra les archives des paquetages installés par le processus de déploiement, puis plusieurs scripts sont lancés exécutant les étapes de la section 2.5.2.1 afin de déployer l'application.

2.7 Conclusion

Les intergiciels ont progressivement évolués d'une approche orientée objet vers une approche orientée composant. C'est le cas des *Entreprise Java Beans* (EJB) et de *CORBA Component Model*. De tels intergiciels vantent l'utilisation de serveurs de containers pour héberger les instances de composants et prendre implicitement en charge les services systèmes. Ainsi, les implantations de composants peuvent enfin ne contenir que leur logique métier. De plus les stratégies des services systèmes requis sont exprimées dans des descripteurs externes aux implantations. Il devient ainsi possible d'appliquer des stratégies systèmes différentes sur une même implantation de composant et donc de disposer de plusieurs comportements non fonctionnels sans modifier le code, et cela en utilisant un vocabulaire XML pour décrire les besoins des instances de composants en terme de sécurité, de transaction et de persistance. Alors, les paquetages de composants sont déployés dans les serveurs de conteneur. Ces conteneurs sont configurés en utilisant les descripteurs pour fournir correctement les services systèmes aux instances de composants.

La spécification CCM offre un fort potentiel pour couvrir globalement le processus de description, de production, de déploiement et d'exécution des composants répartis et hétérogènes. Cependant elle n'est pas à ce jour finalisée.

Dans l'état actuel, OpenCCM offre une chaîne ouverte de production des composants CORBA ainsi qu'un environnement flexible pour le déploiement et l'exécution de ceux-ci. La chaîne de production s'articule autour d'un référentiel pour le langage OMG IDL3 alimenté par un compilateur OMG IDL3 et visité par un ensemble de générateurs d'OMG IDL2 et de squelettes étendus Java, Le déploiement des composants est exprimé de manière flexible par la description du téléchargement des archives de composant, l'instanciation, la configuration et l'interconnexion des composants s'exécutant au sein de serveurs . L'ensemble est construit au dessus des intergiciels CORBA, il offre ainsi à la communauté système une plate-forme libre pour l'expérimentation de future génération d'intergiciels comme l'automatisation de la génération de code d'enregistrement des composants auprés des différentes services de recherche sur propriété et la prise en compte de cet enregistrement comme étant une propriété non fonctionnelle, ce qui fait le sujet de de stage.

Chapitre 3

Enregistrement des composants

Les systèmes d'information modernes sont fréquemment installés sur des réseaux à grande échelle. La mise en place de ces systèmes sur un réseau étendu se base de plus en plus sur l'utilisation d'applications distribuées complexes construites à partir de composants logiciels autonomes et coopératifs pouvant être placées sur de multiples serveurs distants. Par ailleurs chaque composant doit pouvoir être retrouvé par les clients, pour cela il faut l'enregistrer auprès de différents types de services d'enregistrement.
Il existe plusieurs manières d'enregistrer un composant. Enregistrer son nom symbolique peut être suffisant quand il est possible, pour un client, de le retrouver selon ce nom seulement. La recherche de ce composant peut être comparée à la recherche dans des pages blanches où il faut connaître le nom de la personne afin de pouvoir retrouver son numéro de téléphone. Mais parfois ce nom symbolique n'est pas suffisant. Dans ce cas les propriétés du composant sont enregistrés, ce qui est équivalent à un enregistrement dans des pages jaunes.
Dans ce chapitre nous allons voir les deux types de service d'enregistrement : le service de nommage et le service de courtage. Cette étude est nécessaire car l'automatisation de l'enregistrement va se faire auprès de ces deux services. Nous abordons par la suite la manière dont CCM décrit l'enregistrement des composants afin que le lecteur puisse comprendre notre approche d'automatisation de cet enregistrement et ce que nous apportons de plus par rapport à ce qui existe déjà.

3.1 Services d'enregistrement

L'enregistrement d'un composant peut se faire auprès de différents types de service afin de pouvoir le retrouver. La recherche peut s'effectuer selon le nom du composant, dans ce cas il faut l'enregistrer auprès des pages blanches (service de nommage), ou bien selon son type et ses propriétés, dans ce cas son enregistrement doit s'effectuer auprès des pages jaunes (service de courtage).

3.1.1 Service de nommage

Les systèmes répartis ont besoin d'un mécanisme leur permettant de désigner les entités qui les composent, ceci est rendu possible grâce au service de nommage.
Le service de nommage permet de retrouver les ressources en leurs associant des noms symboliques manipulables par des programmes, ces services sont rendus par des serveurs qui tiennent à jour des catalogues mettant en relation des noms externes avec leurs résolutions (des noms internes ou des adresses).

3.1.2 Service de courtage

Le service de courtage est connu comme étant le service de recherche sur propriétés [ODP95]. Il permet de découvrir dynamiquement les services offerts sur un réseau. Quand un serveur désire publier son offre, il l'enregistre auprès du service de courtage, cet enregistrement s'appelle aussi *exportation de service*, et l'offre enregistrée *offre de service*.

3.1.2.1 Service de courtage CORBA

Les besoins d'utiliser des pages jaunes dans l'environnement CORBA à donné naissance à un service de courtage propre à CORBA. Le principe de fonctionnement de ce service est comme suit : une application serveur exporte vers le trader la description et la référence d'un objet qui lui est propre. Les applications clientes désirant utiliser cet objet interrogeront le trader en fournissant un filtre de sélection. Une fois que l'application aura la référence sur l'objet demandé, elle pourra l'utiliser à travers le bus CORBA [OMG00]

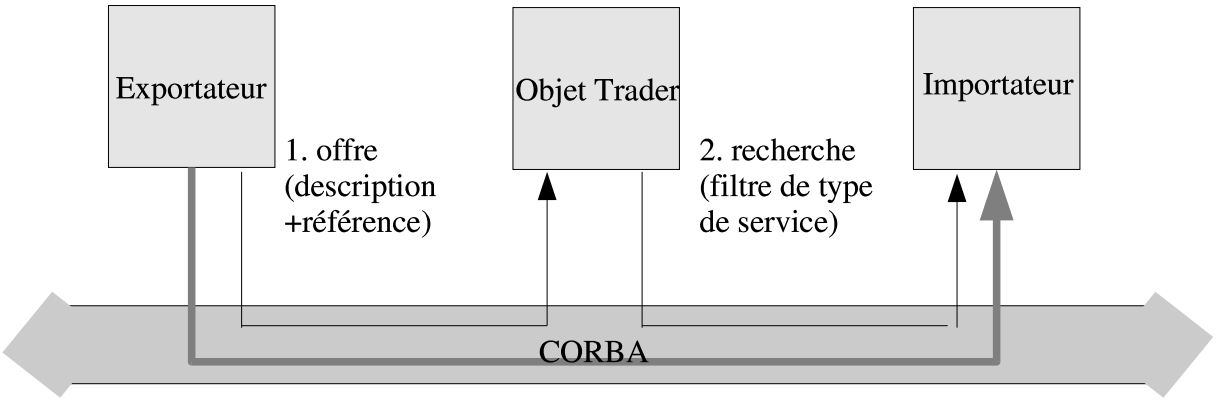

FIG. 3.1 – Fonctionnement du trader CORBA

3.1.2.2 Types de services

Les services exportés sont décrits sur un réseau d'une manière abstraite au niveau du trader à l'aide d'une structure appelée *type de service*. Les types de services peuvent être comparés aux catégories des pages jaunes. La recherche d'un hôtel dans les pages jaunes, par exemple, suppose qu'il y ait un certain nombre d'informations communes à tous les hôtels comme un nom, un numéro de téléphone, une adresse, un prix pour une nuit dans une chambre, le type de service d'une offre représente les types d'informations qu'un importateur peut utiliser pour rechercher un service au niveau du trader. Les types de service sont organisés d'une manière hiérarchique selon un arbre d'héritage.

L'interface ServiceTypeRepository joue le rôle de dépositaire de stockage des types d'offres de services et assure la vérification du typage lors de l'exécution des requêtes d'exportation et de recherche. Dans ce dépositaire, un type d'offre de service est caractérisé par quatre éléments [OMG00].

- Un nom unique identifiant le type de service.
- Des super types hérités permettant de définir une classification des types par héritage multiple.

– Une interface OMG IDL à laquelle les services exportés doivent se conformer.
– Zero ou plusieurs types de propriétés définissant l'aspect comportemental du service.

Le type de propriété est un triplet <nom, type, mode>. Le mode indique si la propriété est optionnelle ou obligatoire lors de l'enregistrement d'une offre de service de ce type et si elle peut changer de valeur après la publication de l'offre. Les modes de propriétés sont les suivants :

– normal : la valeur de la propriété est optionnelle, elle peut être modifiée ou supprimée.
– readonly : idem à normal mais si la valeur de la propriété est fournie à l'exportation alors elle ne peut plus être modifiée.
– mandatory : une valeur doit obligatoirement être affectée à la propriété lors de l'exportation.
– mandatory readonly : idem à mandatory mais non modifiable.

L'interface ServiceTypeRepository [OMG00] offre plusieurs opérations, pour gérer les différents types de service (voir détails dans la section 3.1.2.5).

3.1.2.3 Exportation de services auprès du Trader

Afin d'enregistrer un composant, il faut l'exporter au niveau du service de recherche sur propriété sous forme d'une offre de service

Une offre de service permet d'associer, pour un type de service donné, la référence de l'objet CORBA avec les valeurs de propriétés qui le caractérisent. Chaque offre de service a un identifiant unique renvoyé à l'exportateur par le trader lors d'une demande d'enregistrement. Cet identifiant permet à l'exportateur de modifier ou retirer l'offre. L'exportation de service se fait grâce à l'interface *Register* (voir section 3.1.2.5).

3.1.2.4 Propriétés dynamiques

Généralement, la propriété d'une offre de service a une valeur fixe enregistrée par l'exportateur. Cette valeur ne change pas sauf si quelqu'un la modifie explicitement.

Pour supporter les propriétés qui changent de valeur fréquemment, les traders offrent la possibilité d'utiliser des propriétés dynamiques[OMG00]. Les valeurs de ces propriétés ne résident pas dans le trader, elles seront évaluées à la demande, lors d'une recherche, à travers un objet d'évaluation de propriétés dynamiques désigné par l'exportateur.

La structure d'une propriété dynamique contient l'interface de l'évaluateur dynamique de propriétés, le type de donnée des propriétés dynamiques retournées et des informations supplémentaires d'implémentation. Quand la valeur d'une propriété est demandée, le trader invoque la méthode *evalDP* à partir de l'interface *dynamic_PropertyEval*.

3.1.2.5 Les API du service de courtage

Le trader CORBA offre onze interfaces faisant partie du module *CosTrading*. Ces interfaces permettent l'importation et l'exportation de services, d'offre de types de services ainsi que la configuration du trader. il y a deux types d'interfaces :

– **Interface d'administration**, permet de contrôler le fonctionnement du service de courtage et de définir les types de services.
– **Interface d'utilisation**, permet aux applications clientes de dialoguer avec le service.

Les interfaces d'utilisation les plus importantes sont :

– **L'interface Lookup** : cette interface permet l'importation de services. Elle n'offre que l'opération *query* permettant aux importateurs de rechercher des objets.
– **L'interface Register** : cette interface permet aux différentes applications distribuées de publier leurs services et de mettre à jour le contenu du dépositaire du service de courtage. La

publication d'un service (enregistrement du service auprès du service de courtage) se fait à l'aide de l'opération *export* nécessitant les paramètres suivants :

1. Le nom du type de service,
2. Une référence à l'interface qui offre le service,
3. Des valeurs pour les propriétés du service.

   Il est possible de modifier une description d'un service avec l'opération *modify* ou supprimer des services du dépositaire avec l'opération *withdraw*.

– **L'interface OfferIterator** : cette interface permet le parcours des résultats de la recherche.
– **L'interface Link** : cette interface permet de constituer une fédération de services de recherche.
– **L'interface DynamicPropEval** : cette interface permet de calculer, à l'exécution, la valeur courante d'une propriété dynamique au travers de l'opération *EvalDP*.
– **L'interface Proxy** : cette interface permet de déterminer la référence réelle d'un objet offrant un service.

3.2 Enregistrement dans CCM et OpenCCM

La spécification CCM n'évoque pas explicitement l'enregistrement des composants auprès des services de nommage et de courtage, l'étude du modèle de déploiement nous a permis de déduire que cet enregistrement est automatique et ceci en décrivant les services auprès desquels on veut enregistrer un composant dans le fichier CAD.
Au niveau du déployeur, un code générique permet d'enregistrer chaque composant après sa configuration.
Dans ce qui suit nous allons évoquer les différentes façons qu'OpenCCM propose pour déployer une application et enregistrer les composants.

L'enregistrement des composants dépend de la manière de déployer l'application. Comme il a été spécifié dans [Ope03] une application peut être déployée de deux façons.

– En utilisant un fichier java où le développeur écrit la portion de code relatif à la création, initialisation, instantiation et enregistrement des composants auprès des différents services.
  Dans cette méthode l'enregistrement est toujours à la charge du développeur.
– En utilisant un fichier XML (le fichier *Component Assembly Descriptor*). Dans ce cas, le programmeur doit remplir les différents fichiers de description, en précisant dans le fichier CAD les services auprès desquels chaque instance de composants doit être enregistré. Quand une instance de composant est enregistrée auprès du service de courtage, les différentes propriétés à exporter sont définies et initialisées dans le fichier de description du composant comme l'illustre la figure 3.2 .
  Lors du déploiement de l'application, le fichier XML est parcouru par le déployeur afin d'en extraire les différentes informations qu'il contient, entre autres celles concernant l'enregistrement, puis effectue l'instantiation, l'initialisation et l'enregistrement des différents composants auprès des services spécifiés par le descripteur.
  Certes l'enregistrement est automatique mais les procédures d'enregistrement ne sont pas générés automatiquement, elles sont figées au niveau du déployeur de l'application, de plus les valeurs des propriétés exportées sont statiques, figées dans le CAD et après leurs exportation, les éventuels changement de valeur ne sont pas automatique, enfin cela ne permet pas de prendre en compte les propriétés dynamiques.

Nous proposons que l'enregistrement soit géré par le code même du composant (voir chapitre 4).

Plus concrètement, dans OpenCCM, ces deux méthodes de déploiement existent, l'enregistrement auprès du service de nommage a été pris en compte mais le nom externe du composant ne peut

```
.....
<componentproperties>
        <fileinarchive name= « AaaProperties.cpt »/>
<componentproperties>
<registercomponent
        <registerwithnaming name= « sink »/>
        <registerwithtrader>
                <traderproperties>
                        <traderproperty>
                                <traderpropertyname>ppm</traderpropertyname>
                                <traderpropertyvalue>10</traderpropertyvalue>
                        </traderproperty>
                </traderproperties>
        </registerwithtrader>
</registeromponent>
....
```

FIG. 3.2 – Portion de code du fichier CAD

pas être changé après son exportation. L'enregistrement automatique auprès du service de courtage n'a pas encore été implémenté, il sera probablement intégré ultérieurement.

3.3 Contrat de courtage TORBA

TORBA (*Trader Oriented Request Broker Architecture*) est un serveur de composant de courtage implémenté par un groupe du LIFL. Nous nous sommes inspiré du contrat de courtage TORBA afin de réaliser la description de l'enregistrement dans notre proposition.
L'objectif de TORBA [LMMG01][LM01] est d'offrir une méthodologie et un environnement de conception, de production, de déploiement, d'exécution et d'administration de composants de courtage spécialisés en fonction des besoins applicatifs et ceci à travers le concept de contrat de courtage permettant de définir précisément les besoins des fournisseurs d'offre de services ainsi que ceux des applications clientes, ces contrats de courtage sont clairement spécifiés lors de la phase de conception des applications au même titre que les contrats d'interfaces OMG IDL. La définition de ces contrats se fait à l'aide du langage déclaratif TDL (*TORBA Definition Langage*). Cela se matérialise par la définition de type d'offres de services. D'une part un type d'offres de services permet de clairement indentifier et regrouper les propriétés caractérisant un service. D'autre part, un type contient la liste des requêtes d'importation couramment utilisées par les applications clientes. Enfin, une application cliente peut étendre un type d'offres pour y ajouter des requêtes d'importation spécifiques à ses besoins. Cette notion identifiée par le vocable de « vue »permet la combinaison d'un certain nombre de requêtes d'importation d'un type d'offre, donc une vue permet à une application de définir ses besoins supplémentaires. Il est donc possible d'ajouter modulairement des requêtes de recherche. la figure 3.3 illustre un exemple d'un contrat de courtage TDL pour un service d'importation

```
Abstract offer Peripherique {
        property string          nom;
        query toutes() is TRUE;
};

offer Imprimante : Peripherique {
        interface ServiceImpression;
        property boolean          marche;
        property float            cout_par_page;
        property unsigned short   vitesse;
        dynamic property unsigned long job;
        query marche() is mache ==TRUE;
        query rapides(in unsigned short s) is vitesse > s;
};
view MesImprimantes : Imprimante{
        query mes_preferees is rapides(10);
```

FIG. 3.3 – Un exemple de contrat de courtage TDL

3.4 Conclusion

Dans ce chapitre nous avons parlé de l'enregistrement des composants auprès de deux services de recherche, il s'agit du service de nommage et du service de courtage. OpenCCM a prévu, pour un composant, ces deux types d'enregistrement, bien qu'il n'y ait pas de description explicite de la manière d'enregistrer, il se trouve qu'il existe deux méthodes pour enregistrer un composant, la première est complètement à la charge du développeur alors que la seconde fait un enregistrement automatique par l'outil de déploiement.
Les point faibles de cette deuxième méthode réside dans le fait que le code d'enregistrement est figé, externe au composant et appelé par le déployeur, de plus les propriétés ne peuvent pas être changées après leur exportation.
Dans le but de remédier à cela et à rendre la génération du code d'enregistement automatique, interne au composant et pris en compte comme étant une propriété non fonctionnelle, nous présentons dans le chapitre suivant notre proposition permettant aussi la modification automatique des propriétés après leurs exportation et la prise en compte des propriétés dynamiques.

Chapitre 4

Automatisation de l'enregistrement

Après avoir étudié la spécification CCM et la manière d'enregistrer les composants auprès des différents services, nous présentons la conception de notre approche afin d'automatiser cet enregistrement.
L'approche que nous proposons repose sur la description de l'enregistrement pour chaque composant. La compilation de cette description génère du code permettant d'enregistrer chaque composant auprès des services choisis, de modifier les propriétés exportées et d'annuler le contrat d'enregistrement.

Dans ce chapitre nous spécifions le niveau où doit se faire la description de l'enregistrement, selon ce niveau, le langage de description est défini, nous présentons par la suite le contenu de la description et le code généré après sa compilation, enfin nous désignons l'entité qui se charge d'enregistrer les composants.

4.1 Description de l'enregistrement

Afin de rendre l'enregistrement automatique notre approche utilise un fichier de description désignant les différents services auprès desquels se fait l'enregistrement ainsi que les propriétés à enregistrer .
Selon l'étude de la spécification CCM trois niveaux de descriptions sont possibles (description au niveau du modèle de déploiement, du modèle abstrait et du modèle d'implantation), mais il n'y aura que le niveau de description répondant aux exigences suivantes qui sera retenu :

- La compilation de la description doit permettre la génération du code d'enregistrement à l'intérieur du composant puisque chaque méthode d'enregistrement est propre au composant.
- Il faut que les informations d'enregistrement soient décrites avec des propriétés non fonctionnelles.

Dans les paragraphes suivants nous allons répondre à la question : à quel niveau doit être effectué l'enregistrement et avec quel langage de description ?

4.1.1 Description au niveau du modèle de déploiement

A première vue, la description de l'enregistrement d'un composant peut se faire facilement au niveau du modèle de déploiement, en effet au niveau de ce modèle on peut faire, pour chaque composant, une description des différentes informations à exporter vers les services de recherches sur propriété, et cela en utilisant un descripteur XML.

Cette solution comporte certains inconvenients, car le code généré après la compilation de la description sera externe au composant, en effet selon le processus de développement de CCM (figure2.10) les souches et les squelettes sont déja générés, de plus retenir cette solution n'apporte rien de nouveau car cela ressemble à ce qui est utilisé actuellement dans OpenCCM (pour l'enregistrement des composants auprès du services de nommage) avec les inconvénients suivants :

- pas de modification des propriétés exportées,
- les valeurs des propriétés sont définis statiquement.

En effet puisque les souches et les squelettes ont déja été générés, le code généré sera d'une part externe au composant et d'autre part on est dans l'obligation de générer du code générique, au niveau du déployeur. De ce fait les valeurs des propriétés à exporter sont définies statiquement, sans possibilité de gérer leurs modification.
En conclusion la description de l'enregistrement doit se faire avant la génération des souches et des squelettes.

4.1.2 Description au niveau du modèle abstrait

Il est possible d'effectuer la description d'enregistement des composants au niveau du modèle abstrait, en effet à ce niveau les souches et les squelettes ne sont pas encore générées, donc la compilation de la description peut générer du code d'enregistrement non générique, propre à chaque composant et faisant partie intégrante de ce dernier.
Cette description peut se faire en utilisant un fichier XML. Un parseur doit parcourir cette description afin d'en extraire les informations utiles et générer un autre fichier de description pour qu'il puisse être compilé dans le but de générer les méthodes d'enregistrement du composant et de modification de ses propriétés.

Cette solution semble satisfaisante, mais puisqu'on va générer un autre fichier de description, après le parcours du fichier XML, pourquoi ne pas utiliser ce fichier dès le départ. Donc pour décrire l'enregistrement, une description IDL est utilisée, ce qui nécessite l'ajout de nouvelles productions, concernant l'enregistrement, dans la grammaire de l'IDL. La compilation du fichier IDL permet de générer du code interne au composant, ce qui répond à la première condition, mais les propriétés décrites à l'aide de l'IDL sont considérées comme étant des propriétés fonctionnelles, ce qui ne répond pas à la deuxième exigence. Insérer la description de l'enregistrement au niveau du modèle abstrait ne semble pas approprié pour obtenir les résultats voulus.

4.1.3 Descripteur au niveau du modèle d'implantation

La troisième possibilité est de décrire l'enregistrement au niveau du modèle d'implantation.
Comme le modèle abstrait, le modèle d'implantation permet de générer du code interne aux composants, se traduisant par les souches et les squelettes. De plus il permet de décrire les propriétés non fonctionnelles prises en compte par le composant, ce qui répond bien aux deux conditions posées au départ. Donc le modèle d'implantation est le niveau utilisé afin de décrire les propriétés et les services auprès desquels se fait l'enregistrement de chaque composant, et ceci en utilisant un fichier de description CIDL dont la compilation génère des souches et des squelettes contenant des méthodes d'enregistrement du composant auprès des services spécifiés lors de la description, et des méthodes de modification des propriétés exportées.
Donc nous proposons une extension du langage CIDL afin de permettre une description non fonctionnelle, de l'enregistrement de chaque composant, pour générer le code relatif à l'enregistrement faisant partie intégrante du composant.

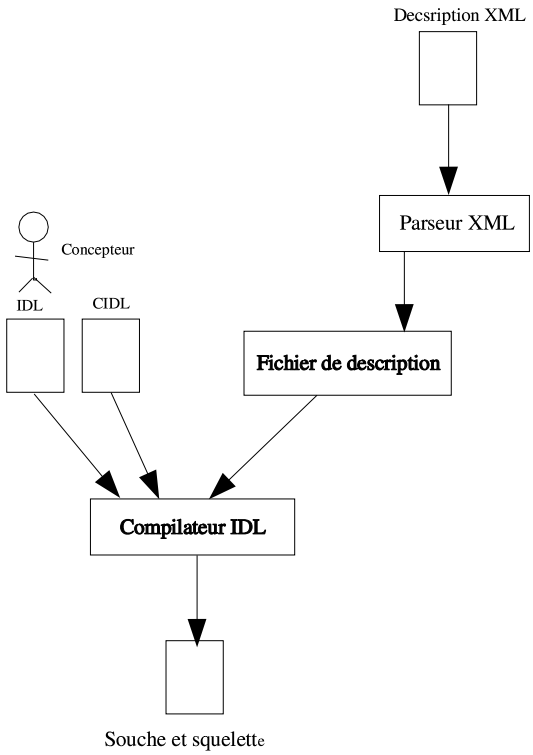

FIG. 4.1 – Descripteur XML au niveau du modèle abstrait

4.2 Contenu de la description

La description de l'enregistrement peut être vue comme un contrat d'enregistrement entre le composant et le/les services de recherche, ce contrat prend en compte les propriétés de chaque composant et cela se matérialise par la définition d'un contrat d'enregistrement.
Chaque propriété exportée est liée à l'un des attributs du composant, ils possèdent les même valeurs et la modification de l'un d'eux engendre la modification de la valeur de l'autre. Chaque propriété liée à un attribut doit être du même type que ce dernier et initialisée avec la même valeur.

La description d'enregistrement CIDL proposée contient les informations suivantes :

- les nom des services vers lesquels se fait l'exportation, c'est à dire le service de nommage et/ou le service de courtage,
- dans le cas du trading, l'offre d'enregistrement, c'est à dire les propriétés à exporter, leurs types(entier, chaîne de caractère ...), l'attribut auquel la propriété est reliée ainsi que le mode de la propriété (section4.2.1) qui spécifient si la propriété est modifiable ou bien dynamique. Contrairement à ce qui a été décrit dans la spécification CCM et pour plus de flexibilité, la proposition concernant l'enregistrement auprès du service de courtage prend en compte les propriétés dynamiques.

Dans le cas du service de nommage, le nom avec lequel un composant doit être enregistré auprès de ce service est une propriété exportée liée à un attribut du composant, le nom de cette propriété est fixe, il s'agit de « **namingExport** », donc la description d'un contrat d'enregistrement auprès du service de nommage contient une seule propriété nommée « **namingExport** »et doit être liée à un attribut du composant de type chaine de caractère.
La description de l'enregistrement a été inspiré du contrat TDL (*Torba Definition Langage*), la description proposée contient le mode de propriété dynamique comme TDL sans la description des

requêtes de recherche. Par rapport à TDL nous avons ajouté le lien entre les propriétés exportées et l'attribut du composant.

La description d'enregistrement est appelée **contrat d'enregistrement**, ce contrat est écrit soit directement dans le fichier CIDL en introduisant de nouvelles production dans la grammaire du CIDL ou bien un fichier externe est utilisé puis exporté dans le fichier CIDL. Le fichier externe utilise une grammaire inspirée du langage déclaratif TDL qu'on appellera **RDL** pour *Registration Definiton Langage*.

4.2.1 Contrat d'enregistrement

Le contrat d'enregistrement contient donc :

- le nom du service auprès duquel est effectué l'enregistrement,
- les propriétés à exporter,
- le lien entre les propriétés et les attributs du composant,
- le type de chaque propriété et son mode (dynamique ou en lecture seule).

Les propriétés peuvent être :

- **readOnly** : dans ce cas la propriété est en lecture seule, et ne peut plus être changée aprés son initialisation et exportation. Si une propriété n'est pas déclarée être en mode readOnly la modification de l'attribut auquel elle est associée engendre sa mise à jour.
  Par défaut, toutes les propriétés sont modifiables, ce qui implique la création d'intercepteurs dans le composant afin d'intercepter les modifications apportées par le client.
- **dynamic** : une propriété dynamique est une propriété dont la valeur n'est pas stockée auprès du serveur de courtage mais évaluée dynamiquement par l'intermédiaire d'évaluateurs.

Comme il a été mentionné précédement une propriété du contrat de courtage est liée avec un attribut du composant, ce lien est soit explicite, en utilisant le mot clé **link** ou bien implicite en donnant à la propriété le même nom que le composant auquel elle est liée.
Une propriété et un attribut liés doivent avoir le même type et le même mode, c'est à dire si la propriété est dynamique, l'attribut auquel elle est associée ne doit pas être en mode readOnly.
Lors de la compilation du contrat d'enregistrement, si on ne défini pas un lien explicite entre la propriété et l'attribut auquel elle est associée, le compilateur s'attend à trouver un attribut ayant le même nom, le même type et un mode compatible avec celui de la propriété, si ce n'est pas le cas une exception est levée.

Dans la section 4.2 nous avons évoquer deux manières d'insérer le contrat d'enregistrement, soit :

- L'écrire directement dans le fichier CIDL.
- L'écrire dans un fichier externe et l'importer dans le fichier CIDL.

Dans ce qui suit nous allons donner un exemple de contrat d'enregistrement pour les deux manières de procéder. La description IDL3 du composant est la suivante :

```
component Imprimante
{
   attribute string the_name;
   attribute int vitesse;
   attribute int job;
   atribute boolean marche;
};


home ImprimanteHome manages Imprimante
{
};
```

FIG. 4.2 – Description IDL3 du composant Imprimante

4.2.1.1 Contrat d'enregistrement dans le CIDL

L'exemple suivant illustre un contrat d'enregistrement pour le service d'impression, ce dernier est enregistré auprès du service de nommage et du service de courtage.

```
.
.
.
.
contract offerTraderImprimante of Imprimante with trader
{
        property string the_name link nom;
        property int vitesse;
        property dynamic int job;
        property boolean marche;
};

contract offerNamigImprimante of Imprimante with ns
{
        property the_name link nmingExport;
};
```

FIG. 4.3 – Contrat d'enregistrement dans le CIDL

4.2.1.2 Contrat d'enregistrement externe au CIDL

Quand le contrat de courtage est externe au fichier CIDL il est écrit dans le fichier RDL comme l'illustre la figure 4.4 , puis importé vers le fichier CIDL comme l'illustre la figure 4.5

```
Contract offerTraderImprimante
{
        property string nom;
        property int vitesse;
        property dynamic int job;
        property  boolean marche
}
```

FIG. 4.4 – Fichier RDL pour le contrat d'enregistrement

```
Import ImprimanteService.rdl
module....
{
.
.
.
        component Imprimante
        {
                registerTrader offerTraderImprimante
                {
                        property the_name link nom;
                }
                registerNS offerNamingImprimante
                {
                        property the_name link namingExport;
                }
        }
}
```

FIG. 4.5 – Importation du RDL dans le fichier CIDL

4.3 Génération de code

Comme il a été mentionné dans le section 4.2.1 la compilation du contrat d'enregistrement permet de générer des méthodes dans les squelettes et les souches des composants, ces méthodes concernent l'enregistrement, la suppression des contrats et la modification des propriétés.
Les méthodes générées par la compilation du contrat d'enregistrement sont citées ci dessous.

- Méthodes d'enregistrement auprès des services spécifiés dans le contrat d'enregistrement.
- Méthodes de modification du nom externe du composant dans le cas de son enregistrement auprès du service de nommage.
- Méthodes de modification de chaque propriété modifiable (qui n'est pas en lecture seule) dans le cas de l'enregistrement du composant auprès du service de courtage.
- Méthode de suppression « annulation »du contrat entre le composant et le service de recherche.

Plus de détails sur ces portions de code seront donnés dans la section 5.3.

4.4 Qui fait l'enregistrement ?

Après avoir décrit le contrat d'enregistrement et les différentes méthodes générées lors de sa compilation, il reste à déterminer l'entité qui prend en charge l'enregistrement, cette entité doit :

- Pouvoir intercepter les différents événements associés au composant afin de pouvoir lancer les méthodes susceptibles d'être déclenchées par ces événements.
- Pouvoir interagir avec le bus ORB afin d'offrir au composant un accès à ce dernier, ainsi qu'un accès aux différents services qu'il offre.
- Permettre de dire que l'enregistrement est une propriété non fonctionnelle.

Il y a deux entités susceptibles d'effectuer cette tâche.

- **La maison de composant** : en effet cette dernière gère le cycle de vie des composants et crée les différentes instances de chaque composant. Elle a connaissance de tous les composants instanciés, et elle peut pour chaque instance effectuer l'enregistrement auprès des services décrits dans son contrat d'enregistrement. Mais la maison n'a pas un accés direct au bus ORB (elle doit passer par le conteneur) et n'a aucune interaction avec les différents services externe, de plus elle ne peut pas intercepter tous les événements associées aux composants (comme la modification de ses attributs).
- **Le conteneur** : selon l'étude effectuée sur la spécification CCM (voir chapitre2) le conteneur répond à toutes ces exigences, en effet selon le principe de fonctionnement du conteneur, il permet d'intercepter les requêtes invoquées sur ses composants pour interposer les gestions non fonctionnelles d'une manière totalement transparente au composant. En effet pour gérer les transactions, le conteneur doit intercepter les appels à ces opérations et décider de commencer ou pas une transaction. De même pour la gestion des événements, le conteneur doit intercepter les événements produits et consommés pour y appliquer les politiques des événements spécifiques.

  Pour toutes ces raisons , le conteneur a été choisi pour effectuer l'enregistrement auprès des services de recherche. Ce dernier offre un accès aux services de nommage et de courtage à travers le bus ORB comme l'illustre la figure 4.6.

Concernant les méthodes générées par la compilation de notre contrat d'enregistrement le conteneur doit les exécuter à l'interception de certains événements illustrés par la figure 4.7 :

- à l'interception de *configuration_complete()*, le conteneur doit enregistrer le composant auprès des services spécifiés par le contrat d'enregistrement,
- à l'interception de la modification d'un attribut associé à une propriété, le conteneur doit lancer

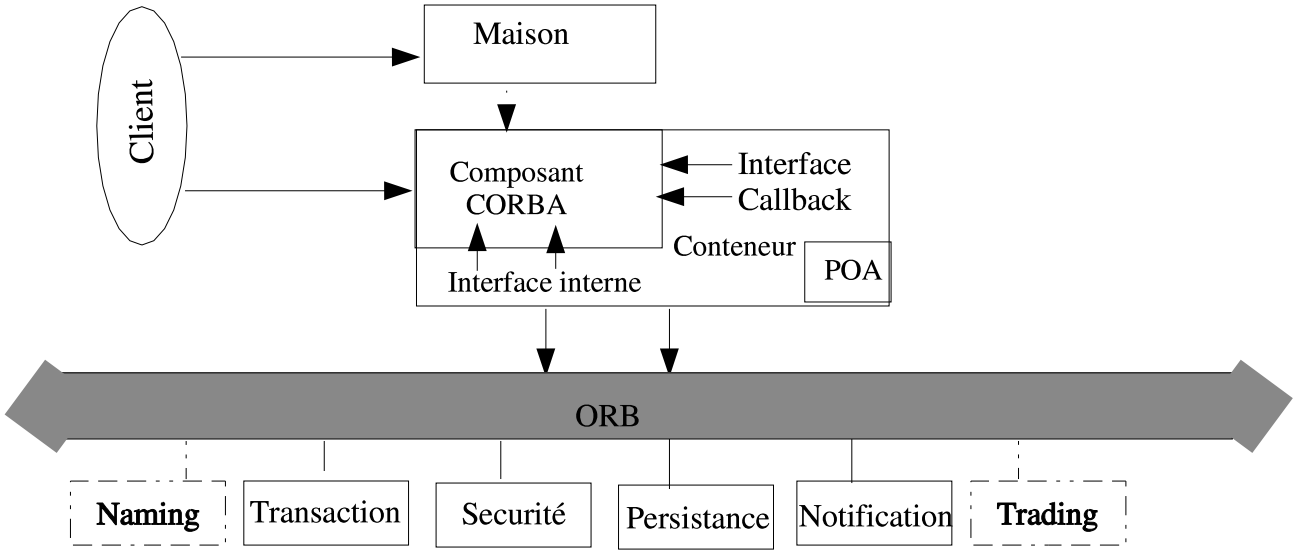

FIG. 4.6 – Interaction du conteneur avec le bus ORB

la procédure de modification de la propriété associée à l'attribut, générée lors de la compilation du contrat d'enregistrement associé au composant contenant l'attribut,

– à la réception de la suppression d'un composant (*ccm_remove()*), les différents enregistrements des offres de composants auprès des services de recherche doivent être supprimés.

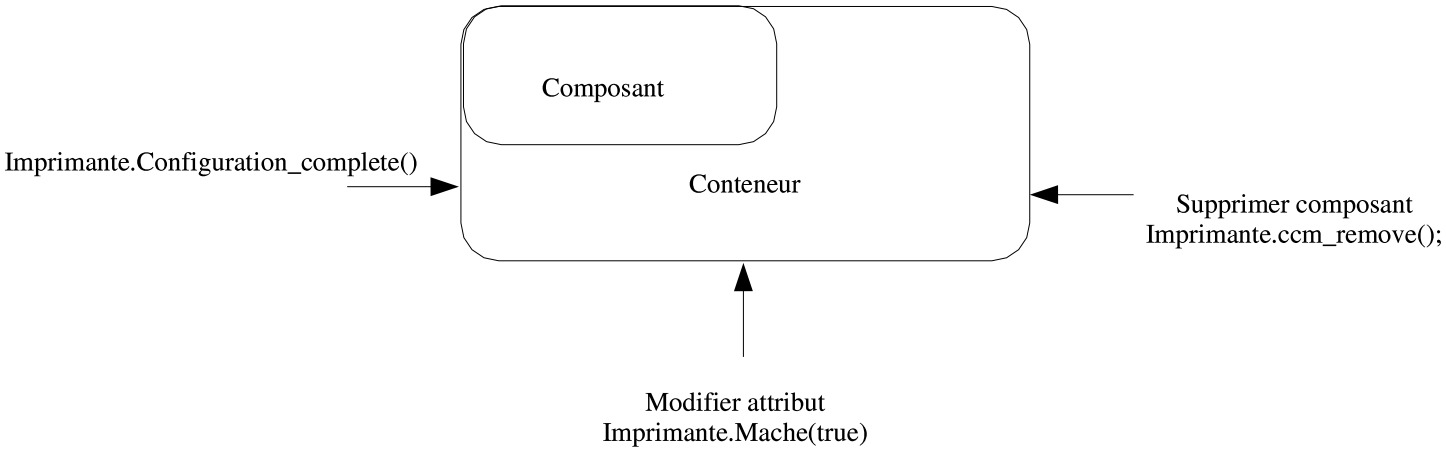

FIG. 4.7 – Interception des événements par le conteneur

4.5 Type de composant et enregistrement

Avant de clore ce chapitre nous allons essayer de répondre à une question qu'on s'est posé à un moment donnée au cour de l'élaboration de la proposition, la question est la suivante : est ce que tous les types de composant peuvent être enregistrés ? Si c'est le cas, à quoi pourrait servire l'enregistrement des composants de type session et service, puisqu'ils n'ont pas un état persistant et leur contenu est volatile, donc on s'est résolu à dire que seul le type entité et process sont concernés par l'enregistrement, car enregistrer un composant dont l'état n'est pas persistant ne sert à rien.

4.6 Conclusion

Dans ce chapitre nous avons présenté notre approche d'enregistrement automatique des composants, en le rendant comme étant des propriétés non fonctionnelles prises en compte par le conteneur de l'application.

Le point fort de notre approche est le fait que les méthodes d'enregistrement soient générées automatiquement et font partie intégrante du composant.

Par rapport à l'enregistrement dans OpenCCM qui malgré son automatisation en utilisant une description XML, notre approche est plus dynamique et permet plus de souplesse. En effet dans OpenCCM lorsqu'on automatise, les propriétés exportées sont statiques, une fois exportées l'utilisateur ne peut pas les changer, de plus les propriétés dynamiques ne peuvent être prises en compte automatiquement. Contrairement à notre approche où les propriétés sont changées automatiquement après leurs exportation, les propriétés dynamiques sont prises en compte et les contrats d'enregistrement peuvent être annulées lors de la suppression du composant.

Dans le chapitre suivant nous vous présentons les différentes étapes que nous avons suivi afin d'implanter cette proposition dans OpenCCM.

Chapitre 5

Réalisation

Après la présentation de notre approche concernant l'automatisation de l'enregistrement dans le chapitre précédant, nous présentons dans ce chapitre les différentes étapes qu'on a suivies afin d'implanter la proposition dans OpenCCM .
Tous au long de notre travail nous sommes passées par plusieurs étapes et nous avons fait certains choix d'implémentation afin de faciliter le travail, ces différentes étapes vont être détaillées dans ce qui suit.

5.1 Contrat d'enregistrement

Comme il a été mentionné dans la section 4.1 un descripteur d'enregistrement va être utilisé afin de décrire les propriétés à enregistrer et les services de recherche auprès desquels se fait l'enregistrement. Nous avions le choix entre l'utilisation d'un descripteur XML et un descripteur CIDL, mais notre choix s'est porté sur le descripteur CIDL pour les raisons mentionnées dans la section 4.1.2.
Une fois le choix du descripteur fait, il reste à décider si ce dernier sera interne ou bien externe au fichier CIDL, le choix d'utiliser un descripteur externe implique l'utilisation d'un fichier RDL (section 4.2.1.2) et l'exportation de ce dernier dans le fichier CIDL. Dans les deux cas plusieurs règles de production seront ajoutées dans la grammaire du CIDL.

Si le contrat d'enregistrement se trouve dans le CIDL les règles de production de la figure 5.1 sont ajoutées à sa grammaire.

Si par contre le contrat d'enregistrement est externe au CIDL, nous définissons un nouveau langage ressemblant au CIDL que nous appelons RDL pour *Registration Definition Langage* inspiré du TDL, qui sera importé dans le fichier CIDL. Ce choix implique la création de grammaire pour ce langage, ainsi que l'ajout de productions dans la grammaire du CIDL. La figure 5.2 illustre la grammaire du RDL alors que la figure 5.3 illustre les productions ajoutées au CIDL pour l'importation du RDL.

Dans notre implémentation, nous avons utilisé un descripteur interne au CIDL, ce qui a impliqué l'ajout de la grammaire de la figure 5.1, ce choix est justifié par le fait que la seconde solution aurait demandé non seulement l'implantation du langage RDL, mais aussi l'ajout de plusieurs scripts pour la compilation de celui ci et la génération du code associé à l'enregistrement, alors qu'en faisant notre choix on a utilisé les scripts d'OpenCCM, en leurs ajoutant quelques lignes de code. Ceci n'est pas la seule cause, certes l'utilisation d'une description RDL permet une modularité de l'application, puisque les productions du contrat d'enregistrement seront externe au CIDL, mais l'approche qu'on a choisi donne plus de facilité à l'utilisateur, il n'aura qu'a décrire l'enregistrement dans le fichier CIDL au lieu de créer un autre fichier, faire la description du contrat d'enregistrement puis revenir

(1) <cidl_definition> ::= <type_dcl> ";"
| <const_dcl> ";"
| <except_dcl> ";"
| <interface> ";"
| <cidl_module> ";"
| <storagehome> ";"
| <abstract_storagehome> ";"
| <storagetype> ";"
| <abstract_storagetype> ";"
| <value> ";"
| <type_id_dcl> ";"
| <type_prefix_dcl> ";"
| <event> ";"
| **<component_register> ";"**
| <home_dcl> ";"
| <composition> ";"
| <component> ";"

(2) **<component_register> ::= <contract_header> "{" <property_decl>*|<NS_body>"}"**

(3) **<property_decl> ::="property" [type_property] <param_type_spec> <identifier> ["link" <simple_declarator>] ";"**

(4) **<contract_header> ::= "contract" <identifier> "of" <identifier> "with" <service_name>**

(5) **<service_name> ::= <TRADER> | <NS>**

(6) **<type_property> ::= « dynamic » | « readOnly »**

(7) **<NS_body> ::= "property" [type_property] <param_type_spec> <identifier> ["link" <simple_declarator>**
| « namingExport] ";"

FIG. 5.1 – Règles de productions ajoutées au CIDL

(1) **<rdl_specification> ::= <import>* <rdl_dcl>+**

(2) **<rdl_dcl> ::=<rdl_header> "{"<rdl_body>*"}**

(3) **<rdl_header> ::= <CONTRACT> <identifier>**

(4) **<rdl_body> ::= <PROPERTY> [<mode_property>] <param_type_spec> <identifier> ";"**

(5) **<mode_property> ::= « dynamic » | « readOnly »**

(6) <param_type_spec> ::= <base_type_spec>
| <string_type>
| <wide_string_type>
| <scoped_name>

FIG. 5.2 – Grammaire du RDL

(1) <cidl_definition> ::= <type_dcl> ";"
| <const_dcl> ";"
| <except_dcl> ";"
| <interface> ";"
| <cidl_module> ";"
| <storagehome> ";"
| <abstract_storagehome> ";"
| <storagetype> ";"
| <abstract_storagetype> ";"
| <value> ";"
| <type_id_dcl> ";"
| <type_prefix_dcl> ";"
| <event> ";"
| **<component_register> ";"**
| <home_dcl> ";"
| <composition> ";"

(2) **<component_register_dcl> ::= <component_header> "{" <component_register_body> "}"**

(3)**<component_register_body>::=(<register_service>"{" <property_attribute_link() >"}")***

(4) **<register_service> ::= "register" <service_name> <identifier> ";"**

(5) **<service_name> ::= <TRADER> | <NS>**

(6) **<property_attribute_link> ::= "property"<simple_declarator> "link" <simple_declarator()> ";"**

FIG. 5.3 – Règles de production pour l'importation du RDL

au CIDL pour déclarer tous les contrats crées ainsi que le lien entre les propriétés et les attributs des composants concernés par l'enregistrement.
Une description interne au CIDL donne plus de facilité à l'utilisateur et moins de description et de fichier à écrire.

La création de la grammaire est la première étape vers l'automatisation de l'enregistrement, cette grammaire va être analysée avec un analyseur lexicale et syntaxique permettant de construire un arbre abstrait sur lequel des vérifications sémantiques doivent être réalisées. Finalement, cet arbre est parcouru pour une génération de code.
Dans la section suivante nous allons présenter la chaîne de compilation d'OpenCCM utilisée afin d'implanter notre approche.

5.2 Chaîne de compilation

Plusieurs types de descripteurs sont à l'entrée de la chaîne de compilation d'OpenCCM, cela inclut :

- les composants CORBA et les interfaces définit via l'OMG IDL,
- les états de persistance via l'OMG PSDL,
- la structure de l'implémentation des composants CORBA et le contrat d'enregistrement via l'OMG CIDL.

Le schéma de la figure 5.4 montre une vue globale de la chaîne de compilation d'OpenCCM.

Afin de réaliser la compilation et la génération de code, plusieurs éléments sont utilisés par la plate forme d'OpenCCM :

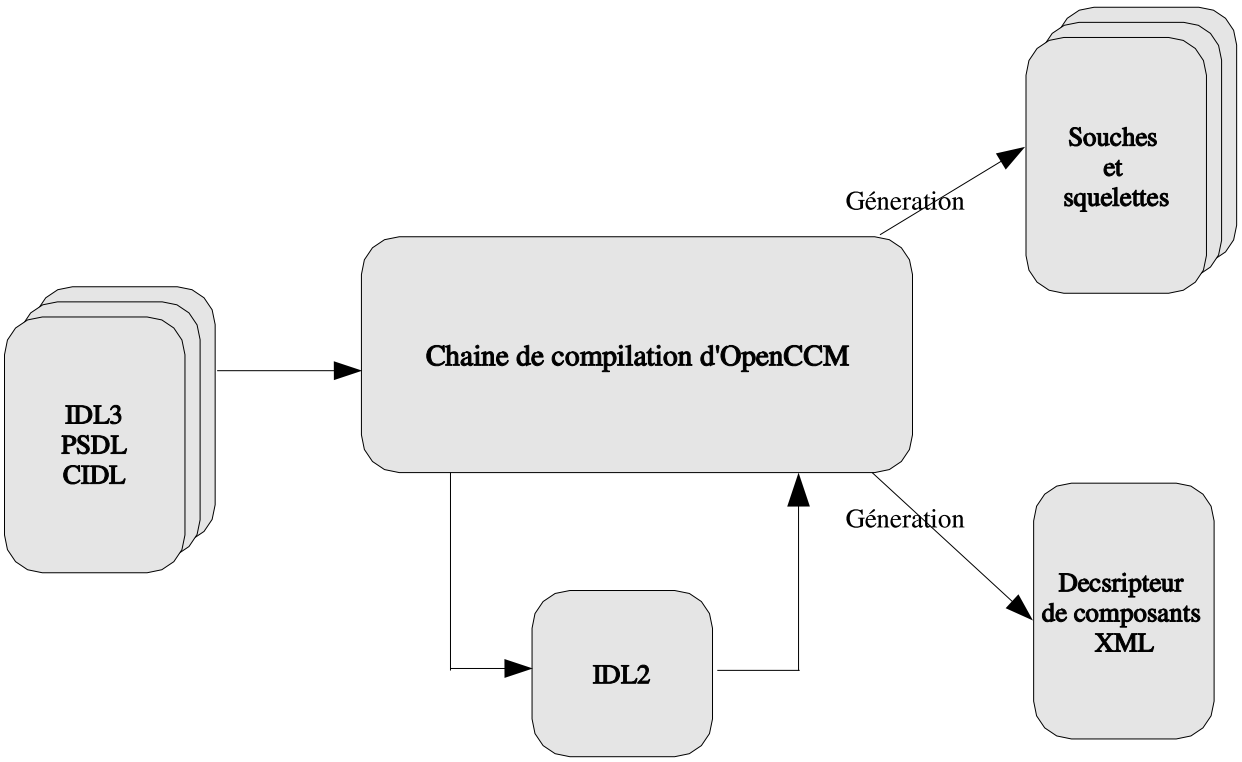

FIG. 5.4 – Vue globale de la chaîne de compilation d'OpenCCM

- un parseur,
- un arbre AST (*Abstract Syntax Tree*),
- un dépositaire d'interface (IR *Interface Repository*),
- des générateurs.

Le parseur utilise l'outil JavaCC pour assurer l'analyse lexicale et syntaxique et créer l'arbre AST [Fli03], cet arbre contient toutes les déclarations des fichiers IDL, CIDL et PSDL et permet de faire le lien entre le parseur java et le dépositaire d'interface.

La grammaire associée au contrat d'enregistrement a été insérée au niveau du fichier *Parser.jj* qui utilise l'outil JavaCC afin de générer des fichiers java permettant de faire l'analyse lexicale et syntaxique. Comme il a été dit dans le paragraphe précédant l'analyseur permet de remplir l'arbre AST en utilisant les fichiers de description IDL, CIDL et PSDL. L'arbre AST est utilisé par des générateurs qui permettent de faire une projection OMG IDL2 associée à des définitions du référentiel OMG IDL3, de générer des squelettes et des implantations java produisant un moule de base à compléter par les développeurs des composants.

5.3 Production de code

La production de code se fait à l'aide des générateurs, ils utilisent des patrons (*template*) pour générer du code java.

Dans le cadre de notre travail, nous avons effectué des changements au niveau de l'une des template afin de prendre en compte les contrat de courtage et générer le code nécessaire à l'enregistrement au niveau des différents services. Le choix de la template s'est fait en se basant sur le fait que le conteneur est l'entité qui se charge d'effectuer l'enregistrement, ces procédures ont été ajoutées à l'interface callback du conteneur « *SessionComponent* »par héritage de la nouvelle classe « *Re-*

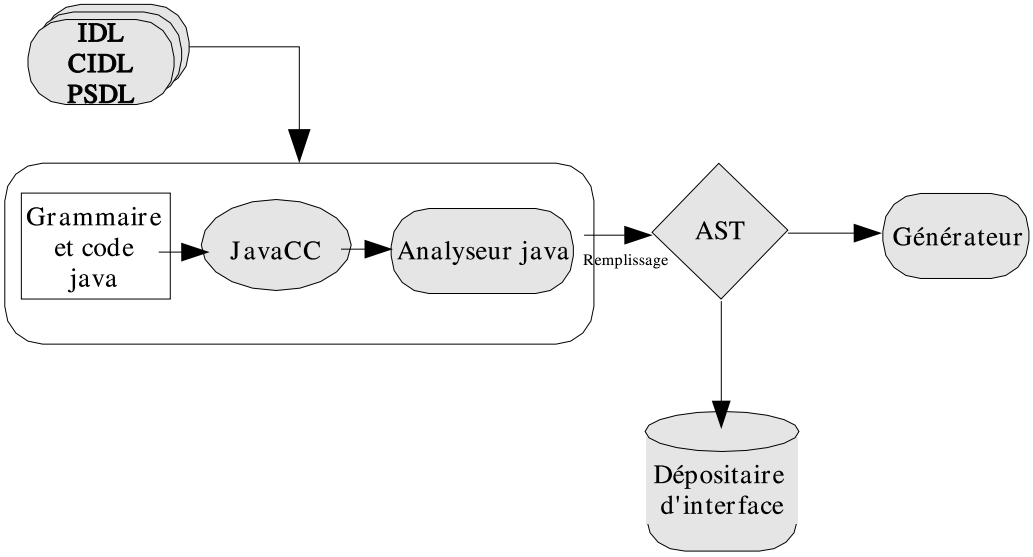

FIG. 5.5 – Chaîne d'analyse et de remplissage de l'arbre AST

gisterComponent »qui contient les différentes procédures d'enregistrement. Le code produit par la template est inséré au niveau du squelette du composant (figure 5.6) .

L'implantation du composant doit respecter quelques conventions de programmation en déclarant à son niveau les procédures d'enregistrement et en faisant appel à ceux générées par les templates au niveau du squelette.

5.4 Lancement des méthodes générées

Lors de l'installation de l'application, chaque composant est enregistré auprès des services désignés par le contrat d'enregistrement. Dans la proposition nous avons dit que l'appel des méthodes d'enregistrement se fait par le conteneur lors de la réception de *configuration_complete()*, en fait au départ nous avions trois événements susceptibles de permettre le lancement des méthodes d'enregistrement :

- **La création d'un composant :** cet événement aurait pu permettre l'appel de la méthode d'enregistrement du composant, mais il n'a pas été retenu car au moment de la création du composant il se peut que toutes les propriétés qui lui sont associées ne soient pas initialisées.
- **L'activation d'un composant :** lors de l'activation d'un composant le conteneur reçoit un événement, à sa réception les procédures d'enregistrement du composant auprès des différents services de recherche peuvent être lancés.
  Cette solution n'a pas été retenue elle aussi car certains composants sont activées à la réception d'une requête et désactivés à la fin de l'exécution de cette dernière, ce qui aurait engendré l'enregistrement de ces composants à chaque réception d'une requête .
  Ce problème peut être réglé en vérifiant à chaque activation du composant si ce dernier est déjà enregistré, si c'est le cas on ne fait rien, sinon on enregistre le composant auprès des différents services spécifiés par le contrat d'enregistrement.
  Mais cette solution semble être lourde, car à chaque activation d'un composant on vérifie si ce dernier a été enregistré ou pas.
  Notons que dans OpenCCM il existe une méthode permettant l'activation d'un composant

```
String oid;
protected org.omg.CORBA.ORB orb;
 private org.omg.CosTrading.Lookup lookup;
private org.omg.CosTrading.Register register;
public
   ImprimanteCCM()
   {
      _servant = null;
      _delegate = null;
      _locator = null;
      oid=null;
      org.omg.CORBA.Object o = orb.resolve_initial_references("TradingService");
      lookup = org.omg.CosTrading.LookupHelper.narrow(o);
      register = this.lookup.register_if();
   }
public void
   register_with_NS(String Name)
                  throws org.omg.Components.CCMException
   {
         org.objectweb.ccm.CORBA.TheNameService.getNamingContext().rebind(
                                          Name
                                          ,  (org.omg.Components.CCMObject) this.get_CCM_object());
   }
public void
   register_with_Trader(Property [] props)
                  throws org.omg.Components.CCMException
   {


                  org.omg.CosTrading.Property[] cosProps = new org.omg.CosTrading.Property[props.length];
                  for (int i=0 ; i<props.length ; i++)
                   {
                        if(props[i].name.equals('job'))
                              cosProps[i] = new org.omg.CosTrading.Property(props[i].name, evaljob());
                        else
                              cosProps[i] = new org.omg.CosTrading.Property(props[i].name, props[i].value);
                  }
                  try{oid = register.export((org.omg.Components.CCMObject) this.get_CCM_object(), 'offerTraderImprimante',
                                                cosProps);
                  } catch (org.omg.CosTrading.RegisterPackage.InvalidObjectRef ex) {}
}
public void
  property_modify( org.omg.CORBA.Any new_val, String name)
                              throws org.omg.Components.CCMException
         {

                  String[] del = new String[0];
                  org.omg.CosTrading.Property[] modif = new org.omg.CosTrading.Property[1];
                  modif[0] = new org.omg.CosTrading.Property(name  , new_val);
                  try{if(name.equals('marche '))
                              register.modify(oid, del, modif);
                        //else, lancer une exception pour dire que la propriété n'est pas modifiable
                  } catch (org.omg.CosTrading.IllegalOfferId ex) {}

         }
public void
         offer_remove()
                              throws  org.omg.Components.CCMException
         {

                  try{register.withdraw(oid);
                  } catch (org.omg.CosTrading.IllegalOfferId ex) {}
         }
public int evaljob()
{
         return _delegate.evaljob();
}
```

FIG. 5.6 – Code généré au niveau du squelette

appelée *ccm_activate()*, cette dernière n'a pas été implanté et n'est jamais appelée [Ope03] l'activation du composant est faite pour l'instant lors de la création de celui ci, mais dans les prochaines versions d'OpenCCM, *ccm_activate()* va être implantée et instanciée.

– ***configuration_complete* :** le lancement de *configuration_complete()* signifie que le composant est prêt a être utilisé, donc toutes les propriétés qui lui sont associées ont été initialisées. *configuration_complete()* est l'événement, intercepté par le conteneur, qu'on à choisi pour lancer la méthode d'enregistrement du composant.

Concernant la méthode de suppression du contrat d'enregistrement, elle est appelée à la réception de l'événement associé à la suppression du composant (*ccm_remove()*).

Quant le conteneur reçoit un événement concernant la modification d'un attribut, on modifie la propriété qui est associée à cet attribut et ceci en lançant la méthode, permettant de changer sa valeur, générée lors de la compilation du contrat d'enregistrement. Notons que pour le moment dans OpenCCM l'interception de la modification d'un attribut par le conteneur n'a pas été implanté. Ce que nous proposons pour l'instant c'est de faire l'appel de la procedure de modification de la propriété à l'interieur de la procedure permettant de changer l'attribut qui lui ait associé.

5.5 Travail en cours

A l'instant où nous écrivons ces lignes, nous avons implanté la grammaire du contrat d'enregistrement et ajouté la nouvelle classe permettant de définir l'interface *callback* d'enregistrement, nous avons généré les différentes méthodes d'enregistrement (auprès du service de nommage et le service de courtage), les méthodes de modification des propriétés et d'annulation du contrat d'enregistrement au niveau du squelette. Il nous reste a faire l'appel de ces différentes méthodes à la réception des événements cités dans la section 5.4.

Chapitre 6

Conclusion

Les middlewares en général permettent de libérer les programmeurs de l'écriture de code non directement lié au métier du composant. Dans le cadre des environnements de développement à base de composant, les middlewares permettent de le libérer encore plus en introduisant la notion de propriétés non fonctionnelles. Les propriétés non fonctionnelles sont prises en charge par le conteneur de composant fourni par le middleware.

Dans ce stage nous avons permis la description non fonctionnelle de l'enregistrement auprès des services de recherche sur propriétés.

Afin d'arriver à ce résultat nous avons étudié dans le deuxième chapitre la spécification CCM, ce qui nous a permis de comprendre le processus de conception et de déploiement d'une application à base de composants. Dans le troisième chapitre nous avons vu la manière dont cette spécification prévoit l'enregistrement des composants auprès des différents services de recherche sur propriété. Nous avons conclu à la fin de cette étude que malgré l'automatisation de l'enregistrement, le processus est figé et ne permet pas beaucoup de flexibilité, d'autant plus que les propriétés exportées sont statiques et ne peuvent plus être modifiées après l'enregistrement du composant.

Nous avons élaboré une solution dans le chapitre 4 permettant une description non fonctionnelle de l'enregistrement en utilisant un contrat d'enregistrement pour définir les différentes propriétés à exporter. La compilation de ce contrat permet la génération automatique de code d'enregistrement interne au composant ainsi que des méthodes de modification des propriétés exportées, de dé-enregistrement et de prise en compte des propriétés dynamiques. Nous avons permis aux middleware d'étendre les conteneurs pour qu'ils réalisent des enregistrements intelligents des composants. Notre proposition a permis d'enrichir l'interface callback du conteneur, puisque les fonctions d'enregistrement sont des méthodes callback que le composant offre au conteneur, nous avons permis l'extension du langage CIDL en introduisant le contrat de courtage et nous avons proposé un modèle d'enregistrement plus dynamique.

La réalisation de la proposition a été faite sous OpenCCM, qui est une implantation de la spécification CCM. La période du stage étant assez courte , nous n'avons implémenté qu'une partie de cette proposition.

Une perspective à court terme au travail effectué durant mon stage serait la modification des propriétés exportées, mais cela ne peu se faire que si l'interception des modifications des attributs par le conteneur est implanté dans OpenCCM ainsi que l'implémentation de ce qui n'a pas été implanté durant le stage. La proposition introduite dans ce rapport peut être étendue pour l'enregistrement des composants auprès d'autres services de recherche sur propriétés.

Bibliographie

[Aya02] D Ayad. Services de recherche intelligents et déploiement dynamique d'applications multi-composants. Rapport de stage du dea informatiques, Université D'EVRY VAL D'ESSONE (France), 2002.

[CCM02] CORBA Components Version 3.0 an Adopted Specification of the Object Management Group, June 2002.

[Fli03] A. Flissi. Inside OpenCCM. Technical report, The ObjectWeb Consortium, mars 2003.

[GGM97] J.M. Geib, C. Gransart, and P. Merle. *Corba : Des concepts à la pratique*. InterEditions, 1997.

[Gro02] OMG CCM Implementations Group. CORBA Component Model Tutorial. *OMG MEETING, Orlando, USA*, June 25th, 2002.

[Lea01] H. Egil S.Markku V Lea, K. Juha. Evaluation and exploitation of OMG CORBA and CORBA Component Model. Technical report, Department of computer science, University of HELINSKI FINLAND, 2001. Series of publicatins C.

[LJES01] K. Lea, H. Juha, Egil, and V S.Markku. CORBA Component Model - status and experiences. Technical report, University of HELINSKI, Department of Computer Science, December 31, 2001.

[LM01] S. Leblanc and P. MERLE. TORBA :vers un serveur de composants de courtage. 2001.

[LMMG01] S. Leblanc, R. Marvie, P. MERLE, and J.M. Geib. TORBA :contrats de courtage pour CORBA. *Calculateurs parallèles*, 2001.

[Mar01a] P. Marvie, R.and Merle. Vers un modèle de composants pour CESURE , le CORBA Component Model. Technical report, Department of computer science, 2001.

[Mar01b] R Marvie. OpenCCM : une plate-forme ouverte pour composants CORBA. Technical report, Laboratoire d'Informatique Fondamentale de Lille, 27 avril 2001.

[Mer02] P. Merle. Getting Started with OpenCCM, building a CORBA Component Application. Technical report, The ObjectWeb Consortium, November 2002.

[MM01] R. Marvie and P. Merle. CORBA Component Model : Discussion and Use with OpenCCM. Technical report, Laboratoire d'Informatique Fondamentale de Lille, France, 2001. Submitted for publication.

[MMG99] R. Marvie, P. Merle, and J. Geib. "Des objets aux composants CORBA, première étude et expérimentation avec CorbaScript". *In Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99), Paris, France,*, Décembre 1999.

[MMG00] R. Marvie, P. Merle, and J. Geib. Towards a dynamic corba component platform, 2000.

[MMGV01a] R. Marvie, P. Merle, J. Geib, and M. Vadet. OpenCCM : une plate-forme ouverte pour composants CORBA. Technical report, Laboratoire d'Informatique Fondamentale de Lille, 27 avril 2001.

[MMGV01b] R. Marvie, P Merle, J Geig, and M. Vadet. OpenCCM : une plate-forme pour composants CORBA. *CFSE'2, Seconde Conférence Française sur les Systèmes d'Exploitation, Paris, France*, Avril 2001.

[MP02] R. Marvie and M-C. Pellegrini. Modèle de composants, un état de l'art. *in Coopération dans les systèmes à objets special issue of L'objet, Ed Hermès, Paris, France*, Septembre 2002.

[ODP95] ODP.Open Distributed Processing Reference Model. parts 1-4.Open Distributed Processing .ISO, 10746-1..4,1995.

[OMG00] Trading Object Service Specification. OMG documents formal/00-06-27, May 2000.

[Ope03] http ://www.objectweb.org/openccm/, February 2003.

[Sab03] N. Sabri. *Une architecture à base de composants CORBA pour des Services Personnalisées*. PhD thesis, Thèse de Doctorat en Informatique de l'Université D'EVRY VAL D'ESSONE (France), juin 2003.

[VM99] V.Matena and M.Hapner. Enterprise Java Beans Specification v1.1specification- Final Release. Sun Microsystems, May 1999.

[WSO] N. Wang, D. Schmidth, and C. O'Ryan. Overview of CORBA Component Model.