RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

OpenCCM : une plate-forme ouverte pour composants CORBA

Raphaël Marvie, Philippe Merle, Jean-Marc Geib, Mathieu Vadet

Laboratoire d'Informatique Fondamentale de Lille
UPRESA CNRS 8022
Bât. M3 – UFR d'I.E.E.A.
F-59655 Villeneuve d'Ascq (France)
{marvie,merle,geib,vadet}@lifl.fr

Résumé
De nos jours, l'ingénierie des larges applications critiques, *e.g.* dans le domaine des télécommunications ou du contrôle aérien, s'oriente vers la mise en œuvre d'intergiciels pour composants logiciels répartis tels que le *Distributed Component Object Model* (Dcom) de Microsoft, les *Entreprise Java Beans* (Ejb) de Sun Microsystems ou le récent *CORBA Component Model* (Ccm) de l'Object Management Group (Omg). Ce dernier semble offrir le plus fort potentiel pour caractériser, produire, déployer et exécuter des applications distribuées hétérogènes. Cependant, cette norme n'est pas à ce jour finalisée et aucune implantation n'est disponible. Dans ce contexte, cet article présente l'état d'avancement de nos travaux de conception et de réalisation de la plate-forme OpenCCM : une implantation libre, ouverte et flexible de la norme Ccm.
Dans l'état actuel, OpenCCM offre une chaîne ouverte de production de 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. L'ensemble est construit au dessus des intergiciels Corba actuellement disponibles et offre ainsi à la communauté système une plate-forme libre pour l'expérimentation de la future génération d'intergiciels pour composants répartis.

Mots-clés : Composants répartis, Corba Component Model, OpenCCM, compilation Omg Idl3

1. Introduction

Actuellement, la plupart des applications distribuées complexes, *e.g.* dans le domaine des télécommunications [6], sont construites au dessus d'intergiciels (*i.e. middleware*) à base d'objets distribués, tels que la *Common Object Request Broker Architecture* (Corba) [17] de l'Object Management Group (Omg), le *Distributed Component Object Model* (Dcom) [5] de Microsoft ou encore *Java Remote Method Invocation* (Rmi) [14, 24] de Sun Microsystems. Cela provient du fait que ceux-ci ont démontré leur capacité à rationnaliser et fiabiliser globalement le processus de caractérisation, de production, de déploiement, d'exécution et d'administration des applications visées. Grâce à cette approche, une application est vue comme l'interconnexion d'un ensemble d'objets distribués. Similairement, les services systèmes tels le nommage, le courtage, la notification, la sécurité, les transactions et la persistance sont vus comme des objets distribués.
Cependant, ces intergiciels comportent certains défauts limitant la possibilité de supporter correctement des applications réparties configurables et à large échelle. Tout d'abord, les connexions entre objets sont enfouies au cœur de ceux-ci et ne peuvent pas être facilement configurées par des architectes externes. Ainsi, il est d'autant plus difficile d'avoir une vision précise de l'architecture d'une application répartie [1]. Ensuite, l'utilisation des services systèmes, *i.e.* les aspects non-fonctionnels d'une application, doit être implantée « en dur » dans le corps des objets applicatifs et ainsi mélangée au code fonctionnel. Ceci rend les développements plus complexes à réaliser pour les experts des domaines applicatifs : ils doivent

en parallèle maîtriser la logique métier de leurs applications et la complexité technique de mise en œuvre des services systèmes. Enfin, la phase de déploiement des objets n'est pas prise en compte. Il n'y a pas de paradigme standard pour installer des objets sur leur site d'exécution de manière automatisée : il n'existe que des solutions et outils *ad hoc*.
Pour corriger ces défauts, les intergiciels ont progressivement évolués d'une approche orientée objet vers une approche orientée composant comme les *Entreprise Java Beans* (EJB) [12] de Sun Microsystems ou le plus récent *CORBA Component Model* (CCM) [15] de l'OMG. 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 d'utilisation 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 sans modifier le code. Par exemple, les spécifications des EJB et du CCM définissent des vocabulaires XML (*eXtended Markup Language*) pour décrire les besoins des instances de composants en termes de sécurité, de transactions et de persistance. Alors, les packages de composants (*i.e.* implantations et descripteurs) sont déployés dans les serveurs de containers. Ces containers sont configurés en utilisant les descripteurs pour fournir correctement les services systèmes aux instances de composants. Le défaut architectural précédemment cité est partiellement résolu : les composants EJB peuvent être interconnectés en utilisant les canevas de conception (*i.e. design patterns* [3]) définis dans les *Java Beans* [2] et le CCM introduit le concept de *ports* comme discuté en § 2.4. Grâce aux intergiciels à composants, les applications sont désormais vues comme une composition de composants packagés, déployables et répartis.

Dans cette évolution, le modèle de composants CORBA semble offrir le plus fort potentiel pour couvrir globalement le processus de caractérisation, de production, de déploiement et d'exécution de composants répartis et hétérogènes. Cependant, cette norme n'est pas à ce jour finalisée et aucune implantation n'est disponible. Ainsi, la communauté système a l'opportunité de contribuer à la conception et au prototypage de cette prochaine génération d'intergiciels pour composants répartis. Dans cette optique, cet article présente l'état d'avancement de nos travaux de conception et de réalisation de la plate-forme OpenCCM : la première implantation libre, ouverte et flexible de la norme CCM.
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 (comme discuté dans [8]) par l'intermédiaire de scripts OMG IDLscript [16] décrivant le téléchargement des archives de composants, l'instanciation, la configuration et l'interconnexion des composants s'exécutant au sein de serveurs génériques. L'ensemble est construit au dessus des intergiciels CORBA actuellement disponibles et offre ainsi à la communauté système une plate-forme libre pour l'expérimentation de la future génération d'intergiciels pour composants répartis à contraintes critiques [22, 21].

L'organisation de cet article est la suivante. La section 2 présente les concepts clés du modèle de composants CORBA et leur expression dans le nouveau langage OMG IDL3. La section 3 discute des choix de conception et de réalisation de notre chaîne de compilation du langage OMG IDL3. La section 4 illustre l'implantation des composants dans le cadre de notre plate-forme OpenCCM. Finalement, les perspectives du travail OpenCCM sont énoncées dans la section 5.

2. Le modèle de composants CORBA

2.1. Une vue d'ensemble

Le modèle de composants CORBA représente le premier modèle industriel pour des composants « métiers » répartis et hétérogènes, c'est-à-dire indépendants des langages de programmation (versus EJB), des systèmes d'exploitation et des fournisseurs d'intergiciels (versus DCOM). Pour cela, il définit un ensemble d'artefacts pour un processus complet de caractérisation, de production, de diffusion, de déploiement et d'exécution de composants.
En terme de caractérisation, le CCM étend le langage OMG IDL pour permettre d'exprimer les ports d'interconnexion des composants et les maisons de composants (voir § 2.4). En terme de production, le

CCM définit le *Component Implementation Definition Language* (CIDL) pour capturer et automatiser la génération de la partie technique de l'implantation des composants. En terme de diffusion, le CCM propose l'utilisation d'archives au format « Zip » caractérisées par des descripteurs exprimés en *Open Software Description* (OSD, un vocabulaire XML). En terme de déploiement, le CCM définit un processus d'installation des archives sur les sites d'exécution, d'instanciation, de configuration et d'interconnexion des maisons et des composants répartis. En terme d'exécution, le CCM fournit des serveurs de containers, fortement inspirés de ceux des EJB, pour prendre en charge implicitement les services de nommage, de courtage, de sécurité, de transactions, de persistance et de notification.
La description complète de tous les artefacts du modèle de composants CORBA dépasse largement le cadre de cet article. Pour plus de détails, le lecteur peut se reporter aux articles [8, 9, 23]. La suite de cette section approfondit plus spécifiquement les artefacts de caractérisation, c'est-à-dire le langage OMG IDL3, la notion de composants, de ports et de maisons à travers un exemple fil rouge.

2.2. Le modèle abstrait

Le modèle abstrait du CCM permet aux concepteurs de caractériser la vision externe offerte par leurs composants, c'est-à-dire les moyens pour créer, rechercher, détruire, configurer, interconnecter et interagir avec les composants indépendamment de toute considération liée à leurs implantations. Cette vision externe s'exprime par l'intermédiaire du nouveau langage OMG IDL3, une extension de l'ancien OMG IDL, permettant de définir des types de composants et de maisons (voir les sous-sections suivantes).
Seuls les concepteurs utilisent ce nouveau langage car il est ensuite projeté en OMG IDL2 pour être exploité par les développeurs d'implantations de composants. Cette approche a été choisie afin de ne pas remettre en cause le savoir-faire dûrement acquis par les développeurs CORBA et de ne pas jeter tous les outils de production déjà existants pour CORBA. Cette projection définit un ensemble de règles de traduction pour générer des équivalents OMG IDL2 des constructions OMG IDL3. Ces équivalences reposent sur une base commune de définitions OMG IDL2, *i.e.* le module `Components` contenant les types de données et les interfaces de base communs à tous les composants et maisons offrant par exemple des opérations génériques (*e.g.* créer, détruire, etc) et des opérations d'introspection. Finalement, le CCM introduit un nouveau référentiel pour stocker des spécifications OMG IDL3 et pouvoir les découvrir et les exploiter dynamiquement durant l'exécution [4].

2.3. L'exemple fil rouge

La suite de l'article s'appuie sur un exemple fil rouge illustré par la Figure 1 et inclus comme démonstration dans la plate-forme OpenCCM. Cet exemple est une application composée d'un ensemble de composants `Client` interagissant de manière synchrone avec un composant `Server`. Ce dernier interagit lui-même de manière asynchrone avec des composants `Consumer`. Les interactions entre les composants sont visualisées par des interconnexions entre les ports des composants (discutées en § 2.4). Ces différents composants sont créés et gérés respectivement par les maisons de composants `ClientHome`, `ServerHome` et `ConsumerHome`.
La spécification OMG IDL3 associée à cet exemple est entièrement listée dans la Figure 2 et les nouvelles constructions OMG IDL3 sont discutées dans la section suivante. Même si cet exemple est relativement simpliste, il est tout de même très représentatif de nombreuses applications distribuées couramment mises en place.

2.4. Le langage OMG IDL3

Le langage OMG IDL3 se présente comme une extension de l'ancien OMG IDL. Ainsi, il en conserve tout le pouvoir d'expression. Une extension mineure est le remplacement des directives d'inclusion de fichiers OMG IDL par l'introduction d'une construction d'importation de modules (*i.e.* le mot-clé `import`). Cette construction permet de mieux exprimer la réutilisation de spécifications OMG IDL. Cela a une forte incidence sur la compilation du langage OMG IDL3 : la nécessité de disposer d'un référentiel OMG IDL3 (comme discuté en § 3). L'exemple OMG IDL3 de la Figure 2 importe le module de base du CCM (`Components`) pour avoir accès à son contenu. Les extensions majeures sont l'ajout de constructions pour caractériser les composants, leurs ports et leurs maisons.
La construction `component` permet de définir des types de composants. Un type de composants peut hériter d'un unique autre type, *e.g.* `Client`, `Server` et `Consumer` héritent de `NamedComponent`. Un composant peut définir un ensemble de propriétés via des attributs (*e.g.* `name` dans `NamedComponent`) configurés

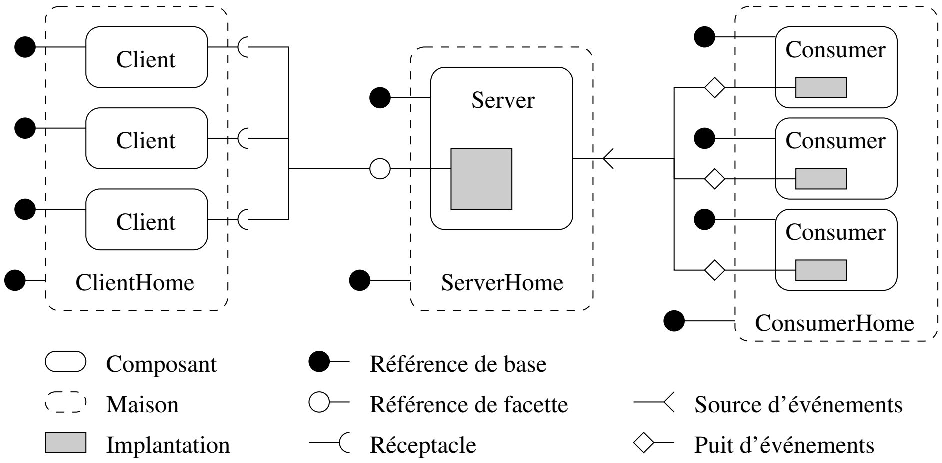

FIG. 1 – *La vision abstraite d'une application à base de composants* CORBA

```
import Components ;
module exemple {
  // Typage des ports.
  interface Service {
    void display (in string text) ;
  } ;
  valuetype Event : Components::EventBase {
    public string text ;
  } ;

  // Typage des composants.
  component NamedComponent {
    attribute string name ;
  } ;
  component Server : NamedComponent {
    provides  Service the_service ;
    publishes Event to_consumers ;
  } ;
  component Client : NamedComponent {
    uses Service the_service ;
  } ;
  component Consumer : NamedComponent {
    consumes Event from_servers ;
  } ;

  // Typage des maisons.
  valuetype NamedPrimaryKey : Components::PrimaryKeyBase {
    public long name ;
  } ;
  home ServerHome manages Server primaryKey NamePrimaryKey {
    factory create_server (in string name) ;
    finder find_server (in string name) ;
  } ;
  home ClientHome manages Client { } ;
  home ConsumerHome manages Consumer { } ;
} ;
```

FIG. 2 – *La spécification* OMG IDL*3 de l'exemple fil rouge*

lors de la phase de déploiement.
Précédemment, un objet CORBA2 était défini par une unique interface listant l'ensemble de ses opérations. Celle-ci ne définit que les services offerts par un objet, elle n'offre pas de moyen d'exprimer un regroupement logique des services (*i.e.* l'héritage multiple OMG IDL est souvent inadapté voire limitatif) et elle ne permet pas de spécifier les services utilisés par l'objet (et fournis par d'autres objets). Il est donc difficile de gérer la connectique entre des objets car les liaisons sont enfouies au sein de leurs implantations. Les composants CORBA apportent ici une solution à travers la notion de *ports*, favorisant l'assemblage des composants au moment du déploiement. Quatre types de *ports* sont définis :

- Une *facette* décrit une interface fournie par un composant et qui est utilisée par des clients en mode synchrone (*e.g.* le port `the_service` de `Server`). Une facette est déclarée à l'aide du mot-clé `provides`. Cette possibilité pour un composant d'offrir plusieurs interfaces est inspirée du modèle COM de Microsoft [20] et permet de découper en points de vues logiques l'interface fonctionnelle d'un composant.
- Un *réceptacle* définit une interface utilisée par un composant en mode synchrone (*e.g.* le port `the_service` de `Client`). Un réceptacle est déclaré à l'aide du mot-clé `uses`. Lors du déploiement, ce point de connexion est lié à une facette adéquate.
- Une *source d'événements* décrit un type d'événements produits de manière asynchrone par un composant (*e.g.* le port `to_consumers` de `Server`). Une source d'événements est déclarée à l'aide des mots-clés `emits`, si la notification est de 1 vers 1, ou `publishes`, si 1 vers n. A l'exécution, un canal du service de notification est alloué à chaque source et diffuse ses événements.
- Un *puit d'événements* décrit un type d'événements consommés de manière asynchrone par un composant (*e.g.* le port `from_servers` de `Consumer`). Un puit d'événements est déclaré à l'aide du mot-clé `consumes`. Lors du déploiement, un puit est connecté aux sources adéquates par la mise en œuvre du canevas de conception *Observer* [3]. Ainsi à l'exécution, un puit devient un consommateur des différents canaux de notification associés aux sources observées.

Le modèle de composants CORBA impose que tous les ports d'un composant soient déclarés statiquement à la définition du type de composants. Ainsi, il est impossible d'ajouter ou retirer des ports dynamiquement durant l'exécution !

Finalement, les maisons de composants, inspirées des EJB, sont des gestionnaires pour la création et la recherche de composants. A la différence des EJB, celles-ci sont explicitement définies par l'intermédiaire de la construction `home`. Une maison est associée à un unique type de composants, cependant un type de composants peut être géré par différentes maisons. Une maison peut définir des opérations spécifiques de création et de recherche, via les mots-clés `factory` et `finder`. Au sein d'une maison, les composants peuvent être indexés par une clé primaire, via le mot-clé `primaryKey`. Dans l'exemple, la maison `ServerHome` gère des composants `Server` indexés par la clé `NamePrimaryKey` et offre l'opération de création `create_server` et l'opération de recherche `find_server`.

3. La chaîne de compilation de la plate-forme OpenCCM

Après cette présentation du modèle de composants CORBA, du nouveau langage OMG IDL3 et de son illustration sur un exemple, cette section discute de nos choix de conception et de réalisation de la chaîne de compilation OMG IDL3 de notre plate-forme OpenCCM.

3.1. Une vue d'ensemble

Généralement, une chaîne de compilation pour un langage donné est constituée d'un analyseur lexical et syntaxique construisant un arbre syntaxique abstrait sur lequel des vérifications sémantiques doivent être réalisées. Finalement, cet arbre est parcouru, soit pour la génération du code cible, soit pour une interprétation / exécution immédiate.
Dans notre contexte, l'OMG IDL3 est décrit par environ 250 règles syntaxiques. L'arbre abstrait doit modéliser plus d'une vingtaine de constructions syntaxiques, principalement `module`, `interface`, `valuetype`, `component`, `home` et les déclarations de types, d'opérations, d'attributs et de ports. La nouvelle construction `import` nécessite de pouvoir accéder aux arbres abstraits construits lors de compilations précédentes. Les vérifications sémantiques sont très nombreuses et souvent récursives. Finalement, la

génération doit produire la projection équivalente en OMG IDL2 et des squelettes pour plusieurs langages de programmation tels que Java, C++, OMG IDLscript, etc.

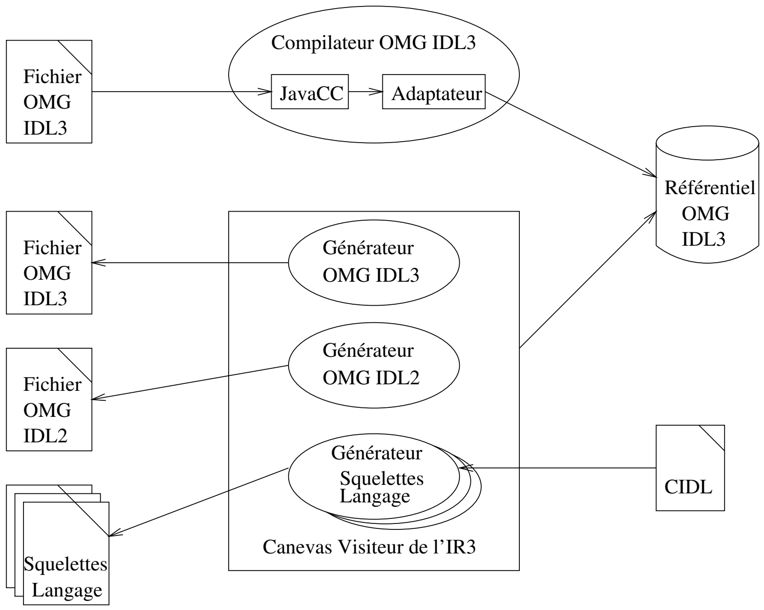

FIG. 3 – *Les composants de la chaîne de compilation* OMG IDL*3 de la plate-fome OpenCCM*

Afin de gérer cette complexité, la chaîne de compilation OMG IDL3 de la plate-forme OpenCCM est décomposée en un ensemble de composants logiciels illustré dans la Figure 3. Cette chaîne 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 IDL et de squelettes pour les langages de programmation cibles. Cette architecture permet de séparer les problèmes, facilitant ainsi la maintenance corrective : le compilateur ne s'occupe que de l'analyse lexicale et syntaxique, le référentiel stocke l'arbre abstrait et effectue les contrôles sémantiques, chaque générateur contient uniquement les règles de production associées à son langage cible. Toutefois, les générateurs partagent le canevas de conception Visiteur [3] appliqué au parcours du référentiel OMG IDL3 et un canevas de production de fichiers de sortie. Cela favorise ainsi la création de nouveaux générateurs sans remettre en cause l'architecture complète de la chaîne.
Les différents composants de cette chaîne sont réalisés dans le langage Java afin d'être fortement portables sur différents systèmes d'exploitation (actuellement Linux, Solaris, MS-Windows). De plus, ces composants dialoguent à travers des interfaces OMG IDL2 permettant ainsi de les assembler selon différentes configurations : il est possible de mettre tous les composants dans un seul processus ou de distribuer ceux-ci dans différents processus dialoguant à travers un bus CORBA. Pour l'instant, cette chaîne fonctionne au dessus des produits ORBacus 4 for Java [18], OpenORB [19] et VisiBroker 4 for Java [7]. Ainsi, le portage sur d'autres ORB écrits en Java ne devrait poser aucun problème insurmontable.

3.2. Le compilateur OMG IDL3

Le rôle du compilateur OMG IDL3 est d'assurer l'analyse lexicale / syntaxique du langage OMG IDL3 et d'alimenter le référentiel OMG IDL3. Pour cela, ce composant est décomposé en deux parties : le *parser* et la couche *adaptateur*.
Le *parser* OMG IDL3 a été implanté à l'aide de l'outil JavaCC [13]. L'intérêt de cet outil est double. Il permet d'exprimer simplement les 250 règles lexicales et syntaxiques du langage OMG IDL3. En sortie, il génère un parser orienté objet facilement réutilisable depuis un programme Java. Afin de ne pas polluer et complexifier les règles de production du parser, celles-ci invoquent les interfaces Java de la couche *adaptateur*.

La couche *adaptateur* propose des interfaces conçues spécifiquement pour être utilisées depuis les règles de production du parser. L'implantation de ces interfaces a pour but de traduire les ordres émis par le parser en ordres pour alimenter le référentiel OMG IDL3. Cela permet d'isoler le parser des détails techniques des interfaces du référentiel OMG IDL3. De plus, comme ces dernières ne sont pas encore finalisées dans la norme CCM, seule l'implantation de la couche *adaptateur* nécessitera d'être revue pour suivre les inévitables futures évolutions.

3.3. Le référentiel OMG IDL3

Le référentiel OMG IDL3 a un rôle pivot dans la chaîne de compilation OMG IDL3 de notre plate-forme OpenCCM. Premièrement, il stocke les spécifications OMG IDL3 sous la forme d'un graphe d'objets CORBA, *i.e.* la matérialisation de l'arbre syntaxique abstrait. Deuxièmement, il assure le contrôle des règles sémantiques lorsque le compilateur OMG IDL3 l'alimente. Troisièmement, le compilateur OMG IDL3 le consulte afin de supporter la nouvelle construction `import`. Finalement, les générateurs le visitent afin de produire leur code cible.

Le référentiel est matérialisé par un graphe d'objets CORBA. Ces objets sont décrits par des interfaces OMG IDL2 telles que `ModuleDef`, `ValueDef`, `InterfaceDef`, `ComponentDef`, `HomeDef`, etc. Chacune de ces interfaces modélise une construction OMG IDL3, *i.e.* les instances contiennent les méta-données OMG IDL3. La norme CCM contient déjà une spécification de ces interfaces. Celles-ci étendent celles du référentiel OMG IDL2 déjà présentes dans CORBA 2.4.1 [17]. Ces extensions ajoutent de nouvelles opérations et interfaces pour modéliser les nouvelles constructions OMG IDL3.

Cependant, ces interfaces ne sont pas finalisées (*i.e.* défauts de conception, oublis, surplus) et sont encore sujet à discussion dans le groupe de travail associé à la norme CCM. Ainsi, nous contribuons techniquement auprès de l'OMG en reportant notre expérience, *i.e.* les défauts de conception et les manques de ces nouvelles interfaces. De plus, nous avons récemment repris, en collaboration avec Alcatel, la rédaction de cette partie de la norme CCM. Dans l'avenir, notre référentiel OMG IDL3 sera toujours « à la norme » car à la base de son élaboration.

3.4. Les générateurs

Comme le modèle de composants CORBA vise différents langages de programmation cibles, l'architecture de notre chaîne de compilation OMG IDL3 a été conçue pour être extensible, *i.e.* pouvoir ajouter de nouveaux générateurs. Pour cela, chaque générateur contient uniquement les règles de production associées à son langage cible.

Toutefois, les générateurs partagent deux canevas de conception. Le premier est le canevas Visiteur [3] appliqué au parcours du référentiel OMG IDL3. Ce canevas offre un ensemble de stratégies de navigation dans le graphe des objets du référentiel. Le second est un canevas de production de fichiers de sortie. Ici, l'idée est de permettre d'exprimer les règles de génération avec un certain niveau d'abstraction, *i.e.* ne pas directement imprimer des chaînes de caractères dans des fichiers texte.

Actuellement, ces canevas ont été appliqués pour réaliser un générateur d'OMG IDL3 visualisant le contenu du référentiel OMG IDL3, un générateur pour la projection OMG IDL2 associée à des définitions OMG IDL3, un générateur de squelettes Java étendus produisant une implantation Java des interfaces OMG IDL2 générées, un générateur d'implantation Java produisant un moule de base à compléter par les développeurs des composants. Dans un souci de simplification, le générateur actuel de squelettes étendus ne s'appuie pas sur le langage CIDL. Cette intégration sera faite dans les prochaines versions de la plate-forme OpenCCM.

4. La production de composants Java

Cette section discute du processus de production de composants Java dans le contexte de la plate-forme OpenCCM. Cette discussion est illustrée sur le composant `Server` et sa maison associée afin de présenter l'OMG IDL2 généré, les choix de conception pour les squelettes générés, ainsi que quelques règles de programmation des composants.

4.1. La projection en OMG IDL2

Dans le cadre du CCM, toute spécification OMG IDL3 est projetée / traduite en une spécification OMG IDL2 équivalente. La Figure 4 illustre un extrait de l'OMG IDL2 associé au composant `Server` et sa maison `ServerHome`. Le lecteur peut constater que la projection consiste principalement à transformer les constructions OMG IDL3 en constructions OMG IDL2.

```
#include "Components.idl"
module exemple {
    interface Service {
        void display(in string text);
    };
    valuetype Event : Components::EventBase {
        public string text;
    };
    interface EventConsumer : Components::EventConsumerBase {
        void push(in Event event);
    };
    interface NamedComponent : Components::CCMObject {
        attribute string name;
    };
    interface Server : NamedComponent {
        Service provide_the_service();
        Components::Cookie subscribe_to_consumers(in EventConsumer consumer)
                           raises(Components::ExceededConnectionLimit);
        EventConsumer unsubscribe_to_consumers(in Components::Cookie ck)
                      raises(Components::InvalidConnection);
    };
    valuetype NamePrimaryKey : Components::PrimaryKeyBase {
        public long name;
    };
    interface ServerHomeExplicit : Components::CCMHome {
        Server create_server(in string name);
        Server find_server(in string name);
    };
    interface ServerHomeImplicit {
        Server create(in NamePrimaryKey key)
               raises(Components::DuplicateKeyValue, Components::InvalidKey);
        Server find_by_primary_key(in NamePrimaryKey key)
               raises(Components::UnknownKeyValue, Components::InvalidKey);
        void remove(in NamePrimaryKey key)
               raises(Components::UnknownKeyValue, Components::InvalidKey);
        NamePrimaryKey get_primary_key(in Server comp);
    };
    interface ServerHome : ServerHomeImplicit, ServerHomeExplicit {};
};
```

FIG. 4 – *La projection* OMG IDL*2 pour le composant* `Server` *et sa maison*

La construction `import` est transformée en `include`. Pour chaque type d'événements, une interface de consommation est générée, *e.g.* `EventConsumer` caractérise l'interface du puit d'événements `Event`. Les définitions de composants sont traduites par des interfaces héritant de l'interface `Components::CCMObject`. La facette `the_service` est traduite par l'opération `provide_the_service` permettant d'obtenir la référence d'un objet CORBA implantant la facette. La source d'événements `to_consumers` est traduite par les deux opérations `subscribe_to_consumers` et `unsubscribe_to_consumers` pour respectivement abonner ou désabonner un puit d'événements `EventConsumer`. Chaque définition de maison de composants est traduite par une interface héritant de deux autres interfaces générées. L'interface explicite contient les opérations explicitement définies pour la maison et hérite de l'interface `Components::CCMHome`. Les opérations de fabrique et de recherche retournent le type de composants géré par la maison. L'interface

implicite fournit des opérations de création, de recherche et de destruction de composants indexés par la clé primaire `NamePrimaryKey`.
Dans le cadre de la plate-forme OpenCCM, l'intérêt de la projection OMG IDL3 vers OMG IDL2 est double. Premièrement, les développeurs d'objets CORBA connaissent déjà les règles de projection OMG IDL2 vers leur langage de programmation. Ainsi, ils pourront plus rapidement produire des composants CORBA. Deuxièmement, les générateurs de souches et de squelettes GIOP/IIOP sont déjà fournis par les produits CORBA. Ainsi, notre plate-forme OpenCCM est naturellement portable sur différents ORB.

4.2. La génération des squelettes étendus

Afin de réaliser les composants et les maisons, la plate-forme OpenCCM permet de générer des squelettes Java étendus. Ceux-ci sont dits étendus car leur rôle est d'implanter toute la mécanique liée au modèle de composants CORBA, *e.g.* la prise en charge des ports et de leur introspection. L'implantation du décodage des requêtes GIOP/IIOP est héritée des classes de squelettes POA générées par le produit CORBA utilisé. La Figure 5 illustre ce choix de conception sur l'architecture des classes squelettes associées au composant `Server`.

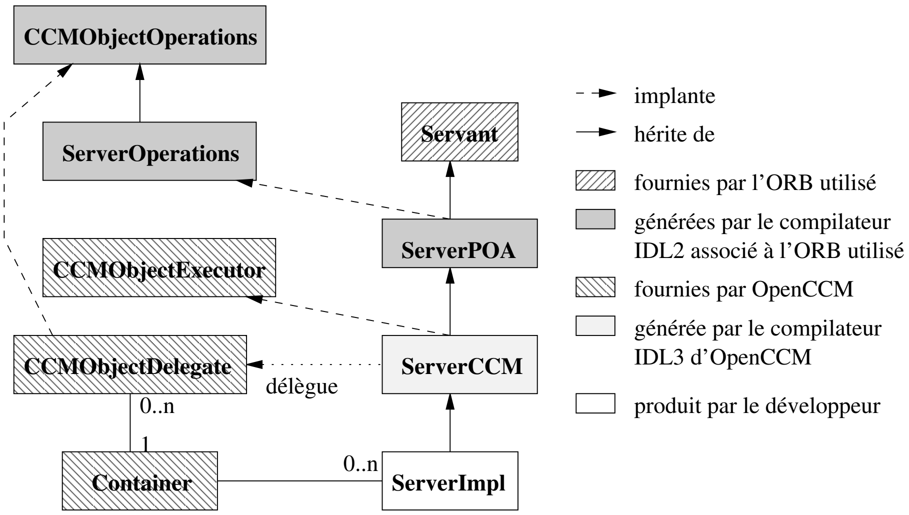

FIG. 5 – *L'architecture des classes du squelette étendu pour le composant `Server`*

Cette architecture montre que la classe `ServerImpl` d'implantation du composant doit hériter du squelette étendu `ServerCCM` généré par la plate-forme OpenCCM. Ce squelette hérite du squelette `ServerPOA` à générer par un compilateur OMG IDL2. Ce squelette hérite lui-même de la classe `Servant` fournie par l'ORB et des interfaces Java contenant les opérations des interfaces OMG IDL2 associées, *i.e.* `CCMObject-Operations` et `ServerOperations`.
Le but de la classe `ServerCCM` est alors d'implanter les opérations OMG IDL2 du composant afin que le développeur ait le minimum de travail à faire (voir § 4.3). Pour cela, cette classe délègue l'implantation des opérations de l'interface OMG IDL2 `CCMObject` à une instance de la classe `CCMObjectDelegate` fournie par le support d'exécution OpenCCM. Ici, le choix s'est naturellement porté sur la délégation car le langage Java n'autorise pas l'héritage multiple. En C++, le choix sera plutôt d'utiliser l'héritage multiple. De plus, la classe `ServerCCM` implante aussi par délégation les opérations de l'interface OMG IDL2 `Server`, *i.e.* `provide_the_service`, `subscribe_to_consumers`, `unsubscribe_to_consumers`. Finalement, le squelette étendu implante l'interface `CCMObjectExecutor` permettant au container OpenCCM de manipuler tout squelette étendu indépendamment de son type.
Chaque instance de composants (*e.g.* `ServerImpl` et `CCMObjectDelegate`) réside dans un container. La classe `Container` offre des services aux composants pour masquer la mise en œuvre des services

systèmes, *e.g.* POA, sécurité, transactions, persistance, notification, etc. Actuellement, les containers OpenCCM sont réduits au strict minimum, *i.e.* simple utilisation du POA. Une des orientations futures de nos travaux portera sur la conception et la réalisation de containers adaptatifs permettant de supporter de manière flexible la gestion des services systèmes (voir § 5).

Cette architecture s'applique aussi à la maison de composants `ServerHome`. Dans ce cas, le squelette étendu n'implante que les opérations de l'interface implicite. L'interface explicite doit, quant à elle, être implantée par le développeur.

4.3. L'implantation des composants

L'implantation des composants a pour objectif de donner une matérialisation au composant, à ses facettes et ses puits d'événements. La Figure 6 présente le code minimal d'implantation du composant `Server`. La classe `ServerImpl` hérite du squelette généré par la plate-forme OpenCCM. Pour chaque facette et puit d'événements, l'implantation doit fournir une méthode, utilisée par le squelette pour déterminer quel est l'objet d'implantation associé. Ainsi, tous les ports d'entrées d'un composant ne sont pas forcément implantés par une seule et unique classe. Ici, la méthode `_get_facet_the_service` signale au squelette que la facette est directement implantée par la classe du composant, *i.e.* héritage de l'interface `ServiceOperations` et implantation de l'opération `display`. Finalement, le squelette fournit la méthode `push_to_consumers()` permettant à l'implantation de publier un événement par la source `to_consumers`.

```
public class ServerImpl extends ServerCCM implements ServiceOperations {
  // requis par la classe de squelette étendu générée ServerCCM.
  public ServiceOperations _get_facet_the_service () {
    return (ServiceOperations) this ;
  }
  // requis par l'interface exemple::Service.
  public void display (String text) {
    /* affichage sur l'ihm */
    // notification aux consommateurs connectés.
    super.push_to_consumers (new EventImpl (text)) ;
  }
}
```

FIG. 6 – *L'implantation Java du composant* `Server`

L'implantation des maisons a pour rôles de décrire comment sont instanciés les nouveaux composants et de fournir le code associé aux opérations explicites. La Figure 7 présente le code minimal d'implantation de la maison `ServerHome`. La classe `ServerHomeImpl` hérite du squelette généré par la plate-forme OpenCCM. Elle doit fournir une méthode statique `_create_home` invoquée par le support d'exécution pour instancier la maison. Le squelette quant à lui invoque la méthode `_create_component` pour instancier l'implantation de composants associée à cette maison. Finalement, les opérations explicites `create_server` et `find_server` peuvent être implantées grâce aux opérations implicites fournies par le squelette.

Comme le lecteur peut le constater, l'implantation de composants et de maisons dans le cadre de la plate-forme OpenCCM doit respecter quelques conventions de programmation. En fait, ces conventions sont des « callback » invoqués par les squelettes pour autoriser une certaine souplesse au niveau de l'implantation, *e.g.* plusieurs classes pour implanter un même composant.
Toutefois, la majeure partie du code technique est automatiquement générée dans les squelettes : la prise en charge des ports, l'introspection, l'indexation des composants, l'activation des composants et maisons via le POA, le décodage des requêtes GIOP/IIOP, etc. De plus, la plate-forme OpenCCM peut aussi générer un moule de l'implantation. Il ne reste alors plus qu'à le remplir avec le code fonctionnel des composants et des maisons.

```
public class ServerHomeImpl extends ServerHomeCCM {
  // requis par le support d'exécution OpenCCM
  // pour créer une instance de la maison lors du déploiement.
  public static ServerHomePOA _create_home () {
    return new ServerHomeImpl () ;
  }
  // requis par le squelette étendu généré.
  public ServerOperations _create_component () {
    return new ServerImpl () ;
  }
 // requis par l'interface explicite de la maison.
 public Server create_server(String name) {
   return super.create( new NamePrimaryKeyImpl(name) );
 }
 public Server find_server(String name) {
   return super.find_by_primary_key( new NamePrimaryKeyImpl(name) );
 }
}
```

FIG. 7 – *L'implantation Java de la maison* `ServerHome`

Finalement, le processus de production des composants OpenCCM se conclut par la construction d'archives au format « Jar ». Celles-ci regroupent le résultat de la compilation des souches / squelettes générés par un compilateur OMG IDL2, des squelettes étendus produits par OpenCCM et les classes d'implantation des composants et des maisons.

5. Perspectives et conclusion

Dans l'état actuel, la plate-forme OpenCCM offre une chaîne flexible et extensible de compilation du nouveau langage OMG IDL3 et de production de composants en Java ainsi qu'un environnement souple de déploiement des composants (présenté dans [8]).
Le travail présenté dans cet article ne représente qu'une première étape et de nombreuses perspectives s'ouvrent à nous. Premièrement, la plate-forme OpenCCM intégrera prochainement un générateur et un support d'exécution pour des composants écrits en C++ et en OMG IDLscript, le support pour le langage CIDL et la prise en compte des descripteurs OSD. Cela permettra de raffiner l'ensemble de nos choix de conception.
Deuxièmement, le support d'exécution fournira des containers pour la prise en charge implicite des services systèmes. Ici, l'idée directrice est de concevoir des containers adaptatifs dans lesquels seront téléchargés des composants systèmes. Ainsi, nos containers ne seront pas juste limités à la persistance, aux transactions et à la sécurité mais pourront accueillir des fonctions d'audit, de tolérance aux pannes, de répartition de charge, de temps-réel, etc. Cela permettra de supporter des applications à contraintes d'exécution critiques. Pour cela, nous étudions une approche mixant méta-programmation et programmation orientée aspect.
Pour décrire et déployer les applications, nous travaillons sur un langage orienté composant à mi-chemin entre un langage de description d'architectures (purement descriptif) et un langage de scripts (purement opératoire). Ce langage sera associé à une machine virtuelle de déploiement d'applications à grande échelle. Cette machine sera fiabilisée par l'intermédiaire des composants systèmes gérés par les containers adaptatifs. Le déploiement et l'administration des applications seront guidés par un canevas de courtage orienté composants (*i.e.* recherche de types, archives, instances, containers, etc.) et les outils associés (console graphique, etc.) [10, 11].

En guise de conclusion, les intergiciels pour composants répartis deviennent progressivement les supports de référence pour la construction des larges applications distribuées. Dans ce contexte, le modèle de composants CORBA s'annonce comme le plus prometteur : il a l'ambition de couvrir globalement le processus de caractérisation, de production, de diffusion, de déploiement et d'exécution d'applications. Afin que la communauté système puisse expérimenter ce fort potentiel, nous avons présenté dans cet article la plate-forme OpenCCM : la première implantation libre, ouverte et flexible du CCM disponible à l'adresse suivante :

`http://corbaweb.lifl.fr/OpenCCM/index.html`

Bibliographie

1. R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.-Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In *Proceedings IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, Angleterre, Septembre 1998.
2. A. DeSoto. Using the Bean Development Kit - a Tutorial. *http://java.sun.com/beans/docs/Tutorial-Nov97.pdf*, Novembre 1997.
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Westley Professional Computing, USA, 1995.
4. J.-M Geib, C. Gransart, and P. Merle. *CORBA : des concepts à la pratique*. Ed. Dunod, Paris, 1999. *ISBN: 2-10-004806-6*.
5. R. Grimes. *Professional DCOM Programming*. Wrox Press ltd., Birmingham, Canada, 1997.
6. S. Grisouard. VIPeR white paper. In *Actes du 3ème Colloque International sur les NOuvelles TEchnolgies de la RÉpartition (NOTERE'2000)*, Paris, France, Novembre 2000.
7. Inprise. VisiBroker 4.5 for Java User Guide, 2001. *http://www.inprise.com*.
8. R. Marvie, P. Merle, and J.-M. Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, Anvers, Belgique, Septembre 2000. IEEE.
9. R. Marvie, P. Merle, J.-M. Geib, and C. Gransart. CCM + IDLscript = Applications Distribuées. *Evolution des plates-formes orientées objets répartis, numéro spécial de Calculateurs Parallèles et Réseau*, 12(1):75–104, 2000. Ed. Hermès, *ISBN: 2-7462-0169-0*.
10. R. Marvie, P. Merle, J.-M. Geib, and S. Leblanc. TORBA: vers des contrats de courtage. In *3ème Colloque International sur les NOuvelles TEchnologies de la RÉpartition (NOTERE'2000)*, Paris, France, Novembre 2000.
11. R. Marvie, P. Merle, J.-M. Geib, and S. Leblanc. TORBA: Trading Contracts for CORBA. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio, Texas, USA, Janvier 2001. USENIX.
12. V. Matena and M. Hapner. *Enterprise Java Beans Specification v1.1 - Final Release*. Sun Microsystems, Mai 1999.
13. Metamata. Java Compiler Compiler. *http://www.metamata.com/javacc/index.html*.
14. Sun Microsystems. *Java Remote Method Invocation Specification*. Sun Microsystems, Octobre 1998.
15. OMG. *CORBA Components: Joint Revised Submission*. Object Management Group, Août 1999. OMG TC Document orbos/99-07-{01..03,05} orbos/99-08{05..07,12,13}.
16. OMG. *IDLscript Specification*. Object Management Group, Août 2000. OMG TC Document orbos/00-08-03.
17. OMG. *CORBA/IIOP 2.4.1 Specification*. Object Management Group, Janvier 2001.
18. OOC. ORBacus 4.0.5 for C++ and Java User Guide, 2001. *http://www.ooc.com*.
19. OpenORB. OpenORB 1.0.1 User Guide, 2001. *http://www.openorb.org*.
20. D. Rogerson. *Inside COM*. Microsoft Press, Redmond VA, USA, 1997.
21. N. Wang, D. Schmidt, M. Kircher, and K. Parameswaran. Applying Reflective Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC 2000)*, Tapei, Taiwan, Octobre 2000.
22. N. Wang, D. Schmidt, and D. Levine. Optimizing the CCM for High-Performance and Real-Time Applications. Middleware'2000, Work-in-Progress Session, 1999. New-York, USA.

23. N. Wang, D. Schmidt, and C. O'Ryan. *Component-Based Software Engineering: Putting the Pieces Together*, chapter An Overview of the CORBA Component Model. Addison-Westley, 2001.
24. A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9, Novembre/Decembre 1996.