Thomas Quinot

École nationale supérieure des télécommunications
Université Paris VI – Pierre-et-Marie-Curie

Traduction de l'annexe des systèmes répartis d'Ada 95 vers OMG IDL : spécification et mise en œuvre

22 mai 2000

Table des matières

Chapitre 1

Introduction

De nombreux modèles d'objets répartis sont actuellement proposés par des industriels ou par des groupes de normalisation. On citera comme exemples le modèle CORBA [Object Management Group, 1998] proposé par l'Open Management Group (OMG), l'annexe des systèmes répartis d'Ada 95 [ISO, 1995] proposé par l'ISO et le modèle RMI (*Remote Method Invocation*) de Java proposé par Sun. Chaque modèle propose des fonctionnalités intéressantes mais disponibles uniquement dans le cadre d'un même modèle. D'autre part, chaque modèle s'accompagne d'un certain nombre de contraintes quant à sa mise en œuvre, ses protocoles et son espace d'adressage.

L'un des axes de recherche du département Informatique et Réseaux de l'ENST est l'interopérabilité entre ces différents modèles. Dans [Pautet et al., 1999], reproduit en annexe de ce mémoire, nous avons présenté une comparaison entre les fonctionnalités offertes par CORBA et l'annexe des systèmes répartis d'Ada 95. Dans CORBA, les services fournis par un objet réparti sont décrits à l'aide d'un langage spécifique, OMG IDL. L'OMG a défini une projection du modèle d'objets et des définitions IDL vers différents langages hôtes tels que C++, Java et Ada. Un service défini par un contrat IDL peut être indifféremment mis en œuvre dans n'importe quel langage hôte, et être appelé à partir d'un client écrit dans n'importe quel autre langage : le mécanisme d'appel utilisé est indépendant des langages utilisés par le client et le serveur. Cette possibilité de faire interagir différents environnements est un avantage certain de CORBA. Au contraire, l'annexe des systèmes répartis d'Ada 95 permet seulement de faire communiquer des objets Ada.

Pour pallier cet inconvénient, nous souhaitons offrir aux implémenteurs de services la possibilité d'utiliser Ada 95 et l'annexe des systèmes répartis, tout en permettant à des clients CORBA d'accéder à ces services. Par ce moyen, nous avons pour objectif de permettre le développement de services en Ada utilisant l'annexe, bénéficiant de ses propriétés de sûreté et de typage fort, mais aussi d'une ouverture vers de nombreux langages et systèmes hôtes apportée par CORBA.

À cet effet, nous souhaitons générer automatiquement la spécification d'un service en OMG IDL, à partir de sa spécification en Ada 95. En utilisant un traducteur CORBA préexistant, nous pouvons générer automatiquement un squelette de serveur en Ada. L'implémentation de ce serveur est également générée automatiquement par un autre module de l'outil que nous avons développé.

La mise en œuvre des méthodes de chaque objet CORBA ainsi défini consiste en des appels aux services Ada 95 correspondants. Nous aurons ainsi réalisé un relais jouant à la fois le rôle de serveur CORBA et de client de l'annexe ; cet agent permettra d'utiliser les services « actifs » de l'annexe à partir du monde CORBA. Nous ne proposons pas, dans ce projet, de méthode d'accès aux services « passifs » de l'annexe (les paquetages *Shared Passive*) à partir de CORBA.

Ce document présente l'ensemble des étapes de ce projet. Dans la section suivante, nous décrivons nos motivations pour définir une traduction formelle d'une spécification de paquetage Ada 95 en contrat OMG IDL, ainsi que la structure générale de cette traduction. Nous présentons ensuite le modèle de traduction proprement dit, à partir de la grammaire d'Ada 95, en utilisant également des propriétés sémantiques du langage. Nous décrivons la mise en œuvre de cette traduction au moyen d'ASIS, une API normalisée pour la réflexivité des programmes Ada, puis la génération automatisée d'une implémentation des objets CORBA ainsi spécifiés. Nous concluons en présentant quelques perspectives d'extensions et de recherches prolongeant ce mémoire.

Chapitre 2

Motivation et contexte de la traduction

2.1 Présentation des systèmes d'objets répartis

2.1.1 Annexe E d'Ada 95

L'annexe des systèmes répartis d'Ada 95 fournit des mécanismes pour la programmation d'applications réparties. Une application peut être divisée en partitions, dont chacune s'exécuter sur un hôte différent. Les partitions communiquent par appels de sous-programmes à distance, en respectant toujours les règles de typage d'Ada. Ces appels distants peuvent être résolus statiquement (au partitionnement) ou dynamiquement (à l'exécution), restituant ainsi deux modèles classiques d'applications réparties : les sous-programmes distants (RPC) et les objets répartis. En effet, l'annexe permet la définition de types *access* particuliers permettant de faire référence à un objet distant. Les appels d'opérations primitives sur un objet désigné par un tel pointeur sont effectués à distance, de façon transparent, sur la partition où réside l'objet.

La définition de services accessibles à distance se fait à l'aide des pragmas de catégorisation `Pure`, `Remote_Types` et `Remote_Call_Interface`. Un quatrième pragma, `Shared_Passive`, permet de définir des données partagées entre partitions. Dans l'annexe, c'est donc le même langage (Ada lui-même) qui est utilisé pour décrire un service, pour y faire appel et pour l'implémenter.

Dans le cadre de ce projet, nous utilisons GLADE, une mise en œuvre de l'annexe pour le compilateur libre GNAT [Schonberg and Banner, 1994]. GLADE a été initialement développé en partenariat par TAMU[1], Ada Core Technologies et l'ENST [Kermarrec et al., 1995] ; Les travaux de TAMU ont évolué parallèlement, en utilisant une ancienne version de la couche de communication de GLADE, pour produire une autre implémentation de l'annexe : ADEPT.

2.1.2 CORBA

Au contraire de l'annexe, CORBA utilise un langage spécifique pour décrire les services accessibles à distance. Ce langage est l'IDL normalisé par l'OMG. IDL

[1] Texas A&M University

permet de définir des modules, des constantes, des types de données et des interfaces. IDL est un langage descriptif purement orienté objets ; dans cette mesure, il présente une certaine ressemblance avec Java. Sa syntaxe est proche de celle de C++. L'OMG a normalisé la projection des spécifications IDL vers différents langages hôtes (C, C++, Ada, Java, etc.) Cette projection est en générale naturelle ; toutefois, quand une construction IDL n'a pas d'équivalent direct dans le langage cible, les documents de l'OMG spécifient comment elle doit être simulée. Un traducteur IDL génère automatiquement des souches clientes et des squelettes d'implémentation pour chaque objet décrit en IDL. Une souche cliente est un module en langage hôte présentant au programme utilisateur l'interface du service décrit par le contrat IDL, et réalisant des appels au sous-système de communication CORBA pour traiter les requêtes de l'utilisateur ; un squelette d'implémentation, réciproquement, reçoit des requêtes en provenance du sous-système de communication, et effectue les appels aux fonctions fournies par l'implémenteur du service avant de retourner les réponses.

2.2 Présentation du projet CIAO

L'objectif du projet CIAO[2] est de permettre à un utilisateur CORBA de faire appel à un service fourni grâce à l'annexe. L'utilisateur CORBA a donc besoin d'une description en OMG IDL de ce service. Ce contrat IDL peut être généré automatiquement si l'on dispose d'un schéma formel de traduction d'une spécification de paquetage de l'annexe vers OMG IDL.

Nous allons donc proposer une application de l'ensemble des spécifications de services de l'annexe (*i. e.* l'ensemble des arbres syntaxiques Ada 95 correspondant à la déclaration d'une library_unit ayant l'une des catégories *Declared Pure*, *Remote Types* ou *Remote Call Interface*) dans l'ensemble des contrats OMG IDL (*i. e.* l'ensemble des arbres syntaxiques OMG IDL).

Cette application est définie dans le même esprit que la projection canonique d'OMG IDL vers Ada. Les constructions « orientées objets » d'Ada sont traduites en leurs équivalents IDL. En Ada, un type *tagged private* pour lequel il existe un type *Remote Access to Class-Wide* (RACW) définit un objet réparti. Nous représentons un tel type par une `interface` IDL, et ses sous-programmes primitifs sont représentés par les opérations de cette `interface`. Un type *Remote Access to Subprogram* (RAS) est également représenté par une `interface` ayant une seule opération « `Call` ». Enfin, les sous-programmes appelables à distance d'un paquetage *Remote Call Interface* sont considérés comme les opérations d'une interface « `Remote_Subprograms` ». Une interface « Constants » est également créée, et contient les traductions des constantes définies par un service Ada.

La traduction doit également conserver la structure générale du service : à chaque paquetage Ada correspond un module IDL ; un paquetages de bibliothèque (*library package*) est traduit par exactement un fichier source IDL ; si des sous-paquetages sont présents, ils sont traduits par des modules imbriqués.

Nous allons présenter dans la partie 3 la traduction que nous proposons. Nous indiquons, pour chaque élément de la grammaire Ada, en partant de l'axiome compilation_unit, une traduction sous forme de zéro ou un élément de la grammaire OMG IDL. La traduction de certains éléments n'est pas actuellement supportée ;

[2]CORBA Interface for Ada (DSA) Objects

cela restreint l'ensemble des paquetages traduisibles. Ces restrictions sont indiquées au fil du texte, et rappelées dans la section 3.7.

Lorsque l'élément Ada est un non-terminal, nous donnons une traduction pour chaque forme alternative. Si l'élément IDL correspondant est un non-terminal, nous indiquons comment ses sous-éléments doivent être construits. En général cette construction se fait récursivement en termes de traductions des sous-éléments de l'élément Ada, et de ses propriétés sémantiques.

Pour certaines formes, nous n'explicitons pas de traduction. La traduction par défaut est alors la concaténation des traductions des éléments qui définissent la forme. Par exemple, si l'on considère un non-terminal foo : := bar baz, alors en l'absence d'autre indication, la traduction d'un foo est la traduction du bar qu'il contient, suivie de celle du baz.

Nous avons choisi, autant que faire se peut, de rester proches de la projection canonique d'OMG IDL vers Ada. Nous en avons dévié chaque fois que cela semblait nécessaire ou souhaitable, en considérant que la traduction doit permettre la mise en œuvre de l'agent d'interface décrit plus loin (partie 5), et que l'IDL généré doit être aisément lisible par un programmeur souhaitant écrire un client CORBA.

2.3 Projets connexes

Plusieurs projets existent déjà qui, comme CIAO, ont pour objectif d'interfacer des modèles d'objets répartis incompatibles. Nous présentons ici deux exemples qui paraissent particulièrement pertinents : *Adept/JxA* et *Java to IDL*.

2.3.1 Adept/JxA

Après la publication des premières versions de Glade, l'équipe ADEPT de TAMU a poursuivi le développement de sa propre implémentation de l'annexe. Elle a ainsi produit ADEPT/JxA, une passerelle entre l'annexe et les objets répartis de Java/RMI [Gargaro et al., 1997b, Gargaro et al., 1997c, Gargaro et al., 1997a]. L'objectif d'ADEPT/JxA est comparable à celui du présent projet : ouvrir l'accès à des services développés à l'aide de l'annexe pour des clients issus d'autres modèles d'objets répartis. Pour simplifier la traduction d'une spécification Ada vers une classe Java, ADEPT impose l'utilisation de types particuliers définis dans un paquetage `Interfaces.Java`. Nous souhaitons adopter une démarche plus générale et permettre, autant que possible, la traduction sans modification de services existants ; lorsqu'un type Ada ne peut pas être traduit « naturellement » en IDL, nous donnons ici une construction permettant de le simuler au mieux à l'aide des types disponibles dans CORBA. Quelques restrictions subsistent ; elles sont regroupées dans la section 3.7

2.3.2 Java to IDL

L'OMG a entrepris de normaliser une projection des objets répartis de RMI sur OMG IDL [Object Management Group, 1999], avec des motivations similaires à celles du projet CIAO. Cette projection est définie pour un sous-ensemble de Java/RMI. À l'instar d'autres documents de l'OMG, la spécification *Java to IDL Mapping* donne une description de haut niveau, peu formelle, de la traduction.

Partant de cette spécification, la réalisation d'un traducteur en Java a été réalisée dans le cadre d'un mémoire de fin d'études ENST [Bourg, 1999]. Ce projet, réalisé en collaboration avec les services de recherche d'EDF, utilise l'API de réflexivité de Java pour explorer la spécification d'un service RMI. Il produit un contrat IDL selon le modèle de traduction défini par la projection de l'OMG.

Cette étude constitute une première étape dans la réalisation d'un système faisant interopérer des entités CORBA et RMI. Le prolongement de ce traducteur par l'implémentation de relais applicatifs reste à faire.

Chapitre 3

Schéma de traduction

3.1 Notations

Les règles syntaxiques Ada sont rappelées dans les paragraphes *Syntaxe*. Elles sont données en utilisant les mêmes notations que le Manuel de référence du langage [ISO, 1995].

Les non-terminaux IDL sont notés comme dans la documentation de l'OMG [Object Management Group, 1998], par exemple <simple_type_spec>. Quand cela sera nécessaire, nous ajouterons à cette notation des informations sémantiques, comme cela est fait dans la notation Ada : <*structure*_identifier>. Nous décrirons certaines constructions IDL par des insertions de code. Dans ces fragments, nous pourrons laisser des non-terminaux non développés :

```
enum \<enumeration_/identifier> { \<enumeration_literal>, ... }
```

FIG. 3.1 – Exemple de fragment de code

3.2 Éléments lexicaux

Au cours de la traduction, nous aurons à représenter des identificateurs Ada par des équivalents en IDL. Nous posons que la traduction d'un identifier Ada est en général l'<identifier> IDL qui a la même représentation. De plus, certaines constructions IDL traduisent des constructions anonymes en Ada. Nous leur donnons un <identifier> formé en ajoutant un suffixe à la représentation d'un identifier Ada. Si un conflit de nommage découle de ce procédé, nous ajoutons des suffixes aux identificateurs IDL pour lever toute ambiguïté.

Ces règles suffisent à la traduction des defining_identifier. La traduction d'un name doit, de plus, tenir compte des règles de résolution de noms d'Ada et d'IDL. Si un name Ada dénote une entité qui n'est pas déclarée dans l'unité en cours de traduction, il doit être traduit par un <scoped_name> contenant les indications de portée nécessaires à sa résolution. Les règles de traduction de compilation_unit (cf. 3.3) garantissent qu'il est toujours possible de désigner sans ambiguïté la construction IDL traduisant une entité Ada visible. Il n'est donc pas nécessaire de traduire les clauses `use` d'Ada ; chaque fois que c'est nécessaire, nous utilisons en IDL des noms (<scoped_name>) totalement qualifiés.

Les identificateurs prédéfinis (déclarés dans le paquetage normalisé `Standard` sont traduits dans le module `CIAO::Standard` (cf. 3.6), avec un préfixe `Ada_` pour éviter toute collision avec les identificateurs prédéfinis IDL.

3.3 Unités de compilation

On considère une compilation_unit qui est la déclaration d'une unité de bibliothèque ayant l'une des catégories *Declared Pure*, *Remote Types* ou *Remote Call Interface*.

Syntaxe

compilation_unit ::=
    context_clause library_item
  | context_clause subunit

context_clause ::= { context_item }

context_item ::= with_clause | use_clause

with_clause ::= **with** *library_unit*_name {, *library_unit*_name };

Traduction

Une compilation_unit est traduite par une <specification> IDL. Nous ne devons considérer ici que la première forme de compilation_unit. En effet, nous ne nous intéressons qu'aux *déclarations* d'unités de bibliothèque, les éléments par lesquels le programmeur définit la *spécification* d'un service.

En premier lieu, la <specification> contient une directive d'inclusion de fichier #include "ciao.idl". Ce fichier fournit des déclarations communes ; son contenu est donné en annexe (cf. 3.6). Ces déclarations sont utilisées pour construire certaines traductions d'éléments Ada au long de ce document.

Les with_clause dont le *library_unit*_name dénote un paquetage de bibliothèque sont traduites par des directives d'inclusion #include "*filename*", où *filename* est pour chaque clause le nom du fichier IDL produit par la traduction du paquetage dénoté. Aucun autre context_item n'est traduit.

Syntaxe

library_item ::= [**private**] library_unit_declaration
  | library_unit_body
  | library_unit_renaming_declaration

library_unit_declaration ::= subprogram_declaration
  | package_declaration
  | generic_declaration
  | generic_instanciation

Seules deux formes de library_item peuvent déclarer effectivement des services appelables à distance : les déclarations de paquetage (package_declaration) et les instanciations de paquetage générique (package_instanciation). La traduction des déclarations de paquetage est traitée dans la section suivante. Celle des instanciations de paquetage générique nécessiterait la mise en œuvre de règles complexes de résolution de noms. Nous choisissons donc de ne pas l'autoriser.

3.4 Déclarations

3.4.1 Déclaration de paquetage

Syntaxe

package_declaration ::= package_specification;

package_specification ::=
    package defining_program_unit_name **is**
        { basic_declarative_item }
   [**private**
        { basic_declarative_item }
    end [[parent_unit_name.]identifier]

defining_program_unit_name ::= [parent_unit_name.]defining_identifier

Traduction

Une package_specification est traduite par un <module>.

Si l'unité est un paquetage enfant, l'<identifier> du <module> doit rendre compte de l'ensemble de ses ascendants. Il est donc construit en concaténant chacun des direct_names qui composent le parent_unit_name avec le defining_identifier, en les séparant par des caractères « souligné » doublés (__). Sinon, l'<identifier> traduit simplement le defining_identifier. De plus, le nom d'un paquetage de bibliothèque est toujours traduit en y préfixant la chaîne `CORBA__`, pour permettre la cohabitation d'un paquetage CORBA pour ce module (généré par un traducteur IDL vers Ada) et du paquetage DSA d'origine dans une même partition.

La déclaration d'un paquetage fils a visibilité sur la déclaration de ses ascendants. Pour conserver cette propriété, la traduction d'un paquetage fils commence par une directive d'inclusion du fichier traduisant le paquetage père. Le module a ainsi visibilité sur toutes les déclarations nécessaires, sans que nous introduisions de collision de nommage.

Les <definition> du module sont les traductions des basic_declarative_item : en général, un basic_declarative_item est traduit par une <definition>, et dans ce cas se rattache au <module>. Cependant, les basic_declarative_item qui définissent des définissant des opération primitives d'objets répartis (subprogram_declaration et abstract_subprogram_declaration) sont traduits comme des <export>, et dans ce cas ils se rattachent à un <interface_body>.

Exemple

```
with Foo.Bar;

package A.B.C is
   -- declarations
end A.B.C;
```

(a) Spécification Ada

```
#include "ciao.idl"
#include "CORBA__Foo__Bar.idl"
#include "CORBA__A__B.idl"

module CORBA__A__B__C {
  // declarations
};
```

(b) Contrat IDL

FIG. 3.2 – Structure de la traduction

Syntaxe

basic_declarative_item ::=
    basic_declaration | representation_clause | use_clause

basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | package_declaration | renaming _declaration
  | exception_declaration | generic_declaration
  | generic_instantiation

Traduction

Une use_clause ne fait que modifier le comportement de résolution des noms dans le processus de traduction, mais ne produit aucune traduction dans le fichier IDL.

Dans les cas où une basic_declaration est :
- une type_declaration ou une subtype_declaration ;
- une object_declaration (qui est nécessairement constante, cf. [ISO, 1995, 10.2.1(16), E.2.2(7), E.2.3(10)]) ou une number_declaration ;
- une package_declaration ;

sa traduction est une <definition>, dont la construction est donnée ci-après.

Une subprogram_declaration ou une abstract_subprogram_declaration est traduite par un <export> (cf. section 3.4.4) qui se rattache à un <interface_body>. Les exception_declaration et generic_declaration ne sont pas traduites. Toutes les exception Ada sont reflétées par l'exception IDL `Ada_Exception`. Cela permet de transmettre aux clients CORBA une information complète sur les exceptions Ada levées par un serveur de l'annexe, sans qu'il soit nécessaire de prévoir à l'avance tous les types d'exceptions que pourrait lever ce serveur. Une déclaration d'unité générique ne crée aucun service appelable à distance (un service peut éventuellement être créé par *l'instanciation* d'une unité générique, mais cette possibilité n'est pas autorisée pour l'instant par le schéma de traduction).

La traduction d'une generic_instantiation n'est pas permise actuellement. De même, les renaming_declaration donnent lieu à des problèmes complexes de visibilité et de résolution de noms, et nous décidons pour l'instant de ne pas permettre leur traduction.

3.4.2 Déclaration de type ou de sous-type

Syntaxe

type_declaration ::= full_type_declaration
  | incomplete_type_declaration
  | private_type_declaration
  | private_extension_declaration

full_type_declaration ::=
    type defining_identifier [known_discriminant_part] **is** type_definition;
  | task_type_declaration
  | protected_type_declaration

Traduction

Une incomplete_type_declaration n'est pas traduite. En effet, elle est toujours complétée par une full_type_declaration, et tant que cette complétion n'a pas été rencontrée, le defining_identifier ne peut être utilisé que pour former des déclarations de pointeurs (access_to_object_definition ou access_definition).

Une access_to_object_definition sera toujours traduite par un type « opaque » indépendant du sous-type désigné si celui-ci est issu d'une full_type_declaration (cf. 3.4.2). De la même façon, la traduction d'une access_definition qui définit un discriminant ne nécessite pas la connaissance exacte de la nature de l'objet désigné.

Le seul cas problématique est celui où une incomplete_type_declaration est utilisée pour former un paramètre formel d'un type pointeur anonyme dans une signature de sous-programme. En Ada, il se peut que la présence de cette signature *avant* la complétion soit requise (par exemple dans le cas où l'on définit un type *access* sur sous-programme et que la complétion est une record_type_definition dont un membre est de ce type). Le sous-programme, s'il est traduit, se rattache à une interface. Or, nous n'émettons qu'une <forward_dcl> là où nous insérerions normalement la traduction de la déclaration du type privé correspondant à cette interface (cela ne modifie en rien par ailleurs la validité ni le sens de la traduction), et nous retardons l'insertion de l'<interface_dcl> jusqu'au moment où la full_type_declaration a été traduite. Nous pouvons ainsi omettre entièrement les déclarations de types incomplets dans la traduction, et ne traduire que leur complétion.

Nous donnons ci-après la traduction des éléments :
- full_type_declaration ;
- private_type_declaration ;
- private_extension_declaration.

Une full_type_declaration déclare un type Ada. Elle est naturellement traduite par une <type_dcl> de la forme `typedef` <type_declarator>, déclarant un type IDL correspondant. Une known_discriminant_part ne peut être présente que pour la déclaration d'un type composite qui n'est pas un tableau, c'est-à-dire un type structure, un type tâche ou un type protégé. Si c'est une task_type_declaration ou une protected_type_declaration, alors le <type_declarator> est le <scoped_name> `CIAO::Opaque` dénotant un type défini dans les déclarations communes, et de nature à contenir un flot quelconque d'octets. En effet, un type tâche ou protégé est un type limité non étiqueté. Lorsqu'un objet d'un tel type doit être transmis au cours d'un appel distant, le langage spécifie que l'utilisateur est responsable de sa mise en forme (*marshalling*) sous forme de flux d'octets transmissible sur le réseau. (cf. [ISO, 1995, E.2.2(14) et E.2.3(14)]). Nous utiliserons encore cette traduction comme « type opaque » plus loin dans ce schéma, pour tous les types Ada dont le mode de transmission est défini par l'utilisateur (au moyen des attributs `'Read` et `'Write`).

Le <declarators> du <type_declarator> est un seul <identifier> traduisant le defining_identifier de la task_type_declaration ou de la protected_type_declaration.

Quand la full_type_declaration contient une type_definition, la construction du <type_declarator> dépend de la forme de la type_definition.

Syntaxe

type_definition ::=
  enumeration_type_definition | integer_type_definition
 | real_type_definition | array_type_definition
 | record_type_definition | access_type_definition
 | derived_type_definition

array_type_definition ::= unconstrained_array_definition
 | constrained_array_definition

unconstrained_array_definition ::=
  array (index_subtype_definition {, index_subtype_definition })
  of component_definition

constrained_array_definition ::=
  array (discrete_subtype_definition {, discrete_subtype_definition })
  of component_definition

component_definition ::= **[aliased]** subtype_indication

Traduction

Si c'est une array_type_definition dont la component_definition dénote un sous-type limité (tel que défini par [ISO, 1995, 7.5(3)]), alors la full_type_declaration déclare aussi un type limité, et comme précédemment la <type_spec> est `CIAO::Opaque`, et le <declarators> provient du defining_identifier.

Une constrained_array_definition dont les éléments sont d'un type non limité définit un type tableau dont la taille est connue à la compilation. Nous pouvons donc le représenter par un tableau IDL de même longueur : la <type_spec> du <type_declarator> est la traduction de la component_definition, telle que définie dans la section 3.5 ; le <declarators> est un <array_declarator> dont l'<identifier> est la traduction du defining_identifier. Pour chaque discrete_subtype_definition, la traduction comporte une <fixed_array_size> égale à la longueur du tableau (son nombre d'éléments) sur la dimension correspondante.

En pratique, cependant, l'API ASIS normalisée ne permet pas de déterminer la valeur des bornes d'un tableau statique. Nous utilisons donc en fait la même traduction, plus générale, pour une constrained_array_definition que pour une déclaration de tableau non contraint.

Une unconstrained_array_definition définit également un tableau, dont la longueur et les bornes ne sont connues qu'à l'exécution. Nous ne pouvons donc pas utiliser un tableau IDL ; en revanche, une *séquence* est adaptée pour représenter ce type d'objet. Dans la traduction, la <type_spec> est le <struct_type> suivant :

```
struct \<T_/identifier>_with_bounds {
   unsigned long long array_bounds[\<N_/positive_int_const>];
   sequence<sequence<...\<component_/type_spec>>> array_values;
};
```

Fig. 3.3 – Traduction de tableau non contraint

où :

- <*T*_identifier> traduit le defining_identifier de la full_type_declaration ;
- <*N*_positive_int_const> est le nombre de dimensions du tableau ;
- le composant `array_bounds` contient la borne inférieure des indices pour

chaque dimension du tableau (la borne supérieure est implicitement définie par la borne inférieure et la longueur de la séquence) ;

- <type_spec> traduit component_definition ;
- le non-terminal <sequence_type> est itéré récursivement une fois pour chaque dimension du tableau.

Nous utilisons les séquences IDL pour représenter des familles d'éléments ordonnées, de longueur quelconque. La borne définie par l'élément d'indice 0 de `array_bounds` est relative à la séquence la plus externe.

Dans le cas des tableaux contraints comme des tableaux non contraints, il appartient au développeur du client CORBA de déterminer comment il doit indicer les tableaux, en tenant compte des bornes qu'il connaît.

Pour toutes les autres formes de type_definition, la <type_spec> est la traduction du sous-élément de type_definition, donnée dans la suite de cette section, et le <declarators> traduit le defining_identifier.

Syntaxe

enumeration_type_definition ::=
    (enumeration_literal_specification {, enumeration_literal_specification})

enumeration_literal_specification ::= defining_identifier
    | defining_character_literal

Traduction

Une enumeration_type_definition se traduit par un <enum_type>, avec :

- un <identifier> formé en ajoutant le suffixe `_enum` au defining_identifier ;
- pour chaque enumeration_literal_specification, un <enumerator> qui est la traduction du defining_identifier pour la première forme, ou bien `CHAR_` suivi du nom de la constante du paquetage normalisé Ada.Characters.Latin_1 correspondant au defining_character_literal pour la seconde forme.

Syntaxe

integer_type_definition ::= signed_integer_type_definition
    | modular_type_definition

real_type_definition ::= floating_point_definition | fixed_point_definition

Traduction

Une integer_type_definition est traduite par un <integer_type>.

Pour une signed_integer_type_definition, cet <integer_type> est `long long` ; dans le cas d'une modular_type_definition, c'est `unsigned long long`. Si une valeur Ada ne peut être représentée comme un entier 64 bits, une erreur devra être générée à l'exécution par l'agent d'interface.

Une real_type_definition est traduite par le <floating_pt_type> `long double`. Si une valeur Ada ne peut être représentée par le type choisi, une erreur devra être générée à l'exécution par l'agent d'interface.

Syntaxe

```
record_type_definition ::= [[abstract] tagged] [limited] record_definition

record_definition ::=
      record
        component_list
      end record
   |  null record

component_list ::=
      component_item { component_item }
   |  { component_item } variant_part
   |  null;

compnent_item ::= component_declaration | representation_clause

component_declaration ::=
      defining_identifier_list : component_definition [:= default_expression];

known_discriminant_part ::=
      (discriminant_specification {, discriminant_specification })

discriminant_specification ::=
      defining_identifier_list : subtype_mark [:= default_expression]
   |  defining_identifier_list : access_definition [:= default_expression]

defining_identifier_list ::= defining_identifier {, defining_identifier}
```

Traduction

Si une record_type_definition contient l'un des mots-clé `tagged` et `limited`, sa traduction le type opaque défini précédemment. Dans le cas contraire, c'est un <struct_type>, avec :

- un <identifier> formé en ajoutant le suffixe `_struct` au defining_identifier ;
- si la full_type_declaration contient une known_discriminant_part définissant un ou plusieurs discriminants scalaires, un <member> pour chacun de ces discriminants[1] ;
- une <member_list> traduisant la component_list, et comprenant :
  - un <member> pour chaque component_declaration ;
  - un <member> pour chaque variant_part, dont la construction est donnée ci-après.

Une representation_clause au sein d'une record_type_definition n'est pas traduite. En effet, une telle clause a pour objectif de spécifier la projection d'un type de donnée sur l'architecture matérielle sous-jacente ([ISO, 1995, 13.1(1)]). Nous prenons le parti de ne pas la refléter dans la traduction, car elle n'intervient pas sur le sens de la description abstraite de service que nous manipulons.

[1] Si une record_definition a un discriminant non scalaire, alors elle contient le mot-clé `limited`, cf. [ISO, 1995, 3.7(10)]

Syntaxe

variant_part ::=
        case *discriminant_*direct_name **is**
          variant
          {variant}
        end case;

variant ::=
        when discrete_choice_list =>
          component_list

discrete_choice_list ::= discrete_choice {| discrete_choice}

discrete_choice ::= expression | discrete_range | **others**

Traduction

Dans une record_type_definition ayant un ou plusieurs discriminants scalaires, une variant_part permet de définir des composants qui dépendent d'un discriminant. Les variant sont mutuellement exclusives. Nous représentons une telle structure en IDL à l'aide d'un type union.

Une variant_part définit un ensemble de champs de structure apparaissant simultanément dans une alternative. Elle est traduite par un <member> ainsi construit : son <identifier> traduit le *discriminant_*direct_name, avec le suffixe `_variant` ; sa <type_spec> est un <union_type> dont l'<identifier> est encore celui du discriminant, suivi de `_variant_union`. Le type du discriminant (IDL) de cette union est `long long int`. En effet, si nous voulions utiliser le type correspondant au discriminant Ada, il serait nécessaire de traduire un discrete_choice, c'est-à-dire une expression statique quelconque. De plus, dans le cas d'un discrete_range, il serait nécessaire d'énumérer explicitement toutes les valeurs possibles de l'intervalle, ce qui pourrait conduire à une traduction de taille déraisonnable.

Les membres de ce type union correspondent aux différentes variantes ; pour chacune d'elles, il nous faut représenter une component_list. Chaque variant est donc traduite par un <member> de l'union, dont l'<identifier> est celui du discriminant suivi de `_var_`*p*, et la <type_spec> est un <struct_type> dont l'<identifier> est celui du discriminant suivi de `_var_`*p*`_struct`. Dans tous ces <identifier>, *p* représente la position de la variante dans la variant_part.

Comme précédemment, la <member_list> est la traduction de la component_list.

On associe un identificateur IDL à chaque discrete_choice. Il est formé en prenant l'identificateur du discriminant indiqué par la variant_part, et en y ajoutant en suffixe `_var_` puis la position de la variante dans la variant_part, ou bien `others` pour l'éventuelle discret_choice_list réduite au mot-clé `others`.

Exemple

```
type Colour is
   (Red, Green, Blue, Yellow);

type My_Record (D : Colour) is
   record
      Foo : Integer;
      case D is
         when Red =>
            Bar : String;
            Baz : Integer;
         when Green
           .. Blue =>
            Fuz : Integer;
         when others =>
            null;
      end case;
   end record;
```

(a) Spécification Ada

```
typedef enum Colour_enum {
   Red, Green, Blue, Yellow
} Colour;

typedef struct
 My_record_struct {
   Colour D;
   long long int Foo;
   union
    D_variant_union
      switch (long long) {

      case 0: // Red
         struct
          D_var_10_struct {
            string Bar;
            long long Baz;
         } D_var_0;

      case 1: // Green .. Blue
         struct
          D_var_1_struct {
            long long Fred;
         } D_var_1;

      default: // others
         struct
          D_var_others_struct {
         } D_var_others;

     } D_variant;
} My_Record;
```

(b) Contrat IDL

FIG. 3.4 – Structure à partie variante

Syntaxe

access_type_definition ::= access_to_object_definition
  | access_to_subprogram_definition

access_to_object_definition ::= **access** [general_access_modifier] subtype_indication

general_access_modifier ::= **all** | **constant**

access_to_subprogram_definition ::=
  access [**protected**] **procedure** parameter_profile
  | **access** [**protected**] **function** parameter_and_result_profile

Traduction

Une access_to_object_definition qui définit un RACW (cf. [ISO, 1995, E.2.2(9)]) crée un type référence vers objet distant. En IDL, on peut utiliser un nom d'interface pour dénoter un tel type. Un telle access_to_object_definition est donc traduite simplement par la <type_spec> correspondant à la subtype_indication (qui dénote le sous-type désigné par le RACW).

Toute autre access_to_object_definition définit un type pourvu de procédures `'Read` et `'Write` fournies par l'utilisateur ; elle est traduite comme un type opaque.

Une access_to_subprogram_definition définit un type sur lequel on peut effectuer un appel à distance. Nous le traduisons par une <interface_dcl> contenant une seule déclaration <op_dcl>. Son <identifier> est `Call` ; la construction de ses autres sous-éléments (<op_attribute>, <op_type_spec> et <parameter_dcls> est détaillée dans la section 3.4.4.

Syntaxe

derived_type_definition ::=
  [abstract] new *parent_*subtype_indication [record_extension_part]

subtype_declaration ::=
  subtype defining_identifier **is** subtype_indication**;**

Traduction

Comme une déclaration de type *tagged*, une derived_type_definition dont le type parent est *tagged* est traduite par un type opaque.

Si le type parent n'est pas un type *tagged*, aucun de ses types dérivés ne peut avoir de record_extension_declaration, et la traduction de la derived_type_definition est celle de sa *parent_*subtype_indication. De même, une subtype_declaration est traduite par une <type_dcl> dont l'<identifier> et la <type_spec> traduisent respectivement le defining_identifier et la subtype_indication.

3.4.3 Déclaration d'objet réparti

Syntaxe

private_type_declaration ::=
  type defining_identifier [discriminant_part] **is**
    [[abstract] tagged] [limited] private;

private_extension_declaration ::= **type** defining_identifier [discriminant_part] **is**
    [abstract] new ancestor_subtype_indication **with private;**

Traduction

Une déclaration de type *tagged limited private* crée un type susceptible d'être accessible à distance (*via* un RACW). Une private_type_declaration d'un tel type est donc traduite par une <interface_dcl> sans <inheritance_spec>, puisque le type Ada n'hérite pas d'un autre type objet réparti. L'<interface_body> est une liste d'<op_dcl> correspondant aux opérations primitives du type ; ces <op_dcl> proviennent de subprogram_declarations ou d'abstract_subprogram_declarations, dont la traduction est donnée ci-après (section 3.4.4).

De même, une private_extension_declaration dont le type parent est *limited private* (donc nécessairement *tagged*, cf. [ISO, 1995, 7.3(8)]) est traduite comme une <interface_dcl> dont l'<inheritance_spec> est le <scoped_name> désignant la traduction du type parent.

Dans l'arbre traduit, une <forward_dcl> est insérée à la position naturelle par rapport aux déclarations Ada. L'<interface_dcl> elle-même, quant à elle, est placée à la fin des déclarations du module courant. Cela garantit que, lorsqu'on produit l'IDL d'une <op_dcl>, tous les types nécessaires ont déjà été traduits. Cela résout, en particulier, le problème des paramètres de types incomplets (cf. 3.4.2).

Toute autre private_type_declaration ou private_extension_declaration définit un type dont le client ne connaît pas le contenu, et se traduit naturellement comme un type opaque.

Exemple

```
type My_Object is
  tagged limited private;

procedure Op_1 (Self: in My_Object);
function Op_2 (Self : in My_Object;
               I : in Integer)
  return Boolean;

type My_Ref is
  access all My_Object'Class;
```

(a) Spécification Ada

```
interface My_Object {
  void Op_1();
  Ada_Boolean
    Op_2(in Ada_Integer I);
};

typedef My_Object My_Ref;
```

(b) Contrat IDL

FIG. 3.5 – Traduction d'un objet réparti

3.4.4 Déclaration de sous-programme

Syntaxe

subprogram_declaration ::= subprogram_specification**;**

abstract_subprogram_declaration ::= subprogram_specification **is abstract;**

subprogram_specification ::=
    procedure defining_program_unit_name parameter_profile
  | **function** defining_designator parameter_and_result_profile

designator ::= [parent_unit_name **.**]identifier | operator_symbol

defining_designator ::= defining_program_unit_name | defining_operator_symbol

parameter_profile ::= [formal_part]

parameter_and_result_profile ::= [formal_part] **return** subtype_mark

formal_part ::= **(**parameter_specification {**;** parameter_specification}**)**

parameter_specification ::=
    defining_identifier_list **:** mode subtype_mark [**:=** default_expression]
  | defining_identifier_list **:** access_definition [**:=** default_expression]

access_definition ::= **access** subtype_mark

mode ::= [**in**] | **in out** | **out**

Traduction

Un sous-programme (procédure ou fonction) est une opération primitive d'au plus un type *tagged*. Une subprogram_specification pour une opération primitive d'un type *tagged limited private* (traduit par une <interface>) définit une méthode d'un objet réparti. Elle est donc traduite par une <op_dcl> qui se rattache à l'<interface_body> correspondant à cet objet. Dans un paquetage *Remote Call Interface*, une déclaration de sous-programme qui n'est pas une primitive d'un objet réparti est traduite comme une opération de l'interface `Remote_Subprograms`.

Dans un paquetage *Declared Pure* ou *Remote Types*, une déclaration de sous-programme qui n'est pas une primitive d'un objet réparti définit une procédure ou une fonction qui ne peut être appelée que localement, et non à distance. Une telle déclaration n'est donc pas traduite.

L'<op_dcl> est ainsi construite :

- l'<op_type_spec> est `void` pour une procédure, ou bien la traduction de la subtype_mark dénotant le type de retour pour une fonction ;
- l'<dentifier> traduit le defining_designator (Pour un operator_symbol, nous le traduisons par `Operator_` suivi du nom de l'opérateur. En effet, IDL ne permet pas la surcharge des opérateurs) ;
- l'élément <parameter_dcls> est une suite de <param_dcl> qui traduisent les parameter_specification, à l'exception de celle qui définit le premier opérande formel contrôlant, c'est-à-dire à l'instance d'objet sur laquelle l'opération est invoquée (dans le cas d'une primitive d'objet réparti), cet opérande correspondant au paramètre implicite `Self` de la méthode IDL ;
- la <raises_expr> `raises CIAO::Ada_Exception` indiquant que la seule exception que peut lever l'opération est l'exception `Ada_Exception` définie dans les déclarations communes.

La traduction d'une parameter_specification est une suite de <param_dcl>, un pour chaque membre de la defining_identifier_list. Si le type du paramètre formel est défini par une subtype_mark, le <param_attribute> traduit naturellement le mode, et la <param_type_spec> traduit la subtype_mark. Si le paramètre formel est défini par une access_definition, le <param_attribute> est `inout` et la <param_type_spec> traduit le sous-type désigné par cette access_definition.

3.4.5 Déclaration de constante

Syntaxe

object_declaration ::=
    defining_identifier_list **:** [**aliased**] [**constant**] subtype_indication
        [:= expression]**;**
  | defining_identifier_list **:** [**aliased**] [**constant**] array_type_definition
        [:= expression]**;**
  | single_task_declaration
  | single_protected_declaration

number_declaration ::=
    defining_identifier_list **: constant :=** static_expression**;**

Traduction

Une object_declaration qui déclare une tâche ou un objet protégé ne donne pas lieu à l'exportation d'un service appelable à distance, et n'est donc pas traduite.

Dans un paquetage *Declared Pure*, *Remote Types* ou *Remote Call Interface*, une object_declaration qui déclare une instance d'un type est toujours une déclaration de constante. Elle est traduite par des <op_dcl> rattachées à l'<interface> `Constants`, une pour chaque defining_identifier. L'<identifier> de chaque <op_dcl> traduit un defining_identifier ; L'<op_type_spec> est la traduction de la subtype_indication, et la fonction ne prend aucun paramètre. Cette fonction retourne la valeur de la constante, telle qu'elle est connue par le serveur.

Une number_declaration est traduite de façon similaire. Si la static_expression est d'un type entier, l'<op_type_spec> de la fonction est `long long int` ; si elle est d'un type réel, l'<op_type_spec> est `long double`.

3.5 Indications de sous-type

En plusieurs points de la traduction, nous faisons appel à la traduction d'une subtype_indication par une <type_spec>.

Syntaxe

subtype_indication ::= subtype_mark [constraint]

subtype_mark ::= *subtype*_name

name ::=
    direct_name | explicit_dereference
  | indexed_component | slice
  | selected_component | attribute_reference
  | type_conversion | function_call
  | character_literal

direct_name ::= identifier | operator_symbol

Traduction

Une indication de sous-type (subtype_indication) se compose d'une marque de sous-type (subtype_mark), à laquelle est éventuellement adjointe une contrainte. Le modèle de types de CORBA ne permet pas d'exprimer les contraintes Ada sur les types. La traduction d'une subtype_indication est donc toujours celle de sa subtype_mark ; en contrepartie, l'agent d'interface vérifie, à l'exécution, que les contraintes sont satisfaites par les données envoyées par les clients CORBA, et lève une exception dans le cas contraire.

Une subtype_mark est un nom qui dénote un sous-type. Il s'agit donc soit d'un identifier ou d'un selected_component (qui se réduit en fait, ici, à une suite d'identifier séparés par des points, et dont chacun dénote un paquetage de bibliothèque), soit d'une attribute_reference (toute autre forme de subtype_mark ne peut dénoter un sous-type). Pour un identifier ou un selected_component, la simple traduction lexicale est utilisée (cf. section 3.2).

Syntaxe

prefix ::= name | implicit_dereference

attribute_reference ::= prefix'attribute_designator

attribute_designator ::= identifier[(*static*_expression)]
  | **Access** | **Delta** | **Digits**

Traduction

Les seuls attribute_designator normalisés pour lesquels une attribute_reference dénote un sous-type sont `Base` ([ISO, 1995, 3.5]) et `Class` ([ISO, 1995, 3.9]). Dans la mesure où nous ne tenons pas compte des contraintes exprimées sur les sous-types pour la traduction, un sous-type contraint d'un type est traduit comme équivalent à son type de base. Réciproquement, la traduction d'une attribute_reference de la forme S`'Base` est donc la traduction de son prefix, qui dénote un sous-type scalaire.

Une attribute_reference de la forme S'Class peut être appliquée à un sous-type *tagged* ou *tagged private*. Nous traduisons une telle attribute_reference comme son prefix. Une unité contenant une subtype_indication dont la traduction appellerait la traduction d'une autre attribute_reference (non normalisée) n'est pas traduisible.

3.6 Définitions communes

Tous les contrats IDL générés conformément à la présente traduction incluent un fichier *ciao.idl* dont le contenu est :

```
/* Common declarations for CIAO IDL contracts. */
/* $Id: //depot/ciao/main/doc/ciao.idl#9 $ */

#ifndef __CIAO_IDL
#define __CIAO_IDL

module CIAO {
  /* Any Ada exception is represented as an Ada_Exception.
     The Name and Information contain the Exception_Name and
     Exception_Information for the occurence. */

  exception Ada_Exception {
    string name;
    string information;
  };
};

#endif /* defined (__CIAO_IDL) */
```

FIG. 3.6 – Déclarations communes

3.7 Restrictions

Si un paquetage contient une renaming_declaration ou une generic_instanciation, sa traduction n'est pas permise.

Si la traduction d'un paquetage appelle la traduction d'une subtype_indication comportant une attribute_reference non décrite dans le présent document, alors ce paquetage n'est pas traduisible.

Chapitre 4

Implémentation

4.1 Présentation d'ASIS

La réalisation du traducteur nécessite de disposer, sous forme exploitable par un programme, de l'arbre sémantique d'une déclaration de paquetage Ada, en d'autres termes d'un mécanisme de *réflexivité* en Ada. À cet effet, nous avons utilisé ASIS.

ASIS [ISO, 1998] (*Ada Semantic Interface Specification*) est une API normalisée permettant à un outil de génie logiciel d'interagir avec un environnement de compilation Ada. En particulier, ASIS permet à une application de parcourir l'arbre syntaxique produit par un compilateur Ada, et d'obtenir des informations sémantiques sur cet arbre. L'utilisation d'ASIS nous permet de profiter des analyseurs syntaxiques déjà existants dans les compilateurs Ada, et nous évite de devoir développer notre propre analyseur (ce qu'on fait les développeurs d'ADEPT/JxA).

Une mise en œuvre d'ASIS (spécifique à un environnement de compilation particulier) offre des types de données permettant de représenter des éléments Ada, ainsi que des fonctions permettant de parcourir l'arbre syntaxique (requêtes syntaxiques). Un itérateur réalisant une traversée systématique en profondeur d'abord de l'arbre est fourni sous forme de la fonction générique *ASIS.Traverse_Element* ; le programmeur peut indiquer les opérations à effectuer sur chaque nœud au cours de la descente à l'aide de sous-programmes génériques formels.

ASIS permet également d'obtenir de nombreuses informations sémantiques sur un programme Ada : les requêtes sémantiques permettent d'obtenir les relations entre éléments Ada définies par les paragraphes *Static semantics* de la norme. Par exemple, considérant un élément qui est un nom d'usage d'une entité dans une expression, la requête *Corresponding_Name_Definition* fournit l'occurence définissante de ce nom d'usage et, partant, donne accès à la définition de l'entité et à l'ensemble de ses propriétés.

Pour la réalisation du traducteur de DSA vers IDL, nous avons utilisé *ASIS-for-GNAT*, l'implémentation d'ASIS développée par *Ada Core Technologies* pour l'environnement Ada du projet GNU. GNAT et ASIS-for-GNAT sont publiés sous forme de logiciel libre, et sont largement disponibles sur l'Internet.

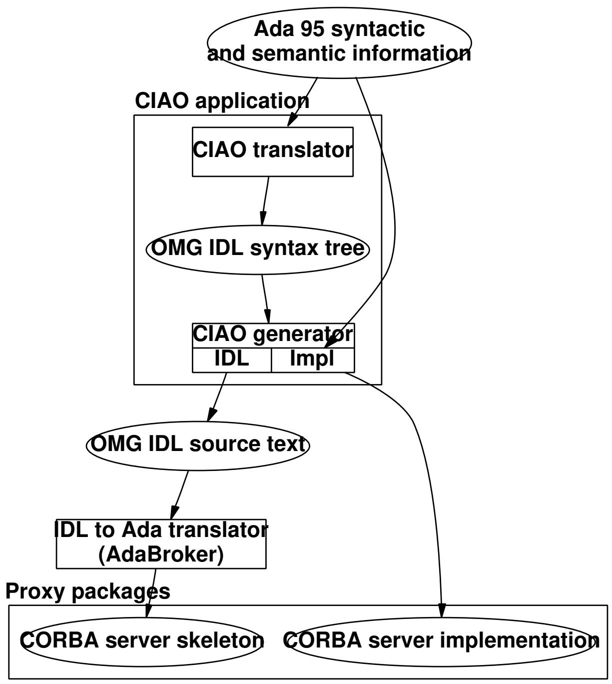

FIG. 4.1 – Flot des données dans le traducteur CIAO

4.2 Le traducteur

4.2.1 Principe de fonctionnement

La traduction d'une spécification de paquetage Ada vers IDL se déroule en deux phases. Dans un premier temps, une descente de l'arbre sémantique est effectuée à l'aide de l'itérateur standard *ASIS.Traverse_Element*. Cet itérateur maintient un « état » interne, que les fonctions d'exploration peuvent modifier. Dans le traducteur, cet état comprend notemment l'arbre syntaxique IDL en train d'être construit, ainsi qu'un pointeur de « position courante » dans cet arbre.

Chaque élément Ada est exploré, et l'élément IDL correspondant est construit. Il est ensuite greffé à la position courant dans l'arbre IDL. S'il doit être rempli par des sous-nœuds, il devient alors l'élément courant, et le parcours de l'arbre Ada se poursuit. Lorsque le traitement d'un élément Ada est terminé, ainsi que celui de tous ses éléments enfants, le pointeur de nœud courant est restauré à la valeur qu'il avait à l'entrée dans l'élément.

Nous construisons ainsi progressivement une image complète en mémoire de l'arbre IDL décrivant le service de l'annexe que l'on cherche à traduire. La seconde phase de la traduction consiste à parcourir cet arbre IDL, et à produire en sortie le texte source correspondant. L'arbre est donc simplement parcouru en profondeur d'abord, en émettant le texte représentant chaque nœud au moment où il est traversé.

Le flot des données dans le traducteur est résumé par la figure 4.1.

Le traducteur est découpé en plusieurs blocs fonctionnels, dont nous allons présenter les principaux. Leurs interactions sont résumées par la figure 4.2.

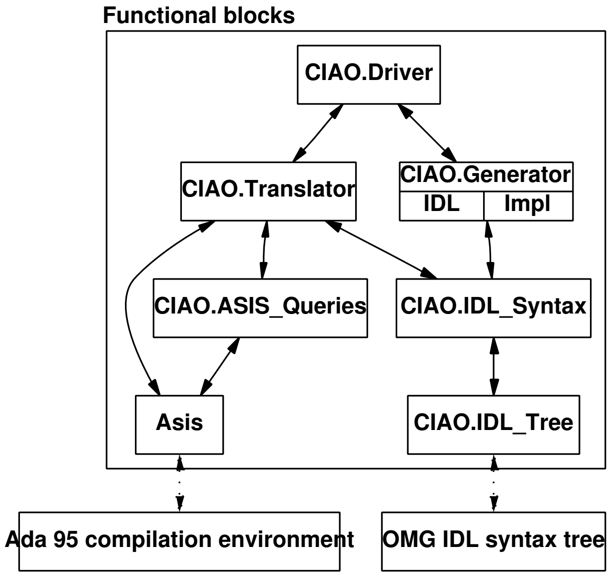

FIG. 4.2 – Découpage fonctionnel

4.2.2 Traduction et accès à l'arbre Ada

Driver constitue le programme principal. Il initialise l'environnement, puis déclenche successivement les différentes phases de la traduction. La première phase est la responsabilité du paquetage *Translator*. Celui-ci contient essentiellement une instanciation de l'itérateur générique d'ASIS. Il définit la fonction de traduction d'un élément Ada vers un élément IDL ; c'est ce paquetage qui implémente le schéma de traduction formel défini précédemment.

Translator utilise de nombreuses requêtes ASIS pour obtenir des informations relatives aux éléments à traduire. Pour certains éléments Ada, le parcours systématique des éléments enfants offert par *Traverse_Element* n'ext pas approprié. Par exemple, si l'on veut traduire une déclaration de paquetage, on ne veut pas faire apparaître dans la traduction la partie privée de cette déclaration, mais seulement la partie visible. Dans ce cas, nous utilisons les requêtes syntaxiques d'ASIS pour obtenir explicitement les éléments fils qui nous intéressent, au lieu de laisser la traversée standard se poursuivre.

Nous utilisons également des requêtes sémantiques, en particulier pour résoudre les noms d'entités. La bibliothèque ASIS dispose de l'arbre syntaxique décoré par le compilateur. Nous n'avons donc pas besoin d'implémenter nous-même les règles de résolution de noms de la norme Ada, et quand au cours de la traduction nous rencontrons le nom d'un type, nous pouvons obtenir immédiatement sa définition.

Nous avons également besoin de connaître certaines propriétés sémantiques de haut niveau du programme. Par exemple, pour traduire une déclaration de sous-programme dans un paquetage de catégorie *Remote Types*, nous avons besoin de savoir quels sont les paramètres formels contrôlants de son profil (i. e. les paramètres formels qui déterminent l'aiguillage de l'appel au moment de l'exécution). Pour cela, nous avons étendu l'ensemble des requêtes ASIS disponibles. Nous nouvelles requêtes (contenues dans le paquetage *ASIS_Queries* correspondent à des propriétés définies formellement par la norme Ada, et sont écrites pour être parfaitement indépendantes du projet CIAO.

4.2.3 Manipulation de l'arbre IDL

Si ASIS permet de manipuler aisément un arbre syntaxique Ada, nous n'avions pas à notre disposition ed bibliothèque permettant de créer facilement un arbre syntaxique IDL. Nous avons donc repris une partie du code du compilateur GNAT, gérant originellement des arbres Ada, et nous l'avons adaptée pour manipuler des arbres IDL. Il est remarquable de noter que nous n'avons effectué que des modifications mineures à ce code : une analyse soigneuse et une documentation abondante ont rendu très facile et profitable sa réutilisation. Ainsi, les paquetages *Nlists* et *Namet* sont intégrés pratiquement sans modifications, et les paquetages *IDL_Tree* et *IDL_Syntax* sont construits sur le modèle des paquetages correspondants de GNAT, *Atree* et *Sinfo*.

IDL_Tree décrit la structure de bas niveau d'un nœud IDL, et définit l'espace de stockage des nœuds. Cet espace est une table dynamique, dont la taille croît de façon contrôlée au fur et à mesure des besoins du programme. Chaque nœud est repéré par un numéro dans la table, de type `Node_Id`. La structure qui correspond au contenu d'un nœud contient notemment le type de nœud, et les identifiants de ses sous-éléments. Ceux-ci sont des champs numériques génériques pouvant recevoir indifféremment des identifiants de nœuds (`Node_Id`) ou des identifiants de listes gérées par le paquetage *Nlists* (de type `List_Id`). Cette structure est faiblement typée ; elle fournit des fonctions de bas niveau pour la manipulation d'arbres syntaxiques.

Le respect des contraintes syntaxiques imposées par la grammaire du langage IDL est assuré par un paquetage fournissant des opérations plus abstraites :le paquetage *IDL_Syntax*. Ce module définit des fonctions accesseurs permettant de manipuler les structures internes des nœuds. Ces fonctions ne permettent la mise en place d'un sous-élément d'un nœud que si elle n'entraîne pas de violation de la grammaire. Le typage de l'arbre est donc vérifié dynamiquement, à l'exécution, et de façon explicite. Cette structure flexible permet d'étendre aisément l'ensemble des constructions grammaticales représentable dans le traducteur.

Nous avons également défini des attributs sémantiques sur les nœuds IDL. Ces attributs peuvent être par exemple des identificateurs de nœuds permettant au traducteur de naviguer dans les structures sémantiques de l'arbre IDL, des indicateurs booléens repérant des nœuds ayant une propriété particulière. Chaque nœud a également un attribut sémantique appelé *Origin*, qui est l'identité de l'élément Ada qui lui a donné naissance. Cet attribut est essentiel pour la génération des relais, car il établit un « pont » entre le sens de la spécification IDL et celle du service DSA.

4.2.4 Génération de source IDL

Le texte source de la spéficiation IDL est produit par le paquetage *Generator.IDL*, qui prend en entrée l'arbre construit par le traducteur et génère le contrat IDL correspondant dans un fichier. La structure de l'information dont nous disposons, à savoir une représentation en mémoire de l'arbre abstrait IDL, est extrêmement proche de la forme du texte à produire. Le fonctionnement du générateur est donc trivial : l'abre IDL est parcouru au moyen des accesseurs syntaxiques du paquetage *IDL_Syntax*, et le texte correspondant est émis dans le fichier, conformément à la définition de la grammaire IDL.

Dans quelques rares cas cependant, la construction des sous-nœuds d'un nœud n'a pas été explicitement représentée par un accesseur, car cela alourdissait le code sans lui apporter plus de sûreté ou de clareté. Dans ces cas, la descente récursive de l'arbre s'effectue de manière non typée, en utilisant un accesseur de bas niveau.

4.2.5 Génération des paquetages relais

Le paquetage *Generator.Proxy*, a pour fonction la production automatique des paquetages d'interface nécessaires à l'interopération entre les clients CORBA et le serveur DSA. Pour chaque interface déclarée dans le contrat IDL issu du traducteur, une implémentation est produite. Cette implémentation est conforme à la projection normalisée d'IDL en Ada « côté serveur » ; elle est destinée à s'agencer avec les squelettes produits par un traducteur IDL à partir du texte source IDL généré.

Le traitement réalisé par les implémentations de méthodes CORBA que nous générons se déroule en trois temps. En premier lieu, les arguments sont convertis des types issus de la projection normalisée de CORBA sur Ada 95 en les types natifs d'Ada, qui sont utilisés par l'annexe. La conversion des types de base est assurée par des fonctions de la bibliothèque d'exécution de CIAO. Pour les types composites, nous générons un paquetage de conversion par module, qui effectue la traduction des types CORBA projetés en types natifs. Un appel DSA est ensuite émis. Enfin, une éventuelle valeur de retour est retraduite en type CORBA et retournée à l'appelant.

La génération de code Ada nécessite une information précise sur le système Ada dans lequel ce code doit s'intégrer. La seule information disponible dans l'arbre IDL lui-même n'est donc pas suffisante ; c'est ici que sont utilisés les pointeurs vers les éléments Ada d'origine. La génération du code est donc dirigée dans sa structure générale par l'arbre IDL, mais contrôlée dans ses détails fins par des informations sur la sémantique Ada obtenues à travers ASIS.

4.3 Le relais généré et le fonctionnement global de l'application

Le traducteur CIAO génère un arbre abstrait IDL, à partir duquel le générateur de relais produit des paquetages convertissant les types de données CORBA projetés en types natifs Ada, ainsi que des paquetages d'implémentation d'interfaces CORBA. Pour obtenir une application DSA interopérable avec CORBA, il reste à agencer ces paquetages et à les enregistrer auprès d'un ORB, afin de constituer une *partition relais*, c'est-à-dire une partition au sens de l'annexe qui soit également un serveur incarnant des objets CORBA.

Chaque paquetage-relais correspondant à un paquetage de l'annexe comporte une procédure d'initialisation du service CORBA. La procédure principale de la partition relais commence par créer et initialiser un ORB, puis elle appelle la procédure d'initialisation des différents relais.

Pour un paquetage RCI, cette procédure instancie l'objet d'implémentation et demande à l'ORB d'activer cette instance et de lui attribuer un nom CORBA (une

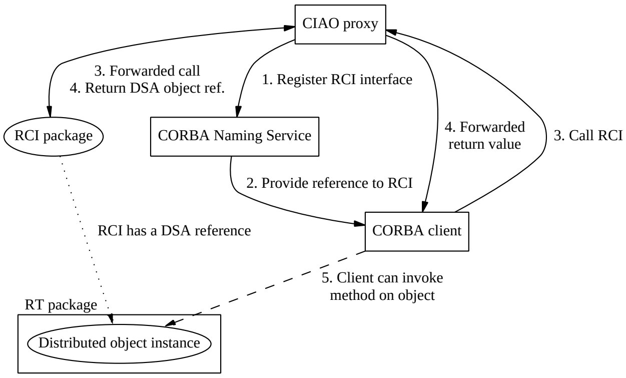

FIG. 4.3 – Initialisation d'une application CIAO

IOR, *Interoperable Object Reference*). Elle associe ensuite cette référence avec un nom sympathique pour l'utilisateur au moyen du service de nommage CORBA. Tout client CORBA peut ainsi obtenir une référence d'objet correspondant à l'interface qui traduit un paquetage RCI.

De telles références sont essentielles pour l'initialisation de l'application. En effet, dans un système réparti, chaque participant doit utiliser un service du système pour prendre connaissance d'une « première » référence d'objet, qui lui permette ensuite d'interroger un service de nommage, un annuaire ou autre service fournisseur de références d'objets. Dans l'annexe, ce sont les paquetages RCI qui jouent ce rôle, car un paquetage RCI est instancié exactement une fois, sur une partition bien déterminée, dans une application Ada répartie ; à l'exécution, le système de communication inter-partitions (PCS) sait les localiser. Ils peuvent donc être utiliser pour fournir aux autres partitions le moyen d'enregistrer, de publier et d'acquérir des références d'objets. Ce mécanisme d'initialisation est synthétisé par la figure 4.3.

Dans une application CIAO, un client CORBA peut ainsi interroger le service de nommage CORBA pour obtenir une référence correspondant à un paquetage RCI, puis effectuer des appels sur ce paquetage pour obtenir des références d'objets répartis.

Pour un paquetage définissant un type objet réparti Ada, le paquetage relais contient, là encore, une implémentation d'objet CORBA. Son activation utilise une fonctionnalité évoluée de CORBA, le POA (*Portable Object Adapter*). Le POA est un service normalisé fourni par un ORB, et permettant au développeur de serveurs CORBA de définir les politiques liant les références d'objets aux entités physiques qui les implémentent (nommées *servants*), et régissant la création, l'activation, la désactivation et la destruction de ces entités. Le modèle d'activation de CORBA 2.0 (le BOA, *Basic Object Adapter*, était beaucoup plus simple : à une référence d'objet correspondait une, et une seule, entité d'implémentation enregistrée dans l'ORB).

En l'occurence, nous créons un seul *servant* pour tous les objets répartis Ada d'un même type. Nous enregistrons ce *servant* dans un POA créé spécialement, mettant en œuvre la politique USE_DEFAULT_SERVANT. Ainsi, tout appel de méthode sur un objet de cette classe sera traité comme un appel sur une méthode du *servant*. Lors de la conversion des arguments CORBA vers les types Ada correspondants, la bibliothèque d'exécution de CIAO extrait un identificateur d'objet. La spécification du POA permet d'utiliser toute séquence d'octets comme identificateur. Nous y indiquons donc tout simplement la référence d'objet de l'annexe, sous sa forme emballée par la bibliothèque d'exécution de Glade. Cela permet de réaliser un relais sans état, que l'on peut redémarrer s'il se termine anormalement ; par ailleurs, nous n'introduisons pas de fuite de mémoire sur la partition relais.

4.4 Exemple de code produit

Nous donnons ci-après un exemple de paquetage DSA, et le source IDL généré corresponant.

```
package Gizmos is
   pragma Remote_Types;

   type Root_Gizmo is
     abstract tagged
     limited private;

   subtype Celsius is Float
     range -273.15 .. Float'Last;
   type Price is digits 5;

   type Gizmo_Params is record
      Temperature : Celsius;
      Cost        : Price;
   end record;

   procedure Heat
    (Self: access Root_Gizmo;
     How_Much : in Celsius);
   function Get_Params
    (G: in Root_Gizmo)
     return Gizmo_Params;

   type Electric_Gizmo is
     new Root_Gizmo with private;
   function Is_Plugged_In
    (Self: access Electric_Gizmo)
     return Boolean;

private
   --  Private part omitted.
end Gizmos;
```

(a) Spécification Ada

```
#include "ciao.idl"

module CORBA__Gizmos {
  interface Root_Gizmo;

  typedef double Celsius;

  typedef double Price;

  typedef struct
   Gizmo_Params_struct
  {
    Celsius Temperature;
    Price Cost;
  } Gizmo_Params;

  interface Electric_Gizmo;

  interface Root_Gizmo
  {
    void Heat
      (in Celsius How_Much);
    Gizmo_Params
      Get_Params ();
  };

  interface Electric_Gizmo
    : Root_Gizmo
  {
    boolean Is_Plugged_In ();
  };
};
```

(b) Contrat IDL

FIG. 4.4 – Exemple de source IDL généré

Chapitre 5

Perspectives

Ce document présente un schéma de traduction des spécifications de services de l'annexe des systèmes répartis d'Ada 95 vers OMG IDL, le langage de description de CORBA. La construction de ce schéma formel a été la première phase d'un projet en cours visant à offrir aux clients CORBA un accès aux services fournis par l'annexe.

Nous présentons également l'implémentation d'un traducteur conforme à ce schéma. Ce logiciel est basé sur ASIS, une API normalisée pour la réalisation d'outils « réflexifs » en Ada. Ce traducteur génère un contrat IDL compilable et lisible par un programmeur, correspondant aux services définis par un paquetage de l'annexe, répondant ainsi aux objectifs de notre mémoire.

Il reste donc à générer automatiquement une implémentation de ce contrat IDL, qui traitera les requêtes CORBA reçues en effectuant les appels correspondants sur les objets de l'annexe, après conversion des arguments et vérification des contraintes de typage. Cette implémentation s'intégrera dans une partition (au sens DSA) jouant le rôle d'interface, de relais entre CORBA et l'annexe, de la même façon que dans ADEPT/JxA les JxStubs (différents pour chaque objet Ada) s'insèrent dans JxAgent. Nous réaliserons cette troisième étape du projet CIAO dans le cadre d'un stage de DEA au département Informatique & Réseaux de l'ENST. Par ailleurs, nous entendons poursuivre l'exploration des moyens permettant l'interopération des systèmes d'objets répartis à travers une thèse de doctorat à l'ENST.

Chapitre 6

Bilan & conclusion

Ce mémoire a établi un schéma de traduction des spécifications de services de l'annexe des systèmes répartis d'Ada 95 vers OMG IDL. À partir de ce schéma ont été développés un traducteur automatisé et un générateur d'implémentations CORBA, en utilisant l'API ASIS pour parcourir des arbres sémantiques Ada. Ces logiciels ont été testé sur un exemple simple de système réparti Ada : le TP de systèmes répartis de la Dominante Informatique de l'ENST.

Outre la réalisation de l'ensemble des fonctionnalités prévues initialement dans le projet CIAO, ce mémoire m'a permis de compléter la mise au point du modèle de traduction. En effet, certaines traductions se sont révélées problématiques, et ont dû être amendées au cours de l'implémentation du traducteur et du générateur de spécifications IDL. Les traductions n'ont pas été modifiées pendant l'écriture du générateur de relais ; le traducteur a seulement été enrichi pour compléter l'arbre IDL avec des informations sémantiques, essentiellement des pointeurs indiquant l'élément Ada à l'origine des nœuds IDL. L'ajout de ces informations sémantiques a été très simple, grâce à l'architecture générique adoptée dès l'origine pour le stockage de l'arbre IDL.

Ce mémoire a également été l'occasion de soumettre le compilateur GNAT et les outils CORBA AdaBroker et Broca à une torture intensive, qui a permis la mise en évidence et la correction de plusieurs problèmes dans ces logiciels.

Bibliographie

[Bourg, 1999] Anne-Laure BOURG. Implémentation d'un compilateur de Java vers IDL. Technical Report, ENST Paris, July 1999.

[Gargaro et al., 1997a] Anthony GARGARO, Gary SMITH, Ronald THERIAULT, Richard VOLZ, and Raymond WALDROP. Ada-Java Communication in ADEPT. In *Proceedings of Tri-Ada'97*. Saint-Louis, Missouri, USA, 1997a.

[Gargaro et al., 1997b] Anthony GARGARO, Gary SMITH, Ronald THERIAULT, Richard VOLZ, and Raymond WALDROP. Interoperation of Ada and Java using ADEPT/JxA. Technical Report, Texas A&M University, College Station, Texas, USA, April 1997b.

[Gargaro et al., 1997c] Anthony GARGARO, Gary SMITH, Ronald THERIAULT, Richard VOLZ, and Raymond WALDROP. Towards Interoperation of Ada and Java Using ADEPT (Advanced Distributed Execution Partioning Toolset). In *Ninth Annual Software Technology Conference*. Salt Lake City, Utah, USA, April 1997c.

[ISO, 1995] ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC/ANSI 8652 :1995.

[ISO, 1998] ISO. *Information Technology – Programming Languages – Ada Semantic Interface Specification (ASIS)*. ISO, 1998.

[Kermarrec et al., 1995] Yvon KERMARREC, Laurent PAUTET, Gary SMITH, Samuel TARDIEU, Ron THERIAULT, and Richard VOLZ. Ada 95 Distribution Annex Implementation for GNAT. Technical Report, Texas A&M University, College Station, Texas, USA, April 1995. (Contract with a grant from Computer Sciences Corp.).

[Object Management Group, 1998] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker : Architecture and Specification, revision 2.2*. February 1998. OMG Technical Document formal/98-07-01.

[Object Management Group, 1999] OBJECT MANAGEMENT GROUP. *Java Language to IDL Mapping*. Object Management Group, March 1999. OMG ptc/99-03-09.

[Pautet et al., 1999] Laurent PAUTET, Thomas QUINOT, and Samuel TARDIEU. CORBA & DSA : Divorce or Marriage? In *Proceedings of AdaEurope'99*. Santander, Spain, June 1999.

[Schonberg and Banner, 1994] Edmond SCHONBERG and Bernard BANNER. The GNAT project : A GNU-Ada 9X compiler. In *Proceedings of Tri-Ada'94*. Baltimore, Maryland, USA, 1994.

Annexe
CORBA & DSA : Divorce or Marriage ?

Cet article offre une comparaison entre CORBA et l'annexe des systèmes répartis d'Ada 95. Il a été présenté au cours de la conférence *Reliable Software Technologies – AdaEurope '99*.