CORBA and CORBA Services for DSA

Laurent PAUTET, Thomas QUINOT, Samuel TARDIEU and the AdaBroker team

{pautet,quinot,sam}@inf.enst.fr
adabroker-devel@adabroker.eu.org

École nationale supérieure des télécommunications
46, rue Barrault
F-75634 Paris CEDEX 13, France

Abstract

Comparing CORBA and the Ada 95 Distributed Systems Annex shows that an advantage of CORBA is its Common Object Services, providing standard, frequently-used components for distributed application development. This paper presents our implementation of similar services for the DSA. We also introduce new developments of our team that aim at providing close interaction between CORBA and Ada applications. Part of the work presented here was accomplished by the AdaBroker team: Fabien Azavant, Emmanuel Chavane, Jean-Marie Cottin, Tristan Gingold, Laurent Kübler, Vincent Niebel, and Sébastien Ponce.

1 Introduction

A software developer who wants to create a distributed heterogeneous, possibly multi-language application faces a difficult choice. Several object models and protocol suites are available for this purpose, each with its own advantages and particular features; they are not currently interoperable. In this paper, we focus more specifically on two particular architectures: OMG CORBA and the Ada 95 Distributed Systems Annex (DSA).

CORBA [9] is sponsored by the Open Management Group, a consortium of software vendors that seek to promote industrial standards for the development of distributed heterogeneous applications. It is based on OMG IDL, an interface description language whose syntax is close to C++. The object model is close to Java, only allowing the definition of distributed objects. The CORBA standards also define mappings of IDL into host languages such as C++, Java, and Ada 95. Client stubs and server skeletons in host language are automatically generated by an IDL compiler; they interface with a communication subsystem, the Object Request Broker (ORB), through a vendor-specific API. An ORB uses a set of standard protocols to communicate with its peers. CORBA thus permits interoperation of clients and servers that are independently coded in different languages, and using products from different vendors.

The Ada 95 Distributed Systems Annex is part of the Ada 95 ISO standard [5]. It aims at providing a framework for programming distributed systems within the Ada language, while preserving strong typing properties. Its distributed application model is more general, as it can not only include distributed objects, but also remote subprograms (providing a classical remote procedure call facility) and references to remote subprograms. It also allows the definition of a shared data space, through the abstraction of Shared Passive packages. In the case of DSA, the IDL is not a separate language (as in CORBA), but the host language itself. This affords developers an integrated approach for application development and test: going from a non-distributed application, which is easy to test and debug, to a full distributed system only requires the addition of one categorization pragma to each package that defines remote objects or subprograms. The Remote Call Interface (RCI) categorization pragma makes the subprograms of a package available for remote procedure calls, while the Remote Types (RT) pragma allows access-to-class-wide types declared in the package to designate remote objects. Such access types are then called RACWs (Remote Access to Class-Wide).

We have developed an implementation of the Distributed Systems Annex for the GNAT compiler [7]; the details of our comparison of DSA and CORBA features can be found in [10]. In this paper we first discuss the implementation of common services for the DSA. Theses services provide functionalities that are frequently required in distributed applications, and are potentially useful to all DSA developers. We then describe our implementation of an Ada IDL precompiler and ORB binding. These are the necessary tools for creating Ada software that will interoperate with CORBA clients and servers. We finally present a new project of our team: an automated tool to make the functionality of a DSA server available to CORBA clients. This allows a server implementor to only code a DSA server, while being able to provide the same service to clients from the DSA and CORBA worlds.

2 CORBA common services for DSA

The CORBA ORB provides a core set of basic communication services. All other distributed services that an application may use are provided by objects described by IDL contracts. The OMG has standardized a set of useful services like Naming, Concurrency, Events, Trading, Licensing, Object Life Cycle, etc. A CORBA vendor is free to provide an implementation of these services. It is unfortunate that the DSA currently does not provide such a set of commonly-used services. We have consequently started to design and implement a set of DSA services that provide similar func-

SIGAda'99 10/99 Redondo Beach, CA, USA

tionalities for DSA application developers. In this section, we also describe our current development of a dynamic invocation facility for DSA, which is quite similar to CORBA's Dynamic Invocation Interface (DII).

2.1 The Naming service

It is impractical for users of distributed applications to deal directly with machine representations of object references, because these are machine-oriented identifiers that designate a particular object instance at a particular physical location, without any consideration for the user-defined semantics of the object. The Naming service allows the association (*binding*) of an object reference with user-friendly names. A name binding is always defined relative to a *naming context* wherein it is unique.

A naming context is an object itself, and so can be bound to a name in another naming context. One thus creates a *naming graph*, a directed graph with naming contexts as vertices and names as edge labels. Given a context in a naming graph, a sequence of names can thus reference an object. This is very similar to the naming hierarchies that exist in the Domain Name System and the UNIX file system.

A typical usage scenario consists in obtaining a well-known remote reference that designates the naming context corresponding to the "root" of a naming hierarchy, and then executing recursive naming operations on this hierarchy. The Trading Service provides a higher level of abstraction than the Naming Service: if the Naming Service can be compared to the White Pages, the Trading Service can be compared to the Yellow Pages, allowing a user to query objects by their properties rather than by their name.

The CORBA naming service is defined as IDL module CosNaming (see code sample 1). This module defines two data types: *name component*, and *name*, which is a sequence of name components. This module also supplies two interfaces: *naming context* and *binding iterator*. The Naming Context interface provides the necessary operations to bind a name to an object, and to resolve (look up) a name in order to obtain the associated object reference. The Binding Iterator interface is used to walk through a collection of names within a context; such a collection is returned by the *list* operation of the Naming Context interface.

Translating the CosNaming service definition to Ada using the standard mapping is not trivial; CosNaming makes use of three OMG IDL features that are not easily represented in Ada: sequences, exceptions with members, and forward interface declarations. Excerpts of the generated code are given in samples 2 and 3.

We have implemented a similar service with native Ada 95 distributed objects. We were thus able to take advantage of standard language features; this yields a simple specification, which is far easier to understand and use than the CORBA one (see samples 4 and 5).

```
module CosNaming {
  typedef string Istring;
  struct NameComponent {
    Istring id;
    Istring kind;
  };
  typedef sequence <NameComponent> Name;
  enum BindingType {nobject, ncontext};
  struct Binding {
    Name binding_name;
    BindingType binding_type;
  };
  typedef sequence <Binding> BindingList;

  interface BindingIterator;

  interface NamingContext {
      exception CannotProceed {
          NamingContext cxt;
          Name rest_of_name;
        };
      void bind (in Name n, in Object obj)
        raises (CannotProceed);
      void list
        (in unsigned long how_many,
         out BindingList bl,
         out BindingIterator bi);
      // Other declarations not shown
    };

  interface BindingIterator {
      boolean next_n
        (in unsigned long how_many,
         out BindingList bl);
      // Other declarations not shown
    };
};
```

Sample 1: CosNaming IDL

2.2 The Events service

The Events service provides a way for servers and clients to interact through asynchronous events between anonymous objects. A *supplier* produces events, while a *consumer* receives event notifications and data. An *event channel* is the mediator between consumers and suppliers. *Consumer admins* and *supplier admins* are in charge of providing *proxies* to allow consumers and suppliers to get access to the event channel (dashed arrows in figure 1). For instance, a pull supplier will query his supplier admin in order to obtain a proxy pull consumer. Suppliers and consumers produce and receive events through their associated proxies (see plain arrows in figure 1). From the event channel point of view, a *proxy supplier* (or *proxy consumer*) is seen as a consumer (or a supplier). Therefore, a proxy supplier (or proxy consumer) is an extended interface of consumer (or supplier). The Events service defines *push* and *pull* methods to exchange events. Four models of events and data exchange can thus be defined.

We have developed an Events service for the DSA. During the implementation of the service, we realized that although the service is nicely specified by an IDL file, most of its semantics are quite vague; the behaviour of some methods is left up to the vendor in such a way portability is seriously compromised. Other CORBA services also suffer similar vagueness in definition. For this reason, we decided to implement only the Naming and Events services as defined by OMG, and to implement other services directly as Ada units with well-specified semantics (see 2.3).

Note that `Proxy_Push_Consumer` defined in `Event_Channel_Admin` inherits from `Push_Consumer` defined in `Event_Communication` (sample 6). The OMG has extended this service to provide typed data operations. An Ada 95 programmer would easily adapt our implementation by using stream operations to get this new service.

2.3 The Mutex service

CORBA defines a Concurrency service that basically offers a complete locking system to serialize concurrent access to a resource. Extended features such as "intent to lock" are

```
with Corba.Object, Corba.Sequences,
   Corba.Forward;
package CosNaming is

   type Istring is new Corba.String;
   type NameComponent is
   record
      id : Istring;
      kind : Istring;
   end record;

   -- Example of a sequences mapping
   package NameComponent_Unbounded is
      new Corba.Sequences.Unbounded
        (NameComponent);
   type Name is
      new NameComponent_Unbounded.Sequence;

   package BindingIterator_Forward is
      new Corba.Forward;
end CosNaming;
```

Sample 2: CosNaming

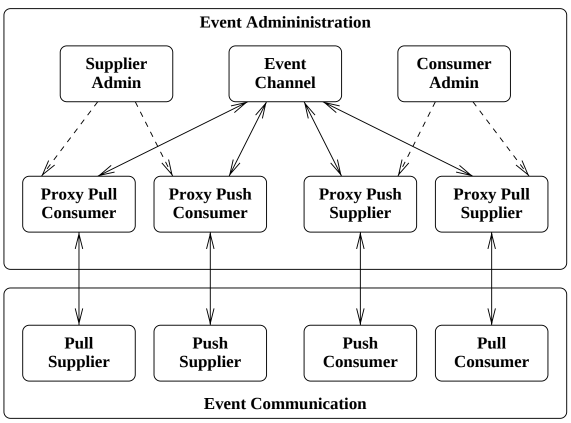

Figure 1: Structure of the Events service IDLs

also defined by this service.

We have chosen to implement basic locking services using a more decentralized approach. Our service is based on a distributed mutual exclusion algorithm described by Li and Hudak in [8], which avoids using a central lock manager. It has been described in [11], while a previous prototype implementation done by ENST students can be found in [2].

2.4 Dynamic invocation in DSA

2.4.1 Introducing the Dynamic Invocation service

In CORBA, the Interface Repository (IR) and Dynamic Invocation Interface (DII) mechanisms allow clients to dynamically discover and invoke services.

The Interface Repository is a database maintained by server ORBs that stores information describing the services that are available in the system (e. g. the list of operations for a given distributed object type, with their names and parameter profiles). It is accessible for all nodes that exist in the distributed application. Clients can query the IR to retrieve the methods associated with an object reference, and the signature of one such method at run time, and then invoke that method, even though its specification was unknown to the client at compile time. The Dynamic Invocation Interface is the API that allows the construction

```
with Corba.Object, Ada.Exceptions;
use CosNaming, Ada.Exceptions;
package CosNaming.NamingContext is

   type Ref is new Corba.Object.Ref with
      null record;
   function To_NamingContext (
      Self: in Corba.Object.Ref'class)
      return Ref'class;

   -- An IDL exception is mapped to an
   -- Ada exception plus a tagged record.

   CannotProceed : exception;
   type CannotProceed_Members is
      new Corba.Idl_Exception_Members with
   record
      ctx : NamingContext;
      rest_of_name : Name;
   end record;

   function Get_Members (
      X: in Exception_Occurrence)
      return CannotProceed_Members ;

   procedure bind (Self: in Ref;
      N: in Name;
      Obj: in Corba.Object.Ref);

   -- Forward reference to BindingIterator.
   procedure list
      (Self : in Ref;
       how_many: in Corba.Unsigned_Long;
       bl: out BindingList;
       bi: out BindingIterator_Forward.Ref);

   -- [some declarations are missing]
end CosNaming.NamingContext;
```

Sample 3: CosNaming.NamingContext

of a method call from the description returned by the IR and client-provided actual parameters.

The DSA does not define a similar facility. However, such a service can easily be provided, and we seek to implement it. In the following two sections, we describe the specification of this future facility.

2.4.2 Implementation of the Interface Repository

In our DSA Dynamic Invocation facility, an RCI package will act as a DSA interface repository; ASIS tools will be used to obtain the necessary interface information from an Ada compilation environment and make it available to the interface repository, and utility packages will be created that provide a dynamic request construction facility.

ASIS [6] is an open, published, vendor-independent API for interaction between CASE tools and an Ada compilation environment. It defines the operations needed by such tools to extract information about compiled Ada code from the compilation environment. The ASIS interface allows the tool developer to take advantage of the parsing facility built in the compiler; it provides an easy access to the syntax tree and associated semantic information built by the compiler from a compilation unit.

ASIS standardizes a set of *queries* that allow an Ada program to manipulate the syntactic information corresponding to another Ada program: for a given Ada element, it gives access to its children element; a systematic recursive traversal iterator is provided, as well as queries that allow the user to explicitly obtain specific children elements of an element. These are the ASIS *syntactic queries*. A set of *semantic*

```
package GLADE.Naming is
   pragma Remote_Types;

   type Istring is private;
   function  Get_Istring
     (I : in Istring) return String;
   procedure Set_Istring
     (I : in out Istring; S : in String);

   type Name_Component is record
      Id, Kind : Istring;
   end record;
   type Name_Component_Sequence is
      array (Natural range <>)
     of Name_Component;
   type Name is private;
   -- [some declarations are missing]
private
   -- [some declarations are missing]
end GLADE.Naming;
```

Sample 4: GLADE.Naming

```
with GLADE.Objects; use GLADE.Objects;
with GLADE.Naming; use GLADE.Naming;
package GLADE.Naming.Interface is
   pragma Remote_Types;

   type Binding_Iterator is
      tagged limited private;
   type Binding_Iterator_Ref is
      access all Binding_Iterator'Class;

   type Naming_Context is
     new Objects.Object with private;
   type Naming_Context_Ref is
      access all Naming_Context'Class;

   procedure Bind
     (Ctx : access Naming_Context;
      N   : in Name;
      Obj : in GLADE.Objects.Object_Ref);
   procedure List
     (Ctx      : access Naming_Context;
      How_Many : in  Natural;
      BL       : out Binding_List;
      BI       : out Binding_Iterator_Ref);
   -- [some declarations are missing]
private
   -- [some declarations are missing]
end GLADE.Naming.Interface;
```

Sample 5: GLADE.Naming.NamingContexts

queries is also defined. These functions provide information about the semantic relathionships between elements. For example, from an element that is a usage name for an entity, they can provide the definition of that entity. We thus can view ASIS as a *reflexivity* interface for Ada.

The interface repository can be implemented as a straightforward DSA server that offers two sets of operations. For DSA service providers (other RCI or Remote Types packages), it shall provide a means to register an interface, comprising a set of primitive operation names and their signatures. This can be achieved by submitting ASIS tree data for the package declaration to the Interface repository. For clients, it shall offer a means of retrieving the description of the operations of a distributed object, given a reference to this object. In other words, it shall provide a means to perform queries on the ASIS data.

The ASIS standard specifies a set of services that may be provided in a client-server implementation. These services closely reflect the low-level queries provided by the standard. We consider that a DSA Interface repository should provide queries that fit smoothly in the model of existing queries, while providing higher-level semantic information, as required by clients' needs. For example, a query that lists all visible primitive operations of a distributed object type would be very useful to DII clients. Consequently, we see the DSA IR essentially as an ASIS server with extended functionalities, as required for the purposes of dynamic invocation.

```
with Ada.Streams; use Ada.Streams;
package GLADE.Event_Communication.Interface is
   pragma Remote_Types;

   type Push_Consumer is
     abstract tagged limited private;
   type Any_Push_Consumer is
      access all Push_Consumer'Class;

   procedure Disconnect
     (Consumer : access Push_Consumer)
      is abstract;
   procedure Push
     (Consumer : access Push_Consumer;
      Event    : in Stream_Element_Array)
      is abstract;
   -- [some declarations are missing]
private
   -- [some declarations are missing]
end GLADE.Event_Communication.Interface;
with GLADE.Event_Communication.Interface;
use  GLADE.Event_Communication.Interface;
package GLADE.Event_Channel_Admin.Interface is
   pragma Remote_Types;

   type Proxy_Push_Consumer is
     abstract new Push_Consumer
     with private;
   type Any_Proxy_Push_Consumer is
      access all Proxy_Push_Consumer'Class;

   procedure Connect
     (Consumer: access Proxy_Push_Consumer;
      Supplier: in Any_Push_Supplier)
     is abstract;
   -- [some declarations are missing]
private
   -- [some declarations are missing]
end GLADE.Event_Channel_Admin.Interface;
```

Sample 6: GLADE Event Interfaces

As soon as it is registered, a service is known to the repository and visible by clients. In the case of the registration of a distributed object type, for example, any client that obtains an access value designating an object of this type can retrieve the description of its operations, even though it knew nothing of them at compile time, and does not semantically depend on the server specification.

2.4.3 Implementation of the request construction library

The DII client will then use a utility function that constructs a request message from an interface description retrieved from the repository, and actual parameters provided by the client. This message will be sent to the server through the Partition Communication Subsystem (PCS), just like a normal remote call generated by the compiler in a "static" service invocation: the client will call a wrapper routine that will build a proper request description and ultimately call one of the standard invocation subprograms of the PCS, `Do_RPC` and `Do_APC`.

Apart from calls to the service registration functions,

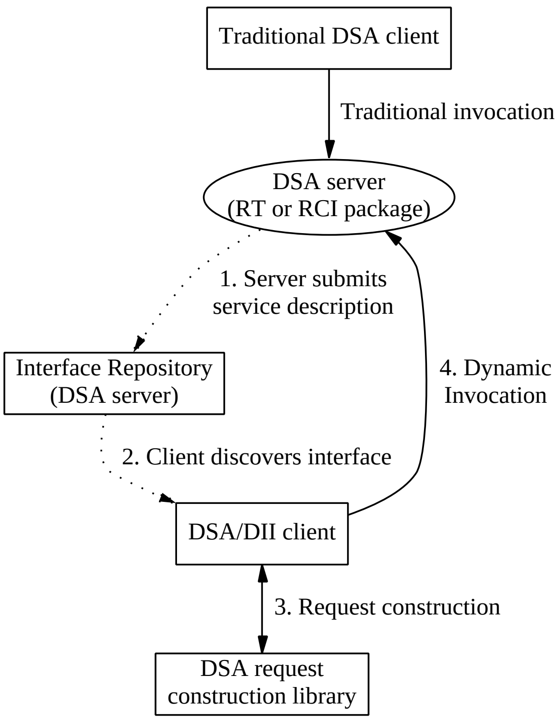

Figure 2: Dynamic Invocation for DSA

no Interface Repository or DII-specific code is required on the server side; it should be noted in particular that, from the server point of view, a dynamically constructed request is treated exactly in the same way as a traditional, static request. The dynamic interface discovery and invocation mechanisms are confined in the DSA interface repository and the client request construction library.

The system outlined above is going to be implemented by our team in the next few months; all DSA users will thus gain the same flexibility with dynamic invocation that is currently offered to CORBA programmers by the most advanced ORBs, which implement the CORBA Interface Repository.

3 Free CORBA ORBs for Ada

3.1 An omniORB-based Ada ORB

The CORBA standard specifies a mapping of IDL to Ada. Using an IDL to Ada precompiler and the corresponding ORB, it should be possible to implement CORBA clients and servers in Ada. Unfortunately, we do not know of any existing free, open-source implementation of such tools. We feel that this situation makes it impractical to evaluate CORBA with Ada during a project's prototyping phase, to integrate them in a critical application where source code availability is required, or to use them in an educational context.

However, free C and C++ ORBs with C and C++ IDL precompilers are readily available. We have therefore decided to develop an Ada binding for a free ORB's internal API, and to implement our own IDL precompiler targeted at Ada 95 using this API. This constitutes the *AdaBroker* project [1]. We selected the C++ ORB omniORB[1] for this project. This ORB, available from AT&T Laboratories Cambridge (formerly ORL) under the GNU General Public License, provides a fairly complete implementation of the CORBA standards and of the C++ language mapping, and has proven extremely performant, particularly under Linux. omniORB's IDL to C++ precompiler is based on the free Sun IDL front-end. We have developed a new back-end targeted at Ada 95, and integrated it in omniORB's precompiler. Our tool complies with the OMG standard Ada language mapping, and generates client stubs and implementation skeletons in Ada.

We also have implemented Ada packages that provide a complete binding to the transport facilities of omniORB. The C++ omniORB library provides two classes that correspond to different views of CORBA objects: `Object`, which is the ancestor class for all server implementations, and `omniObject`, which embodies the network resources associated with an object. Our Ada binding encapsulates `omniObject`, and reimplements a native `Object` class entirely in Ada, thus allowing us to have a clean, well-defined interface between the generated code and the underlying ORB functionality, and to limit the scope of our dependance on a specific ORB implementation.

Starting from this binding, we also developed a complete DII (Dynamic Invocation Interface) implementation, compliant with the CORBA 2.0 specification [3]. It should be noted that the DII implementation does not need to bind directly to any C++ code; it only uses the services provided by the Ada wrappers for `omniObject`.

We have thus effectively provided a free, open-source implementation of an IDL to Ada precompiler, and of matching ORB libraries. Our work is based on omniORB, and will be freely available and redistributable.

3.2 A novel Ada ORB

AdaBroker still lacks some functionalities. Most notably, as omniORB provides no Interface Repository, nor does AdaBroker. It also lacks a POA (Portable Object Adapter). The Object Adapter is the part of the ORB responsible for the creation, activation and destruction of object implementations. The original Object Adapter is the BOA (Basic Object Adapter); the POA provides a more flexible interface to implementation management. For example, it allows server implementors to only register one implementation (a *servant*) for a whole set of objects sharing the same interface. Unfortunately, omniORB currently only provides BOA.

For these reasons, we have decided to implement our own, full Ada ORB, *Abroc* [4]. Abroc already has an IIOP stack, a POA, and an IDL translator. All the system has been successfully tested on simple client and server examples. We plan to continue this project and to extend it into a full-featured CORBA library and toolkit for Ada 95.

4 A CORBA interface for DSA services

4.1 Objective

Services implemented as RT or RCI packages can currently be invoked only from other Ada 95 code using the DSA mechanisms: remote procedure calls and distributed objects. This may be considered a drawback by software component developers when they consider using the DSA to implement distributed services, because this limits the scope of their products to Ada 95 application developers. In order to promote the use of Ada 95 as a competitive platform for

[1]For detailed information about omniORB, see `http://www.uk.research.att.com/omniORB/omniORB.html`

the creation of distributed services, we aim at providing a means for CORBA applications to become clients of DSA services.

This first requires a tool to produce to map the description of a service implemented in DSA (i. e. the declaration of a DSA package) into a CORBA service description, that is an OMG IDL specification. This specification shall be used by CORBA client developers to generate stubs for calling the services exported by the DSA package.

An adaptation layer is also necessary, which will receive the CORBA requests from CORBA clients, map them onto Ada remote invocations, and perform the corresponding DSA calls. This code shall be instantiated in one or more "proxy" partitions in the distributed application containing the DSA packages, so they can receive invocation requests from the CORBA software bus.

We are currently implementing these two software components; the following two subsections discuss their design and internals.

4.2 From DSA specification to IDL file

In order to generate an IDL interface specification from a DSA package declaration, we have first produced a complete, formal specification of a mapping of DSA services (described as Pure, Remote Call Interface or Remote Types package specifications) into OMG IDL [12]. This specification precisely defines a translation of all legal DSA syntax trees into IDL syntax trees.

We have then created an automated translation tool that implements this mapping: the *CIAO*[2] Translator [13]. This tool uses ASIS to extract syntactic and semantic information from the GNAT Ada 95 compilation environment. Although GNAT is the primary implementation platform, we believe that the program is easily portable to any other compilation environment that supports ASIS. It does not depend on compiler internals, but only on some utility libraries that come with GNAT.

The operation of the translator has two main phases. The Ada semantic tree is first recursively traversed using the standard ASIS iterator, *ASIS.Traverse_Element*. This is a generic iterator that provides a framework for a systematic recursive, depth-first travesal of an Ada syntax tree. The internal state of the traversal functions includes the IDL syntax tree being constructed, as well as a "current position" pointer designating a node in the tree. When an Ada element is encountered that must be mapped, a new node is created and becomes the current node. The child elements of the Ada element are then explored, either explicitly using ASIS syntactic queries, or implicitly as a result of the continuation of the iterator. When an element is translated, its translation is attached as a subnode of the current node.

When the whole Ada tree has been explored, we have an in-memory image of the IDL decorated syntax tree. The second phase consists in performing a trivial traversal of this tree, in order to produce the corresponding IDL source file. This merely consists in traversing the tree depth-first, outputting the representation of each node as it is traversed.

The implementation of these first two phases has been completed; see figures 7 and 8 for an example of the translation.

The third phase, still under development, consists in the generation of a server implementation matching the generated IDL specification. This phase will also be driven by the IDL tree image, because the structure of the server implementation is closely described by the IDL specification. It is necessary to generate an implementation subprogram for each operation of every interface in the tree. This is done by derivating the skeleton implementation type produced by a standard Ada IDL compiler, such as the one contained in *AdaBroker*.

The implementation generator will traverse the IDL syntax tree and output the implementation function; it will also use semantic information from the Ada environment in order to construct the necessary parameter type conversions, constraint checks, and DSA method invocations. For this purpose, each node in the IDL tree is annotated with an ASIS element identifier pointing back to the Ada construct it represents.

Figure 3 summarises the flow of data through CIAO tools.

```
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_Parameters is record
      Temperature : Celsius;
      Cost        : Price;
   end record;

   procedure Heat (Self : access Root_Gizmo;
     How_Much : in Celsius);
   function  Get_Parameters (G : in Root_Gizmo)
     return Gizmo_Parameters;

   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;
```

Sample 7: Example DSA package *Gizmos*

4.3 Binding DSA entities with CORBA

In the previous section we presented the motivation and implementation of the tool we have developed, which produces an IDL specification from the declaration of a DSA package. In order to allow CORBA clients to make use of this IDL specification and perform calls to DSA services, we also need to provide a run-time mechanism to translate CORBA requests into DSA method invocations, and translate back the DSA invocation results to CORBA results.

To this effect, we are considering a two-step approach. We will first generate a CORBA server skeleton in Ada using traditional ORB software: we will use the IDL specification automatically generated by the translator described above as input to a standard IDL translator; more specifically, we plan to use our own, AdaBroker, described in section 3. AdaBroker will generate a CORBA server skeleton in Ada 95, providing hooks for the implementation of the various services described by the IDL specification. We will provide actual implementations for these services by gen-

[2] CORBA Interface for Ada (DSA) Objects.

```
#include "ciao.idl"

module Gizmos {
   interface Root_Gizmo;

   typedef ::CIAO::Standard::Ada_Float
     Celsius;

   typedef ::CIAO::Standard::Ada_Root_Real
     Price;

   typedef struct Gizmo_Parameters_struct {
      Celsius Temperature;
      Price Cost;
   } Gizmo_Parameters;

   interface Electric_Gizmo;

   interface Root_Gizmo {
      void Heat (in Celsius How_Much);
      Gizmo_Parameters Get_Parameters ();
   };

   interface Electric_Gizmo : Root_Gizmo {
      ::CIAO::Standard::Ada_Boolean
        Is_Plugged_In ();
   };
};
```

Sample 8: Generated IDL for *Gizmos*

erating Ada source code that handles CORBA requests by invoking primitive operations on DSA objects.

For each DSA package, we will thus automatically generate a CORBA skeleton package and a corresponding implementation package. These packages will together constitute "proxies" that will act as gateways between CORBA clients and DSA servers. The user will be able to assign the proxy packages to one or more partitions in her DSA application, using the partitioning tool provided by the vendor of her compilation system; each partition that contains proxy packages will then behave as a CORBA server, and will be able to serve requests from CORBA clients. The structure of such a distributed application is summarised on figure 4.

Suppose for instance that a CORBA client has obtained a CORBA object address (an IOR, Interface Object Reference) *My_ Gizmo* that designates a DSA object of type *Gizmo'Class*. The type of this IOR is the CORBA mapped interface *Gizmo*. As far as CORBA is concerned, it designates some object that is managed by the ORB on the proxy partition. When the client makes a call to *My_ Gizmo.Heat (37.0)*, a CORBA request is sent to the proxy partition's ORB. The ORB looks up the IOR in its internal tables, and calls the *Heat* method on the CORBA implementation object for interface *Gizmo*. This method first checks that the provided actual for parameter *How_ Much* verifies the constraint on subtype *Celsius*. The implementation object has an internal variable which is a RACW *Self_ Access* designating the actual DSA object. After the constraint check is made, the *Heat* subprogram uses this RACW to perform a dispatching call to *Heat (Self_ Access, 37)*. This call may or may not be remote, depending whether the user has chosen to affect the proxy package on the partition where the DSA object instance was created.

We need to assign an IOR for all DSA constructs that we want to be visible to CORBA clients. On the first time a DSA object reference (RACW) is to be transmitted to a CORBA client, we will register it with the underlying ORB software, and cache the mapping in internal structures of the proxy package, all this processing being completely trans-

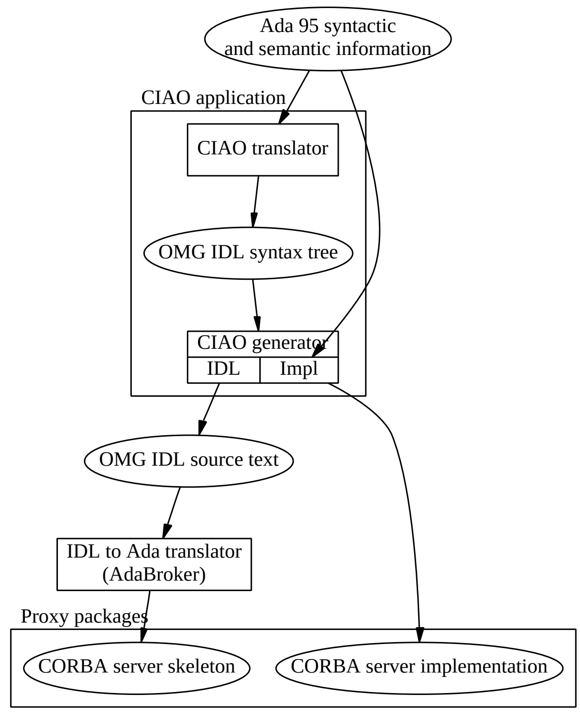

Figure 3: Flow of information through CIAO

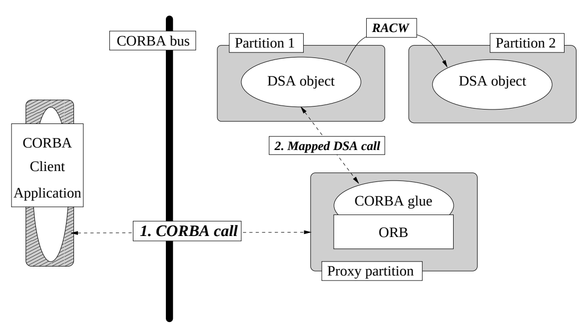

Figure 4: Calling DSA objects from CORBA

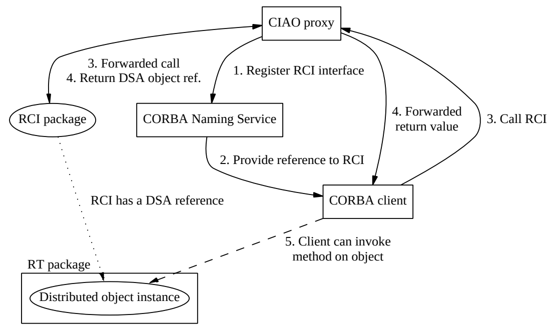

Figure 5: Bootstrapping a CIAO application

parent to both the client and the server.

Remote Access to Subprogram types can designate distributed subprograms. These are conceptually equivalent to objects with no attributes, and with a single operation, *Invoke*. We can thus consider RAS types as a particular form of object references, and handle them in a similar fashion to RACW types.

In distributed systems, initially obtaining a distributed object reference may be difficult. It is often necessary to query a distributed naming service (see section 2.1) to obtain object references, but it is impossible to use the naming service to retrieve its own root reference. In DSA, an object reference can only be created when a value is transported (as a remote call parameter, or as a function return value); when an object is created on a partition, the only way for a client partition to obtain a reference to that object is to get it as a value returned by a remote call, i. e. as the return value or as an *out* mode parameter of a RCI or RAS call, or RACW primitive operation call. Initially, clients always obtain their first object reference using RCI calls, because RCI have a fixed, well-know position within the distributed application. In CIAO, we have thus decided to solve the "chicken & egg" problem in the following way: RCI packages will be mapped to modules with a *Remote_Subprograms* interface; the proxy packages will automatically register the corresponding objects with the CORBA naming service at start-up time. In this scheme, CORBA clients can obtain references to the RCI packages by querying the CORBA Naming service, and then request objects from RCI servers just like normal DSA clients.

The solution presented here is still to be developed. The CIAO translator is already running; we are now starting the implementation of the automated proxy package generator. We expect to have it ready for release by the last quarter of 1999.

5 Conclusion

CORBA defines a set of useful services for development of distributed heterogeneous applications. We offer a specification and implementation of similar services of general interest for users of the Distributed Systems Annex of Ada 95. We also provide an implementation of a free, open-source CORBA ORB for Ada using the omniBroker ORB, as well as a prototype for a new, completely Ada ORB with advanced functionalities such as the Portable Object Adapter. We finally propose a mapping of the DSA object model to OMG IDL, and an automated tool to make DSA services available to CORBA clients.

Acknowledgements

The authors would like to acknowledge the contribution of ENST students: Fabien Azavant, Emmanuel Chavane, Jean-Marie Cottin, Tristan Gingold, Laurent Kübler, Vincent Niebel, and Sébastien Ponce, for their outstanding work on the AdaBroker project.

References

[1] Fabien Azavant, Jean-Marie Cottin, Laurent Kubler, Vincent Niebel, and Sébastien Ponce. AdaBroker, using OmniORB2 from Ada. Technical report, ENST Paris, March 1999.

[2] R. Berrendonner, L. Bousquet, T. Quinot, and S. Thellier. Mururoa: Distributed mutual exclusion using distributed objects in Ada 95. Technical report, ENST Paris, December 1997.

[3] Emmanuel Chavane. AdaBroker development — Dynamic Invocation. Master's thesis, ENST Paris, July 1999.

[4] Tristan Gingold. Broca, an Ada CORBA implementation. Master's thesis, ENST Paris, July 1999.

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

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

[7] 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.).

[8] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

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

[10] Laurent Pautet, Thomas Quinot, and Samuel Tardieu. CORBA & DSA: Divorce or Marriage? In *Proceedings of AdaEurope'99*, Santander, Spain, June 1999.

[11] Laurent Pautet and Samuel Tardieu. Building fault tolerant distributed systems using IP multicast. In *Proceedings of SigAda'98*, Washington, DC, USA, November 1998.

[12] Thomas Quinot. Mapping the Ada 95 Distributed Systems Annex to OMG IDL — Mapping definition. Technical report, ENST Paris and university Paris VI, May 1999.

[13] Thomas Quinot. Mapping the Ada 95 Distributed Systems Annex to OMG IDL — Specification and implementation. Master's thesis, ENST Paris, May 1999.