CIAO
Opening the Ada 95 Distributed Systems Annex to CORBA Clients

Thomas Quinot

École nationale supérieure des télécommunications
46, rue Barrault
F-75643 PARIS CEDEX 13
France
E-mail: quinot@inf.enst.fr

Abstract

While the Distributed Systems Annex of Ada 95 provides developers with a framework for easy contruction of safe distributed systems, integrating distribution seamlessly in the strong typing and well-defined semantics of Ada, it lacks the cross-platform, multiple-languages capabilities offered by CORBA.

*This paper presents CIAO (*CORBA Interface for Ada distributed Objects*), a tool for automated generation of proxies that allow CORBA clients to interact with services created using the Distributed Systems Annex. A representation of DSA service specifications in OMG IDL is first presented. We then describe an automated translation tool based on this representation model. This tool is based on ASIS, a standardized API for the extraction of syntactic and semantic information from an Ada compilation environment. It uses Broca, a full Ada ORB developed at ENST. CIAO and Broca are to be released as Free Software in the forthcoming months.*

1 Introduction

1.1 The Distributed Systems Annex and CORBA

The Distributed Systems Annex of the Ada 95 standard [2] provides application developers with a framework for easy construction of distributed systems. It integrates distribution as a natural extension of the language's abstraction paradigm. This allows applications developed using the annex to get the full benefits of the strongly typed programming model of Ada; the debugging of such applications is made very easy as well, because one can first write a system as a non-distributed, traditional Ada program, then later partition it and transform it into a distributed application. No code specific to distribution needs to be written; the distributed application uses exactly the same native Ada data types as a non-distributed ones; local procedure calls are simply converted by the compiler to remote invocations. When an abstraction is made remote, no modification of its clients is required.

CORBA [4], a set of industry standards promoted by the OMG consortium, is based on OMG IDL, a descriptive language whose syntax is close to C++, which is used to describe the services provided by a class of objects. IDL specifications can be mapped to different host languages such as C++, Ada 95, Java, or SmallTalk. A service developed in C++ can thus be accessed by clients written in Ada or Java, that run on entirely different platforms. The OMG has also standardized the protocols that are used for inter-node communication, theoretically allowing interoperation between products developed using tools from different vendors.

1.2 The CIAO project

We have provided an in-depth comparison of the Distributed Systems Annex and CORBA in [5]. Our team has been researching solutions to take advantage of the safety of the annex, as well as of the emphasis put on code reuse and good software engineering practice by Ada, while opening distributed services to the multiple platforms and languages supported by CORBA. We have thus started the implementation of CIAO (*CORBA Interface for Ada distributed Objects*), a tool for automated generation of proxies between DSA services and CORBA clients. We see the development of CIAO as an opportunity to promote the use of Ada 95 as a competitive platform for the creation of distributed services.

The principle of CIAO is to first generate an IDL specification that represents a service created using the distributed systems annex, and then automatically generate an implementation of this CORBA contract. In sections 2 and 3, we

present a method and tool for the translation of a DSA service specification to IDL. In section 4, we then describe the automated implementation generator, and in section 5 we discuss the operation of the generated implementation.

2 Service specifications in distributed systems

2.1 Specification of DSA services

In Ada 95, abstraction between the interface (or specification) and implementation of a service (*i. e.* a set of data types and operation signatures) is provided by packages. The declaration of a package can contain, in particular, the declaration of data types and subprograms, while the body of the package contains the implementation of the subprograms. This imposes a clear specification of a functional module's interface, and a complete separation between specification and implementation.

The Distributed Systems Annex (DSA) naturally extends this model by allowing the programmer to specify that the interface defined by a specific package declaration is to be considered "remote": several pragmas allow the definition of a *category* on a package, which indicates to the compiler that subprogram calls are potentially to be executed on remote hosts, and that objects declared using the types defined in the package may also reside on other nodes.

The *Remote_Types* pragma is used to define potentially distributed object types. When a tagged limited private type (*e. g.* `type Foo is tagged limited private`) and a corresponding class-wide access type (`type Foo_Ref is access all Foo'Class`) are declared in a *Remote_Types* package, then an instance of (a derived type of) `Foo` that resides on a given node can be designated by a `Foo_Ref` value held on another node: such an access type is called a *remote access to class-wide* type (or RACW). A node that has a RACW which designates a remote object can invoke methods on that object; these method invocations are serviced using a remote procedure call performed by the Partition communication Subsystem (PCS), which is completely hidden within the compiler's run-time library. To the programmer's eyes, the behaviour of his application is exactly as though the object instance lived within the same process.

The *Remote_Call_Interface* allows the creation of pure remote procedure call servers. When a library unit is categorized using that pragma, the Ada partitioning tool guarantees that the package is instantiated on exactly one node, and any call to a subprogram defined in that package made from any partition is treated as a remote call as well.

The specification of a service in the DSA thus consists in the declaration of a package that has the *Remote_Types* or *Remote_Call_Interface* category. Packages with the *Pure* category can also intervene in a service specification, because the Annex allows a *Remote_Types* or *Remote_Call_Interface* to depend on a *Pure* unit.

2.2 Specification of CORBA services

The CORBA distributed programming model is more restrictive than the one of DSA. It is purely object-oriented: any service in a CORBA system is described in terms of objects that implement interfaces. The description of an interface, its associated data types and its operations together constitute an IDL specification. An IDL to host language translator is used to produce client stubs and server skeletons for the object. The generated stubs and skeletons depend on a particular Object Request Broker (ORB, the runtime component that implements the CORBA protocols) implementation. The mapping of IDL constructs to host languages is standardized by OMG. In the case of Ada 95, the mapping does not make use of the distributed systems annex; the IDL base data types are mapped to specific Ada types declared in a standard *CORBA* package, and object references are mapped to private tagged types that extend a root *CORBA.Object.Ref* type. Distribution is thus apparent and intrusive, because it is necessary to explicitly obtain object references in a CORBA-specific way, and to convert data types between the mapped IDL types and their native Ada counterparts.

2.3 Formal mapping of DSA specifications into OMG IDL

In order to open access to DSA services for CORBA clients, we first have to represent the interface of such a service in CORBA's specification paradigm. In other words, we have to translate the declaration of a DSA package into an IDL specification. To this effect, we have defined in [6] a complete formal mapping of such declarations into OMG IDL. That document precisely defines the translation of all legal DSA declarations into IDL abstract trees.

The translation starts with the root Ada non-terminal, compilation_unit. A compilation unit is mapped to an IDL <module>; subunits are mapped to nested modules. The declarations in a compilation units are likewise mapped to IDL declarations. More generally, the translation of an Ada construct is specified as an equivalent IDL element. When this element is a non-terminal of the IDL grammar, we then have to specify how its sub-elements are to be constructed. We give this construction in terms of the translations of other Ada elements, and thus recursively define a translation for all valid constructs that can be encountered in a DSA package declaration.

Most Ada types are mapped to their "natural" IDL counterparts: numeric, enumerated, boolean, and string types are

translated to the corresponding IDL constructs. Ada records are translated to IDL `struct` types. The members of that structure type are the respective translations of the component declarations of the Ada record. Variant records are represented using structures and unions, and arrays are represented as sequences.

Some Ada types have semantics that are inherently local to the node that created them. This is the case, for example, for non-remote access types, as well as limited types. The Distributed Systems Annex mandates that such a type shall have user-defined marshalling and unmarshalling subprograms if they are to be used in the specification of a DSA service. In the CORBA translation of a service, these types are represented as opaque sequences of octets. A CORBA client can thus obtain a value of those types, and send it back unchanged to a DSA server.

Potentially distributed objects are declared in Ada as tagged limited private types. Such a type is mapped to an IDL interface declaration. The methods of this interface are translated from the declarations of the Ada type's primitive operations. Since the "self" parameter of object methods (which designates the object instance that is the target of the call) is implicit in IDL, we omit the first controlling formal parameter of all primitive declarations for distributed object types.

We have thus defined a complete translation of all legal Ada constructs in OMG IDL. The only exceptions are renaming declarations and generic instantiations. These are not taken into account in the current translation, because they introduce complex visibility and name resolution problems; we therefore decided to concentrate on getting a first operational version of the translation without these constructs.

3 Implementing an Ada to IDL translator

Having established a model for the translation of DSA service descriptions into OMG IDL specifications, the CIAO project continued with the development of a translator that implements this model. In this section, we first present a standard library for Ada CASE tool construction, then discuss the implementation of our DSA to IDL translator using that library.

3.1 The Ada Semantic Interface Specification

ASIS [3] (Ada Semantic Interface Specification) 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 parser and semantic analyser that are 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 queries* is also defined. These functions provide information about the semantic relationships 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 CIAO DSA to IDL translator uses ASIS. This makes it independent of any particular Ada compilation environment, although it is developed primarily with the GNAT compiler, and the associated ASIS-for-GNAT implementation. GNAT [8] is a free software Ada compilation environment initially developed at New-York University and now maintained by Ada Core Technologies[1].

3.2 The CIAO translator

3.2.1 Overview

The CIAO Translator [7] uses ASIS to extract syntactic and semantic information from the GNAT Ada 95 compilation environment. The program is easily portable to any other compilation environment that supports ASIS, for it does not depend on compiler internals, but only on some utility libraries that come with GNAT in source form.

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 traversal 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 translated, a new node is created and becomes the current node. The children 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 processed, 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 an IDL abstract tree. This tree gets decorated with semantic information, to allow the implementation generator to obtain sufficient information about

[1] See `http://www.gnat.com/`.

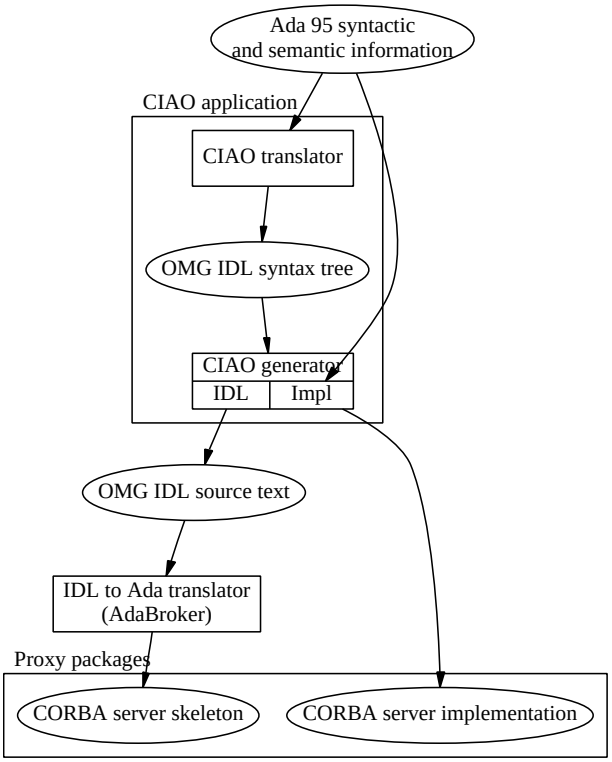

Figure 1. Flow of information through CIAO

the translated DSA service. Once this IDL tree is constructed, a trivial recursive traversal is performed, in order to produce the corresponding IDL source file. This merely consists in descending the tree depth-first, emitting a textual representation of each node as it is traversed into an output file; see samples 1 and 2 for an example of the translation.

The last phase consists in the generation of the CORBA implementation. The code generator is driven by the IDL tree, and uses ASIS queries to retrieve supplementary information about the structure of the DSA service from the Ada environment. Figure 1 summarizes the flow of data through CIAO tools.

3.2.2 Construction of the IDL tree

The overall structure of CIAO is presented on figure 2. The main procedure, *Driver*, initializes the ASIS environment, then schedules the different translation and code generation phases.

The first phase consists in the production of the IDL abstract tree from the DSA package declaration. This is the responsibility of the *Translator* package. This package essentially contains an instantiation of the generic *Traverse_Element* iterator, and defines the function that maps an Ada element into its IDL translation, as defined by the formal translation model.

```
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;
```

Sample 1: Example DSA package *Gizmos*

Translator uses numerous ASIS queries to obtain the descriptions of the Ada elements to be translated. Some of the Ada tree is simply traversed according to the default depth-first traversal provided by the standard iterator. On the other hand, for some particular elements, this default systematic traversal is inappropriate. Some parts of the Ada tree are to be ignored in the translation. For example, the private part of a DSA package declaration does not define a service provided by that package to exterior clients, and is therefore not translated. Similarly, the "self" formal parameter of distributed object methods is implicit in IDL, and must be omitted. In these situations, we explicitly use ASIS syntactic queries to explore the Ada tree, and bypass the default traversal.

We also make use of semantic queries, most notably for the resolution of entity usage names. The ASIS implementation has access to the Ada abstract tree as constructed and decorated by the compiler. All the name resolution and visibility rules are thus already applied; from any usage name, we can immediately obtain the corresponding defining oc-

```
#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 ();
  };
};
```

Sample 2: Generated IDL for *Gizmos*

currence, and thus the denoted entity.

The translator needs to know of some high-level semantic properties that are defined in the language reference manual, but have no corresponding ASIS queries. For example, in order to translate a subprogram declaration in a *Remote_Types* package and associate it with the proper IDL interface, its controlling formal parameters have to be determined. To address this requirement, we have created a set of "extended" ASIS queries that correspond to properties formally defined in the Ada standard, and we have isolated them in a package (*ASIS_Queries*) that can be reused independently of the CIAO project in any ASIS tool that would need access to these properties.

We have also designed a library for manipulation of IDL abstract trees. We based it on existing components for syntax tree data types: this library is an adaptation of parts of the GNAT compiler, which were modified to handle IDL nodes rather than Ada nodes. Like the original, this library has two layers. *IDL_Tree* provides low-level access to a "universal" node structure. Other packages only access

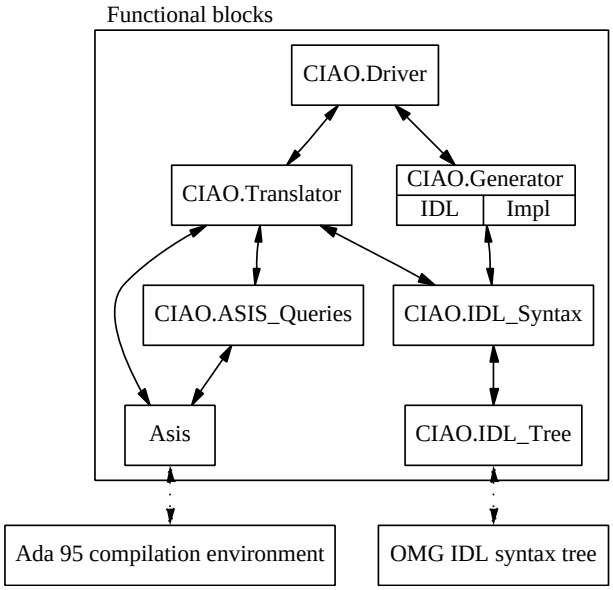

Figure 2. Structure of CIAO

the nodes using a higher-level package, *IDL_Syntax*, which enforces the IDL grammar rules. Each node also includes semantic flags, as well as a pointer to the original Ada element. This attribute is essential for the implementation generator, because it establishes a "bridge" between the semantics of the IDL contract and that of the DSA service.

4 The implementation generator

The next step in the CIAO processing is the generation of an implementation for the constructed IDL specification. Using a standard IDL to Ada compiler, the source file created by the translator can be mapped to CORBA client stubs and server skeletons. The skeleton for an interface contains the declaration of an abstract tagged type whose primitive operations correspond to the methods of the interface. The server developer must then produce an implementation type, which extends the skeleton type and overrides its primitives with actual implementations.

In the context of a CORBA to DSA proxy, these actual implementations must convert the parameters of the method invocation from CORBA mapped types to native Ada types (which are used in DSA calls), then perform a DSA remote call and possibly convert back a return value.

For CIAO, we have produced a code generator that automatically creates the implementation for interfaces generated by the translator described above. These implementations are targeted to the *Broca* CORBA toolkit, a free software ORB and IDL-to-Ada compiler developed at ENST [1]. As *Broca* implements CORBA's Portable Object Adapter, there is very little code in CIAO that actually

depends on the ORB implementation, and we consider the generated code to be fairly portable to another CORBA implementation.

This generator is driven by the IDL abstract tree, because the structure of the produced code corresponds to the layout of the IDL specification as a set of nested modules of interfaces. Semantic information from the Ada environment is also used for the construction of the necessary parameter type conversions, constraint checks, and DSA method invocations. Indeed, a lot of semantic information is lost during the translation of DSA specifications to IDL; IDL has a far less expressiveness, and cannot represent such concepts as subtypes or constrained types. Consequently, the code generator uses annotations inserted in the IDL tree by the translator that point back into the Ada tree, and uses ASIS to complete the code production.

The generator also produces conversion packages for data types that are declared in the DSA packages: when a native DSA type is translated to IDL, the mapped type used by the IDL to Ada compiler is different from the original type. For example, an Ada `String`, translated to an IDL `String`, will be mapped back as a `CORBA.String`. We therefore produce subprograms to convert data back and forth between the two type sets. These functions are used in the generated implementations to convert the call arguments and return values.

5 The run-time operation of the proxy

5.1 Execution of calls

For each DSA package, we have automatically generated a CORBA skeleton package, using Broca's IDL to Ada compiler, and a matching implementation package, using the CIAO code generator. These packages together constitute a "proxy" that acts as a gateway between CORBA clients and DSA servers.

Using a partitioning tool such as *Gnatdist*, the user can then assign the proxy packages onto a partition in her DSA application. That partition is then linked with the Broca run-time library, and behaves as a CORBA server. It may receive requests from CORBA clients; these requests are serviced by calling the generated implementation subprograms, which in turn call the original DSA operations. The structure of such a distributed application is summarised on figure 3.

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*

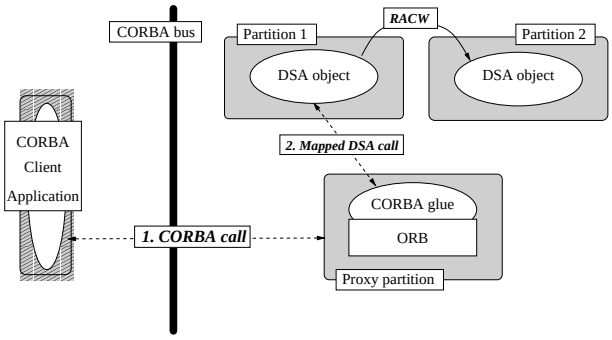

Figure 3. Calling DSA objects from CORBA

(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* satisfies 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.

5.2 Creation of object references

We have discussed the operation of the proxy from the point where CORBA clients have obtained IORs that designate DSA objects; we still need to provide a mechanism for assigning an IOR to a DSA object.

For this purpose, we take advantage of the Portable Object Adapter facility integrated in Broca. In a CORBA system, the Object Adapter (OA) is the software component that implements the management of the entities that incarnate CORBA Objects, and enforces policies over their creation, activation, deactivation and destruction. In former CORBA versions, only one OA existed: the Basic Object Adapter (BOA). In the BOA, exactly one unique implementation object exists for each CORBA object reference. The POA introduces much more flexibility in the design of a server: it allows a server to contain only one implementation object for a whole class of CORBA objects; within a method invocation, standard function calls to the ORB provide a way to identify the specific instance which is the target of the call, using a user-assigned identifier which is embedded in the object reference.

CIAO uses this policy for the implementation of interfaces that represent Ada distributed objects. The object

identifier that we chose is simply the marshalled form of a RACW (a DSA object reference) that designate the object. Each time one such reference is to be transmitted to a CORBA client, we construct an IOR by associating that object identifier with the POA created at initialization time by the proxy for the class of the object. There is no need to register that newly-created IOR with the ORB, thus avoiding any run-time memory leak. Since the RACW can be computed back from the IOR, there is no need to maintain a record of the mapping between DSA objects and IORs: the proxy is stateless, and can be stopped and restarted without loss of functionality in case of a fault.

The POA thus permits the creation of CORBA object references that correspond to RACWs. 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 treat RAS types likewise, as a particular form of object references, and handle them in a similar fashion to RACW types, once more using the marshalling and unmarshalling functions provided by the Annex.

Remote Call Interface packages are much simple to handle; since each RCI package is instantiated only once in a complete DSA application, only one object reference for its CORBA implementation object is needed. In the case of RCIs, the POA is thus used in its mode of operation that is closest to the BOA: a single implementation object is created, and it is activated and assigned a reference by the ORB at proxy start-up.

5.3 Initial distribution of references

In distributed systems, initially obtaining a distributed object reference may be difficult. It is often necessary to query a distributed naming service 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 remote 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; the Partition Communication Subsystem includes its own "hidden" mechanism for name resolution, and initially provides a means for any partition to make a call to any RCI package.

In CIAO, we have thus decided to solve the "chicken & egg" problem in the following way: RCI packages are mapped to modules with a single *Remote_Subprograms* interface; the proxy packages automatically registers 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. We thus naturally transport the DSA model of operation into the CORBA world, by making the procedure for getting initial references to RCI packages explicit (whereas it is implied in DSA).

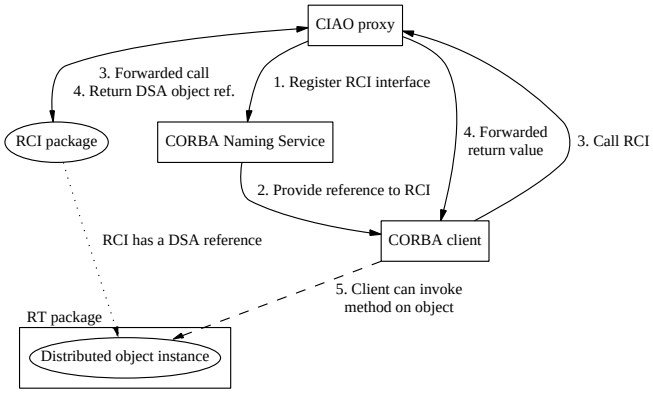

Figure 4. Bootstrapping a CIAO application

An example of this protocol is given on figure 4. If the application was purely based on DSA, then at startup a node would only be able to make remote calls to the RCI package. This is how it would obtain references to objects that live on other partitions. When the CIAO proxy is started, it creates a CORBA implementation object for the interface of the RCI package, and registers a reference to this object with the CORBA naming service (1). A CORBA ORB provides a method by which a CORBA client can obtain an initial reference to the naming service. The client can then query the naming service and obtain the reference to the RCI package (2). This reference is used to perform a remote call on the RCI package (3), which finally returns a reference to a DSA object (4). The CORBA client can thus invoke a method on the DSA object (5) just as any DSA node could.

6 Conclusion

The Distributed Systems Annex provides a superior framework for the development of safe distributed applications. It is well-integrated in the Ada software engineering practice, and allows for an easy transition from non-distributed to distributed code. CORBA, on the other hand, emphasises interoperation between objects created and residing on different platforms, written using different languages, and different CORBA implementations.

We proposed a solution to make the best of both worlds: the tools presented here allow a CORBA client to access services provided by DSA objects and servers. It makes use of ASIS, an ISO standardized API for Ada CASE tools. It is based on free software components developed by ENST and others, and will be made freely available in source code in the near future.

References

[1] T. Gingold. Broca, an Ada CORBA implementation. Master's thesis, ENST Paris, July 1999.

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

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

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

[5] L. Pautet, T. Quinot, and S. Tardieu. CORBA & DSA: Divorce or Marriage? In *Proceedings of AdaEurope'99*, Santander, Spain, June 1999.

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

[7] T. Quinot. Mapping the Ada 95 Distributed Systems Annex to OMG IDL — Specification and implementation. Master's thesis, ENST Paris, Aug 1999.

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