DROOPI*: Towards a generic middleware

Thomas QUINOT, Fabrice KORDON, Laurent PAUTET

{quinot,pautet}@enst.fr
École Nationale Supérieure
des Télécommunications
CS & Networks Department
46, rue Barrault
F-75634 Paris CEDEX 13, France

fabrice.kordon@lip6.fr
Laboratoire d'Informatique de
Paris 6/SRC
Université Pierre-&-Marie-Curie
4, place Jussieu
F-75252 Paris CEDEX 05, France

Abstract

This paper presents our work to bridge the Ada 95 Distributed Systems Annex (DSA) and CORBA to take advantages of both environments facilities. Our project consists in two successive steps. The first one is CIAO, a DSA to CORBA translator. The second one aims at the definition of a generic middleware to be customized to DSA and CORBA. We propose a definition and an architecture of services for a generic middleware, DROOPI*, and explain how it can be customized according various criteria. This generic middleware takes advantage of the lessons learned in the design and the development of several other projects: GLADE [24], AdaBroker [1], and CIAO [25]. Therefore, in its motivation and its targetting, we believe* DROOPI *is an original project.*

1 Introduction

Ada 95 is now widely recognized to be a excellent language for the programming of distributed applications. It has all required facilities (inheritance, polymorphism, data-driven synchronization mechanisms, etc.) However, Ada 95 lacks in openness (ability to cooperation with distributed components written using other languages).

On the contrary, CORBA is dedicated to the distributed cooperation of software components regardless of their programing language. Its dynamic aspects are particularily valuable when a component uses services that were not implemented when the component was compiled. However, it only provides a minimum paradigm for cooperation which lacks safety mechanisms such as type protection, or enhanced parallel capabilities.

*Distributed Reuseable Object-Oriented Polymorphic Infrastructure

Our objective is to get the best from both environments, and to provide Ada 95 with CORBA interoperability facilities. The project started with the design of CIAO: a DSA to CORBA translator. CIAO has successfully allowed CORBA clients to invoke DSA services. However, CIAO has several drawbacks due to the approach based on the definition of a gateway between CORBA and DSA. If the gateway is centralized, it becomes a bottleneck. If the gateway is distributed, it becomes difficult to preserve the identity function (i.e. two distinct references correspond to the same object).

Based on the CIAO experience, we propose a new solution to bridge DSA and CORBA: the definition of a generic middleware that could be specialized. We identified on the `GLADE` and AdaBroker projects that some parts of the code were similar, in regard to the general behaviour. It appears then that a great improvement would be to factorize this code, and to instanciate it in different contexts.

An interesting consequence should be to ease the interoperability between the distribution models of CORBA and DSA. The approach will no longer be top-down, as in CIAO where the distribution model is adapted to low-level requirements. A new bottom-up design is introduced, that leverages the middleware communication layers in various distribution contexts.

The paper is structured as follows. Section 2 presents the features of CORBA and DSA that we aim to share. Section 3 then describes the CIAO approach, and provides some conclusions used to elaborate a new solution. Finally, section 4 presents the main generic middleware functions that have to be defined in order to permit specialization into various middleware schemes. We also provide a short comparison between Droopi and some similar projects, such as Quaterware and Jonathan. We finally draw conclusions and perspectives.

2 Building blocks

This section presents two models of interest for distributed application development:

- The OMG CORBA model has an outstanding interoperability architecture; it addresses the need of interoperation between software components developed using various languages on different machine architectures;

- The Ada 95 Distributed Systems Annex (DSA) provides an interesting approach to distribution: existing constructs for separation of interface and implementation are extended with distribution semantics; all existing properties of the language are preserved.

On the other hand, using a Java Virtual Machine (JVM) creates innovative possibilities for distribution (code migration, stubs download). The JVM can be easily programmed in Ada 95, for example with the JGNAT compiler [4].

We thus also considered Sun's Java/RMI (Remote Method Invocation). However, its approach to distribution is very classical and does not introduce new ideas beyond features already present in CORBA or DSA. Work is in progress within the OMG to integrate RMI as a subset of CORBA [22]. Furthermore, the RMI distribution model exhibits severe shortcomings: although the intent of the RMI specification is to provide distribution-transparent remote invocation, Java language semantics cannot be preserved correctly in a distributed context [5].

In the following subsections, we describe the two models for distributed applications. We start with CORBA, whose main functionnalities are also available in DSA. Then, we present the rich development environnement provided by DSA. But first of all, we justify the choice of Ada 95 as the programming language in all our distributed projects.

2.1 Building distributed systems with Ada 95

The core Ada 95 language [11] contains high-level features available in most modern programming languages such as inheritance and polymorphism [16]. It also defines paradigms more seldom available in other languages, such as data-driven synchronization or hierarchical library units.

A conforming Ada 95 implementation has to support the core programming language. It may support some or all of the optional annexes contained in the reference manual. Those "specialized needs" annexes cover various domains such as real-time programming, systems programming or distributed systems. Each annex has its own test suite used for compiler conformity assessment [13].

Concerning inheritance, types may be derived and optionally extended, provided they are marked as "tagged". A class encompasses a tagged type (the root of the inheritance tree) and all its derived types. All the primitive operations of a type (subprograms declared in the same scope as the type) are inherited by its children, unless explicitly overridden. Sample 1 contains a tagged type "Event" and two derived types "Information" and "Fatal_Error".

As in Java, an Ada type can only derive from one parent. However, this limitation can be easily worked around by using established techniques such as sibling or mix-in inheritance [3].

We have chosen Ada 95 for all our studies and developments because of the following characteristics:

- Ada 95 is one of the very few languages providing all the high-level features required by large applications, and more specifically distributed applications. Concurrency, real-time features, object-oriented constructs, data encapsulation, strong typing, exceptions and name spaces are essential in the real-world practice of distributed application development.

- DSA is actually similar to Java/RMI. It focuses on Ada to Ada communication, just like RMI is oriented toward Java to Java communication. Moreover, DSA has a very rich distribution model, including regular remote subprograms, distributed objects, and shared memory. The use of Ada 95 was thus required to take DSA into account.

```ada
package OO_Example is

   type Event is tagged
      record
         Date    : String (1 .. 8);
         Message : String (1 .. 8);
      end record;
   --  Root class type.

   procedure Output (E : Event);
   --  Primitive operation: screen output

   procedure Handle (E : Event);
   --  Primitive operation: do nothing

   type Information is new Event with
      record
         File : String (1 .. 10);
      end record;
   --  Add field File to Event.

   procedure Output (I : Information);
   --  Overload primitive operation Output
   --  to have a file output. Primitive
   --  Handle is not redefined.

   type Fatal_Error is new Information with
      record
         Level : Natural;
      end record;
   --  Add field Level to Information.

   procedure Handle (E : Fatal_Error);
   --  Overload primitive operation Handle :
   --  execute action depending on severity
   --  level. Primitive Output is not
   --  redefined (use of Information's one).

   type Any_Event is access all Event'Class;
   --  Pointer type to any object in the
   --  Event hierarchy.

end OO_Example;
```

Sample 1: Object-oriented example

- The last but not the least point in favor of Ada 95 is that the CORBA Ada 95 mapping is easier to use than most other mappings. A CORBA mapping specifies how distributed entities are implemented in a given target language. In Ada 95, like in other languages, a few entities are not translated into a one-to-one mapping. However, the Ada 95 mapping exhibits less such non-trivial translations, and they are simpler than other non-trivial mappings to other languages.

2.2 CORBA: An interoperable distributed OO platform

CORBA [20] is an industry-sponsored effort to standardize the distributed object paradigm. In CORBA, a dedicated language is used to describe all services provided by applications or the CORBA platform itself: OMG IDL (Interface Definition Language). The consistent use of IDL makes CORBA more self-describing than any other client/server middleware.

OMG IDL supports a C++-like syntax for constant, type and operation declarations, including the standard preprocessor directives. IDL allows the definition of *modules*, which provide a hierarchical name space for all entities. A module can define *interfaces*. An interface is a specification of the external appearance of a class: it defines a set of methods that a client can invoke on an object.

An interface can also define attributes. An attribute is a component field. For any attribute *Attr*, the implementation contains two subprograms Get_*Attr* and Set_*Attr*. When an attribute is *readonly*, only Get is provided. An interface can derive from one or more interfaces (multiple inheritance).

CORBA offers a description model based solely on distributed objects. In that respect, it can be compared to Java, as this language only provides an object-oriented programming model, and ignores the classical structured programming model.

From an IDL description of a service, a translator generates client stubs and server skeletons in a target language (e.g., C++, C, Java, Ada 95). A number of language mappings have been standardised by the OMG. Each one specifies how IDL entities are implemented in a particular target language. The straightforwardness of a mapping depends on target language features. When an IDL feature is not defined in the target language, the mapping provides a standard way of simulating the missing functionality; these work-arounds are often awkward. In particular, when the target language does not provide object-oriented constructs, the user has to deal with a complex simulation of those functions. C++ programmers also have to follow several rules related to parameters passed by reference.

Generated stubs and skeletons perform appropriate calls to a vendor-supplied library (the *Object Request Broker*, ORB) which deals with the core distribution functions. The interfaces they use to invoke ORB services are specific to a vendor and product. However, they have to provide users with a view of objects complying with a standard canvas defined by the language mapping.

Client stubs convert user method invocations into requests to the ORB, which transmits these through an object adapter to the server skeleton (figure 1). Object adapters are components within the ORB that control several aspects of service implementations. They create and activate the concrete entities implementing services; they control how execution resources are assigned to these entities, and how object references are mapped to them. Finally, object adapters take care of destroying implementation objects and reclaiming the associated resources as they become unused.

2.2.1 CORBA Services

The CORBA ORB provides a core set of basic functions. All other services are provided by objects described using IDL. The OMG has standardized a set

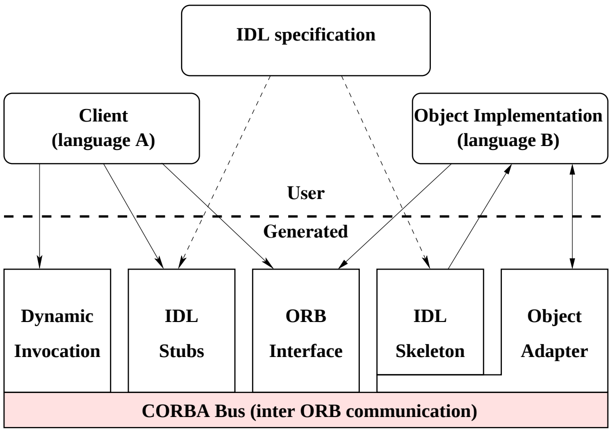


Figure 1: Overview of CORBA

of useful services such as Naming, Trading, Events, etc. Vendors are free to provide a compliant implementation of these services.

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. So, one 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 reference an object. This is very similar to the naming hierarchies that exist in the Domain Name System and in the UNIX file system. A typical scenario consists in providing a well-known object reference that defines the root of a naming context hierarchy, and in executing naming operations on this hierarchy.

The Trading Service provides a higher level of abstraction than the Naming Service. The Naming Service can be compared to the White Pages and the Trading Service to the Yellow Pages.

The Events service provides a way for servers and clients to interact through asynchronous events between anonymous objects. A *supplier* produces events when a *consumer* receives notification 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 2). Suppliers and consumers produce and receive events through their associated proxies (see plain arrows in figure 2). From the event channel point of view, a *proxy supplier* (resp. *proxy consumer*) is seen as a consumer (resp. 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. This allows to define four models to exchange events and data.

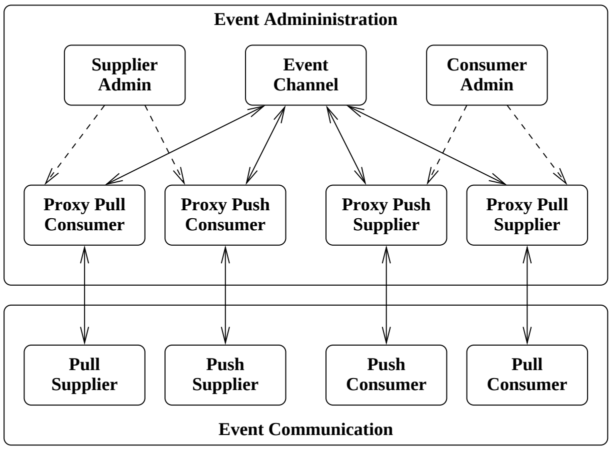


Figure 2: Structure of the Events Service IDLs

2.2.2 Dynamic aspects of CORBA

IDL interface information can be stored on-line in a database called Interface Repository (IR). The OMG specification describes how the database is organized, and how to retrieve information from it. The IR allows a client to discover the signature of a method which it did not know at compile time. The client can subsequently use this knowledge together with actual values for the method parameters to construct a complete request and to invoke the method. The set of functions that permits the construction of a method invocation request at run time is the Dynamic Invocation Interface (DII). The DII has a server-side counterpart, called Dynamic Skeleton Interface (DSI). The DSI allows objects to implement interfaces defined only at runtime.

The IR API allows the client to explore repository classes to obtain a module definition tree. From this tree, the client extracts subtrees defining constants, types, exceptions, and interfaces. From an interface subtree, the client can select an operation with its list of parameters (type, name and mode) and exceptions.

A client has then three ways to make a request. As in the static case, he can send it and wait for the result; he can also perform a one-way call and discard the result. A third mechanism is offered: the client sends the request without waiting for the result, and obtains it later, asynchronously.

2.3 Ada 95 DSA

DSA provides the developer with a standardized way of partitioning an Ada application across a network of computers. Communication between the various partitions relies on remote subprogram calls or method invocations on remote objects.

Remotely-called subprograms may be statically or dynamically bound, allowing applications to use either the classical remote procedure call paradigm [2]

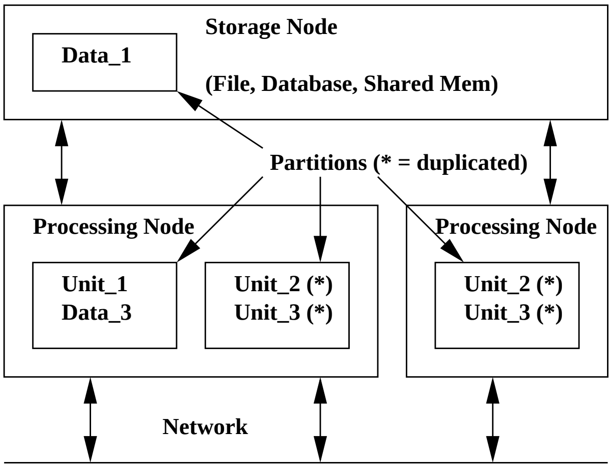


Figure 3: Overview of DSA

(see sample 2) or the increasingly popular distributed object paradigm [30, 20] (see sample 3). High-level semantics of the language are preserved when using DSA; for example, exception propagation works as usual and timed subprogram calls can be used as though the program was not distributed. Another interesting feature is shared objects (see sample 4) whose contents can be modified directly using atomic or non-atomic read and write operations. Such objects may be mapped on a shared memory support [14].

```
package RCI is
   pragma Remote_Call_Interface;

   procedure Remote_Subprogram
     (Parameter : in out Integer);

   procedure Asynchronous_Subprogram
     (Parameter : in String);
   pragma Asynchronous (Asynchronous_Subprogram);

end RCI;
```

Sample 2: Remote subprograms

In DSA, services provided by a module are described using a subset of Ada 95. The user identifies interface packages at compile time using so-called *categorization pragmas*. Those pragmas are:

Remote Call Interface (RCI). Library units categorized with this pragma can declare subprograms to be called and executed remotely. This remote procedure call (RPC) operation is a statically bound operation. In these units, clients and servers do not share their memory space.

Dynamically bound calls are integrated with Ada capabilities to dereference subprograms (remote access to subprogram, RAS) and to dispatch

```
package RT is
   pragma Remote_Types;

   type Remote_Object_Type is
     tagged limited private;

   procedure Remote_Object_Method
     (Object    : in Remote_Object_Type;
      Parameter : out Integer);

   type Remote_Object_Reference is
     access all Remote_Object_Type'Class;
   --  This construct defines a remote
   --  reference type that can point on
   --  both local and remote objects.

private
   type Remote_Object_Type is tagged limited
      record
         Name : String (1 .. 10);
      end record;
end RT;
```

Sample 3: Remote objects

```
package SP is
   pragma Shared_Passive;

   Shared_Integer_Array :
     array (1 .. 10, 1 .. 10) of Integer;
end SP;
```

Sample 4: Shared Objects

on class-wide operands (remote access on class wide types, RACW). These remote access types can be declared in an RCI package.

A remote access type can be implemented as a fat pointer — a structure with a remote address and a local address. The remote address denotes the host on which the entity has been created; the local address describes the service in that host's local address space.

Remote Types (RT). Unlike RCI units, library units categorized with this pragma can define distributed objects and remote methods on them. They can also define the remote access types described above. A subprogram defined in a RT unit may be a primitive subprogram of a distributed object type. In this case, it can be called remotely using a method invocation on a RACW. Unlike RCI units, RT units are not to be placed on only one partitions: as any regular unit, a RT unit can be placed on any number of partitions.

Shared Passive (SP). Entities declared in such library units are mapped onto shared storage space (file, memory, or database). When two partitions use such a library unit, they can communicate by means of a common variable.

This corresponds to the shared variables paradigm. Entry-less protected objects declared in these units provide transaction-like atomic access to shared data.

A categorized package must follow specific legality rules. For example, the declaration of a package whose subprograms can be called remotely must not contain data types whose semantics are purely local (e.g. active types such as task types). Restrictions are also enforced on the dependencies of each package. As a general rule, RCI $\succ$ RT $\succ$ SP. This means that a Remote_Types package declaration can only depend on other Remote_Types or Shared_Passive units.

The DSA does not describe how a distributed application should be configured. It is up to the programmer (using a partitioning tool whose specification is beyond the scope of the annex) to define what the partitions in his program are, and on which machines they should be executed.

2.3.1 Distributed objects

Object-oriented and distribution features can be combined to create distributed objects. Distributed objects in Ada 95 are used in the same way as regular non-distributed ones. However, a few restrictions apply to their type declaration:

1. The target of a distributed reference must be a "private" type, that is a type whose fields are not directly accessible, except from within the package where the type has been declared. One immediate consequence is that the only way to access fields of a remote object is to use one of its methods.

2. The target of a distributed reference must be a "limited" type. A limited type is a type whose instances cannot be duplicated by an assignment operation. This prevents a partition from acquiring a deep copy of a remote object; if the object contains data whose semantics are purely local (such as a regular pointer), it would make no sense to move it from one partition to another. Of course, the implementor of a type is free to provide the programmer with a "copy" operation, which will carefully copy all the fields of the object.

In spite of those limitations, all popular distributed objects constructs can be implemented on top of the Ada 95 model. For example, object migration can be easily achieved by using a "deep copy" method and a redirection object, that will redirect every method call made to the old location of the object to the new copy, using the polymorphism properties of object references.

2.4 Summary of differences between CORBA and DSA

We have briefly described CORBA and Ada 95 DSA. The essential points where they differ are detailed in [23]. We provide here a summary:

Description of services. In CORBA, services are described using OMG IDL, a purely declarative language whose syntax and object model are close to those of C++. In DSA, services are described using Ada 95 package specifications. Additional restrictions are placed on these declarations to ensure that requests can be transmitted across partitions.

Service model. A CORBA service is composed of a set of objects that implement interfaces: the service model is purely object-oriented. In DSA, the service model derives from the abstractions provided by the Ada 95 languages (packages), and is thus more general: a service can consist in tagged types (similar to CORBA objects), but can comprise subprograms accessible through the RPC mechanism as well. DSA also provides a distributed share data facility through Shared Passive packages. Such units provide the functionality of a distributed shared memory.

Application development model. In CORBA, an IDL contract must first be written. Stubs and skeletons are then automatically generated, and client and server modules are derived from, and must integrate with, the generated code. In DSA, a service can first be developed in a stand-alone fashion, tested in a non-distributed context, and then be made into a remotely accessible service using a post-compilation partitioning tool. Programmers thus do not ever have to deal with any generated source code.

Interoperability The Ada 95 Reference Manual [11] does not require a DSA implementation to work on heterogeneous systems (although `GLADE`, like any reasonable implementation, provides default XDR-like marshalling operations, which provides a system-independent representation of data streams). No provision is made for interoperability of DSA applications with other distributed systems.

An ORB is required to implement CORBA's standard Common Data Representation (CDR) and General Inter-ORB Protocol (GIOP), which allows transparent communication among heterogeneous nodes.

CORBA and DSA thus differ in their objectives and philosophy. CORBA aims at providing a means for interoperating components of various origins. Interfaces between CORBA and various legacy systems are defined. Most components of the CORBA platform are defined using the platform's own description language. This makes it very modular and vendor-neutral.

The focus of DSA is seamless integration of distribution in the development process of an Ada 95 application. The intent of the standard is to make distribution as transparent as possible by making distributed program units as similar as possible to non-distributed ones. Consequently, many high-level functionalities such as the location of the unit providing a given service, or the determination of application termination, are part of the platform implementation. In CORBA, these high-level functionalities would be defined and implemented as external services.

The following section presents an attempt at providing interoperation between these platforms, overcoming their divergences. We learn from this experiment that, in spite of these differences, both are constructed around the same core abstractions of the distributed object programming model (object addressing, remote method invocation, and resource encapsulation). These features are essential parts of any distributed object-oriented platform. In section 4 we then define a set of recurring functionalities that one can expect to find in such environment.

3 First approach to CORBA/DSA integration

Based on our comparison of CORBA and DSA, we made a first attempt at interoperating the two platforms, in order to take advantage of their respective strong points:

- DSA's type safety and natural integration in the software engineering process,

- CORBA's openness to multiple languages and environments.

The objective of the CIAO[1] project is to let CORBA clients invoke DSA services. The principle of CIAO is to first generate an IDL specification from the description of a DSA service. We then automatically produce an implementation of this IDL specification, which gateways CORBA requests to DSA services.

In sections 3.1 and 3.2, we present a method and tool for the translation of a DSA service specification to OMG IDL, and automated gateway implementation generation.

3.1 Formal mapping of DSA specifications into OMG IDL

To enable CORBA clients to access DSA services, we first have to represent the interface of these services in the CORBA specification paradigm. In other words, we have to translate the declaration of a DSA package into an IDL specification. We have defined a complete formal mapping of such declarations into OMG IDL [25].

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 unit 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.

[1] CORBA Interface for Ada distributed Objects

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. All methods of this interface are translated using the declarations of the Ada type's primitive operations.

We therefore have 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.

3.2 Implementing an Ada to IDL translator

Having established translation rules of DSA service descriptions to OMG IDL specifications, the CIAO project continued with the development of a translator implementing 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.

3.2.1 The Ada Semantic Interface Specification

ASIS [12] (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 tool developers to take advantage of the compiler's parser and semantic analyser; it provides 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 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 occurrence of an entity name, they can provide the definition of that entity. We thus can view ASIS as a *reflexive description* 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 [27] is a free software Ada compilation environment initially developed at New-York University and now maintained by Ada Core Technologies[2].

3.2.2 The CIAO translator

The CIAO Translator 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 CIAO translator has two main phases. First, the Ada semantic tree is recursively descended using the standard ASIS iterator; an IDL tree is constructed by assembling IDL subtrees that correspond to each encountered Ada element. This IDL tree then gets decorated with semantic information about the DSA service. Two descents of this tree are finally executed:

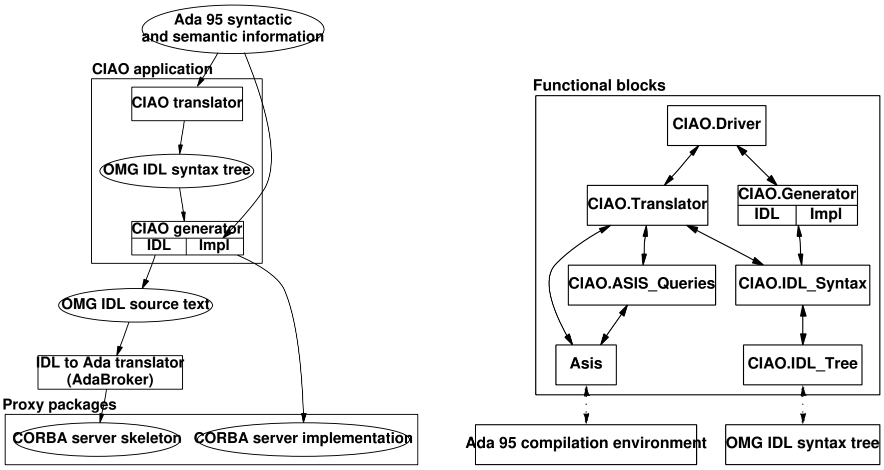
- A trivial traversal is used to produce the corresponding IDL source file.
- An implementation of that CORBA specification is generated. The code generator is driven by the IDL tree, and uses ASIS queries to retrieve additionnal information about the properties of the DSA service from the Ada environment. Figure 4(a) summarizes the flow of data through CIAO tools.

The overall structure of CIAO is presented on figure 4(b). The main procedure, *Driver*, initializes the ASIS environment, then schedules the translation and code generation phases.

The first phase consists in the production of the IDL abstract tree from 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 any Ada element into its IDL translation, as defined by the formal translation model.

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 from a *Remote_ Types* package and associate it with the proper IDL interface, its controlling formal parameters have to be determined. To address this requirement,

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



(a) Flow of information in CIAO

(b) Structure of CIAO

Figure 4: CIAO overview

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.

The next step in the process 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. CIAO also generates an implementation for the interfaces described by the IDL specifications.

This implementation uses CORBA's Portable Object Adapter (POA) to minimise dependencies toward a specific CORBA product. A few vendor-specific points do remain. We have isolated them in a specific package; we have implemented a version targeted at our *AdaBroker* ORB [1]. This package could be easily adapted to other compliant Ada/CORBA tool chains.

The implementation generator is driven by the IDL abstract tree, because the structure of the produced code corresponds to the layout of the IDL specification. Semantic information from the Ada environment is also used for the construction of parameter type conversions, constraint checks, and DSA method invocations. This information is accessed through annotations in the IDL tree that point back to the original Ada tree; these annotations compensate the fact that IDL is less expressive than Ada 95.

3.3 The run-time operation of the proxy

For each DSA package, we have automatically generated a CORBA skeleton package (using AdaBroker'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.

One of the proxy's duties is to assign CORBA IORs to DSA objects. For this purpose, we take advantage of POA features. The POA introduces flexibility in server design: it allows a server to contain only one concrete implementation entity serving requests for a whole set of CORBA objects; within a method invocation, the ORB API provides a way of identifying the specific instance which is the target of the call. Instances can be differentiated by a user-assigned identifier embedded in the object reference.

In CIAO, the object identifier is simply the marshalled form of a RACW (a DSA object reference). Each time such a reference is to be transmitted to a CORBA client, we construct an IOR by associating that identifier with a POA created at initialization time by the proxy (one POA thus exists for each distributed object type). 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.

Remote Access to Subprogram (RAS) types 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.

Remote Call Interface packages are much simpler 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.

3.4 Lessons learned

CIAO has successfully allowed CORBA clients to invoke DSA services. However, the gateway generation approach has a number of drawbacks. Some are related to the current implementation, but others are essential to the approach.

Using a gateway to convert requests from one platform to the other means that all requests will have to go through a bottleneck. Depending on the size of the application, this may be a major drawback in terms of performance. Reliability is also at stake, because the computing node executing the gateway becomes a single point of failure. Both issues may be addressed by establishing several gateways. This introduces another concern: consistence of the information used by all gateways to translate requests. It becomes difficult to ensure that translated object references designating the same DSA object seen through different gateways are identical CORBA references, i. e. to preserve object identity across gateways.

The CIAO approach also requires application developers to perform specific steps to make distributed (DSA) application open to CORBA clients. While most of this process is automated, it requires a modification to the application to include the proxy units in one of its partitions.

To provide a more convenient access to DSA services for CORBA clients, and to further allow CORBA objects to be transparently accessed from DSA applications, we now propose another approach to the interoperability problem, which consists in using CORBA (along with the standard GIOP protocol suite) as the underlying object-oriented distribution mechanism for a DSA implementation. The next section details this approach.

4 A generic middleware specification

In section 2, we have presented two object-oriented distribution middlewares: CORBA and DSA. We have described their respective features in the areas of description of services, communication mechanisms, and ability to interoperate. We have then introduced CIAO, a first approach at integrating both environments.

We now propose an altogether different take at the issue of interoperating distributed object platforms. This second approach consists in defining an architecture for a generic object-oriented distribution middleware. In section 4.1, we give an overview of this generic architecture. In 4.3, we discuss the motivation and benefits of designing customisability into this architecture.

4.1 Recurring services in distributed object-oriented applications

In this section, we present a generic description of the functions that one can expect to find in a distributed OO platform; we thus provide an architectural model for such a platform, and we show how CORBA and DSA can be thought of as instances of this model.

In this paper we restrict ourselves to those middlewares that support the distributed object paradigm. Other distribution models, such as plain RPC or Message Passing, can be implemented in terms of distributed object oriented primitives. An abstract model of the distributed OO architecture thus notionally encompasses those other architectures.

In the following subsections, we present what services are provided in all platforms. A similar classification of functionalities has been described in [29].

Basic services offered by distribution middlewares constitute an abstraction layer over operating system communication facilities. This layer has to preserve the fundamental properties of the object oriented model: addressing (embodying *identity of objects*), exchange of requests (transport, data representation, request protocol, embodying *method calls*), and resource management (activation of objects, requests dispatching, embodying *objects life cycle*).

We now describe these functional areas in greater detail:

Addressing

Definition Each object needs to be assigned an address (or *reference*). An address is an piece of information which can be passed from node to node, and denotes one particular object unambiguously all over its existence.

Properties An address must be a *global* identifier for an object. Addresses need to designate an object unambiguously: they must be globally *unique*.

Rationale One of the fundamental properties of objects is *identity*. Addresses embody that property by allowing nodes to manipulate these object identities.

Example An open addressing scheme that satifies these properties can be established using TSAP (Transport Service Access Point [10]) addresses from the underlying networking environment, associated with a locally unique identifier attributed by the middleware. This is the solution retained in CORBA.

Other addressing schemes can be established, provided they satisfy those conditions. Using TSAP addresses has the advantage that no global mapping of identifiers to network addresses has to be maintained.

Transport

Definition A node must be able to establish a communication link to an object. This link is used to transmit requests for the execution of object methods.

Properties Messages must be reliably delivered.

Rationale Invoking an object method in the object-oriented programming model consists in *sending a message* to the object [9]. Distribution fits nicely in this model: the corresponding message goes through a communication network.

Example Existing message-passing libraries can be reused. This method has already been successfully applied to reimplement the Java/RMI platform over existing distributed processing libraries. [18] presents KaRMI, a drop-in replacement for Sun's RMI, which features an abstraction for the message passing layer. This abstraction can make use of TCP/IP sockets as well as specialised message-passing hardware in a parallel environment; KaRMI also has the ability to create bridge objects between different technology domains on-the-fly.

Marshalling

Definition Request data must be translated into a representation suitable for transmission over a network.

Properties In order to build interoperable applications, this representation has to be previously standardized. For instance, the middleware at the calling and receiving ends of the communication channel have to agree on the size and endianness of integers.

Rationale System representation choices are a tradeoff between portability and cost of data conversion from machine representation to network representation (and back).

Example It is necessary to find a balance between platform related representation (constraints) and application related representation (that can be optimized for a given environemnt). For example, a common data representation may be specified for all data types, while messages may still carry an endianness flag which indicates whether data items are stored low-order byte first or high-order byte first [20]. This provides a reasonably simple and compact representation of information, while alleviating the cost of "byte-swapping" operations. As these operations involve a significant portion of the critical path in request marshalling, this optimisation is greatly beneficial to performance.

Protocol

Definition The middleware implements a protocol for the transmission of requests among instances of distribution runtimes.

Properties The fundamental primitives of a distributed objects protocol are *Request* and *Reply*, which are used to implement remote invocation of subprograms and methods. Abortion of pending requests can also be provided by protocol primitives. Further primitives may be found in particular platforms; these are used to perform various high-level functions, and can be functionnally understood as requests sent to objects that are part of the platform implementation.

Example CORBA defines a Generic Inter-ORB Protocol for this purpose.

Activation

Definition The middleware has to activate and deactivate the concrete entities implementing objects. When a request is received, it ensures that the object reference is mapped onto an actual object implementation. The middleware may have to create an object implementation on-the-fly, or to retrieve an object state from persistent storage.

Properties Each request must be processed within the correct context (including object state and identity, history of preceding requests, etc.)

Example The Portable Object Adaptor of CORBA provides several policies on various issues (automatic activation, system or user id creation, thread creation per-request or per-object, ...)

Dispatching

Definition When a request reaches a node, it has to be assigned onto an execution ressource (a thread of control) for processing. The distribution runtime has to find a suitable thread on the node, or create one if necessary.

Properties The dispatching mechanism should not introduce dead locks. If possible, it should process requests fairly; request prioritization support may also be provided.

Rationale Several policies for deciding when to create a thread, and on which thread to dispatch a given request, can be defined. The choice of a particular threading policy is discussed in [26].

Example `GLADE` implements a pool of dynamically created threads. Most incoming requests can thus be served immediately, while keeping the memory and context switching footprint of the runtime within limits [24].

The previously listed services are sufficient to provide the core functionalities of a distribution middleware. They are built atop the operating system communication facilities and provide distributed object-oriented abstractions.

However, applications often require support for more sophisticated functions related to distribution. These functions include:

Naming

Definition The Naming service allows object references to be associated with symbolic names, and nodes to query the service for any reference associated with a name.

Rationale This service lets applications determine the references of any object they need at run-time. It is essential when an application has to adapt to environment modifications, to changes in technonoly or to the evolution of user requirements. They allow developers to avoid using hard-coded object references, and thus make the entire distributed application reconfigurable.

Example DSA has an implicit naming service: all nodes can invoke operations defined by units categorized as *Remote Call Interfaces*. The runtime takes care of locating on which node the named unit resides. This service is also used to prevent multiple declarations of unique entities like RCI units or to check remote unit versions. For these reasons, the naming service has to be an internal entity of the `GLADE` communication system.

Synchronisation

Definition A synchronisation service may be provided to synchronise the operation of nodes executing actions in parallel.

Rationale The operation of the concurrent, independant processes that participate in a distributed application must be coordinated; most notably, care must be taken that concurrent access to shared resources occurs only in an orderly, consistent way. Distribution platforms may help addressing this issue by providing specialized concurrency support services, such as distributed mutual exclusion [17].

Interface Repository

Definition An application may need to invoke operations on objects whose interface was not known when the application was created. For this invocation to be performed correctly, a description of the interface must be available. The purpose of Interface Repository Services is to propagate such information: they distribute service descriptions to interested nodes.

Properties The abstractions manipulated by the Interface Repository service must reflect the platform's service model.

A dynamic invocation mechanism is necessary to use the information provided by an interface repository. This facility enables nodes to invoke dynamically discovered methods. A dynamic invocation interface reifies the process of creating a remote method invocation request. After the request object is constructed, it can be processed like any ordinary method invocation on a remote object; the receiving object cannot distinguish dynamically-constructed requests from statically-created ones.

Example CORBA defines an Interface Repository ORB service and an associated Dynamic Invocation Interface.

Termination

Definition A distributed application consists of a set of computing nodes that cooperate to reach a goal. If, at some point in time, all nodes agree that this goal has been reached, they may decide to globally terminate processing and perform an orderly shutdown. A Termination service can be implemented to establish this decision.

Properties Such a service must determine a consensus among participating nodes on whether to end the application. The Termination service provides a support for applications to make this decision. Implementation of this service requires specific support from the ORB in addition to the low-level services listed above. Specifically, the implementation of the Termination service needs some way to inspect the state of the ORB's internal threads of control.

Example `GLADE` provides a global termination algorithm for DSA. If all nodes have terminated their own work, and no message is in transit in the communication network, it can be proven that no new computing work can happen [15]. In that case, global termination can be decided.

The user can select for each partition a local or global termination policy [24].

Shared data

Definition Nodes in a distributed application may want to share part of their data. A shared data service provides shared storage transparently for the application developer: distributed shared data appears just as normal local data; the distribution runtime takes care of propagating the effects of data access between nodes.

Properties A shared storage service must provide all nodes with a consistent view of the shared variables. Several models of consistency can be defined.

Rationale This service provides developers with a very straightforward way to define a set of globally available data.

Example DSA offers *Shared Passive* data: library units containing only variables, which can be accessed consistently from any node in a distributed application. `GLADE` provides several shared data supports like file or DVSM [14].

4.2 Example

Let us suppose that we have a very simple object with a single method *echo* taking a string argument, and sending that string back to the caller. Invoking this method involves the following steps:

Addressing The caller determines a reference to the object.

Transport The caller-side ORB extracts a network address from the reference and creates a connexion to the object's ORB.

Marshalling It converts the string passed to the method into a representation suitable for network transport, for example a four-byte, little-endian integer containing the length of the string, followed by the string itself.

Protocol It creates a *Request* message containing the name of the method, the object reference, and the marshalled argument, along with a unique request ID. It sends the message across the transport link to the callee ORB.

The callee ORB receives the message.

Activation The callee ORB looks up the object reference in its internal tables, and associates it with an implementation object.

Dispatching The callee ORB finds an idle thread within its available thread pool, and assigns the execution of the request onto that thread.

Marshalling The callee ORB marshalls the echoed string as above.

Protocol It creates a *Reply* message containing the request ID and the marshalled result string. The reply is sent back to the caller.

The caller ORB unmasrshalls the result and returns it to the calling routine.

In the following two sections, we present several approaches to the problem of interoperating different realisations of this abstract description.

4.3 Towards a generic customizable middleware

We have given a general description of services required to operate a distribution platform based on the *Distributed objects* paradigm. We have layered these services in two abstraction levels. The lower layer consists in the core infrastruture offering the distributed objects model. The higher layer comprises services that are built upon that model, enabling users to efficiently use it, and to control some aspects of its operation.

We have defined a generic model of object-oriented distribution middleware. We now consider this model as a specification for a set of generic components that can be used to build an interoperable distribution runtime library. We may then be able to tailor the distribution runtime according to three criteria.

Environment parameters. A customizable middleware can support multiple application *profiles*. A profile is a proper subset of available functions which is necessary for an application. By omitting services and functionalities that are not required in a particular situation, one can create a "light" middleware with a smaller resource footprint than a full-featured one.

Light profiles where some language functions are prohibited help to improve formal provability of a system's properties. This is essential for many critical, embedded applications. Such a light profile, called the *Ravenscar profile*, has been defined for monolithic Ada 95 programs [7], and work is in progress to extend it to include a light version of DSA. Similar proposals have also been made for a subset of CORBA suitable for embedded applications [21].

Application requirements. Customization of these generic components ensures suitability for the particular requirements of an application. [6, 29] show that customization of distribution middleware to exploit specific features of the underlying environment and hardware architectures helps improving the overall performances of distributed systems.

Such an improvement is of interest when an application requires several objects to be physically located in the same computing node. Network costs may be optimized out by the use of efficient local inter-process communication facility (e. g. shared memory). A similar optimization mechanism exists in `GLADE`; building it into a generic middleware architecture will extend its benefits to CORBA.

Interoperability considerations. Using generic components wherever it is possible in the implementation of a distributed application ensures that as many aspects of the application as possible will operate in a standard manner, and will integrate smoothly in a consistent distribution framework.

Furthermore, we seek to propose a modular design separating support of various platform "personalities" (CORBA, DSA, RMIs) from core object management. This scheme will allow objects to be accessed simultaneously from different platforms.

As a testbed for this generic middleware approach, we intend to specify a generic distribution middleware. This specification will be a refinement of the general description of necessary facilities presented in section 4.1. We will implement it, and attempt to show its fitness as a framework for the implementation of various distribution platforms.

Our first targets will be CORBA and Ada 95 DSA. Implementing DSA using a CORBA ORB as the core distribution middleware implies that servers and objects created using DSA also exist as CORBA objects. They will thus be available to CORBA clients immediately, without having to establish any gateway between incompatible worlds.

DSA will thus gain the interoperability and openness to other languages that it currently lacks. This is similar to OMG's proposal of using CORBA as a way to implement Java RMI [22].

4.4 Related projects

One objective of the Droopi project is to show that the Open Management Architecture (OMA [19]) is suitable to serve as a paradigm for interoperating various object-oriented distribution platforms. While some of the richer semantics of specific models may not be reflected by OMA abstractions, we claim that those aspects of a distributed system that *can* be described within the OMA are sufficient to make it interoperable.

For example, DSA's strong typing constraints (which stems from Ada 95's one) cannot be expressed in the current definition of OMG IDL. However, the domains covered by the CORBA specifications are sufficient to describe the mechanisms that are to be used by a middleware to perform a method invocation on an object: they provide sufficient information for another middleware to *interoperate*.

Quarterware [28] also adopts a CORBA-like model to offer a component-based middleware. Although the *QuarterWare* framework can be specialized into implementations of several distribution platforms (CORBA, Java/RMI), it is not focused on the interoperability of applications across platforms.

The *Jonathan* [8] distributed processing environment for the Java Virtual machine offers the possibility to integrate different types of bindings between objects in an application. Jonathan supports a CORBA and an RMI personality using a generic distribution kernel. Our work will share some features of Jonathan's modular and decoupled design. However, we shall focus our own contributions on customizability of all ORB components for a variety of application requirements. Specifically, we intend to define a light ORB profile specification and implementation to be used in real-time, critical embedded systems.

5 Conclusion

This paper presented our experience in bridging CORBA and Ada 95 DSA to take advantage of both facilities: interoperability and enhanced parallel model.

A first approach was investigated with CIAO: an Ada to CORBA translator that enable distributed application developers to invoke DSA services with CORBA clients. CIAO is operational but we have experimented several drawbacks due to the approach.

We have then proposed another way to bridge DSA and CORBA via the definition of a generic middleware that could be specialized. This is a way to increase the reusability of some parts of the code that are similar on various distributed environemnts. This code is then instanciated according to the contexte (i.e. the personnality of the customized middlware).

An interesting consequence should be to ease the interoperability between the distribution models of CORBA and DSA. We think it should be also possible to enable leveraging with other distributed middleware

References

[1] F. Azavant, J.-M. Cottin, V. Niebel, L. Pautet, S. Ponce, T. Quinot, and S. Tardieu. CORBA and CORBA Services for DSA. In *ACM SigAda'99*, Oct. 1999.

[2] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[3] G. Bracha and W. Cook. Mixin-based Inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 25, pages 303–311, New York, NY, Oct. 1990. ACM Press.

[4] E. Briot. JGNAT: The GNAT Ada 95 environment for the JVM. In *Ada France 1999*, Sept. 1999.

[5] G. Brose, K.-P. Löhr, and A. Spiegel. Java Does not Distribute. In *TOOLS Pacific '97*, Nov. 1997.

[6] R. H. Campbell, J. W. Coomes, A. Dave, N. Islam, Y. Li, W. S. Liao, S. Lim, T. Qian, D. K. Raila, E. Roush, A. Sane, M. Sefika, A. Singhai, and S. T. Tan. Customizable Object-Oriented Operating Systems. Submitted for *Communications of the ACM* Special Issue on Recent Developments in Operating Systems, Sept. 1996.

[7] B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of SigAda'98*, Washington, DC, USA, Nov. 1998.

[8] B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stéfani. Jonathan: an Open Distributed Processing Environment in Java. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer Verlag, Sept. 1998.

[9] D. H. H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 9–16, Jan. 1978.

[10] ISO. *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO, Feb. 1995. ISO/IEC 7498-1:1994.

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

[12] ISO. *Information Technology – Programming Languages – Ada Semantic Interface Specification (ASIS)*. ISO, 1998. ISO/IEC 15291:1998.

[13] ISO. *Information technology – Programming languages – Ada: Conformity assessment of a language processor*. ISO, 1999. ISO/IEC 18009:1999.

[14] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[15] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2(3):161–175, 1987.

[16] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall International, 1988.

[17] M. Naimi, M. Tréhel, and A. Arnold. A Log(N) Distributed Mutual Exclusion Algorithm Based on the Path Reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, Apr. 1996.

[18] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *ACM 1999 Java Grande Conference*, pages 153–159. ACM, June 1999.

[19] Object Management Group. *Discussion of the Object Management Architecture*. Object Management Group, Jan. 1997. OMG formal/00-06-41.

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

[21] Object Management Group. *minimumCORBA*. Object Management Group, Aug. 1998. OMG orbos/98-08-04.

[22] Object Management Group. *Java Language to IDL Mapping*. Object Management Group, Jan. 2000. OMG ptc/00-01-06.

[23] L. Pautet, T. Quinot, and S. Tardieu. CORBA & DSA: Divorce or Marriage? In *Proceedings of the 1999 Ada-Europe International Conference on Reliable Software Technologies*, Santander, Spain, June 1999.

[24] L. Pautet and S. Tardieu. *GLADE User Guide*.

[25] T. Quinot. CIAO: Opening the Ada 95 Distributed Systems Annex to CORBA Clients. In *Ada France 1999*, Sept. 1999.

[26] D. C. Schmidt and S. Vinoski. Comparing Alternative Server Distributed Programming Techniques. *SIGS C++ Report*, 7(8), Oct. 1995.

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

[28] A. Singhai. *QuarterWare: A middleware toolkit of software RISC components.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[29] A. Singhai, A. Sane, and R. H. Campbell. Quarterware for Middleware. In *Proceedings of ICDCS'98*. IEEE, May 1998.

[30] Sun Microsystems, Inc. *RMI – Documentation.*