GLADE: a Framework for Building Large Object-Oriented Real-time Distributed Systems

Laurent PAUTET Samuel TARDIEU

{pautet,tardieu}@enst.fr

École Nationale Supérieure des Télécommunications
Computer Science and Networks Departement
46, rue Barrault
F-75634 Paris CEDEX 13, France

Abstract

This paper describes how GLADE, our implementation of the Ada 95 Distributed Systems Annex, can be used to build large object-oriented real-time distributed systems. In addition to the powerful distribution features included in the Ada 95 language itself, we provide extensions to help the programmer build robust and failsafe distributed applications.

1. Introduction

It is generally admitted that Ada 83 had a strong focus on real-time, mission-critical systems. But Ada 83 has been criticized from a number of standpoints, one of them being its lack of cooperation with foreign programming languages and with the outside world in general. To fix those defects, new features were added in the latest major revision of Ada, called Ada 95 [8]. Moreover, Ada 95 is now the first standardized object-oriented language (ANSI/ISO/IEC-8652:1995). It is also the first standardized language including distribution features.

A great effort was led by Ada Core Technologies (ACT) to provide the Ada community with a free high-quality Ada 95 compiler called GNAT. This compiler, which implements the core Ada 95 language as well as all the optional annexes, belongs to the GCC family and shares its back end with the C and C++ compilers. As the other compilers from the GCC suite, GNAT supports many native and cross configurations.

In collaboration with ACT, we have been developing GLADE, an implementation of the Distributed Systems Annex of the Ada 95 reference manual. GLADE, which is available under the same free license as GNAT, has been designed for this particular compiler, but should be portable to any Ada 95 compilation environment with minimal efforts. We also worked on proposing new extensions to the original Ada 95 model for distributing large programs. Some of those extensions have been integrated in GLADE already as they do not change the syntax or the semantics of the language; some others require in-depth changes of the language, and will be submitted as formal proposals to the committee in charge of the next Ada revision.

Section 2 of this paper highlights the main features of the Distributed Systems Annex. Section 3 describes GLADE, our implementation of this annex. Sections 4 and 5 respectively contain the real-time and fault-tolerance extensions that we have designed and implemented. Finally, section 6 will briefly describe the current work in progress.

2. Ada 95: the perfect language for building large OO real-time distributed systems

The core Ada 95 language contains high-level features available in most modern programming languages such as inheritance and polymorphism. It also defines not-to-be-found-everywhere paradigms such as data-driven synchronization or hierarchical library units.

A validated Ada 95 implementation must support the core programming language, and 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, system programming or distributed systems. Each of these annexes has its own test

suite used for compiler validation.

2.1. Real-time features

Ada 83 already provided a number of tasking features in the core language. New features have been introduced in Ada 95 either in the core language or in the specialized "Real-Time Systems" Annex:

- protected objects, asynchronous transfer of control and requeuing facilities for task entries (core language);
- dynamic priorities and asynchronous task control (real-time systems annex).

A protected object is similar to a conditional critical region [4] and a monitor [7]. Data encapsulated in a protected object is accessed in mutual exclusion. Only procedures, functions and entries can access this data. An entry is a subprogram with an associated guard; a call to a guarded entry is allowed if the guard evaluates to true (see sample 1).

```
procedure PO_Example is

   protected type Semaphore
     (How_Many : Natural := 1)
   is
      entry P;
      --  Seize a resource
      procedure V;
      --  Release a shared resource
   private
      Available : Natural := How_Many;
   end Semaphore;

   protected body Semaphore is

      entry P when Available > 0 is
      begin
         Available := Available - 1;
      end P;

      procedure V is
      begin
         Available := Available + 1;
      end V;

   end Semaphore;

   S : Semaphore (2);  --  Two resources

begin
   [...]
   --  Try to get a resource for 1 minute
   select
      S.P;
   then abort
      delay 60.0;
      raise Unable_To_Seize_Resource;
   end select;
end PO_Example;
```

Sample 1. Protected object example

This example also illustrates timed/conditional entry calls, where a tentative call is made and gets aborted if it is not served before another condition arises (in our case the expiration of a delay).

2.2. Object-oriented features

Ada 83 already had the basic object-oriented features such as abstraction, encapsulation, information-hiding, and modularity. Ada 95 also includes modern data constructions such as inheritance and polymorphism [17].

Types may be derived from and optionally extended as soon as they are marked as "tagged". A class designates a tagged type (the root of the inheritance tree) and all the derived types. All the primitive operations of a type (that is, subprograms declared in the same scope as the type) are inherited by its children, unless explicitly overridden. Sample 2 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 well-known techniques such as sibling or mix-in inheritance [2].

```
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 2. Object-oriented example

2.3. Distribution features

The Distributed Systems Annex provides the developer with a standardized way of partitioning an Ada application across a network of computers. Communication between the various partitions takes place using remote subprogram calls or method calls on remote objects.

The remotely-called subprograms may be statically or dynamically bound, thereby allowing applications to use either the classical remote procedure call paradigm [3] (see sample 3) or the increasingly popular distributed object paradigm [12, 13] (see sample 4). High-level semantics of the language are preserved when using the Distributed Systems Annex; for example, exception propagation works as usual and timed subprogram calls can be used as if the program was not distributed.

```
package RCI is
   pragma Remote_Call_Interface;

   procedure Remote_Subprogram
     (Parameter : in out Integer);

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

end RCI;
```

Sample 3. Remote subprograms

```
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 4. Remote objects

Ada incorporates facilities for programming distributed systems as a consistent and systematic extension to those provided for programming non-distributed systems. Thus, the benefits of a type-safe object-oriented programming language supporting both data-driven synchronization and concurrency are made available for programming distributed systems.

Some packages play a special role in distributed systems; they are identified by means of categorization pragmas, which describe the category each package belongs to. A categorized package has to follow some extra 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). In addition to uncategorized packages (also called "normal packages") three categories are defined by the Distributed Systems Annex:

- A package of category "Remote_Call_Interface" contains subprograms that can be called remotely. Such a package can be present only once in a distributed application, unless transparently replicated.
- A package of category "Remote_Types" defines new types with global semantics. For example, a linked list type can be defined in such a package, provided that additional marshaling and unmarshaling operations are given by the programmer.
- A package of category "Shared_Passive" contains declarations of objects that will be shared between several partitions. This category will not be detailed in this paper since it deals with shared objects, opposed to distributed objects.

Two other pragmas complete these categorizations. When applied to a subprogram declared in a "Remote_Call_Interface" package, the "Asynchronous" pragma allows the caller to proceed immediately without waiting for the completion of the remote subprogram. Another pragma, "All_Calls_Remote", forces all the calls, even local, to go through the communication layers.

The Distributed Systems Annex 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.

2.4. 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.

3. GLADE: distributed systems for GNAT

GLADE, our implementation of the Distributed Systems Annex, provides a configuration tool (GNATDIST) and a partition communication subsystem (GARLIC) that can be used to build a distributed application for a heterogeneous or homogeneous set of machines [18]. Both are described hereafter.

3.1. GNATDIST

The GNATDIST tool, through its configuration language, allows the programmer to partition his program and to take advantage of the numerous extensions that we propose.

GNATDIST reads a configuration file (whose syntax is close to Ada 95), checks the consistency of the distributed program and builds several executables, one for each partition. By default, it also takes care of launching the different partitions with parameters that can be specific to each partition. The principle of this tool is to ease the partitioning of the non-distributed application to build the distributed application. One can also define and build partitions independently from each other; this method is appropriate for building client/server applications.

GNATDIST takes care of generating the stubs and skeletons for units holding a categorization pragma. To create a partition, GNATDIST creates an executable which includes the skeletons of the programmer units assigned to this partition, the stubs needed by the transitive closure as well as the communication subsystem. It also elaborates internal GLADE services explicitly configured by the programmer.

3.2. GARLIC

GARLIC is in some respects the ORB of the Distributed Systems Annex (as shown on figure 1). It includes several internal name and location servers. For instance, each Remote_Call_Interface unit is registered in one of these servers in order to resolve static references (as the COS[1] Naming Service would do in CORBA).

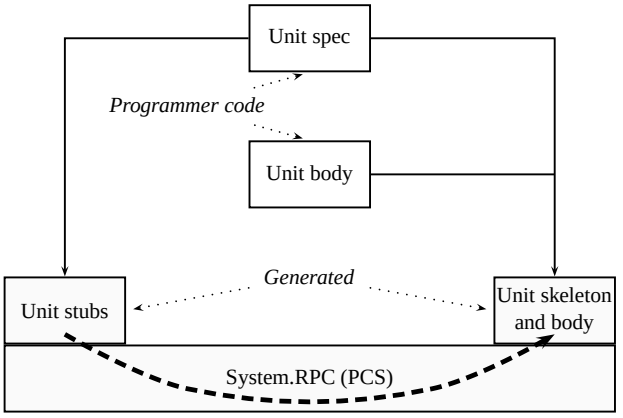

Figure 1. PCS viewed as an ORB

Although the Distributed Systems Annex does not require an Ada 95 compiler to work in a heterogeneous system, GLADE has been especially designed to handle heterogeneity. Of course, the developer can switch off this feature at configuration time if he cannot afford the translation overhead. For the same reason, the classical file operations are still available in an heterogeneous system (the developer can write an object in a file on a 64-bit machine and read it on a 32-bit machine).

4. Extensions to GLADE for real-time systems

GLADE over Ethernet is well-suited for soft real-time systems only; unpredictable network performances make it impossible to develop distributed hard real-time systems. However, using GLADE on multiprocessor machines makes it easy to develop hard real-time applications using dedicated Ada constructs.

When developing the partition communication subsystem, we have tried very hard to keep in mind the possible real-time aspects of the user application. Let us consider two cases.

[1] Common Object Services

4.1. Asynchronous transfer of control

Sample 5 shows a method call, `Action (Object)`, that is limited in time to 10 seconds. After that, if `Action` has not returned, it gets aborted (the code between `then abort` and `end select` is called the *abortable part*) and `Action_Has_Failed` is executed.

```
procedure SelectAbort is
begin
   select
      delay 10.0;
      Action_Has_Failed;
   then abort
      Action (Object);
   end select;
end SelectAbort;
```

Sample 5. Using timed calls

When this code belongs to a distributed application, and partition $\mathcal{A}$ calls `Action` on `Object` located on partition $\mathcal{B}$, the method call `Action (Object)` is a remote subprogram call. If, for any reason, this method does not terminate in 10 seconds, an asynchronous cancellation message is sent from partition $\mathcal{A}$ to partition $\mathcal{B}$. Upon cancellation message reception, partition $\mathcal{B}$ stops the job in progress, in order to not consume useless processing power.

This construction can be used to switch computations from one node to another when the first one is so saturated that it cannot answer timely requests anymore. The code shown on sample 6 illustrates this concept: if the method call on `Object1` cannot be completed within 10 seconds, the computation is aborted asynchronously using a control message and a new computation is started on `Object2`.

```
procedure SelectObject is
begin
   select
      delay 10.0;
      Action (Object2);
   then abort
      Action (Object1);
   end select;
end SelectObject;
```

Sample 6. Switching computation objects

4.2. Priority transmission

Each Ada partition has its own priority range. Each task in a partition has a priority within this range; at any time, no task is ever running if another task with a higher priority is ready to run but not yet bound to a processor.

Partitions running on homogeneous architectures will use the same priority ranges. However, an Ada distributed system may be made of many computers, some of them being SPARC Stations, some of them being PCs running Windows, and some others being embedded boards with a network interface. Those different architectures will run different Ada executives, with possibly different priority ranges.

The behavior on an incoming method call with respect to the rest of the partition is not defined in the reference manual. In `GLADE`, we have chosen to implement, optionally, a mapping between the caller's priority and the callee's priority, the callee being the task which executes the code of the incoming method. Figure 2 illustrates this: the priority Prio_P is brought in the range $[0..255]$, transmitted with the remote method call over the network, then mapped into the receiver's priority range to become priority $\text{Prio}_{P'}$. Prio_P and $\text{Prio}_{P'}$ are at the same level relative to their respective priority ranges. This concept is the same as the Real-Time CORBA specification of mapping priorities between different kinds of partitions.

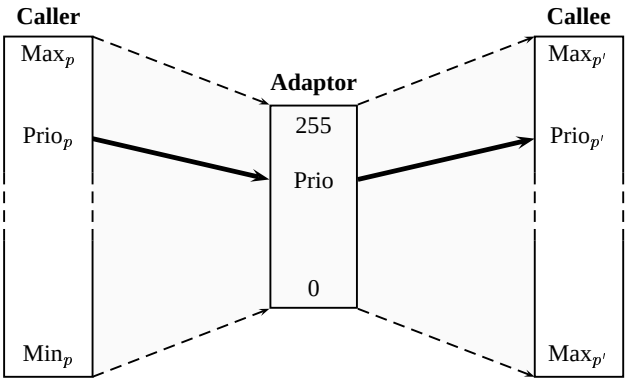

Figure 2. Interpartition priority mapping

4.3. Lightweight executive

The Distributed Systems Annex makes no difference between client and server partitions. However, it is common to have partitions that do not offer any service; those partitions will not export locally created objects and will not accept incoming remote calls.

A client only partition needs not be able to handle incoming messages but the answers to its own queries. In this case, there is no need to embed a complex executive which can deal with asynchronous requests. A simple "send then listen" scheme, along with a primitive way of locating other services is enough. Moreover, if the client only partition does not use any tasking construct on its own, the data structures dedicated to distribution need not be protected against concurrent accesses.

To reduce the memory footprint and CPU usage of those client only partitions, `GLADE` offers a special lightweight

executive. The choice of the executive is made by the binder, which is the part of the compilation chain that deals with elaboration order and closure issues. As soon as the partition does not publish any object and does not use any tasking construct, the binder chooses the light run-time. If one of those conditions is violated, the regular executive will be included instead, offering the full power (and, unfortunately, the full weight) of the Distributed Systems Annex.

4.4. Anonymous task pool

When multiple remote subprogram calls occur on the same partition, they are handled by several anonymous tasks. In a real-time context, it can be useful to control the number of anonymous tasks created to handle remote calls. These tasks can be allocated dynamically or re-used from a pool of (preallocated) tasks. When a remote subprogram call is completed, the anonymous task can be deallocated or queued in a pool in order to be re-used for further remote subprogram calls. The number of tasks in the anonymous tasks pool can be configured by means of three independent parameters.

The task pool minimum size indicates the minimum number of existing anonymous tasks available for the communication subsystem. Preallocating anonymous tasks can be useful in real-time systems to prevent dynamic task allocation.

The task pool high size is a ceiling. When a remote subprogram call is completed, its anonymous task is deallocated if the number of tasks already in the pool is greater than the ceiling. If not, then the task is queued in the pool.

The task pool maximum size indicates the maximum number of anonymous tasks in the communication subsystem. In other words, it provides a way to limit the number of remote calls in the partition. When a RPC request is received, if the number of active remote calls is greater than the task pool maximum size, then the request is kept pending until an anonymous task completes its own remote call and becomes available.

A maximum task pool size of 1 will allow only one active remote call at a time, thus enforcing serialization of incoming calls. This makes the partition act as a task accepting a remote rendez-vous request [6].

5. Extensions for surviving crashes

5.1. Fault-tolerant communications subsystem

When a partition starts its execution, one of the first elaboration steps is a registration with the boot partition which includes the partition id (short for “identifier”) server and with the Remote_Call_Interface name server.

The partition id server is used to allocate a unique partition id when a new partition registers. It also replies to information queries from other partitions. This information includes the IP address, the port on which the partition is waiting for requests and all its configuration parameters (termination and reconnection policies, filters, ...).

The Remote_Call_Interface name server is used to register newly elaborated Remote_Call_Interface packages. This Remote_Call_Interface package registration occurs once the partition has received its partition id. The partition registers its Remote_Call_Interface and Shared_Passive packages with their names, their version numbers and internal information.

These two servers are located on a boot partition. In some respects, GLADE has its own internal name servers when CORBA requires the help of the programmer who is supposed to register her well-known reference manually.

For fault-tolerance issues, it can be critical to prevent the whole distributed system from blocking when the boot partition maintaining those servers dies. The boot partition can be replicated on boot mirrors, in order to prevent this partition from being a single point of failure. A partition has always to connect to the boot partition or to a boot mirror in order to get minimal information about the other partitions.

The boot partition is the first boot mirror of the distributed system. A new partition declared as a boot mirror joins the group of boot mirrors. The group of boot mirrors operates as a token ring: any request from a new partition to a boot mirror is sent on the ring using a token. A request can go over the ring up to two times before being approved by the whole set of boot mirrors.

When the boot partition dies, a new boot partition is elected among the remaining boot mirrors. A boot partition is responsible for the global termination detection. That is why a new boot partition has to be elected.

This group is based on an internal token ring which ensures that each member of the group can replace the boot server partition when it dies. It can be compared to a fault-tolerant COS Naming Service except that the programmer does not have to deal with it manually or implement fault-tolerant algorithms. Dedicated algorithms preserve the global consistency of this group by taking care of message loss detection and retransmission and by computing a consistent global state [10, 18].

5.2. Restarting a partition

On the one hand, a partition holding no Remote_Call_Interface unit can be freely replicated in a distributed application. On the other hand, a partition with Remote_Call_Interface packages cannot be present more than once. If this partition was to be launched repeatedly, it would not be possible to decide which instance should

handle an incoming remote call.

When a partition crashes or gets stopped, all the partitions that knew about it mark this partition as dead. In some cases, the developer may want to start an identical partition that will offer the same services.

If configured with the Reject_On_Restart reconnection policy, a dead partition is kept dead and any attempt to restart it fails. Any remote call to a subprogram located on this partition results in a system exception.

If configured with the Fail_Until_Restart reconnection policy, a dead partition can be restarted. Any remote call to a subprogram located on this partition results in a system exception as long as this partition has not been restarted. As soon as the partition is restarted, remote calls to this partition are executed normally.

If configured with the Wait_Until_Restart reconnection policy, a dead partition can be restarted. Any remote call to a subprogram located on this partition is kept pending until the partition is restarted. As soon as the partition is restarted, remote calls to this partition are executed normally. The suspended remote procedure calls to this partition are resumed.

Note that with the last mechanism, the programmer may have to handle a communication error anyway because the server failure may occur in the middle of a remote invocation. The Communication Subsystem cannot afford to handle the communication error in order to retry the invocation later, because of the *at most once* semantics of a remote call in Ada 95.

6. Work in progress

Some research and development works are currently underway in two major directions: GLADE improvements according to specified profiles (including hard real-time), and GLADE improvements using ideas taken from the CORBA world.

6.1. Profiles for Ada 95 distributed systems

To ensure that a real-time system meets its deadlines, the programmer may want to check that the design and the implementation of her application is feasible. To do so, restricting the computational model can ease the job of the developer. For this reason, Ada 95 allows to define profiles. A profile is a simple subset of Ada 95 that ensures efficiency and high integrity. Such a profile is checked at compilation time by the compiler using restriction pragmas. Typically, the programmer can forbid features with high overheads (dynamic resolution of dispatching), forbid synchronization features to simplify the tasking run-time system (small underlying kernels) or forbid features that would prevent the application from being deterministic, predictable and efficient. Hard real-time working groups have defined such profiles (such as the Ravenscar one [5]) and compiler vendors already provide them.

These profiles do not address distribution issues. But we are already thinking of some useful restrictions. For example, a possibility would be to forbid synchronous remote subprogram calls and to accept only asynchronous remote calls. The execution time of a synchronous remote call may not be predictable. For this reason, asynchronous remote calls could be a better compromise and would provide a higher level of functionality compared to bare message passing. This restriction has to be introduced at the compilation level.

6.2. Improvements coming from the CORBA world

We are actively looking at CORBA experience returns in order to incorporate the best ideas of CORBA in GLADE. Currently, we have identified three points where the CORBA model would be a benefit to GLADE programmers: general purpose or specialized services, dynamic invocation and interaction with CORBA objects.

CORBA-like services

CORBA standardized services are an elegant way of offering extra capabilities to the application developer without encumbering CORBA itself with seldom used services. Although the Ada standard contains several predefined libraries (e.g., for doing input/output services), there exist no predefined service to be used in a distributed application.

We are in the process of porting the most useful CORBA services to distributed Ada. At this time, we already achieved the translation of the naming service and the event service. We also chose a more powerful way of implementing the concurrency service by a fully distributed mutual exclusion service [11]. The next services on our port list are the transaction and the lifecycle service.

Dynamic invocation

The CORBA standard defines a way of performing dynamic invocation on remote objects. This method consists of building a method call *by hand*, by pushing all the arguments in order to make a request. An additional service, called the Interface Repository, allows an application to query an object about its interface (this property is usually called *reflexivity*).

In Ada 95, reflexivity is achieved using ASIS[2] [9]: a legal Ada compilation unit can be traversed, and syntactical and semantical information can be obtained for every node

[2] Ada Semantic Interface Specification

seen. Using this standardized interface, it is possible to provide a remote node with the full description of an object interface. We plan to utilize this capability through a yet-to-be-defined API in order to build requests targeting an object whose static and dynamic types are unknown at compilation and link time.

This feature will ease the process of adding new services in an ever running application without the need to stop any partition. This would bring a hot upgrade framework, as the one found in Erlang [1]: system downtimes would be limited to hardware maintenance or upgrades only. A software upgrade would require no service interruption.

Addressing Ada objects from CORBA

CORBA has been very successful in the last few years and has been widely deployed. This leads to a situation where it is much easier to provide a team with an IDL specification than to convince them to convert their existing application to distributed Ada. We have thus developed a software called CIAO[3] that automatically generates IDL specifications from a set of Ada specifications [14, 15, 16]. This tool also builds an implementation corresponding to the IDL description without human intervention; this implementation acts as a proxy object that transforms every CORBA method call into Ada method calls.

7. Conclusion

Ada 95 was the first object-oriented language to be internationally standardized. The real-time features already found in Ada 83 have been nicely integrated with the object model and with the new concurrency constructs such as protected objects. Moreover, distribution capabilities have been included in the standard, and let a developer build distributed applications painlessly.

Many people were skeptic regarding the integration of those three major features. However, our implementation has shown that they mix very well, and that using Ada for developing large real-time object-oriented distributed systems leads to very elegant and straightforward solutions.

We have taken advantage of the freedom offered by the Distributed Systems Annex to add powerful extensions to the basic distribution model. Those extensions are either focused on pure Ada real-time systems or on interoperability with other object-oriented distributed systems such as CORBA.

[3] CORBA Interface for Ada Objects

References

[1] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[2] S. Barbey, M. Kempe, and A. Strohmeier. Object-oriented programming with Ada 9X. In *OOPSLA'93, Washington DC, USA, September 26 - October 1 1993*, number 18 in Tutorial Notes. ACM Press, 1993.

[3] A. Birrell and B. Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2(1):39–59, february 1984.

[4] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[5] B. Dobbing and A. Burns. The Ravenscar tasking profile for high-integrity real-time programs. *ACM SIGADA Ada Letters*, 18(6):1–6, Nov./Dec. 1998.

[6] N. D. Gammage, R. F. Kamel, and L. M. Casey. Remote rendezvous. *Software Practice and Experience*, 17(10):741–755, Oct. 1987.

[7] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct. 1974. Erratum in *Communications of the ACM*, Vol. 18, No. 2 (February), p. 95, 1975. This paper contains one of the first solutions to the Dining Philosophers problem.

[8] I. Intermetrics. *Ada 95 – Reference Manual*, 1995.

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

[10] G. Le Lann. Algorithms for distributed data sharing systems with use tickets. In *Proceedings of the 3rd Berkeley workshopon Distributed Data Management and Computer Networks*, pages 259–272, August 1978.

[11] K. Li and P. R. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, New York, NY, 1986. ACM.

[12] S. Microsystems. *RMI – Documentation*.

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

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

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

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

[17] A. Strohmeier, S. Barbey, and M. Kempe. Object-oriented programming and reuse in Ada 9X. In *Tutorials of TriAda'93*, volume 2, pages 945–984, Seattle, Washington, USA, September 1993.

[18] S. Tardieu. *GLADE – Une implémentation de l'annexe des systèmes répartis d'Ada 95*. PhD thesis, École Nationale Supérieure des Télécommunications, Oct. 1999.