What future for the Distributed Systems Annex?

Laurent PAUTET Samuel TARDIEU

{Laurent.Pautet,Samuel.Tardieu}@enst.fr
École Nationale Supérieure des Télécommunications
Département Informatique et Réseaux
46, rue Barrault
F-75634 Paris Cedex 13, France

Abstract

In this paper, we report our experience as implementors and users of the Ada 95 Distributed Systems Annex (annex E of the Ada reference manual). We identify the principal strengths and weaknesses of the annex, and make some proposals to improve it either immediately or for the next revision of the language (Ada0X). Our goal is to get an annex that is more open and compatible with other distributed systems such as CORBA, without loosing the capability of developing pure Ada rock solid distributed systems.

We assume that the reader is familiar with Ada. Knowledge of the Distributed Systems Annex is useful but not required to understand the main ideas exposed in this article.

1 The Distributed Systems Annex today

A few years ago, distributed systems were mainly used by teams with very special needs in terms of processing power or reliability, or for teaching the basis of distributed programming. Today, because of the dramatic growth of the Internet and the development of high speed networks, more and more people are becoming familiar with distributed computing. It has become quite common and well accepted to have a part of a computation done locally while the rest is being done on a server located far away. For example, the `SETI@home` project is dedicated to finding extraterrestrial intelligence signs in the universe. To achieve its goal, it uses a giant distributed system whose nodes are personal computers with spare CPU cycles that now look for patterns in a huge set of data collected by radio telescopes [4].

The architects of Ada 95 [13] had foreseen this increasing interest in distributed systems. They chose to add a Distributed Systems Annex (DSA in short) in the latest language revision [10]. This annex, while still fully consistent with the rest of the language, defines how subprograms can be called remotely, and how complex data structures such as pointers on remote objects and remote subprograms can be built and used. However, unlike foreign distributed architectures such as CORBA, those facilities preserve the strong type checking and the safety features of the Ada programming language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.
To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda'99 10/99 Redondo Beach, CA, USA
©1999 ACM 1-58113-127-5/99/0010...$5.00

1.1 Existing implementations

Two implementations of the DSA are available at the time of writing, targeting the freely available GNAT Ada compiler:

1. **GLADE** (GNAT Library for Ada Distributed Execution), developed and maintained jointly by the ENST[1] and by Ada Core Technologies[2]. This implementation is freely available under the same license as GNAT, and commercial support is available through Ada Core Technologies. It includes a partitioning tool called **gnatdist** [16], and a partition communication subsystem called **garlic** (GNAT Ada Reusable Library for Interpartition Communication) [17].

2. **ADEPT** (Ada 95 Distributed Execution and Partitioning Toolset) has originally started as a joint project between Computer Sciences Corporation[3], the Texas A&M university[4] and the ENST. It is based on an early implementation of GLADE [11], and has since then evolved into a bridge between Ada and RMI (see section 3.2). It is now maintained by the Texas A&M university.

1.2 Typical uses of the DSA

Since Ada is a general purpose programming language, distributed systems in Ada can be used potentially in every domain of computer science. This section describes some projects using the DSA; it is based on publicly available information available through public newsgroups and mailing lists.

1.2.1 Industry

GLADE has been successfully used by several major companies, both in the US and in Europe. Of course, the acceptance of the distributed features of the language is bound to the acceptance of Ada 95 itself, and many industrial companies are still using Ada 83, even when compiling with an Ada 95 compiler. However, new projects and major revisions of existing products are now using Ada 95 and its new concepts, such as tagged types and protected types. At the same time, they do evaluate the DSA to see whether it meets their needs in terms of distributed systems.

EDF, the most important electricity provider in France, has been building a prototype of a WWW session-tracker using the DSA. The goal is to keep a link between separate requests made by the same user to a WWW server. This company is also considering using the DSA as a caching proxy for database requests; if a request has already been performed in the past and if the database has not

[1]`http://www.enst.fr/`
[2]`http://www.gnat.com/` and `http://www.act-europe.fr/`
[3]`http://www.csc.com/`
[4]`http://www.tamu.edu/`

been updated in the meantime, the previously computed result will be sent back to the user, thus avoiding unnecessary processing on the database server side.

1.2.2 Military

Most military projects using distributed systems written in Ada had their own communication layer, around which the whole program was designed. This has been made unnecessary with the DSA, whose goal is to let the program architect design her distributed program almost without thinking at the distribution issues. Remote subprogram calls and distributed objects integrate very well with a monolithic design. DSA could be used only as a communication layer that would replace the existing one, but the gain will be too low for such a task. It is thus often difficult to fully move to the DSA without rethinking the whole program architecture. However, GLADE is starting to show up in new developments, in particular when distributed simulations are involved.

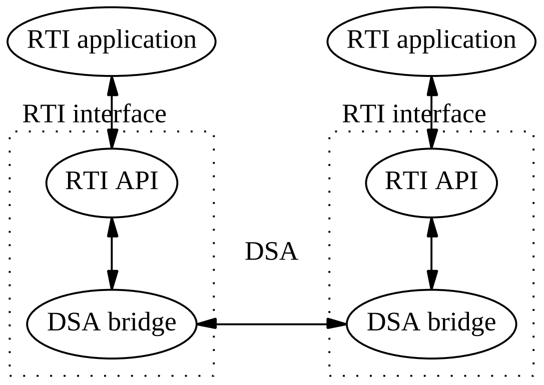

Figure 1: Using DSA as a communication layer

For example, modern distributed interactive simulations use a standardized API called RTI (Run Time Infrastructure). This API can be called from any language, such as Ada or C++. Using RTI, one can connect flight simulators used by human pilots and computer-driven simulation programs, some of them written in C, others in C++ and others in Ada. One implementation of RTI has been developed in Ada and uses the DSA as a communication layer [3]. The DSA is hidden to the RTI programmer, who only needs to use the RTI API, but it takes care of all the communication between RTI nodes, as shown on figure 1.

1.2.3 Education

Ada 83 has been used for years in software engineering classes, because of its high-level features such as genericity, strong-typing, encapsulation and tasking. The fact that an Ada compiler catches most errors at compile time makes it much easier for students to concentrate on the real problem rather than on a trivial mistake uncaught by a C compiler.

Ada 95 extends the power of Ada 83 to object oriented and distributed programming. From our own teaching experience, students enjoy using it when learning the basis of distributed programming (remote subprogram calls, distributed objects) because of its ease of use and its integration in a consistent model. For example, our students have been able to develop a complete multi-users messaging system based on distributed objects in a few hours. It would probably have taken much more time if they had had to cope with raw sockets and message passing.

2 Weaknesses of the current model

In this section, we present the main weaknesses of the DSA in its current form. We expect that those defects will not be present in the next Ada standard; some proposals to fix them and to increase the capabilities of the DSA are given in section 3.

2.1 Interoperability between Ada compilers

The design team of the DSA chose to separate the compiler from the PCS (Partition Communication Subsystem); the PCS has only a few entry points located in a standardized package called `System.RPC`, and those entry points must be the only interface between the code generated by the compiler and the PCS. It is thus theoretically possible to plug any PCS into any DSA-capable compiler[5].

Data exchanged between the compiler and the PCS are encapsulated into Ada streams; the PCS is not supposed to interpret their contents and must manipulate them as opaque data. This raises two major problems:

1. The content of Ada streams is not normalized: for example, one compiler can choose to store an integer in a stream as it is stored in memory, while another will use another format such as XDR (eXternal Data Representation) [29]. This prevents an Ada compiler from reading what has been stored by another, unless they agree on a common format for streams content.

2. The data stored in the streams given to the PCS is unspecified. One compiler can choose, to designate a remote subprogram, to use a package name followed by a subprogram index, while another can use a package index and a subprogram index to mean the same thing. Once again, Ada compilers would have to agree on a common protocol to be able to communicate with each other.

Even using a compiler from the same vendor on heterogeneous systems does not guarantee that your computers will understand each other, as the DSA does not require that heterogeneous systems be supported.

2.2 Interoperability with other languages

Unlike the DSA, which has been designed to write pure Ada distributed programs, CORBA [22] allows parts of a distributed system to be written using different programming languages. Each part is made of one or more objects, whose interfaces are described using the Interface Description Language (IDL). A stub and a skeleton will be generated from an interface. The stub is used to perform outgoing method calls to a remote object, while the skeleton handles incoming requests and redirects them to the actual implementation of the object methods. While the skeleton is generated for the language in which the object implementation has been written, the stub can be generated for many other programming languages (Ada, C, C++, Java, Lisp, etc.). It is thus possible to access a remote object from programs written in another programming language, as shown on figure 2 (the Dynamic Invocation box can be ignored and will be explained later).

The multi-language characteristics of CORBA played an important role in its wide acceptance: a team of developers can choose its favorite programming language to implement a service, and have another team use its own favorite language to access it. As far as interoperability is concerned, the DSA completely missed the train; the interfacing capabilities of Ada 95 compared to Ada 83 (pragma

[5]A validated Ada compiler does not need to be DSA capable as the DSA is an optional annex.

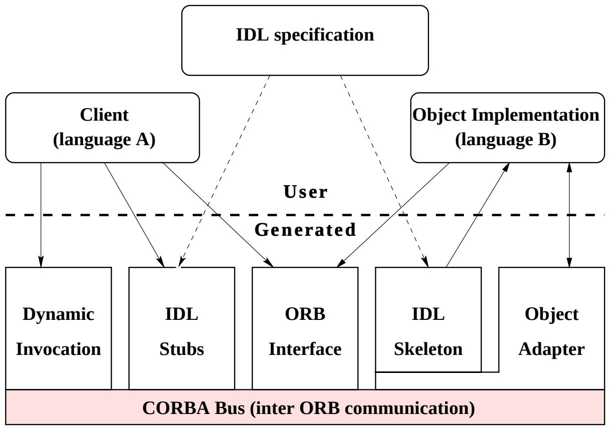

Figure 2: Global CORBA architecture

Import, access to subprograms with foreign conventions) have not shown up in annex E.

On the one hand, the lack of normalization of the protocol used to communicate between the partitions of a distributed Ada program forbids the development of any portable binding with remote Ada services. On the other hand, this allows distributed Ada programs to avoid costly constraint checks as the strong typing is preserved all the path along, while an interface with foreign languages would require additional checks to ensure the validity of externally acquired data.

This led to a situation where people have developed two-headed distributed programs: every data exchange between two Ada partitions is made through the DSA, while CORBA is used to export Ada services to the outside world (a concrete application can be found in [21]). However, this way of doing things is costly because two different interfaces (Ada and CORBA) need to be maintained at the same time. Also, it is error-prone, as a mismatch between the two versions may result into incorrect programs. We propose a solution to this particular problem in section 3.3.

Another solution commonly found to extend access to Ada remote objects to foreign languages is the design of small wrappers that translate a proprietary protocol into calls to the distributed objects (see figure 3). However, this solution has the same maintenance problems as the one exposed below, as two consistent interfaces must be maintained at the same time.

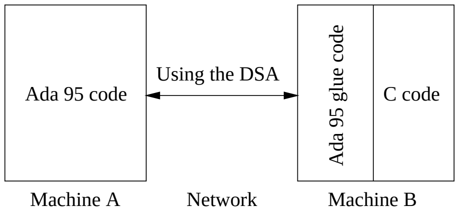

Figure 3: Using a proxy to access DSA services

2.3 Termination of a distributed application

Since 1980, we know that terminating a distributed application is not a trivial issue [8, 6]. To summarize the problem, an application can be globally terminated only when all the partitions are locally ready to terminate and there is no message in transit on the network that can potentially wake up one of the partitions [19].

However, an Ada distributed application can be composed of a dynamic number of partitions; when using a client/server model, the number of partitions that will compose the distributed program is not known in advance. A server cannot guess whether a new client is going to connect or not. The situation is similar to the one where tasks could be created spontaneously in a non-distributed program: even if all the tasks were waiting on a `select` statement with a `terminate` alternative, should the program be terminated if a new task showed up spontaneously and tried to wake up one of those existing tasks?

The reference manual does not contain anything about termination of distributed applications. As a direct consequence, it does not define whether it is erroneous or not to connect a new partition to a globally ready to terminate distributed system. This particular point deserves to be specified in the reference manual.

3 Some proposals to extend the DSA

In this section, we propose extensions that fall in two categories: the first category contains proposals that can be adopted for the current language revision if all Ada vendor agree on their realization. The second one contains proposals to be adopted for the next Ada revision.

3.1 Normalization of layers

The DSA can be decomposed into three independent layers: a high-level one, in charge of the semantics of the DSA, a mid-level one, which defines the communication between compiler generated code and the PCS, and a low-level one, which represents the underlying communication protocol.

3.1.1 The high-level layer

This layer contains the whole spirit of the DSA; it consists into the description, at the Ada language level, of what can be distributed and the semantics of every remote operation. The three categorization pragmas solely dedicated to the DSA (`Remote_Call_Interface`, `Remote_Types` and `Shared_Passive`) open a large and consistent set of possibilities to distribute Ada entities, objects or subprograms.

This layer has been carefully thought and leads to powerful constructs; for example, CORBA only deals with distributed objects, while the DSA also deals with remote subprogram calls in a traditional way. It also acts as a hidden naming service used to silently locate remote subprograms.

However, the set of entities that can be used remotely could be slightly enlarged by introducing the notion of remote rendez-vous for example. This would require allowing a task declaration in the visible part of a `Remote_Call_Interface` package, as well as remote accesses to task types and objects. Of course, appropriate restrictions must be placed on types of entry parameters, just as those restrictions exist for remote subprograms.

3.1.2 The mid-level layer

What we call the mid-level layer here is the declaration of the `System.RPC` package. As written in section 2.1, the design team of the DSA was willing to ensure a compatibility between any DSA-capable compiler and any PCS through this standardized package.

While this definition allowed us to start quickly the implementation of GLADE because one part of the design was implicitly contained in the annex, we soon realized that the requirement to go through `System.RPC` for every remote call introduces a lot of constraints.

To take one example, the only non-opaque parameter given to the procedure used to do a remote subprogram call (`Do_RPC`) is an integer denoting the remote partition. That implies that one of the two following methods is used:

1. This integer is assigned at partitioning time and the system is closed and static (it is not easy to add new clients in a client/server architecture after the first partitioning step while the server is running, and also not easy to launch several instances of a single client as they will have different identifiers).

2. This integer is computed at run time, and the compiler must have a way of retrieving it by talking to the partition communication subsystem using another interface than `System.RPC`, which defeats the capability of using the PCS with another Ada compiler.

After careful thoughts, we deliberately chose to use the second method and implemented a new package named `System.Partition_Interface`. This package contains all the needed subprograms to exchange localization information between generated code and the PCS. As a consequence, it is not possible for another compiler to use GLADE without generating calls to this new package.

After several years of experience in maintaining GLADE, we now firmly believe that the standardization of the PCS interface is useless and should be removed from the next language revision. Moreover, it is the only case in the reference manual where something that can be considered internal to the compiler has been described in an authoritative way.

3.1.3 The low-level layer

This part does not belong to the DSA. This is the cause of all the trouble described in sections 2.1 and 2.2. A standardization of the communication protocol would open the road to interoperability with other systems, either written in Ada (and thus using the DSA themselves) or in foreign languages, using easy-to-use communication libraries.

Using a clearly defined protocol does not cause any safety and efficiency loss, as long as the whole program is written in Ada. However, interfacing with other languages less safe than Ada may require the generation of additional checks to ensure that the externally acquired data meet Ada strong-typing constraints. Those checks could be turned on either by a pragma placed in the declaration of the remote package, meaning that even Ada programs would suffer a performance loss, or by a special flag in each packet indicating whether this packet is considered safe (from an Ada point of view) or not.

A proposal for such a low-level protocol is in progress, but is out of the scope of this paper. We plan to implement it for GNAT and GLADE, for compilers generating processor-specific code and for the future GNAT to JVM (Java Virtual Machine) compiler. This would, amongst other things, allow some partitions of a distributed system to run in native form (typically server partitions) while some others would run on a Java virtual machine (*e.g.*, client applets in a WWW browser).

3.2 Interfacing with RMI

RMI (Remote Method Invocation) is yet another way of writing distributed applications, using the Java programming language. This alternative to CORBA offers the possibility of having objects located on different hosts communicate with each other, and also lets objects with their implementation (in Java byte code) move from one host to another. This very powerful feature called **code migration** is a big step towards the development of mobile agents.

RMI is very interesting for Ada users from several point of views:

- Compilers that compile Ada code into Java byte code can use RMI objects and libraries to build distributed applications.

- A bridge between the DSA and RMI is already fully functional (ADEPT, see section 1.1) and allows Java users to access Ada services.

- Sun Microsystems (author of Java and RMI) is working with the OMG (Object Management Group, the entity in charge of CORBA normalization), to use IIOP (Internet Inter-ORB Protocol) as a basis for RMI implementation, while IIOP will be extended to support the full semantics of RMI. That means that Ada code compiled into Java byte code and using RMI will be able to talk with services written with CORBA, and that the Ada/RMI bridge will be usable as a gate between the DSA and CORBA

3.3 Interfacing with CORBA

As written in section 2.2, nothing prevents a user from maintaining two consistent interfaces for a service, one for the Ada side of the world using the DSA and a second for the other languages using CORBA, although this is a painful and error-prone task. In this section, we will see that other methods can be used to interface DSA and CORBA. As far as distributed objects are concerned, Ada and CORBA are close from each other. The differences and similarities between them have been studied in [27] and [24]. Comparisons from a designer's point of view have been published in [23] and [15].

3.3.1 Exporting DSA services to CORBA

An ongoing project is the automatic translation of DSA services into CORBA specifications so that those services can be used from a CORBA node [25]. Packages categorized as `Remote_Types` or `Remote_Call_Interface` are analyzed using ASIS (Ada Semantic Interface Specification) [14], and the semantic information is then utilized to generate one or more IDL modules; the implementation of each module is also produced automatically. Incoming CORBA calls to DSA objects are automatically transformed into outgoing DSA calls.

The first version of this tool called CIAO (CORBA Interface for Ada (DSA) Objects) [26] is using exclusively the free software OmniORB2 CORBA product. The gate between OmniORB2 and GNAT has been developed as a separate project [1] and can be used independently.

3.3.2 Using a common protocol

We are currently investigating the use of IIOP as the basis of our low-level protocol, to ease the interfacing process between the DSA, CORBA and RMI. The basic idea behind this is effort splitting: implementing a tasking runtime requires a lot of resources from Ada compilers vendors, and so does the implementation of the DSA. If some of the costs could be shared by the CORBA and RMI vendors, there would probably be more implementations of the DSA, as the existing infrastructure could be reused easily.

We already have a full Ada ORB implementation [12], soon to be released as free software. This software, whose code name is Broca, will serve as a basis for both our free software CORBA product, AdaBroker, and a future version of GLADE that will be using IIOP.

Using IIOP as the standard protocol for the DSA would allow accessing CORBA and RMI services directly through the DSA without any need for an additional bridge. Also, it would be possible, using an Ada to Java byte code compiler, to transfer active objects and achieve code migration in Ada using only the DSA. This would be a big step forward in terms of fault tolerance and reliability, as critical services could duplicate themselves automatically in order to keep for example a minimum number of instances available at any time.

3.4 Dynamic interfaces

Another powerful CORBA feature not found in the DSA is the ability to use dynamic interfaces. Two mechanisms, DII (Dynamic Interface Invocation, shown on figure 2) and DSI (Dynamic Skeleton Interface), can be used to register a class by describing its methods using their names and signatures, and to build a call to those methods dynamically.

The most obvious advantage is that the interface does not need to be present at compilation time. For example, a calculator application can be enriched at run time by adding new functions (as remote objects designed by their names) that are called dynamically as the user types names them. Explicit static calls to those functions do not appear anywhere in the calculator source code.

The introduction of such a mechanism in the DSA would considerably ease the interfacing with CORBA, as both side could interface with each other using this protocol.

3.5 Quality of Service

On a completely unrelated topic, a useful extension to the existing DSA specification would be the introduction of Quality of Service (QoS) parameters. For the reader unfamiliar with this notion, a QoS specification may be seen as a numeric value quantifying some properties of the underlying network, for example the maximum delay between a request emission date and its handling on the receiver side, or a guaranteed bit rate from one partition to another.

As the requests for quality of service depend heavily on the location of every service, it makes more sense to include all the QoS related information into the configuration file describing the distributed application rather than in the subprogram specification itself. To this purpose, we intend to propose the format of the gnatdist (our partitioning tool) command file as a standard for describing Ada distributed applications. This file format will first be extended to contain the necessary QoS extensions, as shown in sample 3.5.

Sample 1 gnatdist extension for QoS

```
configuration Demo is
  P1, P2 : partition;
  C : channel := (P1, P2);

  for C'Bandwidth use 1_000_000;
  for C'Peak use 2_000_000;
  for C'Max_Peak use 5.0;
  --  Require up to 1Mbit/s for normal execution,
  --  with peaks up to 2Mbits/s for a maximum of
  --  5 seconds.
[...]
end Demo;
```

3.5.1 Network characteristics

It is common knowledge that QoS and crude packet switching networks do not mix well, as the transmission time of a data fragment is unrelated to the transmission time of other fragments composing the same high-level packet. Unfortunately, packet switching networks are very common both in the industry and in universities (TCP/IP over Ethernet).

The introduction of the new IP revision (IPv6, previously called IPng) [5] adds the notion of traffic class. This field, present in every IPv6 packet, is used by all the routers on a given path to prioritize the flow accordingly to a predefined policy. Within an organization, it is be possible to setup all the routers to exchange data between two partitions at the highest possible priority, thus obtaining the speed and the bandwidth of the slowest physical link on the data path.

However, this solution is only applicable to a company network, as the company system administrator does not control the routing and priority handling policies on external routers. The use of ATM (Asynchronous Transfer Mode) networks by Internet providers can help develop such a priority scheme between distant sites. Those networks can negotiate a guaranteed bandwidth for a complete virtual circuit. Once the required bandwidth has been allocated, it will never be used for something else unless the resource has been explicitly released.

We have already developed a binding between Ada and ATM [18], which is being integrated in AdaSockets[6], one of our free software products. The ultimate goal is to offer the ability to use ATM as the underlying networking protocol for GLADE. We are working on defining a syntax for the indication of the desired networking resources, that could be applied to IPv6, ATM and other protocols dealing with resource reservations such as RSVP [2].

3.5.2 Priority related enhancements

One of the most needed enhancements of the DSA would be a specification of how priorities in partitions of an Ada distributed program are related to each other. Right now, it is unspecified whether the priority of the caller will be used or not for executing the remote subprogram. This is even worse as the various partitions may have different scheduling policies and priority ranges, thus leading to a malfunction if priorities are propagated over the network.

We have integrated new pragmas in the GLADE configuration file to statically describe the behavior of incoming calls. For example, one can set an upper limit of the number of incoming remote calls that can be executing at the same time. Also, a common priority map is used on all the partitions to transform the caller priority into an acceptable priority for the receiver, if the user chooses to do so (a flag in the configuration file toggles this). This method eases the scheduling computation in a distributed application using Rate Monotonic Scheduling techniques [28].

4 Conclusions

We have shown in this paper that the Distributed Systems Annex of Ada 95 is well defined and consistent with the rest of the language, but would benefit from a standardization of the protocol used to communicate between the partitions. This standardization would ease the development of Ada distributed applications using different Ada compilers, and would open the world of distributed Ada to other languages, without loosing any of the Ada safety and security.

Despite those (hopefully constructive) criticisms, we firmly believe that the presence of the Distributed Systems Annex is a major achievement in the language definition. We want to thank again the design team who chose to make Ada even more powerful and user-friendly by including the DSA in the ISO standard.

[6]http://www.infres.enst.fr/ANC/

References

[1] F. Azavant, J.-M. Cottin, L. Kubler, V. Niebel, and S. Ponce. AdaBroker, using OmniORB2 from Ada. Technical report, ENST Paris, March 1999.

[2] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. Technical report, The Internet Society, September 1997. RFC 2205.

[3] D. Cannazzi. yaRTI, an Ada 95 HLA Run Time Infrastructure. In *Proceedings of AdaEurope'99*, Santander, Spain, June 1999.

[4] J. Davis. The Power is Out There. *Business 2.0*, pages 102–104, 1998.

[5] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Technical report, The Internet Society, December 1998. RFC 2460.

[6] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, Aug. 1980.

[7] N. Francez. Corrections: "Distributed Termination". *ACM Transactions on Programming Languages and Systems*, 2(3):463–463, July 1980. See [8, 20, 9].

[8] N. Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, Jan. 1980. See also corrections [7] and remarks [20, 9].

[9] N. Francez. Technical correspondence: Reply from Francez. *ACM Transactions on Programming Languages and Systems*, 3(1):112–113, Jan. 1981. See [8, 20].

[10] A. Gargaro, S. J. Goldsack, C. Goldthorpe, D. Ostermiller, P. Rogers, and R. A. Volz. Towards Distributed Systems in Ada 9X. In *Proceedings of the Conference for Industry, Academia and Government*, pages 49–54, New York, NY, USA, Nov. 1992. ACM Press.

[11] A. Gargaro, Y. Kermarrec, L. Pautet, and S. Tardieu. PARIS: Partitionned Ada for Remotely Invoked Services. In *Proceedings of AdaEurope'95*, Frankfurt, Germany, Mar. 1995.

[12] T. Gingold. BROCA: an Ada Object Request Broker. Master's thesis, École Nationale Supérieure des Télécommunications, July 1999.

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

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

[15] Y. Kermarrec. CORBA vs. Ada 95 DSA – A programmer's view. In *Proceedings of SigAda'99*, Redondo Beach, California, USA, Oct. 1999.

[16] Y. Kermarrec, L. Nana, and L. Pautet. GNATDIST: a configuration language for distributed Ada 95 applications. In *Proceedings of Tri-Ada'96*, Philadelphia, Pennsylvania, USA, 1996.

[17] Y. Kermarrec, L. Pautet, and S. Tardieu. GARLIC: Generic Ada Reusable Library for Interpartition Communication. In *Proceedings Tri-Ada'95*, Anaheim, California, USA, 1995. ACM.

[18] R. Labelle. Interface ATM pour Ada. Master thesis, École Nationale Supérieure des Télécommunications, July 1999.

[19] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[20] C. Mohan. Technical correspondence: On Francez's "Distributed Termination". *ACM Transactions on Programming Languages and Systems*, 3(1):112–112, Jan. 1981. See [8, 9].

[21] S. A. Moody. Object-oriented real-time systems using a hybrid distributed model of Ada 95's built-in DSA capability (Distributed Systems Annex-E) and CORBA. *ACM SIGADA Ada Letters*, 17(5):71–76, Sept./Oct. 1997.

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

[23] I. L. Patton. Impact of Using the Ada 95 Distributed Annex vs. CORBA on the Development of a Distributed System. Master's thesis, George Mason University, Fairfax, Virginia, USA, June/Aug. 1998.

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

[25] L. Pautet, T. Quinot, and S. Tardieu. CORBA and CORBA Services for DSA. In *Proceedings of SigAda'99*, Redondo Beach, California, USA, Oct. 1999.

[26] T. Quinot. Mapping the Ada 95 Distributed Systems Annex to OMG IDL – Specification and implementation. Master's thesis, École Nationale Supérieure des Télécommunications, July 1999.

[27] J. D. Riley. A comparison of two approaches to distributed application development in Ada: The distributed system annex and CORBA. In *Proceedings of the TRI-Ada Conference*, pages 73–82, new York, Dec. 3–7 1996. ACM Press.

[28] J. A. Stankovic and K. Ramamritham. *Advances in Real-Time Systems*. IEEE Computer Society Press, 1993.

[29] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.