Issues with using Ravenscar and the Ada Distributed Systems Annex for High-Integrity Systems

Neil Audsley and Andy Wellings
Real-Time Systems Group, Dept. of Computer Science, University of York, York, UK

Abstract

The Distributed Systems Annex (DSA) was designed for general-purpose distributed systems programming. This paper explores the practical use of the DSA in terms of high-integrity real-time distributed systems. In particular, it defines a high-integrity subset of the DSA and considers how such a subset can be implemented using SPARK and Ravenscar.

1. Introduction

The Distributed Systems Annex (DSA) part of the Ada language standard [1] and rationale [4] was designed for general-purpose distributed systems programming. It has a number of weaknesses for use in real-time systems, documented in detail at the 1998 Real-Time Ada Workshop [9]. This paper further explores the practical use of the DSA in terms of high-integrity real-time distributed systems.

For the purposes of this paper high-integrity systems are those having additional requirements over above real-time requirements, including safety and reliability. Essentially, such systems must be developed and analysed (static offline and dynamic testing) to a greater degree of rigour than normal real-time systems. Often this rigour is defined by some domain standard, e.g. DO-178B [8] for civil aircraft, 00-55 [13] and 00-56 [12] for UK military aircraft. A key characteristic of all domain standards is the ability to statically analyse software (when produced to the highest level of integrity) for its behaviour in terms of data-flow, control-flow, timing and resource usage (i.e. stack). This places a requirement on the software to be amenable to such offline analyses.

The Ada language [1] provides several features to aid developers of high-integrity distributed systems, including the Distributed Systems Annex (DSA) and Annex H for Safety and Security. The work of the Annex H Rapporteur Group has provided a ISO technical report [2] that effectively includes definition of the safe tasking subset of Ada, commonly known as the Ravenscar profile [6]. When combined with suitable restrictions regarding the sequential part of the language (e.g. those imposed by SPARK [5]), the Ravenscar profile and the DSA could provide a practical method for programming high-integrity distributed systems.

A motivation behind this work is to utilise, as far as possible, a standard Ada environment, i.e. use of the DSA as given in the language standard; "off-the-shelf" components, i.e. compiler and run-time (GNAT) together with DSA implementation (Glade). This paper explores two main issues related to the production of high-integrity real-time distributed systems using Ada:

1. Problems with using the standard environment;
2. Changes required to the standard environment.

The paper is presented in five further sections. The next section gives background on the DSA, Glade, Ravenscar and SPARK. Section 3 describes the programming model we wish to use for distributed systems. Subsequently, section 4 gives a discussion of the problems faced using Ada and the DSA for Ravenscar and SPARK compliant systems; section 5 outlines potential solutions. Finally, conclusions are offered in section 6.

2. Background

2.1. Distributed Systems Annex

As stated in the language reference manual [1]:

> *"E(1): This Annex defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program."*

Note that the intention is that the distributed system may be heterogeneous.

Partitions contain one or more library level units. Partitions may be either active, in which case they contain one or more tasks; or passive. Partitions can be allocated any processor. Thus, a processor may have one or more partitions allocated to it.

Inter Partition Communication

Within Annex E, inter-partition communication is achieved by: Remote Procedure Call (RPC), Remote Access Subprogram (RAS) and Remote Access to Class Wide type (RACW).

A RPC requires a task at the called partition to handle the incoming call. A RPC can be either synchronous, in which case the caller is blocked until the called partition returns the result of the call; or asynchronous, in which case the caller is not blocked (effectively the caller can continue in parallel with the invoked procedure in the remote partition). Asynchronous RPCs[1] are distinguished by the use of the `pragma asynchronous` together with only using `in` parameters. Further details of Annex E are given in [1].

Partition Communication System (PCS)

Within each partition, a library level unit called the Partition Communication Subsystem (PCS) is required to effect inter-partition communication. The PCS effectively performs message passing among partitions. The PCS is internally responsible for all routing decisions, low-level message protocols, etc. The interface to the PCS library is defined in Annex D of the language reference manual as package `System.RPC`.

Importantly, `System.RPC` is defined using `Ada.Streams` for communicating the remote call parameters in terms of a byte stream of data from caller to callee. The motivation for this design is to ensure that parameters have similar meaning for caller and callee, considering that partitions can be on different architectures. In turn, `Ada.Streams` can be implemented to use a data exchange standard such as XDR.

Implementation

Whilst the language reference manual leaves much of the implementation of the DSA fairly open, a number of important implementation guidelines are given in the standard [1] and rationale [4]. Firstly, a run-time is associated with each partition, with no inter-dependence between run-times in different partitions. Secondly, the implementation of the RPC receiver on the called partition must be re-entrant. Finally, there is no necessity to implement the RPC receiver in terms of a task or pools of tasks, only that if such a method is used, the number of tasks available should be documented.

2.2. Ravenscar Ada Tasking Subset

Ravenscar is an Ada tasking profile [6]. It limits the functionality of concurrent Ada95 to objects that allow easy timing analysis. The limitations as imposed by Ravenscar include no dynamic allocation of tasks or protected objects, with the latter limited to single entries only, with a maximum of one task queuing on it, with only a simple Boolean guard.

[1]Asynchronous RPCs are known as APCs within the standard.

2.3. SPARK Ada Subset

Ravenscar is only a tasking profile – it does not limit any other part of Ada. Therefore, just imposing the tasking restrictions defined by Ravenscar does not guarantee easy timing analysis. A complementary subset of the sequential parts of the language is needed for high-integrity systems. The choice made for the purposes of this paper is the SPARK Ada subset [5]. This subset of Ada has been widely used in high-integrity systems [7]. Restrictions of SPARK include no use of exceptions and no dynamic operations, including memory or run-time dispatching. The motivation for SPARK is static analysability, hence the disallowing of features such as access types or dynamic operations.

2.4. GNAT / GLADE

At the compiler level, GNAT takes the pragmas as specified in the source and creates client/server stubs for the remote programs being called. GLADE, the GNAT implementation of the DSA, consists of three distinct parts: Gnatdist – a tool that uses a configuration file to produce runnable Ada programs for each partition of the distributed system (required as distributing programs is a post-compilation issue in Ada); TCP – an implementation of the TCP protocol to run underneath Garlic. Garlic was designed to give a full implementation of the Ada Partition Communication Systems model. Underneath Garlic is an interface to TCP/IP. The Garlic implementation of the PCS utilises a pool of tasks to handle incoming remote calls. If the pool is exhausted other tasks are dynamically created.

3. Programming Model for High-Integrity Distributed Ada

Given that the context for this work is high-integrity distributed systems, an appropriate programming model is required that ensures predictability from timing and resource usage perspectives. The model adopted was consistent with the Ravenscar and SPARK subsets; utilised a producer-consumer model of communications and allowed only asynchronous communication between partitions Thus, only asynchronous RPCs, i.e. APCs are permitted. Both RAS and RACW are not permitted within the restrictions of SPARK (i.e. no access types or dynamic dispatching is permitted).

3.1. Asynchronous Communication

To achieve predictability within a distributed system, any communications between processors must be bounded in terms of time and resources used. Clearly, this has an impact upon the low-level communications protocols and media used for physically transmitting a message between processing nodes. For the purposes of this paper, we can assume that predictable protocols and media are available – e.g. CAN [16, 15].

At a higher level, the communications paradigms provided by the DSA must be predictable and efficient. Initially, we observe that both the synchronous and asynchronous RPC provided by the DSA are predictable (given predictable low-level communications). However, use of synchronous RPC has a number of consequences:
Local blocking – the calling task is blocked until the remote call is complete.
Failure – if the remote call could fail (perhaps due to a partition of the communications network) then there is a requirement on the PCS implementation to detect such failure and return some form of error to the caller or provide transparent replication [18].

Whilst blocking does not necessarily lead to unpredictable systems, the timing analysis will be pessimistic, always having to assume worst-case blocking times. Also, given that nested RPCs may occur, the worst-case blocking times are difficult to derive, as the analysis of such systems is inherently holistic as chains of blocking dependencies can arise [17].

3.2. Producer-Consumer Model of Comms

The model of communications implied by asynchronous communication is that of producer-consumer, i.e. a dataflow model. Such a model is more in keeping with real-time or high-integrity systems. These systems are generally characterised by taking input data from the environment, performing some form of computation on the data, finally outputting the data back to the environment (where the environment contains sensors, actuators etc).

In terms of the Ravenscar and SPARK Ada subsets, a producer-consumer model can be realised by tasks performing computation, with some mechanism for communicating data between tasks in an asynchronous manner. As tasks are not allowed entries in Ravenscar, they must communicate via protected objects. The restrictions on Protected Objects mean that only simple synchronization is available via protected entries – equivalent to the functionality provided by `Ada.Synchronous_Task_Control`. This is illustrated below:

```
task Producer;   task Consumer;

protected Shared_Data is
  producer Give(D : in Data);
  entry Take(D : out Data);
private
  The_Data : Data;    Available : Boolean := False;
end Shared_Data;

protected body Shared_Data is
  procedure Give(D : in Data) is
  begin  The_Data := D; Available := True;
  end Give;

  entry Take(D : out Data ) when Available is
  begin D := The_Data; Available := False;
  end Take;
end Shared_Data;

task body Producer is
begin    ... Shared_Data.Give(The_Data); ...
end Producer;

task body Consumer is
begin    ...  Shared_Data.Take(The_Data); ...
end Consumer ;
```

Note that this model would need to be extended if more than one consumer is required for a single data item. In full Ada, one could allow more than one task waiting on the guarded entry. This is not allowed in Ravenscar. Therefore, some form of data copying would be required, so that each consuming task has a dedicated protected object. Note also, that the producer must over-write the data if the consumer is slow in consuming, as only one entry per protected object is allowed. Of course, a buffer could be encapsulated in the protected object.

In terms of the distributed Ada, the above model requires inputs to a partition (arriving in the form of asynchronous RPCs) to place data into a protected object, setting a guard to `true` to indicate its presence. The next task in the producer-consumer chain will either be blocked waiting on the guard (i.e a guarded entry) in which case it will be free to proceed and read the data, or will at some point in the future call into the protected object and proceed to read the data without becoming blocked at all. Note, that the consumer and the protected object must always be within the same partition as only asynchronous RPC are allowed.

The following code implements a distributed producer-consumer, where the shared data is in a different partition to the producer, but in the same partition as the consumer task.

```
-- PARTITION ONE -
package Distributed_Shared_Data is
  procedure Give(D : in Data);
  procedure Take(D : out Data);
end Distributed_Shared_Data;

package body Distributed_Shared_Data is
  protected Shared_Data is
    producer Give(D : in Data);
```

```
      entry Take(D : out Data);
   private
      The_Data : Data;   Available : Boolean := False;
   end Shared_Data;

   protected body Shared_Data is
      procedure Give(D : in Data) is
      begin  The_Data := D; Available := True;
      end Give;
      entry Take(D : out Data ) when Available is
      begin D := The_Data; Available := False;
      end Take;
   end Shared_Data;

   procedure Give(D : in Data) is
   begin Shared_Data.Give(D);
   end Give;

   procedure Take(D : out Data)is
   begin Shared_Data.Take(D);
   end Take;
end Distributed_Shared_Data;

with Distributed_Shared_Data;
package Consumer_Partition_Interface is
   pragma Remote_Call_Interface;
   pragma Asynchronous(Distributed_Shared_Data.Give);
   procedure Distributed_Shared_Data.Give(D : in Data);
end Consumer_Partition_Interface;

task Consumer ;
task body Consumer is
begin
   ... Distributed_Shared_Data.Take(The_Data);   ...
end Consumer;

-- PARTITION TWO -
task Producer;
task body Producer is
begin
   ... Consumer_Partition_Interface.Give(The_Data); ...
end Producer;
```

3.3. Implementation

The standard GNAT/GLADE environment was utilised. This leads to the implementation scheme given in Figure 1, illustrating the facilities required for a call from `Partition_1` to `Partition_2`, including the return (this is only for illustrative purposes, as the there is no return for an APC).

When the application task in `Partition_1` makes the `Partition_2.Proc` the client side stub code is executed, i.e. `Client_Body`, followed by the PCS code (i.e. Garlic) which effects the transmission of the call and its parameters across to `Partition_2`. On message reception on `Partition_2`, the run-time will call PCS code (i.e. Garlic) which identifies a task from the pool to handle the incoming call. This task then executes the `Server_Body` stubs (which includes re-assembly of the parameters of the call), finally making the call to `Partition_2.Proc`.

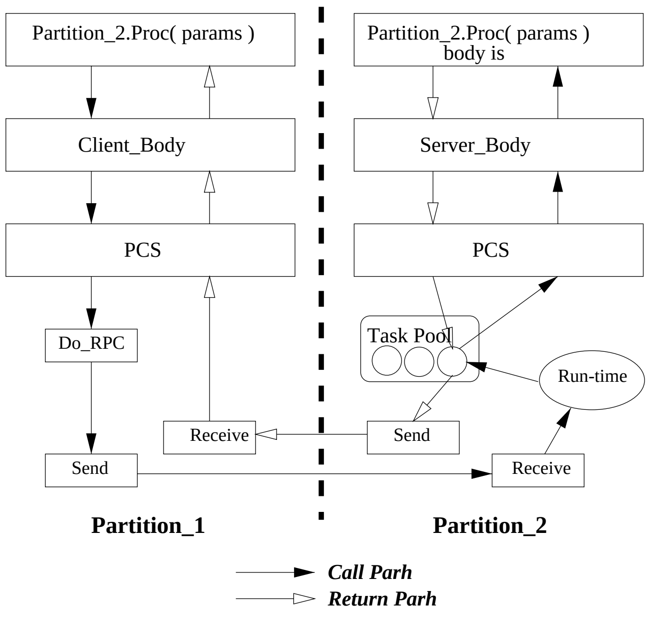

Figure 1. Implementation Schematic

Once the call is complete, the return of any `out` parameters to `Proc` proceeds to `Partition_1`.

4. Problems with Standard Ada

4.1. Ravenscar Limitations for PCS

No dynamic creation of Tasks

To be able to handle concurrent remote calls, the partition communication system must be reentrant. This is specified as an implementation requirement in section E.5.24 of the Ada Language Reference Manual [1].

A natural solution would be to spawn a new task whenever a request is made. This approach was considered by the designers of GLADE but rejected because of the overhead involved in task creation and finalization [14, 11, 10]. Instead, they use a pool of tasks to handle the remote procedure calls (see `System.RPCpool`). The initial size of this task pool is parameterized and can be set in the gnatdist configuration file. If the pool cannot cope with the incoming load, more tasks are created dynamically. This solution is not available to us, as Ravenscar forbids dynamic creation of tasks.

The only realistic solution is to provide a fixed pool of tasks that serve remote calls. The problem with a fixed pool size is that, in conjunction with a fixed buffer size, there is a static upper bound on the number or requests that can be handled in a period of time. This bound stays fixed for the *entire* lifetime of the program. Therefore the initial bound must be sufficient to cope with the worst case scenario, and will therefore require more system resources than an extensible solution that deals with the average case before adding more tasks.

No Protected Object Entry Queues

This means that multiple tasks cannot simply wait on a protected entry until a synchronization signal comes through.There are several places in the PCS where this kind of structure is apparent:

At Initialization – At startup, calls to remote partitions must wait until that partition has finished elaboration. As there may be several processes wishing to communicate remotely, some central object must coordinate all of this synchronization information. The natural way to do this would be via a protected object, where requests were queued until the remote partition was ready to accept them.

When a new request is received – In between remote calls, the handler should not be busy-waiting for the next call. In addition, several requests may come in almost simultaneously; they should be handled in parallel, rather than waiting for each preceding call to be processed first. There must therefore be several handlers suspended until an incoming request is received.

Waiting for a Remote Call to return a result – When a local process makes a synchronous remote call (be it to a procedure or remote object) it must wait for a return value before recommencing operation.

4.2. SPARK Limitations for PCS

The interface to the PCS as described by `System.RPC` is not compliant with SPARK. This is due to the `System.RPC` being defined in terms of `Ada.Streams`. Whilst `Ada.Streams` is clearly the Ada method for implementing byte streams between two places, it is outside SPARK as it relies heavily on tagged types. A second problem caused by SPARK is that the PCS defines the exception `Communication_Error` to be raised on communication error, although it is only really defined in terms of synchronous RPC calls. Clearly for asynchronous calls, receiving an exception at the caller for an error either at the remote node or the communication system is of little value as the caller will not have blocked and therefore may not have the context to deal with the exception. Indeed, it is for this reason that the ARM allows these exceptions to be discarded.

4.3. GNAT Run-time Mismatch with SPARK / Ravenscar

It is reasonably clear that the GNAT run-time is not compliant with either SPARK / Ravenscar. For example, dynamic operations are used within the data-structures held by the run-time. This issue is largely beyond the scope of this paper, noting that Ravenscar compliant run-times have been designed and implemented [6]. It is not clear how SPARK compliant such run-times are.

One observation is that it is not easy to use a bespoke run-time with the GNAT compiler, or indeed any other commercial Ada compiler. In the case of GNAT, it is not easy to interface a bespoke run-time to GNARL (via GNARLI) – even if this were achieved, the GNARLI interface is different to that provided by other Ada compilers. Thus, if a Ravenscar and SPARK compliant (and even verified) run-time were available, it would be difficult to incorporate this with existing Ada compilers without significant effort.It would be useful if some form of standard run-time interface were available and used by all Ada compiler vendors (e.g. similar to MRTSI [3] only for Ada 95 rather than 83).

4.4. GLADE Expects Full Communications Protocol Stack

The GLADE implementation of the DSA at the lowest level contains Garlic. This utilises the TCP communications stack for passing messages between processors. The motivation for this design decision was that GLADE, and therefore Garlic, was targeted at processors with relatively complete operating system implementations, e.g. UNIX. Clearly, for general-purpose systems written in Ada this is not a problem. Unfortunately for high-integrity systems, where predictability is required, operating system and communications functionality is restricted. It is unclear whether the Garlic implementation would function as designed when using a restricted “real-time” TCP/IP; or perhaps on raw low-level communications without such a communications stack.

4.5. GNAT / GLADE Mismatch with Ravenscar

The current GLADE implementation uses many (Ravenscar) illegal Protected Objects. Also (as noted above) dynamic task creation could be used if the task pool is exhausted.

4.6. Ravenscar Causes Tasking Inefficiencies

The removal of dynamic task creation from Ravenscar makes it particularly difficult to perform many operations common for high-integrity distributed systems (noting that SPARK also bans dynamic task creation). For example, consider mode changes.

Ravenscar dictates that all tasks have to exist for the full lifetime of the program. If a system has many potential modes, all tasks for all modes would be elaborated. Tasks outside the current mode would be waiting for an event (i.e. blocked waiting for a mode change). It would need to be shown that events could not occur that trigger tasks not in the current mode to execute.

Alternative approaches include:

- Application polled mode changes – here an application could program a mode change into a task, such that a task executes different code for each mode. This only helps if there are equal numbers of tasks in each mode. Also, a task cannot be stopped to force a mode change, it merely responds next time it polls the global mode.

- Support for modes in run-time – if Ada had support for modes in the language and hence run-time, the run-time support system could, when changing to a new mode, re-use the task storage from the previous mode.

- Dynamic task allocation – Finally, it is noted that if dynamic task allocation were available, tasks could be aborted at the end of their mode – perhaps saving state so that if the same task were to run again later, it could recover its state.

4.7. Ravenscar Difficulties for Synchronisation

The limitations that Ravenscar places on the tasking model means that inter-task communication must be performed indirectly via protected objects. The fact that entry queues are disallowed means that synchronization can only be used on a one to one scale. Thus the standard producer/consumer Protected Object can only be implemented for one consumer (and the producer can never block).

This might be fine in trivial concurrent systems. However, on a realistic system, better synchronization may be required – specifically several producer tasks and several consumer tasks for a single shared data construct. A full Ada implementation would involve the consuming tasks waiting on a single entry. Within Ravenscar, one way to manufacture such a construct is to associate a `synchronous_task_control` object with each consuming task. A consumer is suspended until data is available. A producer task effectively sets true the suspension object of the first waiting consumer task.

4.8. SPARK / Ravenscar Cause Obfuscation

The constrained manner of programming that SPARK enforces could be argued to make code less easy to read. Certainly, it can be less concise. A similar problem exists with Ravenscar compliance. Here, the concurrent aspects of the program could become obscured and difficult to understand due to the unavailability of relatively well-accepted and understood synchronisation and communication mechanisms. If a programmer wishes to have a buffer for communication between two tasks, it can still be achieved using Ravenscar, only with much less readability – increasing the possibilities of error.

4.9. Efficiency of Generated Code

There is a concern over the efficiency of GNAT / GLADE in terms of “short-circuiting” the lower parts of the PCS for calls between partitions *on the same processor*. Under the GNAT / GLADE the only saving for such a call is the transmission time compared to a truly remote call.

5. Potential Changes to DSA

It is clear from the discussions in the previous section that a number of changes need to be made for the DSA to be compliant with SPARK and Ravenscar. Whilst it is beyond the scope of this paper to detail potential changes, we make suggestions for further discussion, with relation to programming distributed high-integrity systems.

- *Is the partition concept applicable?*
  There do not seem to be any problems with the partition concept for use with Ravenscar / SPARK. However, it would be useful if the issues of single run-time per processor as against single run-time per partition were discussed further.

- *Should priorities be further refined for distributed calls?*
  Currently, there is no real guidance for the priority of a remote call, which, as has been discussed previously [9], can lead to priority inversions. The problem with defining the remote priority is that the priority level may not even be supported on a different processor. That is, the possibility of heterogeneous systems must be catered for.

- *Where should the interface to the communications system be?*
  The current definition of the DSA suggests a high level paradigm (i.e. RPC) as the interface to the PCS. The motivation for this is to aid portability of applications as only (part of) the PCS needs to be re-implemented when moving to a different architecture or compiler. This approach makes it difficult to use different vendor implementations of the DSA in a single system, as it is unlikely that they will communicate at a low-level. Thus, perhaps a lower-level standard interface is required instead, or in addition. This interface would allow different vendor implementations of the DSA to be compatible.

- *Should the DSA be restructured?*
  This identifies that for high-integrity systems, only part of `System.RPC` is usable. There is an argument for splitting this package into the synchronous and asynchronous parts. There is also an issue with a client-server model being seen as the "chosen paradigm" within the DSA, where producer-consumer is just as, or perhaps more applicable, for many high-integrity systems. Perhaps the DSA should merely be a set of bindings, e.g. CORBA for general purpose distributed systems; to ISO standard RPC for non-CORBA client-server. This approach is useful, in that it removes the problems of stub generation from the Ada compiler, which would make interchangeability of compilers and communications systems easier.

- *Can* `Ada.Streams` *be part of a high-integrity system?*
  A major issue is the use of tagged types and dynamic dispatching within `Ada.Streams` which forms the basis of the DSA. An argument can be made that tagged types *could* be used in this instance, as long as the calls can be statically bound, thus compile time determinable – especially as this can be checked via the ASIS interface to the Ada environment. However, such an approach is a departure from the SPARK / Ravenscar model. Alternatively, a different implementation scheme for, and therefore definition of, `System.RPC` is required that does not use `Ada.Streams`.

- *Lack of functionality?*
  The DSA as it stands does not provide the ability to perform standard distributed systems communications such as multicast. Whilst this can be "manufactured" by the application, this is not particularly satisfactory.

6. Conclusions

There are two prime motivations for Ravenscar. The first is is to ensures that timing analysis can be carried out on a conforming application. The second is that the supporting run-time support system is small, fast, efficient and capable of being certified. Arguably this second requirement has been taken too far. It is clear that in order to facilitate distributed high-integrity real-time programming, the run-time support for distributed programming itself should conform to the Ravenscar profile. We have illustrated in this paper that this support requires greater expressive power than that afforded by Ravenscar. The result is greater complexity in the run-time – the code is almost certainly less analyzable, and definitely harder to produce and read.

References

[1] *Ada 95 Reference Manual*. Intermetrics, January 1995.

[2] Guide for the use of the Ada Programming Language in High Integrity Systems. Technical Report ISO/IEC TR 15942:2000, ISO, 2000.

[3] Ada Runtime Environment Working Group: MRTSI Task Force. A Model Runtime System Interface for Ada. Technical report, ACM SIGAda, 1990.

[4] J. Barnes. *Ada 95 Rationale*. Spinger Verlag, 1995.

[5] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[6] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Uppsala*, pages 263 – 275. Springer Verlag, 1998.

[7] R. Chapman and R. Dewar. Re-engineering a Safety-Critical System in SPARK 95 and GNORT. In *LNCS 1622: Proceedings of the Ada Europe Conference*, pages 39–51. Springer Verlag, 1999.

[8] European Organisation for Civil Aviation Electronics. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[9] M. G. Harbour and S. Moody. Session Summary: Distributed Ada and Real-Time. *Ada Letters – Proceedings of the 9th International Real-Time Ada Workshop*, 19(2):15–18, June 1999.

[10] Y. Kermarrec and L. Nana. Implementation of the Ada 95 Distributed Systems Annex into GNAT: code expansion details and front-end issues, 1996.

[11] Y. Kermarrec and L. Pautet. Evaluation of the Distributed Systems Annex of Ada9X and its implementation in GNAT. Technical report, New York University, February 1994.

[12] Ministry of Defence. *Defence Standard 00-56: Requirements for Safety Management Requirements for Defence Systems (Part 1: Requirements, Part 2: Guidance)*, December 1996.

[13] Ministry of Defence. *Defence Standard 00-55: Requirements for Safety-Related Software in Defence Equipment (Part 1: Requirements, Part 2: Guidance)*, August 1997.

[14] L. Pautet and S. Tardieu. Inside the Distributed Systems Annex, 1997.

[15] K. Tindell and A. Burns. Guaranteeing Message Latencies on Control Area Networks (CAN). *Proceedings 1st International CAN Conference*, September 1994.

[16] K. Tindell, A. Burns, and A. Wellings. Calculating Control Area Network (CAN) Message Response Times. *Proceedings IFAC Workshop on Distributed Computer Control Systems (DCCS)*, September 1994.

[17] K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, November-December 1993.

[18] T. Woolf. Transparent Replication for Fault-Tolerance in Distributed Ada 95. *Ada Letters – Proceedings of the 9th International Real-Time Ada Workshop*, 19(2):33–40, June 1999.