A Reusable Lightweight Executive for Command and Control Systems

Nathan Fleener
Boeing Phantom Works
P.O. Box 3999 MS 3F-70
Seattle, WA 98124-2499
253-657-1192
Nathan.C.Fleener@boeing.com

Laura Moody
Boeing Phantom Works
P.O. Box 3999 MS 3F-70
Seattle, WA 98124-2499
253-657-2890
Laura.L.Moody@boeing.com

Mary Stewart
Boeing Phantom Works
P.O. Box 3999 MS 3F-70
Seattle, WA 98124-2499
253-657-3878
Mary.E.Stewart@boeing.com

1. ABSTRACT

Customized systems are expensive to build, maintain and upgrade. Companies have large investments in legacy systems running on obsolete hardware. These systems carry high hardware maintenance costs with limited growth potential. New systems today must be resilient to changing hardware and system requirements.

The Lightweight Executive contains a cooperating set of scheduling and data management services. A layered architecture allows systems to evolve without impacting proven components. Ada95 [1] provides the mechanisms needed to support these customizations across multiple product lines.

The Lightweight Executive utilizes the Ada95 runtime and leverages off of compiler vendors for portability. Legacy systems have been successfully upgraded utilizing the Lightweight Executive. New product lines now benefit from this proven infrastructure.

1.1 Keywords

Ada, executive, portability

2. INTRODUCTION

The Boeing Open Systems Architecture (OSA) is a suite of reusable applications and core components for Command, Control and Communication (C3I) systems. This reusable software is currently being used for several defense contracts in the domain of airborne and ground warning and control systems. Civilian applications include maritime search and rescue, fire fighting, transit, and flight deck systems.

OSA was originally written in Ada83 and has been ported to Ada95 [2]. New components and revisions are incorporating the richness of the Ada95 constructs and solving problems that were not easily solved with Ada 83.

A layered architecture is used to isolate dependencies and minimize coupling. Units may "with" any package provided as an Application Programmer Interface (API) in a lower layer. Units may not "with" packages from higher or adjacent layers. This allows systems to evolve without impacting proven components.

The lightweight executive contains a cooperating set of scheduling and data management services. Several Boeing subcontractors are using the lightweight executive framework to develop subsystems that interface with OSA . The lightweight executive encapsulates common services and isolates them in a layer between application plugs, the COTS operating system, and Ada Runtime Environment as shown in Figure 1.

3. LIGHTWEIGHT EXECUTIVE REQUIREMENTS

There are many requirements common to most airborne computing systems. Ground based systems also share many of the same requirements. The lightweight executive evolved to support multiple command and control products while maintaining a single source baseline. Ada95 was instrumental in meeting our single baseline objective.

All C3I systems go through a startup, operational and shutdown mode. These systems may need to recover from an inadvertent failure (e.g. lost of power, or a failure of hardware). System wide events also require synchronization.

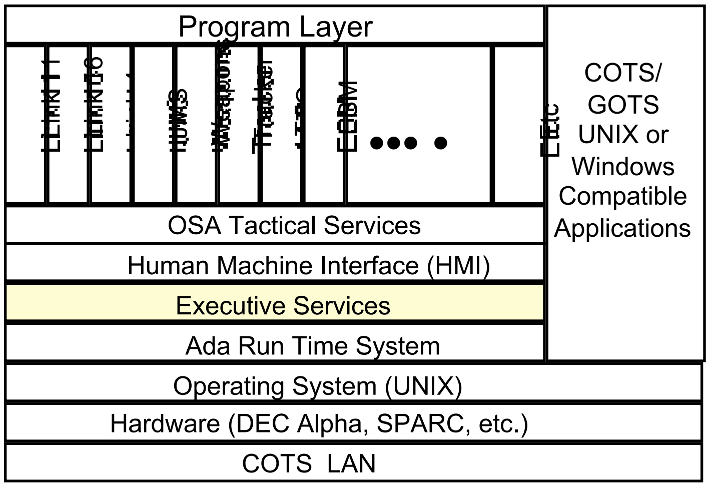


Figure 1 OSA Software Architecture is layered to isolate dependencies

Data recording and playback is another key requirement. What did the operator really see before he committed a fighter to that target? The playback needs to recreate exactly what was in the system. This involves not only recording and time stamping data but a mechanism to synchronize time during playback.

In a large complex system, developers, testers, and end-users require visibility into the system's full-up operational behavior. Performance monitoring, common logging and trace back mechanisms provide the needed visibility. These features allow hyper-accurate isolation/location of problem code. These continue to be helpful in the maintenance phase of a program.

Communication between system nodes is critical for building integrated systems. Both large synchronous data base broadcasts, and small point to point asynchronous message transmission are required. A configurable network topology is needed to support development, integration testing and operational system configurations.

3.1 Why Lightweight?

The lightweight executive utilizes the standard Ada runtime features and a minimum of standard operating system libraries. There are many other services within OSA that depend on larger compile and runtime libraries (e.g. X libraries [3]). The lightweight executive deliberately avoids these in order to maintain its smaller size and thus utility for our thinner subsystems. This also proves useful in writing prototypes and test drivers. The link time is significantly increased when larger libraries and OSA tactical services are included. A shorter link time permits a developer to quickly iterate through the (design-a-little, code-a-little, test-a-little) development cycle. This encourages incremental changes which are easier to troubleshoot. The result is a higher quality product.

The founding goal of OSA is to leverage off COTS hardware and software through the use of proven standards. "Just in time hardware" can be selected based on performance requirements, budget or compatibility with existing systems. Dependence on specific hardware, runtime kernel and compiler vendor must be minimized. The lightweight executive's role is to isolate these dependencies in a minimal set of environmental specific library units.

The lightweight executive utilizes the Ada95 runtime and leverages off of compiler vendors providing the port to other environments. We are hindered when vendors don't provide the complete language implementation including the Annexes which are very appropriate to our application domain.

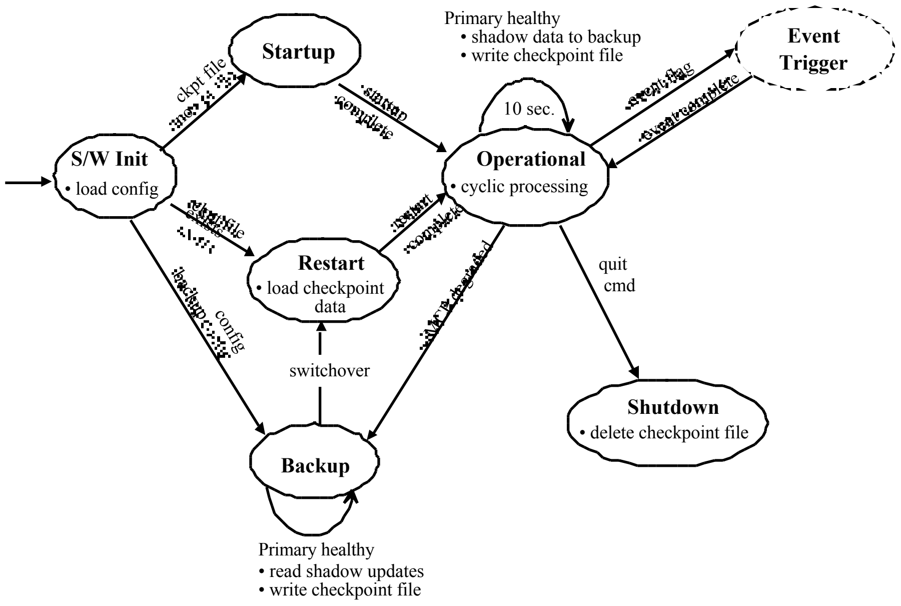
An application plug-in is free to utilize other COTS services to interface with external components. The plug-in application can be removed if support for it isn't provided for a particular environment. OSA has plug-in applications that interface with systems that utilize CORBA [4] and DIS [5] standards.

4. LIGHTWEIGHT EXECUTIVE SERVICES

4.1 Scheduling and Mode Control

The OSA plug and play scheduler provides synchronized startup and shutdown. Cyclic processing may be synchronous, asynchronous or a combination of both. Handling of system event triggers is coordinated. Additionally, failure recovery control is invoked during a restart or a switch over from backup processor mode. These system states are shown in Figure 2. Current OSA event triggers are time reset (used when starting playback), coordinate system jump (mission databases require translation), and purge simulated data. Other event triggers can be specified. The event trigger causes cyclic processing to halt and each application event handler is called. Cyclic processing then resumes.

An application utilizes the abstract plug type and defines the required subprograms. These subprograms represent the processing required during the corresponding system states



(startup, restart, cyclic, shutdown and event trigger). A plug represents a system capability (e.g. navigation). A plug can be synchronous or asynchronous.

Synchronous plugs are executed in the same cyclic task. Their cyclic rate is defined relative to a base frame cycle. An offset cycle can also be specified. This allows 2 heavy plugs to run at the same frequency but at different beats. The advantage of utilizing synchronous plugs is less overhead associated with a separate task and resource locking is not required. Each plug runs its cycle to completion. Legacy code that doesn't provide resource locking can be configured to run synchronously. The disadvantage is that a heavy plug can overrun its time budget. This may cause other plugs to be late starting their cycle. This adds latency to the processing performed by the plug.

Asynchronous plugs execute in their own cyclic task. If a plug provides a resource to another plug in the system it needs to control its own resource locking. The Ada95 protected object provides an efficient and reliable resource locking mechanism. An asynchronous plug will not be late because of processing by another plug. It may still be late due to limited CPU resources. A priority is specified when an asynchronous plug is defined. This allows the critically of the plug to be specified. Using rate monatomic analysis to set the priority will result in optimal system schedulability. (e.g. the highest frequency plug gets the highest priority, etc.) [6][7]

4.2 Time Services

In a command and control system, time plays a particularly critical role. Many independent tactical sightings are reported from a multitude of 'sensors.' Each sensor report is tagged with a time. Knowledge and identification of a given object comes from combining and correlating these events using characteristics of the object, position data and the time of the reports.

Time can be set, reset or speeded up according to playback Because the system can be run in a live mode, or in a simulation or playback mode, time can be live or virtual. An example of the need for virtual time is the playback of recorded data. The state diagram in figure 3 shows events which trigger the time mode changes.

The family of time services is shown in figure 4. Mission_cal.time is an extension of calendar.time. Most operations are inherited from the parent, calendar, but clock is overloaded. It is here that the OSA time services can dictate just what the time really is in "OSA land". Clock returns the real (system) time or some virtual time. The time mode is transparent to all the other software components.

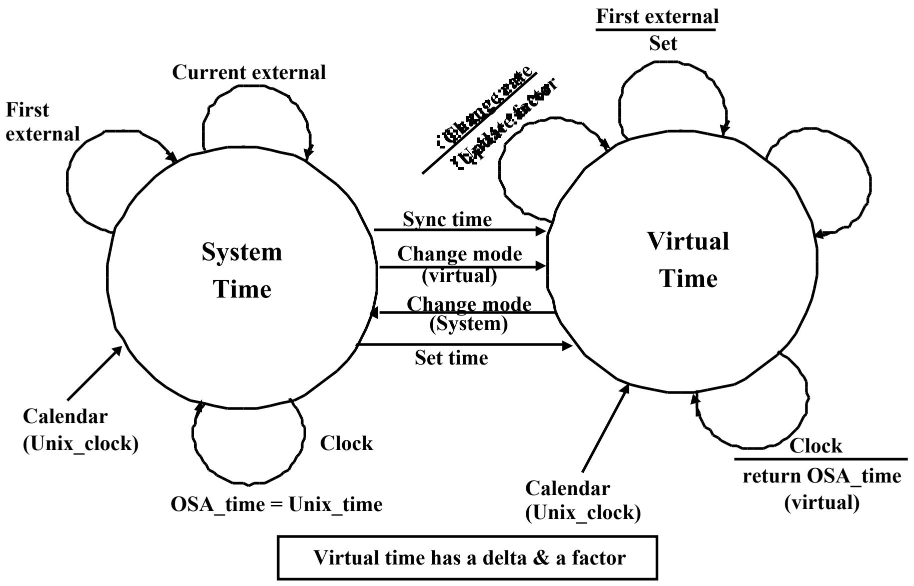


Figure 3 OSA Time Service supports a live or virtual mode.

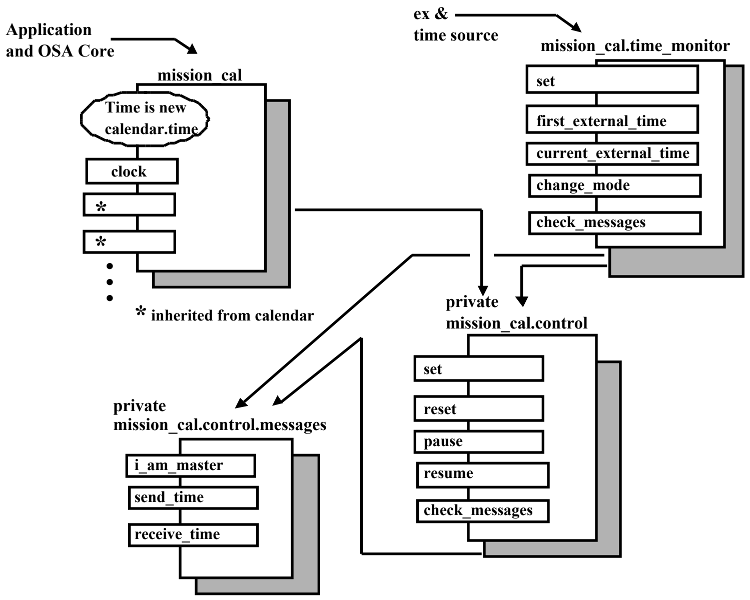


Figure 4 OSA Time Service package hierarchy protects private data from public visibility.

The time services packages are designed to minimize recompiling but still accommodate new requirements. It is important that the parent package, mission_cal, remain stable because it is "withed" by almost every package in OSA. On the other hand, Mission_cal.time_monitor is 'withed' by very few packages. It is here that the time is shifted. Only an outside source of time needs to call these routines. This package can be changed with little recompiling overhead or risk breakage of the parent unit. For another system, the time source might require the concept of a 'tick' to synchronize time. The interface and logic can easily be added to mission_cal.time_monitor and mission_cal.control with mission_cal itself not affected. Mission_cal.control is a private child and its specification safely contains the delta and factor that are used to produce a virtual time.

Another child package, mission_cal.control.messages provides a distributed time service. It currently uses the OSA Network Management services. As we continue to develop this distributed service we plan to use the Distributed Annex-E. When vendor support for the Distributed System Annex matures this will be the preferred implementation. This, of course, will make it necessary to change the base type, mission_cal.time, to a pure type. [8]

4.3 Recording and Playback

The Lightweight Executive provides general record and replay devices. These can be extended and used in composition with data stream objects such as communication tasks.

Record_replay supports recording and playback by performing all the storing and retrieving of data to and from files. As shown in figure 5, the parent package, record_replay provides the interface for declaring private storage objects, opening and closing of files, retrieving the next time, obtaining system status, etc. A child package record_replay.types is a generic that can be instantiated for any type and which provides store and retrieve functions only. Another child package, record_replay.bytes is used to record and replay byte arrays of any size and it provides the store and retrieve functions only.

All functions have an identifier argument which points to an instance of the private storage object. The data, which includes a time stamp for each entry is stored in Ada streams contained in the private type.

File access is performed from another flow of control. A lower priority task reads from and writes to files by calling read and write routines in the body of record_replay using tagged type dispatching. Because the temporary buffers are streams, it is possible to read and write heterogeneous files. The rr_streams package can be used as a standalone utility, as well.

4.4 Communications

OSA Network Management/Configuration provides a solution to the general problem of asynchronous communications between two tasks. The tasks can either reside within the same process or in different processes. Within OSA, inter-task communication is encapsulated into three component types:

- FIFOs: Provide a standard interface for sending and receiving messages between all tasks.
- Socks: Encapsulate communication between two processes using IP based protocols via Berkley Sockets.
- Network Configuration Tables: Provide a static, service-mapped view of the hosts/ports composing an OSA system.

Communication between two tasks, within a single process, requires only a mutual FIFO. Communication

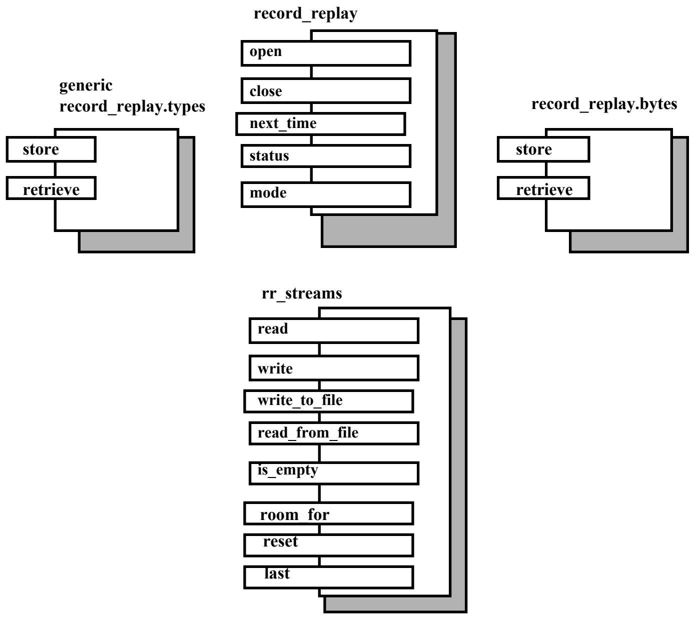


Figure 5 OSA Record Replay Interface

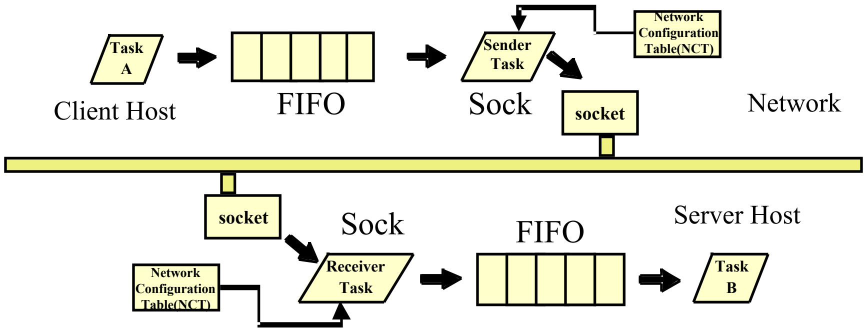


Figure 6 Communication components isolate applications from the network layer.

across multiple processes requires: Socks to provide an application layer for IP, and a Network Configuration Table. Note that, although extra functional layers are required once communication is extended beyond a single process, the extra complexity is easily hidden from the communicating tasks (Task A and B in Figure 6). For each participating process, the definition and instantiation of each sock task is coordinated by the executive.

The FIFO implementation provides a "data" focus point at which general non-type related operations on data are easily implemented. For example, by providing hooks for marshalling and unmarshalling functions on the FIFOs, OSA can easily support operation in heterogeneous environments.

Several different FIFO behaviors are available and can be tightly fitted to the behavior of the data being transmitted. Some of the different behaviors provided include, static versus dynamic message size, queued data versus latest/freshest data, and database array updates that operate transparently for an entire array.

Socks can be associated with one or more FIFOs and have a configurable task priority. This configurable a allows an OSA system architect to group F streams by their priority, and then associate that g a Sock that has the desired task priority. granularity of control over network comm provides determinism desired in real-time, systems.

Socks also encapsulate either TCP or UDP prot implement a retry mechanism for both initializ message transmission. Sock tasks support both p blocking I/O.

Network Management can also record and r traveling through a Sock. Since each Sock coin

a service defined in the Network Configuration Table, the operator can easily re-create the system state attained from network data received by OSA.

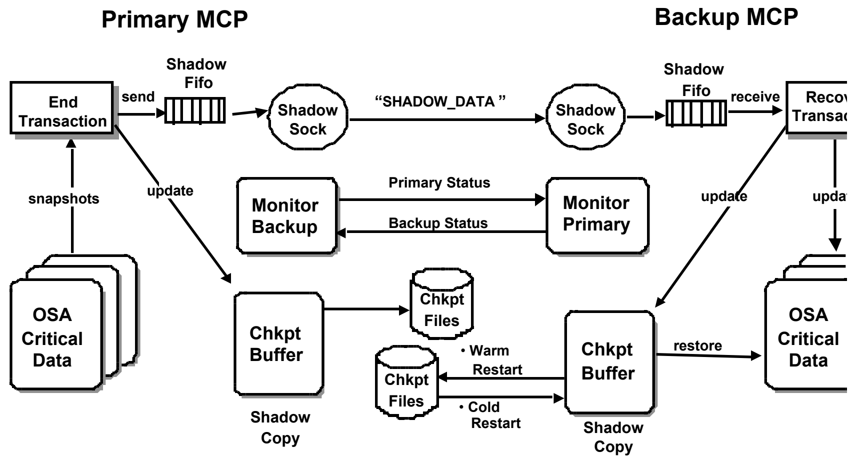
Given the universality of Berkley Sockets and IP, OSA can support system members across several different types of networks (Ethernet, ATM, and FDDI). [9] [10] The implementations of Object Broker standards, such as CORBA, incur a processing overhead that is not justified for our target environments, which are mostly homogenous in platform and operating system. By creating our own Network Management services, we avoid the extra processing overhead, and maintain a greater control over system architecture networking constraints.

4.5 Failure Recovery

System failure recovery requires the following provisions:

- Health and status monitoring for detecting failure
- Snapshot of system state to persistent data storage and/or duplication of system state on a redundant system
- Recovery control mechanism

Health and status monitoring is provided between a



The checkpoint buffer is periodically written to disk by a low priority task. It alternates between two files and places a time stamp at the end of the file. This time stamp is used to recover the latest complete system snapshot. When a backup process is connected the checkpoint copies are also sent to the backup. This is illustrated in figure 7.

The final part of failure recovery is the recovery control mechanism. This is embedded in the OSA scheduler. If the presence of checkpoint files are detected then a restart instead of a startup is performed. Applications restore their data from the checkpoint buffer using the provided API. The same processing state occurs when a backup switches to primary. Figure 2 shows this state diagram.

4.6 System Monitoring and Logging

The package perfm collects runtime timing data and outputs statistics upon request. The OSA scheduler uses the performance monitor to instrument plug-in applications. Applications can use perfm directly for finer timing granularity. Performance monitoring is enabled/disabled by a configuration file that is read during system startup.

The package save_error is used by the OSA scheduler to instrument which plug-in application has been called. In the event of an exception each task is output with the current user.

Enhancement plans include providing a standalone monitor program which can view the state of save_error and perfm packages.

The simple_alert package provides an event log file. This low level capability stores messages for latter analysis. Higher level packages send alerts to operators. The simple_alert package provides a similar service with no dependencies on the display software.

The logging package provides text output routines that can be enabled and disabled during a run. Logging can be used for debugging purposes or system process information. A name is registered by an application and if the name also exists in the logging configuration file, the data associated with the name will be buffered and written to a file, "name".log. Logging to a file can be turned on and off by adding names to and deleting names from the configuration file. The file is inspected at five second intervals. Names can be nested within families and the output associated with groups of names can all go to the same file if the family is activated, i.e. the family name is contained in the config file. A lower priority task does the actual file access. In addition to the family structure, there are 3 priority levels: debug, information, and system. Debug is the most verbose and system is reserved for the fewest messages. The level of priority is established by the appropriate keyword in the configuration file.

Figure 7 OSA Checkpoint function provides cold or warm failure recovery.

5. FUTURE DIRECTIONS

The current Lightweight Executive runs as a single Ada program. A system can have multiple instances of the Lightweight Executive. Network management and time services currently provide cooperation between services across program boundaries. Our plan is to extend this to the complete set of services with minimal change to the API. This capability would allow an OSA system to be divided into smaller, easier to manage programs. This will assist developers and significantly reduce link, load and debug time. Performance is always a concern and difficult to anticipate when supporting multiple product lines with varying system capacities. These smaller program partitions give us flexibility in selecting a hardware configuration that will meet the required performance.

6. CONCLUSIONS

The lightweight executive consists of about 10,000 Ada semicolons. The dynamic configuration features of Ada95 allow systems to be configured during startup instead of compile time. The previous "all knowing" executive required under Ada83 can be stripped down to "know nothing" at compile time. Published APIs and Ada95 child units help maintain stability to our customers but still allow developers to add capabilities. The real-time additions to Ada95 provide independence from vendor supplied capabilities previously used under Ada83. Our strategy relies on compiler vendors providing mature and complete Ada implementations.

7. ACKNOWLEDGMENTS

The work described in this paper was performed under Boeing Internal Research and Development Funds. The authors want to thank our Management Sponsors that entrusted us to carry out this vision. Additional thanks to the OSA development team for their contribution to the evolution of the lightweight executive.

8. REFERENCES

[1] ANSI/ISO/IEC-8652:1995, "Ada 95, The Language Reference Manual and Standard Libraries," January 1995.

[2] S. Moody, "Migrating Well Engineered Ada 83 Applications into Newer Architecture and Reuse Based Ada 95 Systems," Proceedings of the Tri Ada conference, Philadelphia, PA, 1996.

[3] X11 Ada: Ada95 X-Windows Bindings, http://www.inmet.com/~mg/x11ada/x11ada.html, March 1996.

[4] The Common Object Request Broker Architecture and Specification; Revision 2.0: Object management Group, Inc., Framingham, MA, July 1995.

[5] "Standard for Distributed Interactive Simulation - Application Protocols", Version 2.1.2 (Working Draft)

Sept 14, 1995; Institute for Simulation and Training, 3280 Progress Drive Orlando FL 32826.

[6] A. Burns, A. Wellings, "Concurrency in Ada," Cambridge University Press, 1995.

[7] ANSI/ISO/IEC-8652:1995, page 9-2, "Ada 95 Rationale, The Language and Standard Libraries," January 1995.

[8] S. Moody, "object Oriented Abstractions for Real-Time Distributed Systems. Foundation Steps of Ada95 Purity and Generics". http://www.seanet.com/~moody/papers/europe98.ps.gz

[9] M. Laubach, "Classical IP and ARP over ATM," RFC 1577, January 1994.

[10] D. Katz, "Transmission of IP and ARP over FDDI Networks," RFC 1390, October 1990.