# USE OF ADA FOR SHIPBOARD EMBEDDED APPLICATIONS

D. F. Sterne, M. E. Schmid, M. J. Gralia
The Johns Hopkins University
Applied Physics Laboratory
Laurel, MD

T. A. Grobicki
SYSCON Corp.
Columbia, MD

R. A. R. Pearce
TAAG, Inc.
College Park, MD

## Introduction

The Johns Hopkins University Applied Physics Laboratory (APL) performs a variety of research, engineering, and advisory activities under contract to the U.S. Navy. The Navy's planned transition to the use of the Ada* programming language and its associated technology has become an area of increasing APL involvement. APL serves on the KAPSE Interface Team (KIT), the APSE Evaluation and Validation Team, and several Navy-specific Ada working groups. In addition, APL has initiated the applied research project described below.

The general goals of the project are to explore the issues of applying Ada technology to Navy tactical software systems and to provide lessons learned for future combat system upgrades. The primary vehicle for achieving these objectives is the top-down redesign, in Ada, of a simplified version of a tactical software system, the AEGIS Command and Decision System computer program (C&D). This program coordinates the activities of shipboard sensor and weapons subsystems in response to operator commands and automated rules of engagement. The objective of redesigning the program is not to produce operational software but to identify problems and successful techniques of applying Ada to a typical embedded system.

The redesign of the C&D program is proceeding in three steps, or "cuts." During each cut, a simplified executable prototype or "model" of the program will be built, based on functional requirements selected from the C&D program performance specification. Each model will be of wider scope and higher fidelity than its predecessor, and will provide an opportunity to introduce new design decisions and techniques and to discard others.

## Results

The First Cut, including documentation, has recently been completed by a team of five people after approximately six months and slightly over one man-year of effort. The First Cut uses a Digital Equipment Corporation VAX-11/780 computer and the Unix operating system for both program development and program execution, and an Ada compiler developed by the Department of Computer Science at the University of York, England. The First Cut model implements only a few highly simplified C&D functions. These functions include managing a small file of track reports from a single simulated sensor, displaying track positions, velocities, and engagement status on a Navy standard AN/UYA-4 (OJ-194) console (Fig. 1), and responding to operator commands to engage tracks, drop tracks, and change track identification. The First Cut also includes a "wraparound simulation program" (WASP) to create input stimuli and record output responses (Fig. 2). The WASP and C&D model

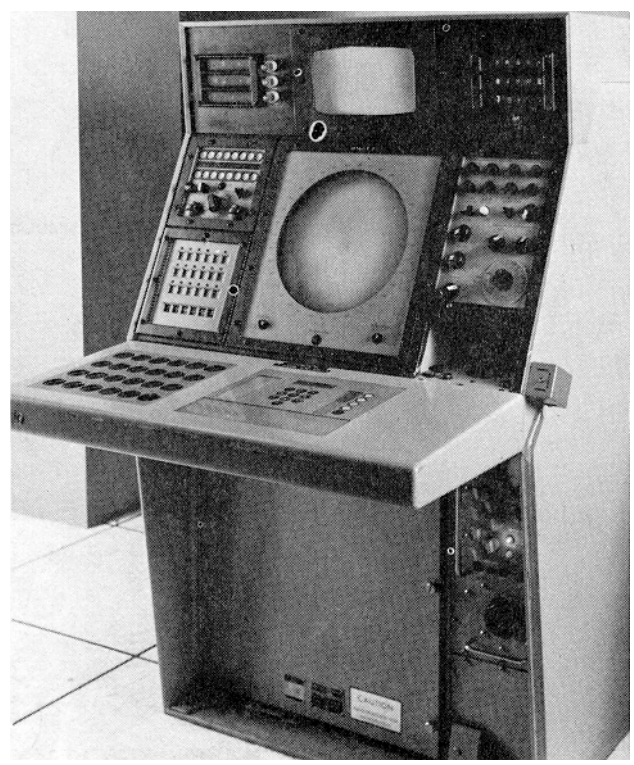

**Fig. 1 AN/UYA-4 operator console.**

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

together comprise approximately 4000 lines of Ada code excluding comments. Approximately 70% of the code is concerned with C&D functions; the remainder belongs to the WASP subsystem. In addition, approximately 2000 to 3000 lines of test software (drivers and stubs) were developed.

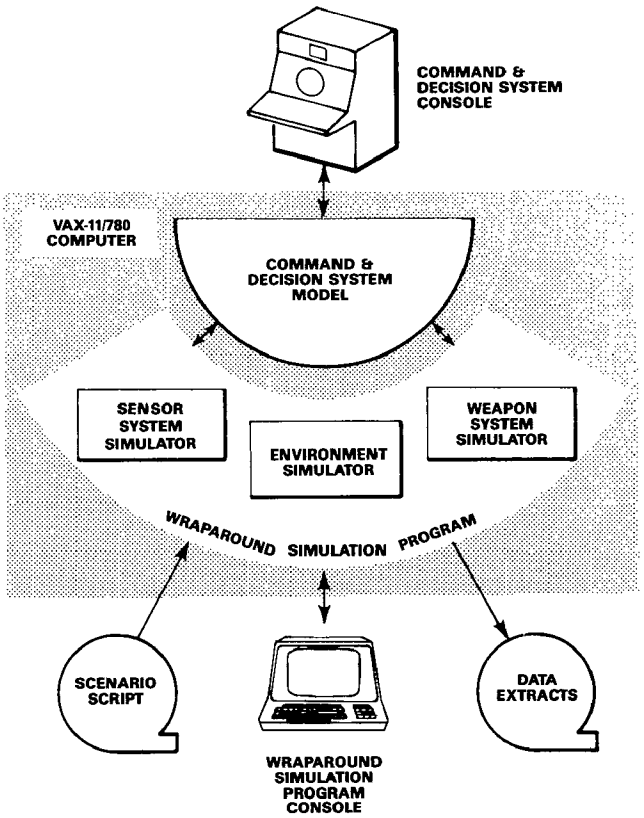


Fig. 2 First Cut system software/hardware configuration.

Figure 3 provides a top-level view of the system. The WASP, operating under interactive operator control, reads a scenario script of events occurring at specified simulation times. These events generally concern the appearance or disappearance of objects detectable by radar (ships, aircraft, and missiles), or changes in the velocities of these objects. Periodically, the WASP sends simulated radar reports of detected object positions and velocities to C&D, where they are stored in a memory-resident "track file." Tracks not reported by the radar simulator are eventually deleted from the track file. The C&D operator can delete tracks from the track file, causing further reports for the deleted tracks to be ignored.

A geographic display of relative track positions and velocities is maintained on the AN/UYA-4 plan position indicator. The C&D operator can request that the system provide additional information about tracks on the AN/UYA-4's alphanumeric display. The operator can also identify a track as being friendly or hostile and can engage hostile tracks. Engagement orders are sent to a simulated weapon control system, where an intercept between the target and a surface-to-air missile is calculated. At the predicted intercept time, a pseudo-random hit or miss decision is made based on an intercept probability parameter. The results are sent back to the C&D system, where they are stored in the track file and reported to the C&D operator. Targets that have been successfully intercepted are no longer reported by the radar simulator, and subsequently are dropped from the track file. Because the track file can be accessed by several processes concurrently, it includes a mechanism for exclusive access during update operations.

The WASP operator can change the intercept probability, the rate at which simulated time advances, and the radar reporting rate at any time during a simulation run. In addition, the WASP operator can request that data extracts (state information snapshots) be produced by various system components.

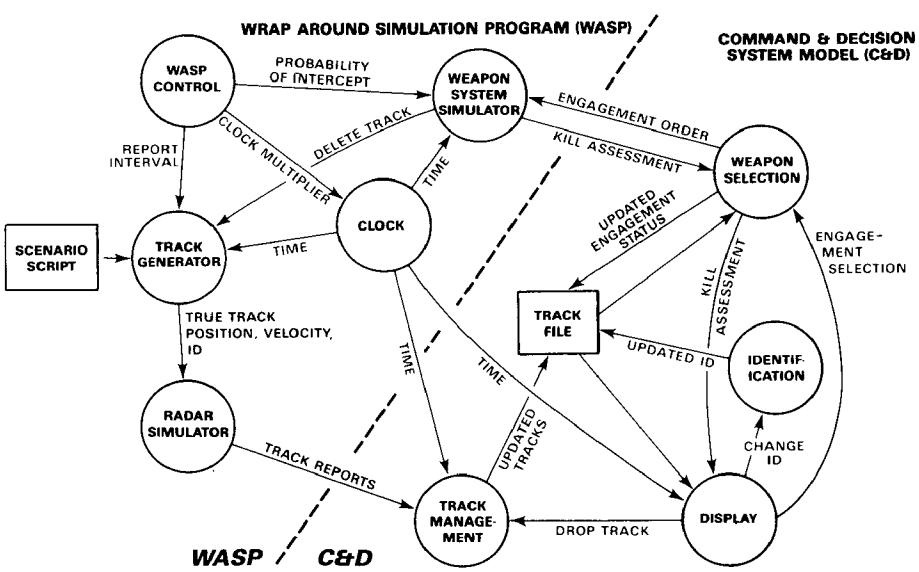


Fig. 3 First Cut system top-level data flow.

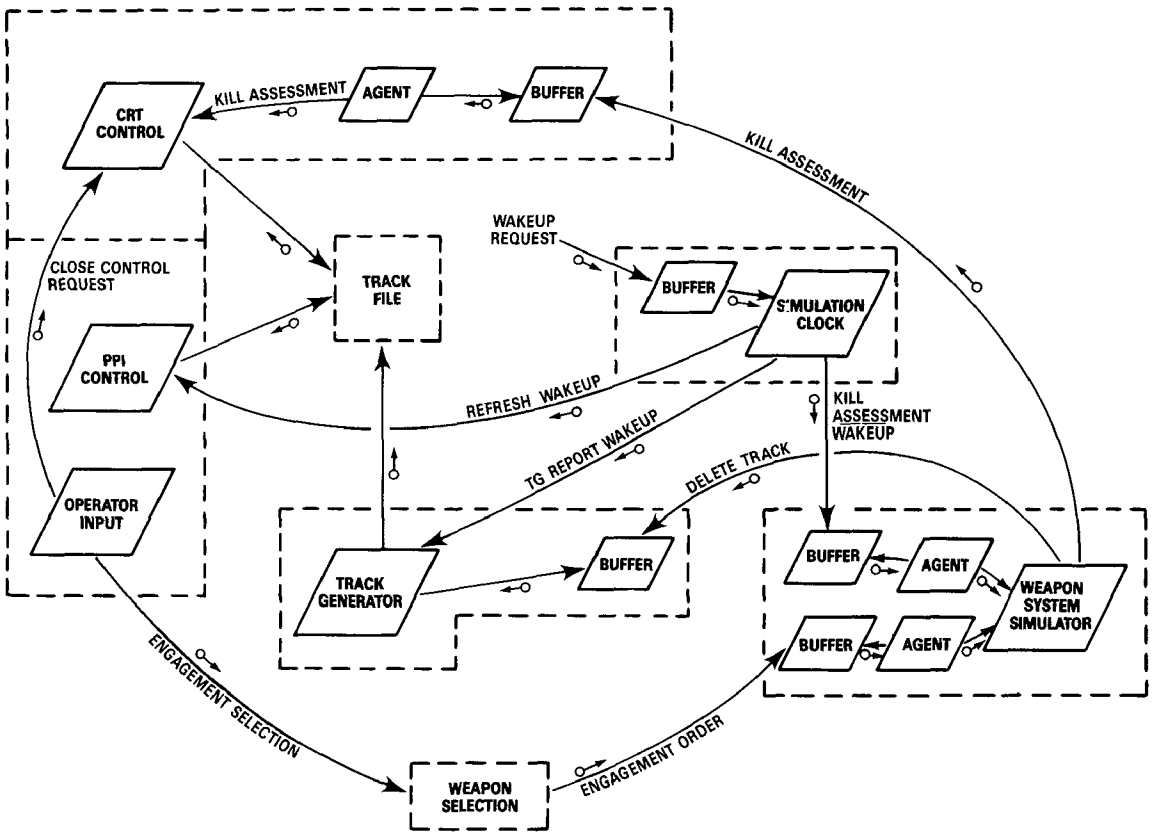


Fig. 4 Top-level package and task organization.

Structured analysis [1] was used to depict the selected program performance requirements, partition them into functions or processes, and identify the inter-process data flow. The top-level partitioning, as shown in Fig. 3, is based on the division of functions in a typical shipboard combat system. Each process was described using pseudo-code or informal program design language.

Major threads of control in the system were then identified, and the processes that embodied them were designated as active processes. The remaining processes were designated as passive, meaning that their external behavior was driven by the active processes with which they communicate. Using a graphic notation based on the ideas of Booch [2] and Buhr [3], top-level control flow decisions, deliberately omitted from the data flow diagram, were sketched in for each top-level data flow. These decisions must be made before deciding which services shall be provided by various Ada packages that will ultimately make up the system. For example, if data must flow from process A to process B, which process should invoke the data-transfer operation? Clearly, if the processes reside in separate Ada packages, the answer must be known before the package interfaces can be defined.

Next, each active process was assigned to one or more Ada tasks, and additional third party tasks (agents and buffers) were added to decouple intertask communication and to avoid obvious opportunities for deadlock. Each passive process was treated as a subprogram or package of subprograms. Active and passive processes were then grouped and encapsulated within Ada packages to provide access control and information hiding. Figure 4 shows a portion of the resulting top-level program architecture (some components are not shown). Note that the packages, shown as dotted lines, are only "roughed in;" the visible parts of the packages are not explicity identified.

Interfaces for the top-level Ada packages were then specified, identifying the data types, callable subprograms, parameters, and exceptions each package makes accessible to the other packages in the system. The package specifications were compiled to assure consistency. Top-level packages were then assigned to the various team members, who carried out detailed design and coding based on the top-level task structure diagram, the compiled package specifications, and the functional specifications captured in informal program design language. A more detailed Ada-based graphic notation was used in the formulation of lower-level packages and tasks. As an example, Fig. 5 shows the Ada diagram developed during the detailed design of the "clock" package.

The First Cut program was completed on schedule without significant problems. Figure 6 shows a sample AN/UYA-4 display generated by the program.

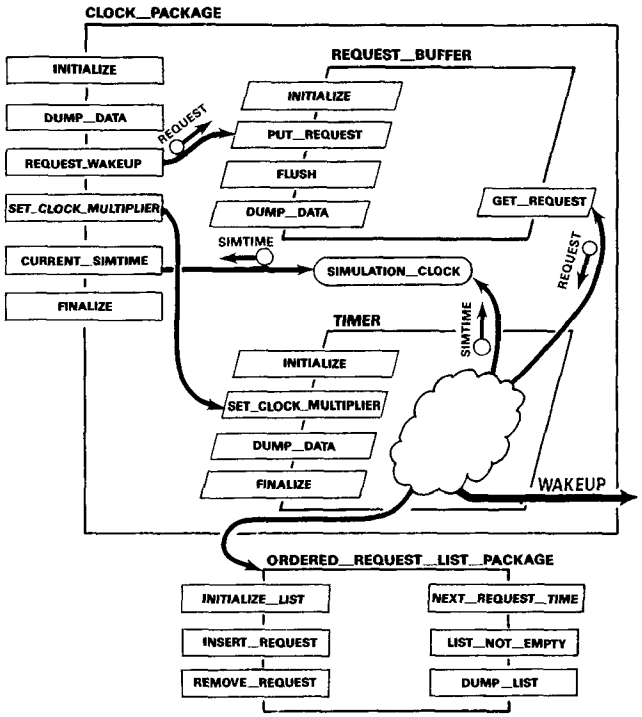


**Fig. 5 Ada diagram of the simulation clock package.**

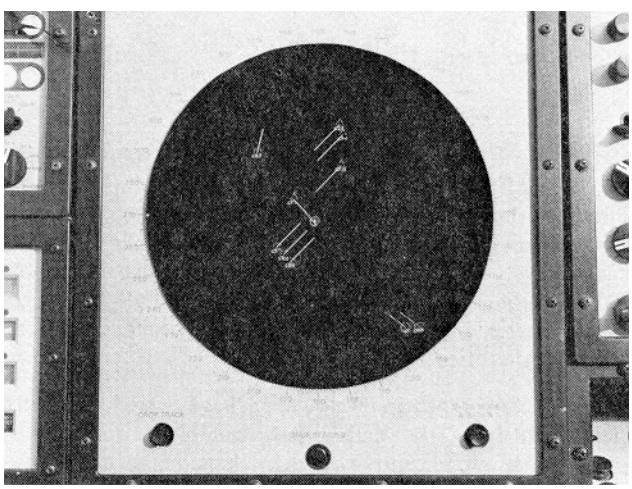

**Fig. 6 Graphic display generated by First Cut program.**

## Evaluation

As a whole, our experience with Ada has been overwhelmingly positive. In particular, the integration phase proceeded remarkably smoothly; this was attributed to Ada's automatic verification of interface consistency among packages, and its strong typing rules. Automatic error recovery was easily implemented using Ada's exception mechanism. This mechanism allowed the 15 tasks in the system to be coded for crash-proof execution.

In general, Ada's tasking features were found powerful and easy to use, and valuable as a conceptual tool for problem decomposition. However, we frequently chose to add or were forced to add third party tasks to decouple intertask communications. In freely adding third party tasks, we shared the concern expressed by others [4] that excessive interactions with the runtime scheduler would result, significantly reducing the efficiency of the system. We found the combination of a buffer task and an agent task (Fig. 7) especially useful in several circumstances in which buffered communication was required between two tasks and the destination task was unwilling to be suspended while waiting for data to enter an empty buffer, or to repeatedly check the buffer's status. Gehani [5] cites situations like these as evidence of Ada's "polling bias." This combination of a buffer and an agent constitutes a tasking idiom similar to idioms suggested by Buhr [3], Maclaren [6], and Hilfinger [7]. We agree with Hilfinger that Ada compiler writers must strive to make these and other idioms especially cheap computationally so that designers may use them liberally in their designs. Because this issue is of such importance, we recommend that execution speed tests of a collection of tasking idioms be developed as an essential part of any serious Ada compiler and runtime system evaluation.

```
task sender;

task body sender is
     m:message;
begin
     loop
        .
        .
        .
        buffer.put(m);
        .
        .
     end loop;
end sender;
```

```
task agent;

task body agent is
    m:message;
begin
    loop
       buffer.get(m);
       receiver.put(m);
    end loop;
end agent;
```

```
task receiver is
    entry put(m: in message);
    entry other_service (...);
end receiver;

task body receiver is
.
.
.
begin
    loop
        select
            accept put (....) do
            .
            .
            .
            end put;
        or
            accept other_service (...) do
            .
            .
            .
            end other_service;
        end select;
    end loop;
end receiver;
```

```
task buffer is
    entry put (m: in message);
    entry get (m: out message);
end buffer;

task body buffer is
    .
    .
    .
    - - standard bounded buffer
    .
    .
    .
end buffer;
```

**Fig. 7 Buffer/agent tasking idiom.**

Another issue concerning Ada's tasking features arose in developing the wraparound simulation program. To provide flexibility in demonstrating and debugging the system, we wanted to be able to start, stop, hasten, and retard the simulation clock, thereby controlling rates at which targets appear and travel, and the rates at which time-related processing in the C&D model occurs. This requirement proved at odds with Ada's time-related constructs. The time base that regu-

lates the expiration of Ada's delay statements and rendezvous time-outs is hidden within the Ada runtime system and is, hence, not under programmer control. We first considered including a global time multiplier in all delay and time-out statements that would scale all time interval values up or down to slow or accelerate time-dependent processing. Although this scheme would require an infinite multiplier value to stop the system clock, near-stops could be achieved with very large multipliers. However, rate changes would not take place uniformly, because delay statements in progress when the multiplier changed would continue at the old clock rate. This effect would create major problems when attempting to resume a normal execution rate after a near-stop.

Instead, we constructed a simulation clock package (Fig. 5) containing its own time multiplier, which can be set by the WASP operator. All delay statements in the system were then replaced by procedure calls to the clock package. These calls request the future delivery of an "alarm" at a specified simulation time. The clock package keeps track of real elapsed time and maintains a list of pending alarm requests. At the appropriate simulation time, as governed by the multiplier, the clock package sends alarms (rendezvous calls) to the requesters. This scheme provides a satisfactory substitute for Ada's delay statement but appears inadequate for imitating Ada's rendezvous time-outs. This is due to the inability to correctly cancel either the rendezvous or the time-out, depending on the amount of simulated time that has elapsed.

It can be argued that the facilities we sought are more properly provided by a runtime symbolic debugger, or by allowing the programmer to make service requests to the runtime system through an additional privileged interface. In either case, we believe these facilities are not particularly exotic, and that they should be available in one form or other on most development systems.

The University of York Ada Workbench Compiler System (Release 1) was chosen because it appeared to be the best of several candidate compilers available to the project. Nevertheless, the completion of the First Cut was impeded by shortcomings in the compiler and runtime system including unimplemented features, incorrectly implemented features, and task-scheduling inefficiencies. Based on our experience with several other compilers, we believe these shortcomings to be representative of the limitations of most Ada compilers available in 1984. Technology immaturity still represents a major obstacle to using Ada, even for research purposes. The University of York compiler was designed to be used with several existing Unix tools including a symbolic debugger, a tool that enforced recompilation order rules, and a performance analyzer. These tools proved invaluable, and similar tools should be considered mandatory for the development of any large real-time system. In particular, the performance analyzer identified time-of-day calculations hidden within the calendar package that consumed 20% of the First Cut program's execution time. By substituting a simpler subprogram tailored to our needs, the consumption dropped to 5%.

A few significant tool limitations were encountered, stemming from a lack of cooperation between the Unix tools and the Ada compiler. For example, under some circumstances, we were unable to make the Unix symbolic debugger recognize the names of variables declared in the outermost block of a task or package. The Unix profiler captured execution timing statistics for procedures, but not for tasks. Since procedures can be shared by many tasks, the statistics provided few clues about the distribution of execution time across tasks. These experiences suggest that other existing non-Ada tools may need to be extended or modified to support the greater demands of the Ada language.

In the absence of a recognized comprehensive Ada-based development methodology, the collection of software development techniques used in this project appears to be a good starting point for an interim methodology. We believe that structured analysis should be augmented by including estimates of data traffic frequencies along key data flows and required response time for key processes. These estimates are needed to provide a basis for choosing between task structuring options during the design phase. Our experience suggests that Ada-based graphic notations are needed during the initial, high-level design phase to bridge the gap between functional specifications expressed as data flow diagrams and detailed design typically expressed in a program design language.

## Remaining Issues

Despite the success of projects like this one that have used Ada for laboratory prototypes, for small non-real-time applications, or as a program design language, several crucial questions about Ada remain unanswered. The most important question is whether Ada programs will run too slowly or occupy too much memory to be useful for real-time embedded applications. Unfortunately, until compilers and runtime system mature, no one can accurately predict what the ultimate speed and memory limits for Ada programs may be. Thus, a commitment to begin implementing a large time-critical system in Ada today, on a tight schedule, entails substantial risk. This risk will diminish as compilers and runtime systems improve and as the technical community develops more experience using Ada in real-time embedded applications.

Based on our contact with tactical embedded systems and people who have built these systems, we believe that the greatest obstacle to acceptance of Ada is a philosophical one. The Ada "design philosophy" is based on the premise that it is possible to build acceptably efficient real-time programs that are machine independent, that are built from reusuable components, and that are relatively easy to maintain. This premise is contrary to current practice as embodied in many existing real-time systems. Most of today's real-time systems had to be painstakingly designed, coded, and tuned to maximize efficiency; in the process, transportability, reusability, and maintainability were inevitably sacrificed. To the designers of these systems, the Ada philosophy appears incompatible with real-world efficiency requirements, and ac-

cepting the Ada philosophy, especially in the absence of representative example systems written in Ada, constitutes a giant leap of faith.

This gap between the philosophy of Ada and the philosophy underlying current practice has important implications that must be recognized and dealt with. If efficiency must be sacrificed to gain these other desirable characteristics, then Ada programs may need to be executed by much more powerful computers. The significance of Ada then goes beyond the software development process, influencing system engineering. More sophisticated tools will be needed so that an application can be tuned and optimized without sacrificing its understandability and maintainability. The importance of these tools is so great that system development schedules should be based on their availability. A software development team for a large embedded system must not only be required to have mastered Ada programming skills, but to have embraced the Ada design philosophy—otherwise the software it develops will be no better than if written in any other language. Thus, sufficient training and competency requirements should be established to insure that Ada does not serve as a facade hiding continued use of outdated design philosophies.

As a programming language, Ada has many attractive and useful properties. However, the excellence of Ada's features alone is not sufficient to insure that use of Ada will produce good results. We believe that Ada can and will be used with great success, but only if larger issues like those described above are clearly recognized and addressed.

## Current Status and Future Work

The First Cut system is the first of three planned prototypes. The Second Cut is currently under way and expected to be completed by the end of 1985. The objectives are to build a larger, higher fidelity model of the AEGIS C&D system (approximately 15,000 lines), and to use this model as a stress test for Ada compiler and APSE technology, and for Ada-based software development techniques.

## Summary

A simplified model of an embedded Navy computer program has been developed using the Ada programming language, environment tools, and Ada-adapted design methods. The experience has made evident the numerous real and potential benefits of Ada, and the immaturity of today's Ada technology. Although temporary technological and cultural obstacles remain, we believe that Ada can be the vehicle for modernization of the software development process and for software cost reduction both within and outside the Department of Defense.

## REFERENCES


1. T. DeMarco, *Structured Analysis and System Specification,* Prentice-Hall, New York, 1979.

2. G. Booch, *Software Engineering with Ada,* Benjamin Cummings Pub. Co., Menlo Park, Calif., 1983.

3. R. J. Buhr, *System Design with Ada,* Prentice-Hall, Englewood Cliffs, N.J., 1984.

4. E. S. Roberts et al, "Task Management in Ada—A Critical Evaluation for Real-Time Multiprocessors," *Software—Practice and Experience,* Vol. 11, pp. 1019-1051, 1981.

5. N. H. Gehani and T. A. Cargill, "Concurrent Programming in the Ada language: The Polling Bias," *Software—Practice and Experience,* Vol. 14(5), pp. 413-427, May 1984.

6. L. Maclaren, "Evolving Toward Ada in Real Time Systems," *Proc. ACM-SIGPLAN Symp. on the Ada Programming Language,* Boston, December 9-11, 1980.

7. P. N. Hilfinger, "Implementation Strategies for Ada Tasking Idioms," *Proc. AdaTEC Conf. on Ada,* Arlington, Va., October 6-8, 1982.